MATLAB®
Object-Oriented Programming

R2018b
How to Contact MathWorks

Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

Object-Oriented Programming

© COPYRIGHT 1984–2018 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.
<table>
<thead>
<tr>
<th>Date</th>
<th>Format</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>New for MATLAB 7.6 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for MATLAB 7.7 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for MATLAB 7.8 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for MATLAB 7.9 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for MATLAB 7.10 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for Version 7.11 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for Version 7.12 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for Version 7.13 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for Version 7.14 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for Version 8.0 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for Version 8.1 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for Version 8.2 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for Version 8.3 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for Version 8.4 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for Version 8.5 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for Version 8.6 (Release 2015b)</td>
</tr>
<tr>
<td>March 2016</td>
<td>Online only</td>
<td>Revised for Version 9.0 (Release 2016a)</td>
</tr>
<tr>
<td>September 2016</td>
<td>Online only</td>
<td>Revised for Version 9.1 (Release 2016b)</td>
</tr>
<tr>
<td>March 2017</td>
<td>Online only</td>
<td>Revised for Version 9.2 (Release 2017a)</td>
</tr>
<tr>
<td>September 2017</td>
<td>Online only</td>
<td>Revised for Version 9.3 (Release 2017b)</td>
</tr>
<tr>
<td>March 2018</td>
<td>Online only</td>
<td>Revised for MATLAB 9.4 (Release 2018a)</td>
</tr>
<tr>
<td>September 2018</td>
<td>Online only</td>
<td>Revised for Version 9.5 (Release 2018b)</td>
</tr>
</tbody>
</table>
# Using Object-Oriented Design in MATLAB

1

## Why Use Object-Oriented Design

- Approaches to Writing MATLAB Programs ........................... 1-2
- When Should You Create Object-Oriented Programs ............ 1-6

## Handle Object Behavior

- What Is a Handle? ........................................ 1-12
- Copies of Handles ........................................ 1-12
- Handle Objects Modified in Functions ....................... 1-13
- Determine If an Object Is a Handle ......................... 1-15
- Deleted Handle Objects ................................ 1-15

## Basic Example

2

## Create a Simple Class

- Design Class ............................................. 2-2
- Create Object ........................................... 2-3
- Access Properties ....................................... 2-3
- Call Methods ............................................ 2-4
- Add Constructor ......................................... 2-4
- Vectorize Methods ....................................... 2-5
- Overload Functions ..................................... 2-6
- BasicClass Code Listing ................................ 2-7
### MATLAB Classes Overview

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role of Classes in MATLAB</td>
<td>3-2</td>
</tr>
<tr>
<td>Classes</td>
<td>3-2</td>
</tr>
<tr>
<td>Some Basic Relationships</td>
<td>3-4</td>
</tr>
<tr>
<td>Developing Classes — Typical Workflow</td>
<td>3-8</td>
</tr>
<tr>
<td>Formulating a Class</td>
<td>3-8</td>
</tr>
<tr>
<td>Specifying Class Components</td>
<td>3-9</td>
</tr>
<tr>
<td>BankAccount Class Implementation</td>
<td>3-10</td>
</tr>
<tr>
<td>Formulating the AccountManager Class</td>
<td>3-14</td>
</tr>
<tr>
<td>Implementing the AccountManager Class</td>
<td>3-15</td>
</tr>
<tr>
<td>AccountManager Class Synopsis</td>
<td>3-15</td>
</tr>
<tr>
<td>Using BankAccount Objects</td>
<td>3-17</td>
</tr>
<tr>
<td>Class to Represent Structured Data</td>
<td>3-19</td>
</tr>
<tr>
<td>Objects as Data Structures</td>
<td>3-19</td>
</tr>
<tr>
<td>Structure of the Data</td>
<td>3-19</td>
</tr>
<tr>
<td>The TensileData Class</td>
<td>3-20</td>
</tr>
<tr>
<td>Create an Instance and Assign Data</td>
<td>3-20</td>
</tr>
<tr>
<td>Restrict Properties to Specific Values</td>
<td>3-21</td>
</tr>
<tr>
<td>Simplifying the Interface with a Constructor</td>
<td>3-22</td>
</tr>
<tr>
<td>Calculate Data on Demand</td>
<td>3-23</td>
</tr>
<tr>
<td>Displaying TensileData Objects</td>
<td>3-24</td>
</tr>
<tr>
<td>Method to Plot Stress vs. Strain</td>
<td>3-25</td>
</tr>
<tr>
<td>TensileData Class Synopsis</td>
<td>3-26</td>
</tr>
<tr>
<td>Class to Implement Linked Lists</td>
<td>3-31</td>
</tr>
<tr>
<td>Class Definition Code</td>
<td>3-31</td>
</tr>
<tr>
<td>dlnode Class Design</td>
<td>3-31</td>
</tr>
<tr>
<td>Create Doubly Linked List</td>
<td>3-32</td>
</tr>
<tr>
<td>Why a Handle Class for Linked Lists?</td>
<td>3-34</td>
</tr>
<tr>
<td>dlnode Class Synopsis</td>
<td>3-34</td>
</tr>
<tr>
<td>Specialize the dlnode Class</td>
<td>3-46</td>
</tr>
</tbody>
</table>
Static Data .................................................. 4-2
  What Is Static Data ........................................ 4-2
  Static Variable ............................................ 4-2
  Static Data Object ....................................... 4-3
  Constant Data ............................................ 4-5

Class Definition—Syntax Reference ........................................ 5

Class Files and Folders ........................................... 5-2
  Class Definition Files ...................................... 5-2
  Options for Class Folders .................................. 5-2
  Options for Class Files .................................... 5-2
  Group Classes with Package Folders ....................... 5-3

Class Components ................................................... 5-5
  Class Building Blocks ...................................... 5-5
  Class Definition Block ..................................... 5-5
  Properties Block .......................................... 5-6
  Methods Block ............................................. 5-6
  Events Block .................................................. 5-7
  A Complete Class ........................................... 5-8
  Enumeration Classes ....................................... 5-8
  Related Information ....................................... 5-9

Classdef Block .................................................... 5-10
  How to Specify Attributes and Superclasses ............... 5-10
  Class Attribute Syntax .................................... 5-10
  Superclass Syntax .......................................... 5-11
  Local Functions in Class File ............................... 5-11

Properties ....................................................... 5-12
  The Properties Block ...................................... 5-12
  Access to Property Values ................................ 5-13
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Methods and Functions</strong></td>
<td>5-15</td>
</tr>
<tr>
<td>The Methods Block</td>
<td>5-15</td>
</tr>
<tr>
<td>Method Calling Syntax</td>
<td>5-15</td>
</tr>
<tr>
<td>Private Methods</td>
<td>5-17</td>
</tr>
<tr>
<td>More Detailed Information On Methods</td>
<td>5-17</td>
</tr>
<tr>
<td>Class-Related Functions</td>
<td>5-17</td>
</tr>
<tr>
<td>How to Overload Functions and Operators</td>
<td>5-18</td>
</tr>
<tr>
<td>Rules for Defining Methods in Separate Files</td>
<td>5-18</td>
</tr>
<tr>
<td><strong>Events and Listeners</strong></td>
<td>5-20</td>
</tr>
<tr>
<td>Define and Trigger Events</td>
<td>5-20</td>
</tr>
<tr>
<td>Listen for Events</td>
<td>5-20</td>
</tr>
<tr>
<td><strong>Attribute Specification</strong></td>
<td>5-22</td>
</tr>
<tr>
<td>Attribute Syntax</td>
<td>5-22</td>
</tr>
<tr>
<td>Attribute Descriptions</td>
<td>5-22</td>
</tr>
<tr>
<td>Attribute Values</td>
<td>5-23</td>
</tr>
<tr>
<td>Simpler Syntax for true/false Attributes</td>
<td>5-23</td>
</tr>
<tr>
<td><strong>Call Superclass Methods on Subclass Objects</strong></td>
<td>5-25</td>
</tr>
<tr>
<td>Superclass Relation to Subclass</td>
<td>5-25</td>
</tr>
<tr>
<td>How to Call Superclass Methods</td>
<td>5-25</td>
</tr>
<tr>
<td>How to Call Superclass Constructor</td>
<td>5-26</td>
</tr>
<tr>
<td><strong>Representative Class Code</strong></td>
<td>5-28</td>
</tr>
<tr>
<td>Class Calculates Area</td>
<td>5-28</td>
</tr>
<tr>
<td>Description of Class Definition</td>
<td>5-31</td>
</tr>
<tr>
<td><strong>MATLAB Code Analyzer Warnings</strong></td>
<td>5-34</td>
</tr>
<tr>
<td>Syntax Warnings and Property Names</td>
<td>5-34</td>
</tr>
<tr>
<td>Variable/Property Name Conflict Warnings</td>
<td>5-34</td>
</tr>
<tr>
<td>Exception to Variable/Property Name Rule</td>
<td>5-35</td>
</tr>
<tr>
<td><strong>Objects In Conditional Statements</strong></td>
<td>5-37</td>
</tr>
<tr>
<td>Enable Use of Objects In Conditional Statements</td>
<td>5-37</td>
</tr>
<tr>
<td>How MATLAB Evaluates Switch Statements</td>
<td>5-37</td>
</tr>
<tr>
<td>How to Define the eq Method</td>
<td>5-39</td>
</tr>
<tr>
<td>Enumerations in Switch Statements</td>
<td>5-41</td>
</tr>
<tr>
<td><strong>Operations on Objects</strong></td>
<td>5-44</td>
</tr>
<tr>
<td>Object Operations</td>
<td>5-44</td>
</tr>
<tr>
<td>Help on Objects</td>
<td>5-45</td>
</tr>
<tr>
<td>Functions to Test Objects</td>
<td>5-47</td>
</tr>
</tbody>
</table>
Functions to Query Class Components ............................. 5-47

Use of Editor and Debugger with Classes .......................... 5-48
Write Class Code in the Editor ........................................ 5-48
How to Refer to Class Files .......................................... 5-48
How to Debug Class Files ............................................ 5-49

Automatic Updates for Modified Classes ............................ 5-50
When MATLAB Loads Class Definitions ............................ 5-50
Consequences of Automatic Update .................................. 5-51
What Happens When Class Definitions Change ..................... 5-51
Actions That Do Not Trigger Updates ............................... 5-52
Multiple Updates to Class Definitions .............................. 5-52
Object Validity with Deleted Class File ............................ 5-52
When Updates Are Not Possible ..................................... 5-52
Potential Consequences of Class Updates ........................... 5-53
Updates to Class Attributes ......................................... 5-53
Updates to Property Definitions ..................................... 5-54
Updates to Method Definitions ....................................... 5-55
Updates to Event Definitions ......................................... 5-56

Compatibility with Previous Versions ............................... 5-58
New Class-Definition Syntax Introduced with MATLAB Software
Version 7.6 ...................................................................... 5-58
Changes to Class Constructors ........................................ 5-59
New Features Introduced with Version 7.6 ......................... 5-60
Examples of Old and New .............................................. 5-60

Comparison of MATLAB and Other OO Languages ............... 5-61
Some Differences from C++ and Java Code ......................... 5-61
Object Modification ..................................................... 5-62
Static Properties ....................................................... 5-66
Common Object-Oriented Techniques ............................... 5-67

Defining and Organizing Classes

User-Defined Classes ..................................................... 6-2
What Is a Class Definition ............................................. 6-2
Attributes for Class Members ......................................... 6-2
Comparison of Handle and Value Classes .......................... 7-2
  Basic Difference ........................................... 7-2
  Behavior of MATLAB Built-In Classes .......................... 7-3
  User-Defined Value Classes ................................... 7-4
  User-Defined Handle Classes ................................... 7-5
  Determining Equality of Objects ............................... 7-8
  Functionality Supported by Handle Classes .................... 7-10

Which Kind of Class to Use ....................................... 7-11
  Examples of Value and Handle Classes ........................ 7-11
  When to Use Value Classes .................................... 7-11
  When to Use Handle Classes ................................... 7-12

The Handle Superclass ........................................... 7-13
  Building on the Handle Class ................................ 7-13
  Handle Class Methods ......................................... 7-13
  Event and Listener Methods ................................... 7-14
  Relational Methods .......................................... 7-14
  Test Handle Validity .......................................... 7-14
  When MATLAB Destroys Objects ............................... 7-15

Handle Class Destructor ........................................... 7-16
  Basic Knowledge ............................................. 7-16
  Syntax of Handle Class Destructor Method .................... 7-16
  Support Destruction of Partially Constructed Objects ....... 7-17
  When to Define a Destructor Method .......................... 7-19
  Destructors in Class Hierarchies .............................. 7-19
  Object Lifecycle ............................................ 7-20
  Restrict Access to Object Delete Method ..................... 7-21
  Nondestructor Delete Methods ................................ 7-22
  Java Objects Referencing MATLAB Objects .................... 7-22

Find Handle Objects and Properties ............................. 7-25
  Find Handle Objects ........................................ 7-25
  Find Handle Object Properties ................................ 7-25

Implement Set/Get Interface for Properties ..................... 7-27
  The Standard Set/Get Interface .............................. 7-27
  Subclass Syntax .......................................... 7-27
  Get Method Syntax ........................................ 7-28
Properties — Storing Class Data

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ways to Use Properties</td>
<td>8-2</td>
</tr>
<tr>
<td>What Are Properties</td>
<td>8-2</td>
</tr>
<tr>
<td>Types of Properties</td>
<td>8-3</td>
</tr>
<tr>
<td>Property Syntax</td>
<td>8-5</td>
</tr>
<tr>
<td>Property Definition Block</td>
<td>8-5</td>
</tr>
<tr>
<td>Access Property Values</td>
<td>8-6</td>
</tr>
<tr>
<td>Inheritance of Properties</td>
<td>8-7</td>
</tr>
<tr>
<td>Specify Property Attributes</td>
<td>8-7</td>
</tr>
<tr>
<td>Property Attributes</td>
<td>8-9</td>
</tr>
<tr>
<td>Purpose of Property Attributes</td>
<td>8-9</td>
</tr>
<tr>
<td>Specifying Property Attributes</td>
<td>8-9</td>
</tr>
<tr>
<td>Table of Property Attributes</td>
<td>8-9</td>
</tr>
<tr>
<td>Property Definition</td>
<td>8-17</td>
</tr>
<tr>
<td>What You Can Define</td>
<td>8-17</td>
</tr>
<tr>
<td>Initialize Property Values</td>
<td>8-18</td>
</tr>
<tr>
<td>Property Default Values</td>
<td>8-18</td>
</tr>
<tr>
<td>Assign Property Values from Constructor</td>
<td>8-19</td>
</tr>
<tr>
<td>Initialize Properties to Unique Values</td>
<td>8-19</td>
</tr>
<tr>
<td>Property Attributes</td>
<td>8-19</td>
</tr>
<tr>
<td>Property Access Methods</td>
<td>8-20</td>
</tr>
<tr>
<td>Reference Object Properties Using Variables</td>
<td>8-21</td>
</tr>
<tr>
<td>Mutable and Immutable Properties</td>
<td>8-22</td>
</tr>
<tr>
<td>Set Access to Property Values</td>
<td>8-22</td>
</tr>
<tr>
<td>Define Immutable Property</td>
<td>8-22</td>
</tr>
</tbody>
</table>
**Validate Property Values** .................................................. 8-24
  Property Validation in Class Definitions .......................... 8-24
  Sample Class Using Property Validation ......................... 8-25
  Order of Validation ....................................................... 8-27
  Abstract Property Validation ........................................... 8-28
  Objects Not Updated When Changing Validation .................. 8-28
  Validation During Load Operation .................................... 8-28

**Property Class and Size Validation** ................................. 8-31
  Property Class and Size .................................................. 8-31
  Property Size Validation ................................................ 8-31
  Property Class Validation ............................................... 8-33
  Default Values Per Size and Class ................................... 8-37

**Property Validation Functions** ........................................ 8-39
  Validate Property Using Functions ................................. 8-39
  MATLAB Validation Functions .......................................... 8-42
  Define Validation Functions ............................................ 8-44
  Add Support for Validation Functions ............................... 8-45

**Metadata Interface to Property Validation** ......................... 8-47

**Property Access Methods** .............................................. 8-49
  Properties Provide Access to Class Data ........................... 8-49
  Property Setter and Getter Methods ................................... 8-50
  Set and Get Method Execution and Property Events ................. 8-52
  Access Methods and Properties Containing Arrays ................ 8-53
  Access Methods and Arrays of Objects .............................. 8-53
  Modify Property Values with Access Methods ....................... 8-53

**Property Set Methods** .................................................. 8-55
  Overview of Property Access Methods ............................... 8-55
  Property Set Method Syntax ............................................ 8-55
  Validate Property Set Value ............................................ 8-56
  When Set Method Is Called ............................................. 8-56

**Property Get Methods** .................................................. 8-59
  Overview of Property Access Methods ............................... 8-59
  Property Get Method Syntax ............................................ 8-59
  Calculate Value for Dependent Property ............................ 8-59
  Errors Not Returned from Get Method ................................. 8-60
  Get Method Behavior .................................................... 8-60
Table of Method Attributes ........................................ 9-5

**Ordinary Methods** ........................................ 9-8
Ordinary Methods Operate on Objects ......................... 9-8
Methods Inside classdef Block .................................. 9-8
Method Files .................................................. 9-9

**Methods in Separate Files** .................................... 9-11
Class Folders .................................................. 9-11
Define Method in Function File .................................. 9-12
Specify Method Attributes in classdef File ................. 9-12
Methods You Must Define in the classdef File ............... 9-13

**Method Invocation** .......................................... 9-15
Determining Which Method Is Invoked ......................... 9-15
Referencing Names with Expressions—Dynamic Reference .... 9-17
Controlling Access to Methods ................................. 9-18
Invoking Superclass Methods in Subclass Methods .......... 9-19
Invoking Built-In Functions .................................. 9-20

**Class Constructor Methods** ................................ 9-21
Purpose of Class Constructor Methods ......................... 9-21
Basic Structure of Constructor Methods ....................... 9-21
Guidelines for Constructors ................................... 9-23
Default Constructor ........................................... 9-23
When to Define Constructors ................................... 9-24
Related Information ............................................ 9-24
Initializing Objects in Constructor ......................... 9-24
No Input Argument Constructor Requirement ................. 9-25
Subclass Constructors ......................................... 9-26
Implicit Call to Inherited Constructor ....................... 9-28
Errors During Class Construction .............................. 9-29
Output Object Suppressed ..................................... 9-30

**Static Methods** ............................................. 9-31
What Are Static Methods ..................................... 9-31
Why Define Static Methods ................................... 9-31
Defining Static Methods ...................................... 9-31
Calling Static Methods ....................................... 9-32
Inheriting Static Methods .................................... 9-32

**Overload Functions in Class Definitions** ................... 9-33
Why Overload Functions ....................................... 9-33
Accessing Dynamic Properties in Arrays ..................... 10-14

Implicit Class Conversion ........................................ 10-16
  Class Conversion Mechanism .................................... 10-16
  Concatenation .................................................... 10-16
  Subscripted Assignment ......................................... 10-17

Concatenating Objects of Different Classes ..................... 10-18
  Basic Knowledge ................................................. 10-18
  MATLAB Concatenation Rules .................................... 10-18
  Concatenating Objects .......................................... 10-19
  Calling the Dominant-Class Constructor ....................... 10-19
  Converter Methods .............................................. 10-21

Designing Heterogeneous Class Hierarchies ..................... 10-24
  Creating Classes That Support Heterogeneous Arrays .......... 10-24
  MATLAB Arrays .................................................. 10-24
  Heterogeneous Hierarchies ..................................... 10-25
  Heterogeneous Arrays .......................................... 10-25
  Heterogeneous Array Concepts .................................. 10-26
  Nature of Heterogeneous Arrays ................................ 10-26
  Unsupported Hierarchies ....................................... 10-29
  Default Object .................................................. 10-31
  Conversion During Assignment and Concatenation ............. 10-32
  Empty Arrays of Heterogeneous Abstract Classes ............. 10-32

Heterogeneous Array Constructors ............................... 10-34
  Building Arrays in Superclass Constructors .................. 10-34
  When Errors Can Occur ......................................... 10-34
  Initialize Array in Superclass Constructor ................... 10-35
  Sample Implementation ......................................... 10-36
  Potential Error ................................................. 10-38

Events — Sending and Responding to Messages

Overview Events and Listeners ................................... 11-2
  Why Use Events and Listeners .................................. 11-2
  Events and Listeners Basics ................................... 11-2
  Events and Listeners Syntax ................................... 11-3
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Listener</td>
<td>11-4</td>
</tr>
<tr>
<td><strong>Define Custom Event Data</strong></td>
<td>11-6</td>
</tr>
<tr>
<td>Class Event Data Requirements</td>
<td>11-6</td>
</tr>
<tr>
<td>Define and Trigger Event</td>
<td>11-6</td>
</tr>
<tr>
<td>Define Event Data</td>
<td>11-7</td>
</tr>
<tr>
<td>Create Listener for Overflow Event</td>
<td>11-8</td>
</tr>
<tr>
<td>Observe Changes to Property Values</td>
<td>11-9</td>
</tr>
<tr>
<td><strong>Implement Property Set Listener</strong></td>
<td>11-11</td>
</tr>
<tr>
<td>PushButton Class Design</td>
<td>11-11</td>
</tr>
<tr>
<td><strong>Event and Listener Concepts</strong></td>
<td>11-14</td>
</tr>
<tr>
<td>The Event Model</td>
<td>11-14</td>
</tr>
<tr>
<td>Limitations</td>
<td>11-15</td>
</tr>
<tr>
<td>Default Event Data</td>
<td>11-16</td>
</tr>
<tr>
<td>Events Only in Handle Classes</td>
<td>11-16</td>
</tr>
<tr>
<td>Property-Set and Query Events</td>
<td>11-17</td>
</tr>
<tr>
<td>Listeners</td>
<td>11-17</td>
</tr>
<tr>
<td><strong>Event Attributes</strong></td>
<td>11-19</td>
</tr>
<tr>
<td>Specify Event Attributes</td>
<td>11-19</td>
</tr>
<tr>
<td><strong>Events and Listeners Syntax</strong></td>
<td>11-22</td>
</tr>
<tr>
<td>Components to Implement</td>
<td>11-22</td>
</tr>
<tr>
<td>Name Events</td>
<td>11-22</td>
</tr>
<tr>
<td>Trigger Events</td>
<td>11-23</td>
</tr>
<tr>
<td>Listen to Events</td>
<td>11-23</td>
</tr>
<tr>
<td>Define Event-Specific Data</td>
<td>11-26</td>
</tr>
<tr>
<td><strong>Listener Lifecycle</strong></td>
<td>11-28</td>
</tr>
<tr>
<td>Control Listener Lifecycle</td>
<td>11-28</td>
</tr>
<tr>
<td>Temporarily Deactivate Listeners</td>
<td>11-28</td>
</tr>
<tr>
<td>Permanently Delete Listeners</td>
<td>11-28</td>
</tr>
<tr>
<td><strong>Listener Callback Syntax</strong></td>
<td>11-30</td>
</tr>
<tr>
<td>Specifying Listener Callbacks</td>
<td>11-30</td>
</tr>
<tr>
<td>Input Arguments for Callback Function</td>
<td>11-30</td>
</tr>
<tr>
<td>Additional Arguments for Callback Function</td>
<td>11-31</td>
</tr>
<tr>
<td><strong>Callback Execution</strong></td>
<td>11-34</td>
</tr>
<tr>
<td>When Callbacks Execute</td>
<td>11-34</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Listener Order of Execution</td>
<td>11-34</td>
</tr>
<tr>
<td>Callbacks That Call notify</td>
<td>11-34</td>
</tr>
<tr>
<td>Manage Callback Errors</td>
<td>11-35</td>
</tr>
<tr>
<td>Invoke Functions from Function Handles</td>
<td>11-35</td>
</tr>
<tr>
<td><strong>Determine If Event Has Listeners</strong></td>
<td>11-37</td>
</tr>
<tr>
<td>Do Listeners Exist for This Event?</td>
<td>11-37</td>
</tr>
<tr>
<td>Why Test for Listeners</td>
<td>11-37</td>
</tr>
<tr>
<td>Coding Patterns</td>
<td>11-37</td>
</tr>
<tr>
<td>Listeners in Heterogeneous Arrays</td>
<td>11-38</td>
</tr>
<tr>
<td><strong>Listen for Changes to Property Values</strong></td>
<td>11-40</td>
</tr>
<tr>
<td>Create Property Listeners</td>
<td>11-40</td>
</tr>
<tr>
<td>Property Event and Listener Classes</td>
<td>11-42</td>
</tr>
<tr>
<td><strong>Assignment When Property Value Unchanged</strong></td>
<td>11-44</td>
</tr>
<tr>
<td>AbortSet When Value Does Not Change</td>
<td>11-44</td>
</tr>
<tr>
<td>How MATLAB Compares Values</td>
<td>11-44</td>
</tr>
<tr>
<td>When to Use AbortSet</td>
<td>11-45</td>
</tr>
<tr>
<td>Implement AbortSet</td>
<td>11-45</td>
</tr>
<tr>
<td><strong>Techniques for Using Events and Listeners</strong></td>
<td>11-48</td>
</tr>
<tr>
<td>Example Overview</td>
<td>11-48</td>
</tr>
<tr>
<td>Techniques Demonstrated in This Example</td>
<td>11-49</td>
</tr>
<tr>
<td>Summary of fcneval Class</td>
<td>11-49</td>
</tr>
<tr>
<td>Summary of fcnview Class</td>
<td>11-50</td>
</tr>
<tr>
<td>Methods Inherited from Handle Class</td>
<td>11-52</td>
</tr>
<tr>
<td>Using the fcneval and fcnview Classes</td>
<td>11-52</td>
</tr>
<tr>
<td>Implement UpdateGraph Event and Listener</td>
<td>11-54</td>
</tr>
<tr>
<td>The PostSet Event Listener</td>
<td>11-58</td>
</tr>
<tr>
<td>Enable and Disable Listeners</td>
<td>11-61</td>
</tr>
<tr>
<td>@fcneval/fcneval.m Class Code</td>
<td>11-62</td>
</tr>
<tr>
<td>@fcnview/fcnview.m Class Code</td>
<td>11-63</td>
</tr>
<tr>
<td><strong>How to Build on Other Classes</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>Hierarchies of Classes — Concepts</strong></td>
<td>12-2</td>
</tr>
<tr>
<td>Classification</td>
<td>12-2</td>
</tr>
<tr>
<td>Develop the Abstraction</td>
<td>12-3</td>
</tr>
</tbody>
</table>

12
### Design of Class Hierarchies
- Super and Subclass Behavior .................................. 12-4
- Implementation and Interface Inheritance .................. 12-5

### Subclass Syntax
- Subclass Definition Syntax .................................. 12-7
- Subclass double .............................................. 12-7

### Design Subclass Constructors
- Call Superclass Constructor Explicitly .................. 12-9
- Call Superclass Constructor from Subclass ................. 12-9
- Subclass Constructor Implementation ...................... 12-11
- Call Only Direct Superclass from Constructor .......... 12-12

### Control Sequence of Constructor Calls
- Modify Inherited Methods .......................... 12-16
  - When to Modify Superclass Methods .................. 12-16
  - Extend Superclass Methods .......................... 12-16
  - Redefine Superclass Methods in Subclass .......... 12-17
  - Override Superclass Methods .......................... 12-18

### Modify Inherited Properties
- Superclass Property Modification ...................... 12-20
- Private Local Property Takes Precedence in Method .... 12-20

### Subclassing Multiple Classes
- Specify Multiple Superclasses .......................... 12-22
- Class Member Compatibility .......................... 12-22
- Multiple Inheritance .................................. 12-23

### Specify Allowed Subclasses
- Basic Knowledge .................................. 12-25
- Why Control Allowed Subclasses .................. 12-25
- Specify Allowed Subclasses .................. 12-25
- Define Sealed Hierarchy of Classes .................. 12-27

### Class Members Access
- Basic Knowledge .................................. 12-28
- Applications for Access Control Lists .................. 12-29
- Specify Access to Class Members .................. 12-29
- Properties with Access Lists .......................... 12-31
- Methods with Access Lists .......................... 12-31
Abstract Methods with Access Lists .............................. 12-35

**Property Access List** ........................................... 12-36

**Method Access List** ........................................... 12-37

**Event Access List** ............................................ 12-38

**Handle Compatible Classes** ................................. 12-39
  - Basic Knowledge ........................................... 12-39
  - When to Use Handle Compatible Classes ............. 12-39
  - Handle Compatibility Rules .............................. 12-40
  - Identify Handle Objects ................................ 12-40

**How to Define Handle-Compatible Classes** .............. 12-41
  - What Is Handle Compatibility? ......................... 12-41
  - Subclassing Handle-Compatible Classes .......... 12-43

**Methods for Handle Compatible Classes** ................. 12-46
  - Methods for Handle and Value Objects ............. 12-46
  - Modify Value Objects in Methods .................. 12-46

**Handle-Compatible Classes and Heterogeneous Arrays** .... 12-48
  - Heterogeneous Arrays .................................. 12-48
  - Methods Must Be Sealed ................................ 12-48
  - Template Technique ...................................... 12-48

**Subclasses of MATLAB Built-In Types** .................... 12-50
  - MATLAB Built-In Types ................................ 12-50
  - Built-In Types You Can Subclass ................... 12-50
  - Why Subclass Built-In Types ......................... 12-51
  - Which Functions Work with Subclasses of Built-In Types ... 12-51
  - Behavior of Built-In Functions with Subclass Objects ... 12-51
  - Built-In Subclasses That Define Properties .......... 12-52

**Behavior of Inherited Built-In Methods** ................. 12-54
  - Subclass double ........................................ 12-54
  - Built-In Data Value Methods .......................... 12-55
  - Built-In Data Organization Methods .................. 12-56
  - Built-In Indexing Methods ................................ 12-57
  - Built-In Concatenation Methods ....................... 12-57
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclasses of Built-In Types Without Properties</td>
<td>12-60</td>
</tr>
<tr>
<td>Specialized Numeric Types</td>
<td>12-60</td>
</tr>
<tr>
<td>A Class to Manage uint8 Data</td>
<td>12-60</td>
</tr>
<tr>
<td>Using the DocUint8 Class</td>
<td>12-61</td>
</tr>
<tr>
<td>Subclasses of Built-In Types with Properties</td>
<td>12-68</td>
</tr>
<tr>
<td>Specialized Numeric Types with Additional Data Storage</td>
<td>12-68</td>
</tr>
<tr>
<td>Subclasses with Properties</td>
<td>12-68</td>
</tr>
<tr>
<td>Property Added</td>
<td>12-69</td>
</tr>
<tr>
<td>Methods Implemented</td>
<td>12-69</td>
</tr>
<tr>
<td>Class Definition Code</td>
<td>12-69</td>
</tr>
<tr>
<td>Using ExtendDouble</td>
<td>12-72</td>
</tr>
<tr>
<td>Concatenation of ExtendDouble Objects</td>
<td>12-75</td>
</tr>
<tr>
<td>Use of size and numel with Classes</td>
<td>12-77</td>
</tr>
<tr>
<td>size and numel</td>
<td>12-77</td>
</tr>
<tr>
<td>Built-In Class Behavior</td>
<td>12-77</td>
</tr>
<tr>
<td>Subclasses Inherit Behavior</td>
<td>12-78</td>
</tr>
<tr>
<td>Classes Not Derived from Built-In Classes</td>
<td>12-79</td>
</tr>
<tr>
<td>Change the Behavior of size or numel</td>
<td>12-81</td>
</tr>
<tr>
<td>Overload numArgumentsFromSubscript Instead of numel</td>
<td>12-82</td>
</tr>
<tr>
<td>Class to Represent Hardware</td>
<td>12-83</td>
</tr>
<tr>
<td>Objective</td>
<td>12-83</td>
</tr>
<tr>
<td>Why Derive from int32</td>
<td>12-83</td>
</tr>
<tr>
<td>Implementation</td>
<td>12-83</td>
</tr>
<tr>
<td>Construct MuxCard Object</td>
<td>12-84</td>
</tr>
<tr>
<td>Call Methods of int32</td>
<td>12-85</td>
</tr>
<tr>
<td>Determine Array Class</td>
<td>12-87</td>
</tr>
<tr>
<td>Query the Class Name</td>
<td>12-87</td>
</tr>
<tr>
<td>Test for Array Class</td>
<td>12-87</td>
</tr>
<tr>
<td>Test for Specific Types</td>
<td>12-88</td>
</tr>
<tr>
<td>Test for Most Derived Class</td>
<td>12-89</td>
</tr>
<tr>
<td>Abstract Classes</td>
<td>12-91</td>
</tr>
<tr>
<td>Abstract Classes</td>
<td>12-91</td>
</tr>
<tr>
<td>Declare Classes as Abstract</td>
<td>12-92</td>
</tr>
<tr>
<td>Determine If a Class Is Abstract</td>
<td>12-93</td>
</tr>
<tr>
<td>Find Inherited Abstract Properties and Methods</td>
<td>12-94</td>
</tr>
<tr>
<td>Define an Interface Superclass</td>
<td>12-96</td>
</tr>
<tr>
<td>Interfaces</td>
<td>12-96</td>
</tr>
</tbody>
</table>
Saving and Loading Objects

Save and Load Process for Objects ........................................ 13-2
  Save and Load Objects .................................................. 13-2
  What Information Is Saved? ........................................... 13-2
  How Is the Property Data Loaded? ................................. 13-3
  Errors During Load ...................................................... 13-3

Reduce MAT-File Size for Saved Objects .............................. 13-5
  Default Values ............................................................ 13-5
  Dependent Properties .................................................. 13-5
  Transient Properties ................................................... 13-5
  Avoid Saving Unwanted Variables .................................. 13-6

Save Object Data to Recreate Graphics Objects ..................... 13-7
  What to Save ............................................................... 13-7
  Regenerate When Loading ............................................... 13-7
  Change to a Stairstep Chart ......................................... 13-8

Improve Version Compatibility with Default Values ................ 13-10
  Version Compatibility .................................................. 13-10
  Using a Default Property Value ................................... 13-10

Avoid Property Initialization Order Dependency ........................ 13-12
  Control Property Loading .............................................. 13-12
  Dependent Property with Private Storage ......................... 13-13
  Property Value Computed from Other Properties ............... 13-14

Modify the Save and Load Process ........................................ 13-16
  When to Modify the Save and Load Process ...................... 13-16
  How to Modify the Save and Load Process ....................... 13-16
  Implementing saveobj and loadobj Methods ..................... 13-17
  Additional Considerations ........................................... 13-17

Basic saveobj and loadobj Pattern ..................................... 13-19
  Using saveobj and loadobj .......................................... 13-19
  Handle Load Problems ................................................ 13-20
Maintain Class Compatibility ........................................ 13-23
Rename Property ......................................................... 13-23
Update Property When Loading ...................................... 13-25
Maintaining Compatible Versions of a Class ..................... 13-26
Version 2 of the PhoneBookEntry Class ......................... 13-27

Initialize Objects When Loading .................................. 13-30
Calling Constructor When Loading Objects ....................... 13-30
Initializing Objects in the loadobj Method ...................... 13-30

Save and Load Objects from Class Hierarchies ................. 13-33
Saving and Loading Subclass Objects .............................. 13-33
Reconstruct the Subclass Object from a Saved struct ........ 13-33

Restore Listeners ...................................................... 13-36
Create Listener with loadobj ....................................... 13-36
Use Transient Property to Load Listener ......................... 13-36
Using the BankAccount and AccountManager Classes .......... 13-38

Save and Load Dynamic Properties ............................... 13-39
Saving Dynamic Properties ......................................... 13-39
When You Need saveobj and loadobj Methods ................ 13-39
Implementing saveobj and loadobj Methods .................... 13-39

Enumerations

Named Values ............................................................ 14-2
Kinds of Predefined Names .......................................... 14-2
Techniques for Defining Enumerations ......................... 14-3

Define Enumeration Classes ....................................... 14-5
Enumeration Class ..................................................... 14-5
Construct an Enumeration Member ................................ 14-5
Convert to Superclass Value ........................................ 14-6
Define Methods in Enumeration Classes ......................... 14-7
Define Properties in Enumeration Classes ...................... 14-7
Enumeration Class Constructor Calling Sequence ............... 14-8
Values of the Indexing Structure ............................................ 17-23
Typical Patterns for Indexing Methods ................................. 17-23

Indexed Reference .............................................................. 17-28
How Indexed Reference Works ........................................... 17-28
Compound Indexed References .............................................. 17-29

Indexed Assignment ............................................................ 17-31
How Indexed Assignment Works ......................................... 17-31
Indexed Assignment to Objects ............................................ 17-33
Compound Indexed Assignments ........................................... 17-33

end as Object Index ............................................................ 17-35
Define end Indexing for an Object ...................................... 17-35
The end Method ................................................................. 17-36

Objects in Index Expressions ............................................... 17-37
Objects Indexes ................................................................. 17-37
Ways to Implement Objects as Indices ................................. 17-37
subsindex Implementation .................................................... 17-37

Class with Modified Indexing ............................................... 17-39
How to Modify Class Indexing ............................................ 17-39
Class Description .............................................................. 17-39
Specialize Subscripted Reference — subsref ........................ 17-40
Specialize Subscripted Assignment — subsasgn .................... 17-42
Implement Addition for Object Data — double and plus ........ 17-43
MyDataClass.m ................................................................. 17-44

Operator Overloading .......................................................... 17-47
Why Overload Operators ..................................................... 17-47
How to Define Operators ..................................................... 17-47
Sample Implementation — Addable Objects ......................... 17-48
MATLAB Operators and Associated Functions ..................... 17-50

18

Customizing Object Display

Custom Display Interface ..................................................... 18-2
Default Object Display ....................................................... 18-2
Using Object-Oriented Design in MATLAB

- “Why Use Object-Oriented Design” on page 1-2
- “Handle Object Behavior” on page 1-12
Why Use Object-Oriented Design

In this section...

| “Approaches to Writing MATLAB Programs” on page 1-2 |
| “When Should You Create Object-Oriented Programs” on page 1-6 |

Approaches to Writing MATLAB Programs

Creating software applications typically involves designing the application data and implementing operations performed on that data. Procedural programs pass data to functions, which perform the necessary operations on the data. Object-oriented software encapsulates data and operations in objects that interact with each other via the object's interface.

The MATLAB language enables you to create programs using both procedural and object-oriented techniques and to use objects and ordinary functions together in your programs.

Procedural Program Design

In procedural programming, your design focuses on the steps that must execute to achieve a desired state. Typically, you represent data as individual variables or fields of a structure. You implement operations as functions that take the variables as arguments. Programs usually call a sequence of functions, each one of which is passed data, and then returns modified data. Each function performs an operation or many operations on the data.

Object-Oriented Program Design

The object-oriented program design involves:

- Identifying the components of the system or application that you want to build
- Analyzing and identifying patterns to determine what components are used repeatedly or share characteristics
- Classifying components based on similarities and differences

After performing this analysis, you define classes that describe the objects your application uses.
**Classes and Objects**

A class describes a set of objects with common characteristics. Objects are specific instances of a class. The values contained in an object's properties are what make an object different from other objects of the same class. The functions defined by the class (called methods) are what implement object behaviors that are common to all objects of a class.

**Using Objects in MATLAB Programs**

The MATLAB language defines objects that are designed for use in any MATLAB code. For example, consider the `try/catch` programming construct.

If the code executed in the `try` block generates an error, program control passes to the code in the `catch` block. This behavior enables your program to provide special error handling that is more appropriate to your particular application. However, you must have enough information about the error to take the appropriate action.

MATLAB provides detailed information about the error by passing an `MException` object to functions executing the `try/catch` blocks.

The following `try/catch` blocks display the error message stored in an `MException` object when you call a function (`surf` in this case) without the necessary arguments:

```plaintext
try
    surf
catch ME
    disp(ME.message)
end
```

Not enough input arguments.

In this code, `ME` is an object of the `MException` class, which the `catch` statement creates to capture information about the error. Displaying the value of the object `message` property returns the error message (Not enough input arguments). Your program can access other properties to get information about the error.

List all the public properties of an object with the `properties` function:

```plaintext
properties(ME);
```

Properties for class `MException`:
identifier
message
cause
stack

**Objects Organize Data**

Properties store the information returned in MException objects. Reference a property using dot notation, as in ME.message. This reference returns the value of the property. For example, assign the property value to a variable.

```matlab
msg = ME.message;
```

The *stack* property contains a MATLAB struct:

```matlab
s = ME.stack
s =
    struct with fields:
        file: D:\myMATLAB\matlab\toolbox\matlab\graph3d\surf.m
        name: 'surf'
        line: 49
```

You can treat ME.stack as a structure and reference its fields without assigning the value:

```matlab
f = ME.stack.file;
disp(f)
D:\myMATLAB\matlab\toolbox\matlab\graph3d\surf.m
```

The file field of the struct contained in the stack property is a character array:

```matlab
c = class(ME.stack.file);
disp(c)
char
```

You could, for example, use a property reference in MATLAB expressions:

```matlab
if ME.stack.name == 'surf'
    disp('Error in surf')
end
```
**Objects Manage Their Own Data**

You could write a function that generates a report from the data returned by MException object properties. This function could become complicated because it would have to be able to handle all possible errors. Perhaps you would use different functions for different try/catch blocks in your program. If the data returned by the error object must change, you would have to update the functions to use the new data.

Objects define their own operations as part of their interface. The MException object can generate its own report. The methods that implement an object’s operations are part of the object definition (that is, specified by the class that defines the object). The object definition can be modified many times, but the interface your program use does not change. Objects isolate your code from the object's code.

To see what methods exist for MException objects, use the methods function:

```matlab
methods(ME)
```

Methods for class MException:

```
addCause       getReport      ne             throw
eq             isequal        rethrow        throwAsCaller
```

Static methods:

```
last
```

You can use these methods like any other MATLAB statement when there is an MException object in the workspace. For example:

```matlab
rpt = ME.getReport;
disp(rpt)
```

Error using surf (line 49)
Not enough input arguments.

Objects often have methods that overload (redefined for the particular class of the object) MATLAB functions. Overloading enables you to use objects just like other values. For example, MException objects have an isequal method. This method enables you to compare these objects in the same way you would compare variables containing numeric values. If ME1 and ME2 are MException objects, you can compare them with this statement:

```matlab
isequal(ME1,ME2)
```
However, what really happens in this case is MATLAB calls the MException isEqual method because you passed MException objects to isEqual.

Similarly, the eq method enables you to use the == operator with MException objects:

\[ ME == ME2 \]

Objects should support only those methods that make sense. For example, it would probably not make sense to multiply MException objects so the MException class does not implement methods to do so.

**When Should You Create Object-Oriented Programs**

You can implement simple programming tasks as simple functions. However, as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage.

As functions become too large, you can break them into smaller functions and pass data from one to function to another. However, as the number of functions becomes large, designing, and managing the data passed to functions becomes difficult and error prone. At this point, consider moving your MATLAB programming tasks to object-oriented designs.

**Understand a Problem in Terms of Its Objects**

Thinking in terms of objects is simpler and more natural for some problems. Think of the nouns in your problem statement as the objects to define and the verbs as the operations to perform.

Consider the design of classes to represent money lending institutions (banks, mortgage companies, individual money lenders, and so on). It is difficult to represent the various types of lenders as procedures. However, you can represent each one as an object that performs certain actions and contains certain data. The process of designing the objects involves identifying the characteristics of a lender that are important to your application.

**Identify Commonalities**

What do all money lenders have in common? All MoneyLender objects can have a loan method and an InterestRate property, for example.

**Identify Differences**

How does each money lender differ? One can provide loans to businesses while another provides loans only to individuals. Therefore, the loan operation might need to be
different for different types of lending institutions. Subclasses of a base MoneyLender class can specialize the subclass versions of the loan method. Each lender can have a different value for its InterestRate property.

Factor out commonalities into a superclass and implement what is specific to each type of lender in the subclass.

**Add Only What Is Necessary**

These institutions might engage in activities that are not of interest to your application. During the design phase, determine what operations and data an object must contain based on your problem definition.

**Objects Manage Internal State**

Objects provide several useful features not available from structures and cell arrays. For example, objects can:

- Constrain the data values assigned to any given property
- Calculate the value of a property only when it is queried
- Broadcast notices when any property value is queried or changed
- Restrict access to properties and methods

**Reducing Redundancy**

As the complexity of your program increases, the benefits of an object-oriented design become more apparent. For example, suppose that you implement the following procedure as part of your application:

1. Check inputs
2. Perform computation on the first input argument
3. Transform the result of step 2 based on the second input argument
4. Check validity of outputs and return values

You can implement this procedure as an ordinary function. But suppose that you use this procedure again somewhere in your application, except that step 2 must perform a different computation. You could copy and paste the first implementation, and then rewrite step 2. Or you could create a function that accepted an option indicating which computation to make, and so on. However, these options lead to more complicated code.

An object-oriented design can factor out the common code into what is called a base class. The base class would define the algorithm used and implement whatever is common
to all cases that use this code. Step 2 could be defined syntactically, but not implemented, leaving the specialized implementation to the classes that you then derive from this base class.

Step 1
function checkInputs()
    % actual implementation
end

Step 2
function results = computeOnFirstArg()
    % specify syntax only
end

Step 3
function transformResults()
    % actual implementation
end

Step 4
function out = checkOutputs()
    % actual implementation
end

The code in the base class is not copied or modified. Classes you derive from the base class inherit this code. Inheritance reduces the amount of code to be tested, and isolates your program from changes to the basic procedure.

**Defining Consistent Interfaces**

The use of a class as the basis for similar, but more specialized classes is a useful technique in object-oriented programming. This class defines a common interface. Incorporating this kind of class into your program design enables you to:

- Identify the requirements of a particular objective
- Encode requirements into your program as an interface class

**Reducing Complexity**

Objects reduce complexity by reducing what you must know to use a component or system:

- Objects provide an interface that hides implementation details.
• Objects enforce rules that control how objects interact.

To illustrate these advantages, consider the implementation of a data structure called a doubly linked list. See “Class to Implement Linked Lists” on page 3-31 for the actual implementation.

Here is a diagram of a three-element list:

To add a node to the list, disconnect the existing nodes in the list, insert the new node, and reconnect the nodes appropriately. Here are the basic steps:

First disconnect the nodes:

1. Unlink n2.Prev from n1
2. Unlink n1.Next from n2

Now create the new node, connect it, and renumber the original nodes:

1. Link new.Prev to n1
2. Link new.Next to n3 (was n2)
3. Link n1.Next to new (will be n2)
4. Link n3.Prev to new (will be n2)
The details of how methods perform these steps are encapsulated in the class design. Each node object contains the functionality to insert itself into or remove itself from the list.

For example, in this class, every node object has an `insertAfter` method. To add a node to a list, create the node object and then call its `insertAfter` method:

```matlab
nnew = NodeConstructor;
nnew.insertAfter(n1)
```

Because the node class defines the code that implements these operations, this code is:

- Implemented in an optimal way by the class author
- Always up to date with the current version of the class
- Properly tested
- Can automatically update old-versions of the objects when they are loaded from MAT-files.

The object methods enforce the rules for how the nodes interact. This design removes the responsibility for enforcing rules from the applications that use the objects. It also means that the application is less likely to generate errors in its own implementation of the process.

**Fostering Modularity**

As you decompose a system into objects (car -> engine -> fuel system -> oxygen sensor), you form modules around natural boundaries. Classes provide three levels of control over code modularity:

- **Public** — Any code can access this particular property or call this method.
- **Protected** — Only this object's methods and methods of the object's derived from this object's class can access this property or call this method.
- **Private** — Only the object's own methods can access this property or call this method.

**Overloaded Functions and Operators**

When you define a class, you can overload existing MATLAB functions to work with your new object. For example, the MATLAB serial port class overloads the `fread` function to read data from the device connected to the port represented by this object. You can define various operations, such as equality (`eq`) or addition (`plus`), for a class you have defined to represent your data.
See Also

More About
- “Role of Classes in MATLAB” on page 3-2
Handle Object Behavior

**In this section...**

- “What Is a Handle?” on page 1-12
- “Copies of Handles” on page 1-12
- “Handle Objects Modified in Functions” on page 1-13
- “Determine If an Object Is a Handle” on page 1-15
- “Deleted Handle Objects” on page 1-15

More than one variable can refer to the same handle object. Therefore, users interact with instances of handle classes differently than instances of value classes. Understanding how handle objects behave can help you determine whether to implement a handle or a value class. This topic illustrates some of those interactions.

For more information on handle classes, see “Handle Classes”.

**What Is a Handle?**

Certain kinds of MATLAB objects are handles. When a variable holds a handle, it actually holds a reference to the object.

Handle objects enable more than one variable to refer to the same object. Handle-object behavior affects what happens when you copy handle objects and when you pass them to functions.

**Copies of Handles**

All copies of a handle object variable refer to the same underlying object. This reference behavior means that if h identifies a handle object, then,

\[ h2 = h; \]

Creates another variable, h2, that refers to the same object as h.

For example, the MATLAB `audioplayer` function creates a handle object that contains the audio source data to reproduce a specific sound segment. The variable returned by the `audioplayer` function identifies the audio data and enables you to access object functions to play the audio.
MATLAB software includes audio data that you can load and use to create an audiostream object. This sample load audio data, creates the audio player, and plays the audio:

```matlab
load gong Fs y
gongSound = audiostream(y,Fs);
play(gongSound)
```

Suppose that you copy the gongSound object handle to another variable (gongSound2):

```matlab
gongSound2 = gongSound;
```

The variables gongSound and gongSound2 are copies of the same handle and, therefore, refer to the same audio source. Access the audiostream information using either variable.

For example, set the sample rate for the gong audio source by assigning a new value to the SampleRate property. First get the current sample rate and then set a new sample rate:

```matlab
sr = gongSound.SampleRate;
disp(sr)
8192
```

```matlab
gongSound.SampleRate = sr*2;
```

You can use gongSound2 to access the same audio source:

```matlab
disp(gongSound2.SampleRate)
16384
```

Play the gong sound with the new sample rate:

```matlab
play(gongSound2)
```

**Handle Objects Modified in Functions**

When you pass an argument to a function, the function copies the variable from the workspace in which you call the function into the parameter variable in the function’s workspace.
Passing a nonhandle variable to a function does not affect the original variable that is in the caller’s workspace. For example, `myFunc` modifies a local variable called `var`, but when the function ends, the local variable `var` no longer exists:

```matlab
function myFunc(var)
    var = var + 1;
end
```

Define a variable and pass it to `myfunc`:

```matlab
x = 12;
myFunc(x)
```

The value of `x` has not changed after executing `myFunc(x)`:

```matlab
disp(x)
```

```
12
```

The `myFunc` function can return the modified value, which you could assign to the same variable name (`x`) or another variable.

```matlab
function out = myFunc(var)
    out = var + 1;
end
```

Modify a value in `myfunc`:

```matlab
x = 12;
x = myFunc(x);
disp(x)
```

```
13
```

When the argument is a handle variable, the function copies only the handle, not the object identified by that handle. Both handles (original and local copy) refer to the same object.

When the function modifies the data referred to by the object handle, those changes are accessible from the handle variable in the calling workspace without the need to return the modified object.

For example, the `modifySampleRate` function changes the `audioplayer` sample rate:
function modifySampleRate(audioObj,sr)
    audioObj.SampleRate = sr;
end

Create an audioplayer object and pass it to the modifySampleRate function:

load gong Fs y
gongSound = audioplayer(y,Fs);
disp(gongSound.SampleRate)

8192

modifySampleRate(gongSound,16384)
disp(gongSound.SampleRate)

16384

The modifySampleRate function does not need to return a modified gongSound object because audioplayer objects are handle objects.

**Determine If an Object Is a Handle**

Handle objects are members of the handle class. Therefore, you can always identify an object as a handle using the isa function. isa returns logical true (1) when testing for a handle variable:

load gong Fs y
gongSound = audioplayer(y,Fs);
isa(gongSound,'handle')

To determine if a variable is a valid handle object, use isa and isValid:

if isa(gongSound,'handle') && isValid(gongSound)
    ...
end

**Deleted Handle Objects**

When a handle object has been deleted, the handle variables that referenced the object can still exist. These variables become invalid because the object they referred to no longer exists. Calling delete on the object removes the object, but does not clear handle variables.

For example, create an audioplayer object:
load gong Fs y
gongSound = audioplayer(y,Fs);

The output argument, `gongSound`, is a handle variable. Calling `delete` deletes the object along with the audio source information it contains:

delete(gongSound)

However, the handle variable still exists:

disp(gongSound)

handle to deleted audioplayer

The `whos` command shows `gongSound` as an `audioplayer` object:

```
Name       Size          Bytes  Class          Attributes
---------- -------          ------ ----------          ----------
Fs         1x1               8 double
gongSound  1x1            0 audioplayer
y         42028x1       336224 double
```

**Note** The value for Bytes returned by the `whos` command does not include the data referenced by a handle because many variables can reference the same data.

The handle `gongSound` no longer refers to a valid object, as shown by the `isvalid` handle method:

```
isvalid(gongSound)
```

```
an =

    logical

    0
```

Calling `delete` on a deleted handle does nothing and does not cause an error. You can pass an array containing both valid and invalid handles to `delete`. MATLAB deletes the valid handles, but does not issue an error when encountering handles that are already invalid.

You cannot access properties with the invalid handle variable:
gongSound.SampleRate
Invalid or deleted object.

Functions and methods that access object properties cause an error:

play(gongSound)
Invalid or deleted object.

To remove the variable, gongSound, use clear:

clear  gongSound
whos

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fs</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>42028x1</td>
<td>336224</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

See Also

More About

• “Handle Class Destructor” on page 7-16
• “Comparison of Handle and Value Classes” on page 7-2
Basic Example
Create a Simple Class

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Design Class” on page 2-2</td>
</tr>
<tr>
<td>“Create Object” on page 2-3</td>
</tr>
<tr>
<td>“Access Properties” on page 2-3</td>
</tr>
<tr>
<td>“Call Methods” on page 2-4</td>
</tr>
<tr>
<td>“Add Constructor” on page 2-4</td>
</tr>
<tr>
<td>“Vectorize Methods” on page 2-5</td>
</tr>
<tr>
<td>“Overload Functions” on page 2-6</td>
</tr>
<tr>
<td>“BasicClass Code Listing” on page 2-7</td>
</tr>
</tbody>
</table>

Design Class

The basic purpose of a class is to define an object that encapsulates data and the operations performed on that data. For example, BasicClass defines a property and two methods that operate on the data in that property:

- **Value** — Property that contains the data stored in an object of the class
- **roundOff** — Method that rounds the value of the property to two decimal places
- **multiplyBy** — Method that multiplies the value of the property by the specified number

Here is the definition of BasicClass:

```matlab
classdef BasicClass
    properties
        Value
    end
    methods
        function r = roundOff(obj)
            r = round([obj.Value],2);
        end
        function r = multiplyBy(obj,n)
            r = [obj.Value] * n;
        end
    end
end
```
For a summary of class syntax, see \texttt{classdef}.

To use the class:

- Save the class definition in a \texttt{.m} file with the same name as the class.
- Create an object of the class.
- Access the properties to assign data.
- Call methods to perform operation on the data.

**Create Object**

Create an object of the class using the class name:

\begin{verbatim}
    a = BasicClass
    a =
        BasicClass with properties:
            Value: []
\end{verbatim}

Initially, the property value is empty.

**Access Properties**

Assign a value to the \texttt{Value} property using the object variable and a dot before the property name:

\begin{verbatim}
    a.Value = pi/3;
\end{verbatim}

To access a property value, use dot notation without the assignment:

\begin{verbatim}
    a.Value
\end{verbatim}

\begin{verbatim}
    ans =
        1.0472
\end{verbatim}

For information on class properties, see “Properties” on page 5-12.
Call Methods

Call the roundOff method on object a:

roundOff(a)
ans =
    1.0500

Pass the object as the first argument to a method that takes multiple arguments:
multiplyBy(a,3)
ans =
    3.1416

You can also call a method using dot notation:
a.multiplyBy(3)

It is not necessary to pass the object explicitly as an argument when using dot notation. The notation uses the object to the left of the method name.

For information on class methods, see “Methods and Functions” on page 5-15

Add Constructor

Classes can define a special method to create objects, called a constructor. Constructor methods enable you to pass arguments to the constructor, and to validate and assign property values. Here is a constructor for the BasicClass class:

```matlab
methods
    function obj = BasicClass(val)
        if nargin > 0
            if isnumeric(val)
                obj.Value = val;
            else
                error('Value must be numeric')
            end
        end
end
end
```
By adding this constructor to the class definition, you can create an object in one step:

```plaintext
a = BasicClass(pi/3)
```

```plaintext
a =

    BasicClass with properties:
        Value: 1.0472
```

This constructor also performs type checking on the input argument. For example:

```plaintext
a = BasicClass('A character array')
```

```
Error using BasicClass (line 11)
Value must be numeric
```

For information on constructors, see “Class Constructor Methods” on page 9-21

### Vectorize Methods

MATLAB enables you to vectorize operations. For example, you can add a number to a vector:

```plaintext
[1 2 3] + 2
```

```plaintext
ans =

   3   4   5
```

MATLAB adds the number 2 to each of the elements in the array [1 2 3]. To vectorize the arithmetic operator methods, enclose the `obj.Value` property reference in brackets, where `obj` is an object array.

```plaintext
[obj.Value] + 2
```

This syntax enables the method to work with arrays of object. For example, given objects `a1`, `a2`, and `a3`:

```plaintext
[a1.Value,a2.Value,a3.Value] + 2
```

By using vector notation, `a` can be an array:

```plaintext
a(1) = BasicClass(2.7984);
a(2) = BasicClass(sin(pi/3));
```
a(3) = BasicClass(7);
roundoff(a)

ans =

2.8000  0.8700  7.0000

**Overload Functions**

Classes can implement existing functionality, such as addition, by defining a method with the same name as the existing MATLAB function. For example, suppose that you want to add two `BasicClass` objects. It makes sense to add the values of the `ObjectValue` properties of each object.

Here is an overload of the MATLAB `plus` function. It defines addition for this class as adding the property values:

```matlab
method
    function r = plus(o1,o2)
        r = [o1.Value] + [o2.Value];
    end
end
```

By implementing a method called `plus`, you can use the “+” operator with objects of `BasicClass`.

```matlab
a = BasicClass(pi/3);
b = BasicClass(pi/4);
a + b
```

```matlab
ans =

1.8326
```

**Related Information**

For information on overloading functions, see “Overload Functions in Class Definitions” on page 9-33.

For information on overloading operators, see “Operator Overloading” on page 17-47.
BasicClass Code Listing

Here is the BasicClass definition after adding the features discussed in this topic:

```matlab
classdef BasicClass
    properties
        Value
    end
    methods
        function obj = BasicClass(val)
            if nargin == 1
                if isnumeric(val)
                    obj.Value = val;
                else
                    error('Value must be numeric')
                end
            end
        end
        function r = roundOff(obj)
            r = round([obj.Value],2);
        end
        function r = multiplyBy(obj,n)
            r = [obj.Value] * n;
        end
        function r = plus(o1,o2)
            r = o1.Value + o2.Value;
        end
    end
end
```

See Also

Related Examples

- “Class Syntax Guide”
MATLAB Classes Overview

- “Role of Classes in MATLAB” on page 3-2
- “Developing Classes — Typical Workflow” on page 3-8
- “Class to Represent Structured Data” on page 3-19
- “Class to Implement Linked Lists” on page 3-31
Role of Classes in MATLAB

In the MATLAB language, every value is assigned to a class. For example, creating a variable with an assignment statement constructs a variable of the appropriate class:

```matlab
a = 7;
b = 'some text';
s.Name = 'Nancy';
s.Age = 64;
whos
```

```
whos
Name      Size            Bytes  Class     Attributes
a         1x1                 8  double
b         1x9                18  char
s         1x1               370  struct
```

Basic commands like `whos` display the class of each value in the workspace. This information helps MATLAB users recognize that some values are characters and display as text while other values are double precision numbers, and so on. Some variables can contain different classes of values like structures.

Predefined Classes

MATLAB defines fundamental classes that comprise the basic types used by the language. These classes include numeric, logical, char, cell, struct, and function handle.

User-Defined Classes

You can create your own MATLAB classes. For example, you could define a class to represent polynomials. This class could define the operations typically associated with MATLAB classes, like addition, subtraction, indexing, displaying in the command window, and so on. These operations would need to perform the equivalent of polynomial addition, polynomial subtraction, and so on. For example, when you add two polynomial objects:
p1 + p2

the `plus` operation must be able to add polynomial objects because the polynomial class defines this operation.

When you define a class, you can overload special MATLAB functions (such as `plus.m` for the addition operator). MATLAB calls these methods when users apply those operations to objects of your class.

See “Class Design for Polynomials” on page 19-2 for an example that creates just such a class.

**MATLAB Classes — Key Terms**

MATLAB classes use the following words to describe different parts of a class definition and related concepts.

- **Class definition** — Description of what is common to every instance of a class.
- **Properties** — Data storage for class instances
- **Methods** — Special functions that implement operations that are usually performed only on instances of the class
- **Events** — Messages defined by classes and broadcast by class instances when some specific action occurs
- **Attributes** — Values that modify the behavior of properties, methods, events, and classes
- **Listeners** — Objects that respond to a specific event by executing a callback function when the event notice is broadcast
- **Objects** — Instances of classes, which contain actual data values stored in the objects' properties
- **Subclasses** — Classes that are derived from other classes and that inherit the methods, properties, and events from those classes (subclasses facilitate the reuse of code defined in the superclass from which they are derived).
- **Superclasses** — Classes that are used as a basis for the creation of more specifically defined classes (that is, subclasses).
- **Packages** — Folders that define a scope for class and function naming
Some Basic Relationships

This section discusses some of the basic concepts used by MATLAB classes.

**Classes**

A class is a definition that specifies certain characteristics that all instances of the class share. These characteristics are determined by the properties, methods, and events that define the class and the values of attributes that modify the behavior of each of these class components. Class definitions describe how objects of the class are created and destroyed, what data the objects contain, and how you can manipulate this data.

**Class Hierarchies**

It sometimes makes sense to define a new class in terms of existing classes. This approach enables you to reuse the designs and techniques in a new class that represents a similar entity. You accomplish this reuse by creating a subclass. A subclass defines objects that are a subset of those objects defined by the superclass. A subclass is more specific than its superclass and might add new properties, methods, and events to those components inherited from the superclass.

Mathematical sets can help illustrate the relationships among classes. In the following diagram, the set of Positive Integers is a subset of the set of Integers and a subset of Positives. All three sets are subsets of Reals, which is a subset of All Numbers.

The definition of Positive Integers requires the additional specification that members of the set be greater than zero. Positive Integers combine the definitions from both Integers and Positives. The resulting subset is more specific, and therefore more narrowly defined, than the supersets, but still shares all the characteristics that define the supersets.
The “is a” relationship is a good way to determine if it is appropriate to define a particular subset in terms of existing superset. For example, each of the following statements makes senses:

- A Positive Integer is an Integer
- A Positive Integer is a Positive number

If the “is a” relationship holds, then it is likely you can define a new class from a class or classes that represent some more general case.

**Reusing Solutions**

Classes are usually organized into taxonomies to foster code reuse. For example, if you define a class to implement an interface to the serial port of a computer, it would probably be similar to a class designed to implement an interface to the parallel port. To reuse code, you could define a superclass that contains everything that is common to the two types of ports, and then derive subclasses from the superclass in which you implement only what is unique to each specific port. Then the subclasses would inherit all the common functionality from the superclass.
Objects

A class is like a template for the creation of a specific instance of the class. This instance or object contains actual data for a particular entity that is represented by the class. For example, an instance of a bank account class is an object that represents a specific bank account, with an actual account number and an actual balance. This object has built into it the ability to perform operations defined by the class, such as making deposits to and withdrawals from the account balance.

Objects are not just passive data containers. Objects actively manage the data contained by allowing only certain operations to be performed, by hiding data that does not need to be public, and by preventing external clients from misusing data by performing operations for which the object was not designed. Objects even control what happens when they are destroyed.

Encapsulating Information

An important aspect of objects is that you can write software that accesses the information stored in the object via its properties and methods without knowing anything about how that information is stored, or even whether it is stored or calculated when queried. The object isolates code that accesses the object from the internal implementation of methods and properties. You can define classes that hide both data and operations from any methods that are not part of the class. You can then implement whatever interface is most appropriate for the intended use.

References


See Also

Related Examples

- “Create a Simple Class” on page 2-2
- “Developing Classes — Typical Workflow” on page 3-8
- “Class to Represent Structured Data” on page 3-19
- “Class to Implement Linked Lists” on page 3-31
Developing Classes — Typical Workflow

In this section...

“Formulating a Class” on page 3-8
“Specifying Class Components” on page 3-9
“BankAccount Class Implementation” on page 3-10
“Formulating the AccountManager Class” on page 3-14
“Implementing the AccountManager Class” on page 3-15
“AccountManager Class Synopsis” on page 3-15
“Using BankAccount Objects” on page 3-17

Formulating a Class

This example discusses how to approach the design and implementation of a class. The objective of this class is to represent a familiar concept (a bank account). However, you can apply the same approach to most class designs.

To design a class that represents a bank account, first determine the elements of data and the operations that form your abstraction of a bank account. For example, a bank account has:

- An account number
- An account balance
- A status (open, closed, etc.)

You must perform certain operations on a bank account:

- Create an object for each bank account
- Deposit money
- Withdraw money
- Generate a statement
- Save and load the BankAccount object

If the balance is too low and you attempt to withdraw money, the bank account broadcasts a notice. When this event occurs, the bank account broadcasts a notice to other entities.
that are designed to listen for these notices. In this example, a simplified version of an account manager program performs this task.

In this example, an account manager program determines the status of all bank accounts. This program monitors the account balance and assigns one of three values:

- **open** — Account balance is a positive value
- **overdrawn** — Account balance is overdrawn, but by $200 or less.
- **closed** — Account balance is overdrawn by more than $200.

These features define the requirements of the BankAccount and AccountManager classes. Include only what functionality is required to meet your specific objectives. Support special types of accounts by subclassing BankAccount and adding more specific features to the subclasses. Extend the AccountManager as required to support new account types.

### Specifying Class Components

Classes store data in properties, implement operations with methods, and support notifications with events and listeners. Here is how the BankAccount and AccountManager classes define these components.

#### Class Data

The class defines these properties to store the account number, account balance, and the account status:

- **AccountNumber** — A property to store the number identifying the specific account. MATLAB assigns a value to this property when you create an instance of the class. Only BankAccount class methods can set this property. The SetAccess attribute is private.

- **AccountBalance** — A property to store the current balance of the account. The class operation of depositing and withdrawing money assigns values to this property. Only BankAccount class methods can set this property. The SetAccess attribute is private.

- **AccountStatus** — The BankAccount class defines a default value for this property. The AccountManager class methods change this value whenever the value of the AccountBalance falls below 0. The Access attribute specifies that only the AccountManager and BankAccount classes have access to this property.
• **AccountListener** — Storage for the *InsufficientFunds* event listener. Saving a BankAccount object does not save this property because you must recreate the listener when loading the object.

**Class Operations**

These methods implement the operations defined in the class formulation:

• **BankAccount** — Accepts an account number and an initial balance to create an object that represents an account.
• **deposit** — Updates the *AccountBalance* property when a deposit transaction occurs
• **withdraw** — Updates the *AccountBalance* property when a withdrawal transaction occurs
• **getStatement** — Displays information about the account
• **loadobj** — Recreates the account manager listener when you load the object from a MAT-file.

**Class Events**

The account manager program changes the status of bank accounts that have negative balances. To implement this action, the BankAccount class triggers an event when a withdrawal results in a negative balance. Therefore, the triggering of the *InsufficientFunds* event occurs from within the withdraw method.

To define an event, specify a name within an *events* block. Trigger the event by a call to the notify handle class method. Because *InsufficientFunds* is not a predefined event, you can name it with any char vector and trigger it with any action.

**BankAccount Class Implementation**

It is important to ensure that there is only one set of data associated with any object of a BankAccount class. You would not want independent copies of the object that could have, for example, different values for the account balance. Therefore, implement the BankAccount class as a handle class. All copies of a given handle object refer to the same data.
BankAccount Class Synopsis

### BankAccount Class

```matlab
classdef BankAccount < handle

    properties (Access = ?AccountManager)
        AccountStatus = 'open'
    end

    properties (SetAccess = private)
        AccountNumber
        AccountBalance
    end

    properties (Transient)
        AccountListener
    end

    events
        InsufficientFunds
    end

    methods

end
```

<table>
<thead>
<tr>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handle class because there should be only one copy of any instance of BankAccount. “Comparison of Handle and Value Classes” on page 7-2.</td>
</tr>
</tbody>
</table>

AccountStatus contains the status of the account determined by the current balance. Access is limited to the BankAccount and AccountManager classes. “Class Members Access” on page 12-28. AccountStatus property access by AccountManager class methods.

AccountNumber and AccountBalance properties have private set access.

AccountListener property is transient so the listener handle is not saved.

See “Specify Property Attributes” on page 8-7.

Class defines event called InsufficientFunds. withdraw method triggers event when account balance becomes negative.

For information on events and listeners, see “Events”.

Block of ordinary methods. See “Methods and Functions” on page 5-15 for syntax.


### BankAccount Class

**function**  
```
function BA = BankAccount(AccountNumber,InitialBalance)
    BA.AccountNumber = AccountNumber;
    BA.AccountBalance = InitialBalance;
    BA.AccountListener = AccountManager.addAccount(BA);
end
```

Constructor initializes property values with input arguments.

**Discussion**

*AccountManager.addAccount* is static method of *AccountManager* class. Creates listener for *InsufficientFunds* event and stores listener handle in *AccountListener* property.

*deposit* adjusts value of *AccountBalance* property. If *AccountStatus* is closed and subsequent deposit brings *AccountBalance* into positive range, then *AccountStatus* is reset to open.

*withdraw* updates *AccountBalance* property. If value of account balance is negative as a result of the withdrawal, notify triggers *InsufficientFunds* event.

For more information on listeners, see “Events and Listeners Syntax” on page 11-22.

**function**  
```
function getStatement(BA)
    disp('-------------------------')
    disp(['Account: ',num2str(BA.AccountNumber)])
    ab = sprintf('%0.2f',BA.AccountBalance);
    disp(['CurrentBalance: ',ab])
    disp(['Account Status: ',BA.AccountStatus])
    disp('-------------------------')
end
```

Display selected information about the account.

**Discussion**

End of ordinary methods block.

BankAccount Class

Discussion

loadobj method:

- If the load operation fails, create the object from a struct.
- Recreates the listener using the newly created BankAccount object as the source.

For more information on saving and loading objects, see “Save and Load Process for Objects” on page 13-2

End of static methods block

End of classdef

Expand for Class Code

classdef BankAccount < handle
    properties (Access = ?AccountManager)
        AccountStatus = 'open'
    end
    properties (SetAccess = private)
        AccountNumber
        AccountBalance
    end
    properties (Transient)
        AccountListener
    end
    events
        InsufficientFunds
    end
    methods
        function BA = BankAccount(accNum,initBal)
            BA.AccountNumber = accNum;
            BA.AccountBalance = initBal;
            BA.AccountListener = AccountManager.addAccount(BA);
        end
        function deposit(BA,amt)
            BA.AccountBalance = BA.AccountBalance + amt;
            if BA.AccountBalance > 0
                BA.AccountStatus = 'open';
            end
        end
        function withdraw(BA,amt)
            if strcmp(BA.AccountStatus,'closed')&& BA.AccountBalance <= 0
                disp(['Account ',num2str(BA.AccountNumber),' has been closed.'])
                return
            end
        end
    end
end
newbal = BA.AccountBalance - amt;
BA.AccountBalance = newbal;
if newbal < 0
    notify(BA,'InsufficientFunds')
end
end

function getStatement(BA)
disp('-------------------------')
disp(['Account: ',num2str(BA.AccountNumber)])
ab = sprintf('%0.2f',BA.AccountBalance);
disp(['CurrentBalance: ',ab])
disp(['Account Status: ',BA.AccountStatus])
disp('-------------------------')
end

methods (Static)
function obj = loadobj(s)
    if isstruct(s)
        accNum = s.AccountNumber;
        initBal = s.AccountBalance;
        obj = BankAccount(accNum,initBal);
    else
        obj.AccountListener = AccountManager.addAccount(s);
    end
end
end

Formulating the AccountManager Class

The purpose of the AccountManager class is to provide services to accounts. For the BankAccount class, the AccountManager class listens for withdrawals that cause the balance to drop into the negative range. When the BankAccount object triggers the InsufficientFunds event, the AccountManager resets the account status.

The AccountManager class stores no data so it does not need properties. The BankAccount object stores the handle of the listener object.

The AccountManager performs two operations:

- Assign a status to each account as a result of a withdrawal
- Adds an account to the system by monitoring account balances.

Class Components

The AccountManager class implements two methods:
• **assignStatus** — Method that assigns a status to a `BankAccount` object. Serves as the listener callback.

• **addAccount** — Method that creates the `InsufficientFunds` listener.

## Implementing the AccountManager Class

The `AccountManager` class implements both methods as static because there is no need for an `AccountManager` object. These methods operate on `BankAccount` objects.

The `AccountManager` is not intended to be instantiated. Separating the functionality of the `AccountManager` class from the `BankAccount` class provides greater flexibility and extensibility. For example, doing so enables you to:

- Extend the `AccountManager` class to support other types of accounts while keeping the individual account classes simple and specialized.

- Change the criteria for the account status without affecting the compatibility of saved and loaded `BankAccount` objects.

- Develop an `Account` superclass that factors out what is common to all accounts without requiring each subclass to implement the account management functionality.

## AccountManager Class Synopsis

<table>
<thead>
<tr>
<th>AccountManager Class</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classdef AccountManager</code></td>
<td>This class defines the <code>InsufficientFunds</code> event listener and the listener callback.</td>
</tr>
</tbody>
</table>

| methods (Static) | There is no need to create an instance of this class so the methods defined are static. See “Static Methods” on page 9-31. |
### AccountManager Class

<table>
<thead>
<tr>
<th>Function</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignStatus(BA)</td>
<td>The <code>assignStatus</code> method is the callback for the <code>InsufficientFunds</code> event listener. It determines the value of a <code>BankAccount</code> object <code>AccountStatus</code> property based on the value of the <code>AccountBalance</code> property. The <code>BankAccount</code> class constructor calls the <code>AccountManager</code> <code>addAccount</code> method to create and store this listener.</td>
</tr>
<tr>
<td>addAccount(BA)</td>
<td><code>addAccount</code> creates the listener for the <code>InsufficientFunds</code> event that the <code>BankAccount</code> class defines. See “Control Listener Lifecycle” on page 11-28</td>
</tr>
</tbody>
</table>

### Expand for Class Code

```matlab
classdef AccountManager
    methods (Static)
    function assignStatus(BA)
        if BA.AccountBalance < 0
            if BA.AccountBalance < -200
                BA.AccountStatus = 'closed';
            else
                BA.AccountStatus = 'overdrawn';
            end
        end
    end
    function lh = addAccount(BA)
        lh = addlistener(BA, 'InsufficientFunds', @(src, ~)AccountManager.assignStatus(src));
    end
end
```

end statements for methods and for `classdef`.

3-16
Using BankAccount Objects

The BankAccount class, while overly simple, demonstrates how MATLAB classes behave. For example, create a BankAccount object with an account number and an initial deposit of $500:

BA = BankAccount(1234567, 500)

BA =

BankAccount with properties:

    AccountNumber: 1234567
    AccountBalance: 500
    AccountListener: [1x1 event.listener]

Use the getStatement method to check the status:

getStatement(BA)

-----------------------------------------------
Account: 1234567
CurrentBalance: 500.00
Account Status: open
-----------------------------------------------

Make a withdrawal of $600, which results in a negative account balance:

withdraw(BA, 600)
getStatement(BA)

-----------------------------------------------
Account: 1234567
CurrentBalance: -100.00
Account Status: overdrawn
-----------------------------------------------

The $600 withdrawal triggered the InsufficientFunds event. The current criteria defined by the AccountManager class results in a status of overdrawn.

Make another withdrawal of $200:

withdraw(BA, 200)
getStatement(BA)
Now the **AccountStatus** has been set to **closed** by the listener and further attempts to make withdrawals are blocked without triggering the event:

```
withdraw(BA,100)
```

Account 1234567 has been closed.

If the **AccountBalance** is returned to a positive value by a deposit, then the **AccountStatus** is returned to open and withdrawals are allowed again:

```
deposit(BA,700)
getStatement(BA)
```
Class to Represent Structured Data

In this section...

“Objects as Data Structures” on page 3-19
“Structure of the Data” on page 3-19
“The TensileData Class” on page 3-20
“Create an Instance and Assign Data” on page 3-20
“Restrict Properties to Specific Values” on page 3-21
“Simplifying the Interface with a Constructor” on page 3-22
“Calculate Data on Demand” on page 3-23
“Displaying TensileData Objects” on page 3-24
“Method to Plot Stress vs. Strain” on page 3-25
“TensileData Class Synopsis” on page 3-26

Objects as Data Structures

This example defines a class for storing data with a specific structure. Using a consistent structure for data storage makes it easier to create functions that operate on the data. A MATLAB struct with field names describing the particular data element is a useful way to organize data. However, a class can define both the data storage (properties) and operations that you can perform on that data (methods). This example illustrates these advantages.

Background for the Example

For this example, the data represents tensile stress/strain measurements. These data are used to calculate the elastic modulus of various materials. In simple terms, stress is the force applied to a material and strain is the resulting deformation. Their ratio defines a characteristic of the material. While this is an over simplification of the process, it suffices for this example.

Structure of the Data

This table describes the structure of the data.
<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Material</td>
<td>char vector identifying the type of material tested</td>
</tr>
<tr>
<td>SampleNumber</td>
<td>Number of a particular test sample</td>
</tr>
<tr>
<td>Stress</td>
<td>Vector of numbers representing the stress applied to the sample during the test.</td>
</tr>
<tr>
<td>Strain</td>
<td>Vector of numbers representing the strain at the corresponding values of the applied stress.</td>
</tr>
<tr>
<td>Modulus</td>
<td>Number defining an elastic modulus of the material under test, which is calculated from the stress and strain data</td>
</tr>
</tbody>
</table>

### The TensileData Class

This example begins with a simple implementation of the class and builds on this implementation to illustrate how features enhance the usefulness of the class.

The first version of the class provides only data storage. The class defines a property for each of the required data elements.

```matlab
classdef TensileData
    properties
        Material
        SampleNumber
        Stress
        Strain
        Modulus
    end
end
```

### Create an Instance and Assign Data

The following statements create a `TensileData` object and assign data to it:

```matlab
td = TensileData;
td.Material = 'Carbon Steel';
td.SampleNumber = 001;
td.Stress = [2e4 4e4 6e4 8e4];
td.Strain = [.12 .20 .31 .40];
td.Modulus = mean(td.Stress./td.Strain);
```
Advantages of a Class vs. a Structure

Treat the TensileData object (td in the previous statements) much as you would any MATLAB structure. However, defining a specialized data structure as a class has advantages over using a general-purpose data structure, like a MATLAB struct:

- Users cannot accidentally misspell a field name without getting an error. For example, typing the following:
  
  ```
td.Modulis = ...
  ```

  would simply

  Add a field to a structure, but returns an error when td is an instance of the TensileData class.
- A class is easy to reuse. Once you have defined the class, you can easily extend it with subclasses that add new properties.
- A class is easy to identify. A class has a name so that you can identify objects with the `who` and `class` functions and the Workspace browser. The class name makes it easy to refer to records with a meaningful name.
- A class can validate individual field values when assigned, including class or value.
- A class can restrict access to fields, for example, allowing a particular field to be read, but not changed.

Restrict Properties to Specific Values

Restrict properties to specific values by defining a property set access method. MATLAB calls the set access method whenever setting a value for a property.

Material Property Set Function

The `Material` property set method restricts the assignment of the property to one of the following strings: aluminum, stainless steel, or carbon steel.

Add this function definition to the methods block.

```matlab
classdef TensileData
    properties
        Material
        SampleNumber
        Stress
    end
end
```
When there is an attempt to set the `Material` property, MATLAB calls the `set.Material` method before setting the property value.

If the value matches the acceptable values, the function sets the property to that value. The code within set method can access the property directly to avoid calling the property set method recursively.

For example:

```matlab
td = TensileData;  
td.Material = 'brass';
```

Error using TensileData/set.Material
Invalid Material

**Simplifying the Interface with a Constructor**

Simplify the interface to the `TensileData` class by adding a constructor that:

- Enables you to pass the data as arguments to the constructor
- Assigns values to properties

The constructor is a method having the same name as the class.

```matlab
methods
  function td = TensileData(material,samplenum,stress,strain)
    if nargin > 0
```

Create a `TensileData` object fully populated with data using the following statement:

```matlab
td = TensileData('carbon steel', 1, ...
    [2e4 4e4 6e4 8e4], ...
    [.12 .20 .31 .40]);
```

**Calculate Data on Demand**

If the value of a property depends on the values of other properties, define that property using the `Dependent` attribute. MATLAB does not store the values of dependent properties. The dependent property get method determines the property value when the property is queried.

**Calculating Modulus**

`TensileData` objects do not store the value of the `Modulus` property. The constructor does not have an input argument for the value of the `Modulus` property. The value of the `Modulus`:

- Is calculated from the `Stress` and `Strain` property values
- Must change if the value of the `Stress` or `Strain` property changes

Therefore, it is better to calculate the value of the `Modulus` property only when its value is requested. Use a property `get` access method to calculate the value of the `Modulus`.

**Modulus Property Get Method**

The `Modulus` property depends on `Stress` and `Strain`, so its `Dependent` attribute is `true`. Place the `Modulus` property in a separate `properties` block and set the `Dependent` attribute.

The `get.Modulus` method calculates and returns the value of the `Modulus` property.
properties (Dependent)
    Modulus
end

Define the property get method in a methods block using only default attributes.

methods
    function modulus = get.Modulus(obj)
        ind = find(obj.Strain > 0);
        modulus = mean(obj.Stress(ind)./obj.Strain(ind));
    end
end

This method calculates the average ratio of stress to strain data after eliminating zeros in the denominator data.

MATLAB calls the get.Modulus method when the property is queried. For example,

td = TensileData('carbon steel',1,...
    [2e4 4e4 6e4 8e4],...
    [.12 .20 .31 .40]);
td.Modulus

ans =
    1.9005e+005

**Modulus Property Set Method**

To set the value of a Dependent property, the class must implement a property set method. There is no need to allow explicit setting of the Modulus property. However, a set method enables you to provide a customized error message. The Modulus set method references the current property value and then returns an error:

methods
    function obj = set.Modulus(obj,~)
        fprintf('%s%d
','Modulus is: ',obj.Modulus)
        error('You cannot set the Modulus property');
    end
end

**Displaying TensileData Objects**

The TensileData class overloads the disp method. This method controls object display in the command window.
The `disp` method displays the value of the Material, SampleNumber, and Modulus properties. It does not display the Stress and Strain property data. These properties contain raw data that is not easily viewed in the command window.

The `disp` method uses `fprintf` to display formatted text in the command window:

```matlab
methods
    function disp(td)
        fprintf(1,...
            'Material: %s
Sample Number: %g
Modulus: %1.5g
',
            td.Material,td.SampleNumber,td.Modulus);
    end
end
```

**Method to Plot Stress vs. Strain**

It is useful to view a graph of the stress/strain data to determine the behavior of the material over a range of applied tension. The `TensileData` class overloads the MATLAB `plot` function.

The `plot` method creates a linear graph of the stress versus strain data and adds a title and axis labels to produce a standardized graph for the tensile data records:

```matlab
methods
    function plot(td,varargin)
        plot(td.Strain,td.Stress,varargin{:})
        title(['Stress/Strain plot for Sample',...
            num2str(td.SampleNumber)])
        ylabel('Stress (psi)')
        xlabel('Strain %')
    end
end
```

The first argument to this method is a `TensileData` object, which contains the data.

The method passes a variable list of arguments (`varargin`) directly to the built-in `plot` function. The `TensileData` `plot` method allows you to pass line specifier arguments or property name/value pairs.

For example:

```matlab
td = TensileData('carbon steel',1,...
    [2e4 4e4 6e4 8e4],[.12 .20 .31 .40]);
plot(td,'-+b','LineWidth',2)
```
**TensileData Class Synopsis**

<table>
<thead>
<tr>
<th>Example Code</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classdef TensileData</code></td>
<td>Value class enables independent copies of object. For more information, see “Comparison of Handle and Value Classes” on page 7-2</td>
</tr>
</tbody>
</table>
### Example Code

```matlab
properties
    Material
    SampleNumber
    Stress
    Strain
end

properties (Dependent)
    Modulus
end
```

### Discussion

See “Structure of the Data” on page 3-19

Calculate Modulus when queried. For information about this code, see “Calculate Data on Demand” on page 3-23.

For general information, see “Set and Get Methods for Dependent Properties” on page 8-61

For general information about methods, see “Ordinary Methods” on page 9-8

For information about this code, see “Simplifying the Interface with a Constructor” on page 3-22.

For general information about constructors, see “Class Constructor Methods” on page 9-21

Restrict possible values for Material property.

For information about this code, see “Restrict Properties to Specific Values” on page 3-21.

For general information about property set methods, see “Property Set Methods” on page 8-55.
Example Code

```matlab
function m = get.Modulus(obj)
    ind = find(obj.Strain > 0);
    m = mean(obj.Stress(ind)./obj.Strain(ind));
end
```

Discussion

Calculate Modulus property when queried.

For information about this code, see “Modulus Property Get Method” on page 3-23.

For general information about property get methods, see “Property Get Methods” on page 8-59.

```matlab
function obj = set.Modulus(obj,~)
    fprintf('%s%d
','Modulus is: ',obj.Modulus)
    error('You cannot set Modulus property');
end
```

Add set method for Dependent Modulus property. For information about this code, see “Modulus Property Set Method” on page 3-24.

For general information about property set methods, see “Property Set Methods” on page 8-55.

```matlab
function disp(td)
    fprintf(1,'Material: %s
Sample Number: %g
Modulus: %1.5g
','
',td.Material,td.SampleNumber,td.Modulus)
end
```

Overload disp method to display certain properties.

For information about this code, see “Displaying TensileData Objects” on page 3-24.

For general information about overloading disp, see “Overloading the disp Function” on page 18-42.

```matlab
function plot(td,varargin)
    plot(td.Strain,td.Stress,varargin{:})
    title(['Stress/Strain plot for Sample',
        num2str(td.SampleNumber)])
    ylabel('Stress (psi)')
    xlabel('Strain %')
end
```

Overload plot function to accept TensileData objects and graph stress vs. strain.

“Method to Plot Stress vs. Strain” on page 3-25

end statements for methods and for classdef.
classdef TensileData
    properties
        Material
        SampleNumber
        Stress
        Strain
    end
    properties (Dependent)
        Modulus
    end
    methods
        function td = TensileData(material,samplenum,stress,strain)
            if nargin > 0
                td.Material = material;
                td.SampleNumber = samplenum;
                td.Stress = stress;
                td.Strain = strain;
            end
        end
        function obj = set.Material(obj,material)
            if (strcmpi(material,'aluminum') ||...
                strcmpi(material,'stainless steel') ||...
                strcmpi(material,'carbon steel'))
                obj.Material = material;
            else
                error('Invalid Material')
            end
        end
        function m = get.Modulus(obj)
            ind = find(obj.Strain > 0);
            m = mean(obj.Stress(ind)./obj.Strain(ind));
        end
        function obj = set.Modulus(obj,~)
            fprintf('%s%d
','Modulus is: ',obj.Modulus)
            error('You cannot set Modulus property');
        end
        function disp(td)
```
sprintf('Material: %s
Sample Number: %g
Modulus: %1.5g
',...
    td.Material, td.SampleNumber, td.Modulus)
end

function plot(td, varargin)
    plot(td.Strain, td.Stress, varargin{:})
    title(['Stress/Strain plot for Sample ',...
        num2str(td.SampleNumber)])
    xlabel('Strain %')
    ylabel('Stress (psi)')
end
end
end

See Also

More About
•  “Class Components” on page 5-5
Class to Implement Linked Lists

In this section...

“Class Definition Code” on page 3-31
“dlnode Class Design” on page 3-31
“Create Doubly Linked List” on page 3-32
“Why a Handle Class for Linked Lists?” on page 3-34
“dlnode Class Synopsis” on page 3-34
“Specialize the dlnode Class” on page 3-46

Class Definition Code

For the class definition code listing, see “dlnode Class Synopsis” on page 3-34.

To use the class, create a folder named @dlnode and save dlnode.m to this folder. The parent folder of @dlnode must be on the MATLAB path. Alternatively, save dlnode.m to a path folder.

dlnode Class Design

dlnode is a class for creating doubly linked lists in which each node contains:

- Data array
- Handle to the next node
- Handle to the previous node

Each node has methods that enable the node to be:

- Inserted before a specified node in a linked list
- Inserted after a specific node in a linked list
- Removed from a list

Class Properties

The dlnode class implements each node as a handle object with three properties:
- **Data** — Contains the data for this node
- **Next** — Contains the handle of the next node in the list (SetAccess = private)
- **Prev** — Contains the handle of the previous node in the list (SetAccess = private)

This diagram shows a list with three-nodes n1, n2, and n3. It also shows how the nodes reference the next and previous nodes.

### Class Methods

The `dlnode` class implements the following methods:

- **dlnode** — Construct a node and assign the value passed as an input to the Data property
- **insertAfter** — Insert this node after the specified node
- **insertBefore** — Insert this node before the specified node
- **removeNode** — Remove this node from the list and reconnect the remaining nodes
- **clearList** — Remove large lists efficiently
- **delete** — Private method called by MATLAB when deleting the list.

### Create Doubly Linked List

Create a node by passing the node's data to the `dlnode` class constructor. For example, these statements create three nodes with data values 1, 2, and 3:

```matlab
n1 = dlnode(1);
n2 = dlnode(2);
n3 = dlnode(3);
```

Build these nodes into a doubly linked list using the class methods designed for this purpose:
\texttt{n2.insertAfter(n1)} \% Insert n2 after n1
\texttt{n3.insertAfter(n2)} \% Insert n3 after n2

Now the three nodes are linked:

\texttt{n1.Next} \% Points to n2
\begin{verbatim}
an =

dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
\end{verbatim}

\texttt{n2.Next.Prev} \% Points back to n2
\begin{verbatim}
an =

dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]
\end{verbatim}

\texttt{n1.Next.Next} \% Points to n3
\begin{verbatim}
an =

dlnode with properties:

    Data: 3
    Next: []
    Prev: [1x1 dlnode]
\end{verbatim}

\texttt{n3.Prev.Prev} \% Points to n1
\begin{verbatim}
an =

dlnode with properties:

    Data: 1
    Next: [1x1 dlnode]
    Prev: []
\end{verbatim}
Why a Handle Class for Linked Lists?

Each node is unique in that no two nodes can be previous to or next to the same node.

For example, a node object, node, contains in its Next property the handle of the next node object, node.Next. Similarly, the Prev property contains the handle of the previous node, node.Prev. Using the three-node linked list defined in the previous section, you can demonstrate that the following statements are true:

n1.Next == n2
n2.Prev == n1

Now suppose that you assign n2 to x:

x = n2;

The following two equalities are then true:

x == n1.Next
x.Prev == n1

But each instance of a node is unique so there is only one node in the list that can satisfy the conditions of being equal to n1.Next and having a Prev property that contains a handle to n1. Therefore, x must point to the same node as n2.

There has to be a way for multiple variables to refer to the same object. The MATLAB handle class provides a means for both x and n2 to refer to the same node.

The handle class defines the eq method (use methods('handle') to list the handle class methods), which enables the use of the == operator with all handle objects.

Related Information

For more information on handle classes, see “Comparison of Handle and Value Classes” on page 7-2.

dlnode Class Synopsis

This section describes the implementation of the dlnode class.
<table>
<thead>
<tr>
<th>Example Code</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>classdef</strong> dlnode &lt; handle</td>
<td>“dlnode Class Design” on page 3-31</td>
</tr>
<tr>
<td></td>
<td>“Why a Handle Class for Linked Lists?” on page 3-34</td>
</tr>
<tr>
<td></td>
<td>“Comparison of Handle and Value Classes” on page 7-2</td>
</tr>
<tr>
<td><strong>properties</strong></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>“dlnode Class Design” on page 3-31</td>
</tr>
<tr>
<td><strong>properties</strong> (SetAccess = private)</td>
<td>“Property Attributes” on page 8-9: SetAccess.</td>
</tr>
<tr>
<td>Next = dlnode.empty</td>
<td>Initialize these properties to empty dlnode objects.</td>
</tr>
<tr>
<td>Prev = dlnode.empty</td>
<td>For general information about properties, see “Property Syntax” on page 8-5</td>
</tr>
<tr>
<td><strong>methods</strong></td>
<td>For general information about methods, see “Methods in Class Design” on page 9-2</td>
</tr>
<tr>
<td><strong>function</strong> node = dlnode(Data)</td>
<td>Creating an individual node (not connected) requires only the data.</td>
</tr>
<tr>
<td></td>
<td>For general information about constructors, see “Guidelines for Constructors” on page 9-23</td>
</tr>
<tr>
<td><strong>function</strong> insertAfter(newNode, nodeBefore)</td>
<td>Insert node into a doubly linked list after specified node, or link the two specified nodes if there is not already a list. Assigns the correct values for Next and Prev properties.</td>
</tr>
<tr>
<td>removeNode(newNode);</td>
<td>“Insert Nodes” on page 3-39</td>
</tr>
<tr>
<td>newNode.Next = nodeBefore.Next;</td>
<td></td>
</tr>
<tr>
<td>newNode.Prev = nodeBefore;</td>
<td></td>
</tr>
<tr>
<td>if ~isempty(nodeBefore.Next)</td>
<td></td>
</tr>
<tr>
<td>nodeBefore.Next.Prev = newNode;</td>
<td></td>
</tr>
<tr>
<td>Example Code</td>
<td>Discussion</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
</tbody>
</table>
| function insertBefore(newNode, nodeAfter)  
removeNode(newNode);  
newNode.Next = nodeAfter;  
newNode.Prev = nodeAfter.Prev;  
if ~isempty(nodeAfter.Prev)  
    nodeAfter.Prev.Next = newNode;  
end  
nodeAfter.Prev = newNode;  
end | Insert node into doubly linked list before specified node, or link the two specified nodes if there is not already a list. This method assigns correct values for Next and Prev properties.  
See “Insert Nodes” on page 3-39 |
| function removeNode(node)  
if ~isscalar(node)  
    error(‘Nodes must be scalar’)  
end  
prevNode = node.Prev;  
nextNode = node.Next;  
if ~isempty(prevNode)  
    prevNode.Next = nextNode;  
end  
if ~isempty(nextNode)  
    nextNode.Prev = prevNode;  
end  
ode.Next = = dlnode.empty;  
ode.Prev = = dlnode.empty;  
end | Remove node and fix the list so that remaining nodes are properly connected.  
ode argument must be scalar.  
Once there are no references to node, MATLAB deletes it.  
“Remove a Node” on page 3-41 |
| function clearList(node)  
prev = node.Prev;  
ext = node.Next;  
removeNode(node)  
while ~isempty(next)  
    node = next;  
ext = node.Next;  
    removeNode(node);  
end  
while ~isempty(prev)  
    prev = node.Prev;  
    node = prev;  
    removeNode(node)  
end | Avoid recursive calls to destructor as a result of clearing the list variable. Loop through list to disconnect each node.  
When there are no references to a node, MATLAB calls the class destructor (see the delete method) before deleting it. |
| methods (Access = private)  
function delete(node)  
    clearList(node)  
end | Class destructor method. MATLAB calls the delete method you delete a node that is still connected to the list. |
| end | End of private methods and end of class definition. |
Expand for Class Code

classdef dlnode < handle
% dlnode A class to represent a doubly-linked node.
% Link multiple dlnode objects together to create linked lists.
properties
    Data
end
properties(SetAccess = private)
    Next = dlnode.empty
    Prev = dlnode.empty
end

methods
    function node = dlnode(Data)
        % Construct a dlnode object
        if nargin > 0
            node.Data = Data;
        end
    end

    function insertAfter(newNode, nodeBefore)
        % Insert newNode after nodeBefore.
        removeNode(newNode);
        newNode.Next = nodeBefore.Next;
        newNode.Prev = nodeBefore;
        if ~isempty(nodeBefore.Next)
            nodeBefore.Next.Prev = newNode;
        end
        nodeBefore.Next = newNode;
    end

    function insertBefore(newNode, nodeAfter)
        % Insert newNode before nodeAfter.
        removeNode(newNode);
        newNode.Next = nodeAfter;
        newNode.Prev = nodeAfter.Prev;
        if ~isempty(nodeAfter.Prev)
            nodeAfter.Prev.Next = newNode;
        end
        nodeAfter.Prev = newNode;
    end

    function removeNode(node)
% Remove a node from a linked list.
if ~isscalar(node)
    error('Input must be scalar')
end
prevNode = node.Prev;
nextNode = node.Next;
if ~isempty(prevNode)
    prevNode.Next = nextNode;
end
if ~isempty(nextNode)
    nextNode.Prev = prevNode;
end
node.Next = dlnode.empty;
node.Prev = dlnode.empty;
end

function clearList(node)
% Clear the list before
% clearing list variable
prev = node.Prev;
next = node.Next;
removeNode(node)
while ~isempty(next)
    node = next;
    next = node.Next;
    removeNode(node);
end
while ~isempty(prev)
    node = prev;
    prev = node.Prev;
    removeNode(node);
end
end
end

methods (Access = private)
function delete(node)
clearList(node)
end
end
end
Class Properties

Only dlnode class methods can set the Next and Prev properties because these properties have private set access (SetAccess = private). Using private set access prevents client code from performing any incorrect operation with these properties. The dlnode class methods perform all the operations that are allowed on these nodes.

The Data property has public set and get access, allowing you to query and modify the value of Data as required.

Here is how the dlnode class defines the properties:

```plaintext
properties
  Data
end

properties(SetAccess = private)
  Next = dlnode.empty;
  Prev = dlnode.empty;
end
```

Construct a Node Object

To create a node object, specify the node's data as an argument to the constructor:

```plaintext
function node = dlnode(Data)
  if nargin > 0
    node.Data = Data;
  end
end
```

Insert Nodes

There are two methods for inserting nodes into the list — insertAfter and insertBefore. These methods perform similar operations, so this section describes only insertAfter in detail.

```plaintext
function insertAfter(newNode, nodeBefore)
  removeNode(newNode);
  newNode.Next = nodeBefore.Next;
  newNode.Prev = nodeBefore;
  if ~isempty(nodeBefore.Next)
    nodeBefore.Next.Prev = newNode;
  end
  nodeBefore.Next = newNode;
end
```
How insertAfter Works

First, insertAfter calls the removeNode method to ensure that the new node is not connected to any other nodes. Then, insertAfter assigns the newNode Next and Prev properties to the handles of the nodes that are after and before the newNode location in the list.

For example, suppose that you want to insert a new node, nnew, after an existing node, n1, in a list containing n1—n2—n3.

First, create nnew:

nnew = dlnode(rand(3));

Next, call insertAfter to insert nnew into the list after n1:

nnew.insertAfter(n1)

The insertAfter method performs the following steps to insert nnew in the list between n1 and n2:

- Set nnew.Next to n1.Next (n1.Next is n2):
  
  nnew.Next = n1.Next;

- Set nnew.Prev to n1
  
  nnew.Prev = n1;

- If n1.Next is not empty, then n1.Next is still n2, so n1.Next.Prev is n2.Prev, which is set to nnew
  
  n1.Next.Prev = nnew;

- n1.Next is now set to nnew
  
  n1.Next = nnew;
Remove a Node

The `removeNode` method removes a node from a list and reconnects the remaining nodes. The `insertBefore` and `insertAfter` methods always call `removeNode` on the node to insert before attempting to connect it to a linked list.

Calling `removeNode` ensures that the node is in a known state before assigning it to the `Next` or `Prev` property:

```matlab
function removeNode(node)
    if ~isscalar(node)
        error('Input must be scalar')
    end
    prevNode = node.Prev;
    nextNode = node.Next;
    if ~isempty(prevNode)
        prevNode.Next = nextNode;
    end
    if ~isempty(nextNode)
        nextNode.Prev = prevNode;
    end
    node.Next = dlnode.empty;
    node.Prev = dlnode.empty;
end
```

For example, suppose that you remove `n2` from a three-node list (`n1–n2–n3`):

```matlab
n2.removeNode;
```
removeNode removes n2 from the list and reconnects the remaining nodes with the following steps:

\[
n1 = n2.\text{Prev}; \\
n3 = n2.\text{Next}; \\
\text{if } n1 \text{ exists, then} \\
\quad n1.\text{Next} = n3; \\
\text{if } n3 \text{ exists, then} \\
\quad n3.\text{Prev} = n1
\]

The list is rejoined because \( n1 \) connects to \( n3 \) and \( n3 \) connects to \( n1 \). The final step is to ensure that \( n2.\text{Next} \) and \( n2.\text{Prev} \) are both empty (that is, \( n2 \) is not connected):

\[
n2.\text{Next} = \text{dlnode.empty}; \\
n2.\text{Prev} = \text{dlnode.empty};
\]

**Removing a Node from a List**

Suppose that you create a list with 10 nodes and save the handle to the head of the list:

\[
\text{head} = \text{dlnode}(1); \\
\text{for } i = 10:-1:2 \\
\quad \text{new} = \text{dlnode}(i); \\
\quad \text{insertAfter(new,head);} \\
\text{end}
\]

Now remove the third node (Data property assigned the value 3):

\[
\text{removeNode(head.Next.Next)}
\]

Now the third node in the list has a data value of 4:
head.Next.Next
ans =
    dlnode with properties:
        Data: 4
        Next: [1x1 dlnode]
        Prev: [1x1 dlnode]

And the previous node has a Data value of 2:
head.Next
ans =
    dlnode with properties:
        Data: 2
        Next: [1x1 dlnode]
        Prev: [1x1 dlnode]

Delete a Node

To delete a node, call the removeNode method on that node. The removeNode method disconnects the node and reconnects the list before allowing MATLAB to destroy the removed node. MATLAB destroys the node once references to it by other nodes are removed and the list is reconnected.
Delete the List

When you create a linked list and assign a variable that contains, for example, the head or tail of the list, clearing that variable causes the destructor to recurse through the entire list. With large enough list, clearing the list variable can result in MATLAB exceeding its recursion limit.

The `clearList` method avoids recursion and improves the performance of deleting large lists by looping over the list and disconnecting each node. `clearList` accepts the handle of any node in the list and removes the remaining nodes.

```matlab
function clearList(node)
    if ~isscalar(node)
        error('Input must be scalar')
    end
    prev = node.Prev;
    next = node.Next;
    removeNode(node)
    while ~isempty(next)
        node = next;
        next = node.Next;
    end
end
```
removeNode(node);
end
while ~isempty(prev)
    node = prev;
    prev = node.Prev;
    removeNode(node)
end

For example, suppose that you create a list with many nodes:

head = dlnode(1);
for k = 100000:-1:2
    nextNode = dlnode(k);
    insertAfter(nextNode,head)
end

The variable head contains the handle to the node at the head of the list:

head

head =

    dlnode with properties:

    Data: 1
    Next: [1x1 dlnode]
    Prev: []

head.Next

ans =

    dlnode with properties:

    Data: 2
    Next: [1x1 dlnode]
    Prev: [1x1 dlnode]

You can call clearList to remove the whole list:

clearList(head)

The only nodes that have not been deleted by MATLAB are those nodes for which there exists an explicit reference. In this case, those references are head and nextNode:
head
head =

dlnode with properties:

Data: 1
Next: []
Prev: []

nextNode
nextNode =

dlnode with properties:

Data: 2
Next: []
Prev: []

You can remove these nodes by clearing the variables:
clear head nextNode

**The delete Method**

The `delete` method simply calls the `clearList` method:

```matlab
methods (Access = private)
    function delete(node)
        clearList(node)
    end
end
```

The `delete` method has private access to prevent users from calling `delete` when intending to delete a single node. MATLAB calls `delete` implicitly when the list is destroyed.

To delete a single node from the list, use the `removeNode` method.

**Specialize the dlnode Class**

The `dlnode` class implements a doubly linked list and provides a convenient starting point for creating more specialized types of linked lists. For example, suppose that you want to create a list in which each node has a name.
Rather than copying the code used to implement the `dlnode` class, and then expanding upon it, you can derive a new class from `dlnode` (that is, subclass `dlnode`). You can create a class that has all the features of `dlnode` and also defines its own additional features. And because `dlnode` is a handle class, this new class is a handle class too.

**NamedNode Class Definition**

To use the class, create a folder named `@NamedNode` and save `NamedNode.m` to this folder. The parent folder of `@NamedNode` must be on the MATLAB path. Alternatively, save `NamedNode.m` to a path folder.

The following class definition shows how to derive the `NamedNode` class from the `dlnode` class:

```matlab
classdef NamedNode < dlnode
    properties
        Name = ''
    end
    methods
        function n = NamedNode (name, data)
            if nargin == 0
                name = ''; data = [];
            end
            n = n@dlnode(data);
            n.Name = name;
        end
    end
end
```

The `NamedNode` class adds a `Name` property to store the node name.

The constructor calls the class constructor for the `dlnode` class, and then assigns a value to the `Name` property.

**Use NamedNode to Create a Doubly Linked List**

Use the `NamedNode` class like the `dlnode` class, except that you specify a name for each node object. For example:

```matlab
n(1) = NamedNode('First Node', 100);
n(2) = NamedNode('Second Node', 200);
n(3) = NamedNode('Third Node', 300);
```
Now use the insert methods inherited from `dlnode` to build the list:

```matlab
n(2).insertAfter(n(1))
n(3).insertAfter(n(2))
```

A single node displays its name and data when you query its properties:

```matlab
n(1).Next
ans =

    NamedNode with properties:

      Name: 'Second Node'
      Data: 200
      Next: [1x1 NamedNode]
      Prev: [1x1 NamedNode]

n(1).Next.Next
ans =

    NamedNode with properties:

      Name: 'Third Node'
      Data: 300
      Next: []
      Prev: [1x1 NamedNode]

n(3).Prev.Prev
ans =

    NamedNode with properties:

      Name: 'First Node'
      Data: 100
      Next: [1x1 NamedNode]
      Prev: []
See Also

More About

• “The Handle Superclass” on page 7-13
Static Data
Static Data

In this section...

“What Is Static Data” on page 4-2
“Static Variable” on page 4-2
“Static Data Object” on page 4-3
“Constant Data” on page 4-5

What Is Static Data

Static data refers to data that all objects of the class share and that you can modify after creation.

Use static data to define counters used by class instances or other data that is shared among all objects of a class. Unlike instance data, static data does not vary from one object to another. MATLAB provides several ways to define static data, depending on your requirements.

Static Variable

Classes can use a persistent variable to store static data. Define a static method or local function in which you create a persistent variable. The method or function provides access to this variable. Use this technique when you want to store one or two variables.

Saving an object of the class defining the persistent variable does not save the static data associated with the class. To save your static data in an object, or define more extensive data, use the static data object technique “Static Data Object” on page 4-3.

Implementation

The StoreData class defines a static method that declares a persistent variable Var. The setgetVar method provides set and get access to the data in the persistent variable. Because the setgetVar method has public access, you can set and get the data stored in the persistent variable globally. Control the scope of access by setting the method Access attribute.

```
classdef StoreData
    methods (Static)
```
function out = setgetVar(data)
    persistent Var;
    if nargin
        Var = data;
    end
    out = Var;
end
end

Set the value of the variable by calling `setgetVar` with an input argument. The method assigns the input value to the persistent variable:

```
StoreData.setgetVar(10);
```

Get the value of the variable by calling `setgetVar` with no input argument:

```
a = StoreData.setgetVar
a =
   10
```

Clear the persistent variable by calling `clear` on the `StoreData` class:

```
clear StoreData
a = StoreData.setgetVar
a =
   []
```

Add a method like `setgetVar` to any class in which you want the behavior of a static property.

**Static Data Object**

To store more extensive data, define a handle class with public properties. Assign an object of the class to a constant property of the class that uses the static data. This technique is useful when you want to:

- Add more properties or methods that modify the data.
- Save objects of the data class and reload the static data.
Implementation

The SharedData class is a handle class, which enables you to reference the same object data from multiple handle variables:

```matlab
classdef SharedData < handle
    properties
        Data1
        Data2
    end
end
```

The UseData class is the stub of a class that uses the data stored in the SharedData class. The UseData class stores the handle to a SharedData object in a constant property.

```matlab
classdef UseData
    properties (Constant)
        Data = SharedData
    end
    % Class code here
end
```

The Data property contains the handle of the SharedData object. MATLAB constructs the SharedData object when loading the UseData class. All subsequently created instances of the UseData class refer to the same SharedData object.

To initialize the SharedData object properties, load the UseData class by referencing the constant property.

```matlab
h = UseData.Data
```

```matlab
h =

    SharedData with properties:

        Data1: []
        Data2: []
```

Use the handle to the SharedData object to assign data to property values:

```matlab
h.Data1 = 'MyData1';
h.Data2 = 'MyData2';
```

Each instance of the UseData class refers to the same handle object:
a1 = UseData;

a2 = UseData;

Reference the data using the object variable:

a1.Data.Data1

ans =

MyData1

Assign a new value to the properties in the SharedData object:

a1.Data.Data1 = rand(3);

All new and existing objects of the UseData class share the same SharedData object. a2 now has the rand(3) data that was assigned to a1 in the previous step:

a2.Data.Data1

ans =

0.8147    0.9134    0.2785
0.9058    0.6324    0.5469
0.1270    0.0975    0.9575

To reinitialize the constant property, clear all instances of the UseData class and then clear the class:

clear a1 a2
clear UseData

**Constant Data**

To store constant values that do not change, assign the data to a constant property. All instances of the class share the same value for that property. Control the scope of access to constant properties by setting the property Access attribute.

The only way to change the value of a constant property is to change the class definition. Use constant properties like public final static fields in Java®.

**See Also**
clear | persistent
Related Examples

- “Define Class Properties with Constant Values” on page 15-2
- “Static Methods” on page 9-31

More About

- “Method Attributes” on page 9-5
- “Property Attributes” on page 8-9
- “Static Properties” on page 5-66
Class Definition—Syntax Reference

- “Class Files and Folders” on page 5-2
- “Class Components” on page 5-5
- “Classdef Block” on page 5-10
- “Properties” on page 5-12
- “Methods and Functions” on page 5-15
- “Events and Listeners” on page 5-20
- “Attribute Specification” on page 5-22
- “Call Superclass Methods on Subclass Objects” on page 5-25
- “Representative Class Code” on page 5-28
- “MATLAB Code Analyzer Warnings” on page 5-34
- “Objects In Conditional Statements” on page 5-37
- “Operations on Objects” on page 5-44
- “Use of Editor and Debugger with Classes” on page 5-48
- “Automatic Updates for Modified Classes” on page 5-50
- “Compatibility with Previous Versions” on page 5-58
- “Comparison of MATLAB and Other OO Languages” on page 5-61
Class Files and Folders

Class Definition Files

Put class definition code in files that have the .m extension. The name of the file must be the same as the name of the class followed by the .m extension.

For information on the code that defines a class, see “Class Components” on page 5-5.

Options for Class Folders

There are two ways to create folders that contain class-definition files:

- Path folder — a folder that is on the MATLAB path.
- Class folder — a folder that is in a path folder and is named with the @ character and the class name. For example:
  
  @MyClass

Class folders are not directly on the MATLAB path. The path folder that contains the class folder is on the MATLAB path.

Options for Class Files

There are two ways to specify classes with respect to files and folders:

- Create a single, self-contained class definition file in a path folder or a class folder
- Define a class in multiple files, which requires you to use a class folder inside a path folder
Create a Single, Self-Contained Class Definition File

Create a single, self-contained class definition file in a folder on the MATLAB® path. The name of the file must match the class (and constructor) name and must have the .m extension. Define the class entirely in this file. You can put other single-file classes in this folder.

The following diagram shows an example of this folder organization. `pathfolder` is a folder on the MATLAB path.

```
     pathfolder
       ClassNameA.m  Contains classdef and methods for ClassNameA
       ClassNameB.m  Contains classdef and methods for ClassNameB
       ClassNameC.m  Contains classdef and methods for ClassNameC
       ...
       ordinaryFunction.m  A function on the path
```

Distribute the Class Definition to Multiple Files

If you use multiple files to define a class, put all the class-definition files (the file containing the classdef and all class method files) in a single `@ClassName` folder. That class folder must be inside a folder that is on the MATLAB path. You can define only one class in a class folder.

```
     pathfolder
       @ClassNameA
         ClassNameA.m  Contains classdef
         classMethod.m  Class method in separate file
         ClassNameB.m  Contains entire class definition
```

A path folder can contain classes defined in both class folders and single files without a class folder.

Group Classes with Package Folders

The parent folder to a package folder is on the MATLAB path, but the package folder is not. Package folders (which always begin with a + character) can contain multiple class
definitions, package-scoped functions, and other packages. A package folder defines a new name space in which you can reuse class names. Use the package name to refer to classes and functions defined in package folders (for example, packagefld1.ClassNameA(), packagefld2.packageFunction()).

```
pathfolder
    +packagefld1
        @ClassNameA
            ClassNameA.m  Contains classdef
            classMethod.m  Class method in separate file
            ClassNameB.m  Contains entire class definition

    +packagefld2  Defines a new name space
        packageFunction.m
        ClassNameA.m
        ClassNameB.m
```

See Also

More About

- “Folders Containing Class Definitions” on page 6-17
- “Packages Create Namespaces” on page 6-25
- “Methods in Separate Files” on page 9-11
Class Components

In this section...
“Class Building Blocks” on page 5-5
“Class Definition Block” on page 5-5
“Properties Block” on page 5-6
“Methods Block” on page 5-6
“Events Block” on page 5-7
“A Complete Class” on page 5-8
“Enumeration Classes” on page 5-8
“Related Information” on page 5-9

Class Building Blocks

MATLAB organizes class definition code into modular blocks, delimited by keywords. All keywords have an associated end statement:

• classdef...end — Definition of all class components
• properties...end — Declaration of property names, specification of property attributes, assignment of default values
• methods...end — Declaration of method signatures, method attributes, and function code
• events...end — Declaration of event name and attributes
• enumeration...end — Declaration of enumeration members and enumeration values for enumeration classes.

properties, methods, events, and enumeration are keywords only within a classdef block.

Class Definition Block

The classdef block contains the class definition within a file that starts with the classdef keyword and terminates with the end keyword.
classdef (ClassAttributes) ClassName < SuperClass
...
end

For example, this classdef defines a class called MyClass that subclasses the handle class, but cannot be used to derive subclasses:

```matlab
classdef (Sealed) MyClass < handle
...
end
```

See, “Classdef Block” on page 5-10 for more syntax information.

**Properties Block**

The properties block (one for each unique set of attribute specifications) contains property definitions, including optional initial values. The properties block starts with the properties keyword and terminates with the end keyword.

```matlab
classdef ClassName
    properties (PropertyAttributes)
        ...
    end
    ...
end
```

For example, this class defines a property called Prop1 that has private access and has a default value equal to the output of the date function.

```matlab
classdef MyClass
    properties (SetAccess = private)
        Prop1 = date
    end
    ...
end
```

See “Property Definition” on page 8-17 for more information.

**Methods Block**

The methods block (one for each unique set of attribute specifications) contains function definitions for the class methods. The methods block starts with the methods keyword and terminates with the end keyword.
classdef ClassName
    methods (MethodAttributes)
        ...
    end
    ...
end

For example:

classdef MyClass
    methods (Access = private)
        function obj = myMethod(obj)
            ...
        end
    end
end

See “Methods and Functions” on page 5-15 for more information.

**Events Block**

The events block (one for each unique set of attribute specifications) contains the names of events that this class declares. The events block starts with the events keyword and terminates with the end keyword.

classdef ClassName
    events (EventAttributes)
        EventName
            ...
    end
end

For example, this class defined an event called StateChange with a ListenAccess set to protected:

classdef EventSource
    events (ListenAccess = protected)
        StateChanged
            end
            ...
    end
end

See “Events and Listeners” on page 5-20 for more information.
A Complete Class

A complete class definition contains any combination of properties, methods, and events code blocks.

```matlab
classdef (Sealed) MyClass < handle
    properties (SetAccess = private)
        Prop1 = datenum(date)
    end
    properties
        Prop2
    end
    methods
        function obj = MyClass(x)
            obj.Prop2 = x;
        end
    end
    methods (Access = {?MyOtherClass})
        function d = myMethod(obj)
            d = obj.Prop1 + x;
        end
    end
    events (ListenAccess = protected)
        StateChanged
    end
end
```

Enumeration Classes

Enumeration classes are specialized classes that define a fixed set of names representing a single type of value. Enumeration classes use an enumeration block that contains the enumeration members defined by the class.

The enumeration block starts with the `enumeration` keyword and terminates with the `end` keyword.

```matlab
classdef ClassName < SuperClass
    enumeration
        EnumerationMember
    end
end
```
For example, this class defines two enumeration members that represent logical false and true:

```matlab
classdef Boolean < logical
    enumeration
        No (0)
        Yes (1)
    end
end
```

See, “Define Enumeration Classes” on page 14-5 for more information.

**Related Information**

“Folders Containing Class Definitions” on page 6-17
Classdef Block

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“How to Specify Attributes and Superclasses” on page 5-10</td>
</tr>
<tr>
<td>“Class Attribute Syntax” on page 5-10</td>
</tr>
<tr>
<td>“Superclass Syntax” on page 5-11</td>
</tr>
<tr>
<td>“Local Functions in Class File” on page 5-11</td>
</tr>
</tbody>
</table>

How to Specify Attributes and Superclasses

The classdef block contains the class definition. The classdef line is where you specify:

- Class attributes
- Superclasses

The classdef block contains the properties, methods, and events subblocks.

Class Attribute Syntax

Class attributes modify class behavior in some way. Assign values to class attributes only when you want to change their default value.

No change to default attribute values:

classdef ClassName
    ...
end

One or more attribute values assigned:

classdef (attribute1 = value,...)
    ...
end

For example, the TextString class specifies that it cannot be used to derive subclasses:

classdef TextString (Sealed)
    ...
end
See “Class Attributes” on page 6-6 for a list of attributes and a discussion of the behaviors they control.

**Superclass Syntax**

Derive a class from one or more other classes by specifying the superclasses on the `classdef` line:

```plaintext
classdef ClassName < SuperclassName ...
end ...
```

For example, the `LinkedList` class inherits from classes called `Array` and `handle`:

```plaintext
classdef LinkedList < Array & handle ...
end ...
```

**Local Functions in Class File**

You can define only one class per file. However, you can add local functions to a file containing the `classdef` block. Local functions are scoped to the `classdef` file and have access to private and protected class members.

```plaintext
classdef ClassName ...
end
function localFunction ...
end
```

**See Also**

**Related Examples**

- “User-Defined Classes” on page 6-2
- “Design Subclass Constructors” on page 12-9
- “Local Functions”
Properties

In this section...

“The Properties Block” on page 5-12
“Access to Property Values” on page 5-13

The Properties Block

Define class properties within a properties block:

```matlab
classdef ClassName
    properties (PropertyAttributes)
        PropertyName class = DefaultValue
    end
end
```

Property attributes apply to all properties defined within the block. To define properties with different attributes, use multiple properties block. All property attributes have default values. For a list of property attributes, see “Property Attributes” on page 8-9.

Optionally assign default values to the property in the properties block. MATLAB evaluates the assignment statement when the class is first referenced or when loading a saved object.

**Note** Evaluation of property default values occurs only when the value is first needed, and only once when MATLAB first initializes the class. MATLAB does not reevaluate the expression each time you create an instance of the class.

For more information on the evaluation of expressions that you assign as property default values, see “When MATLAB Evaluates Expressions” on page 6-13.

Properties with Different Attributes

The following class defines three properties. Model and Color use default attribute values, resulting in public read and write access. SerialNumber has read-only access by object users. Assign the SerialNumber property value from a class member function, such as the constructor or other class method.
classdef NewCar
    properties
        Model
        Color
    end
    properties (SetAccess = private)
        SerialNumber
    end
end
methods
    ...
end
end

**Access to Property Values**

Use dot notation to access property value.

A = NewCar

A = 

    NewCar with properties:

        Model: []
        Color: []
        SerialNumber: []

Set the **Model** and **Color** properties:

A.Model = 'XGT7000';
A.Color = 'Red';

Add a constructor to the NewCar class to set property values:

classdef NewCar
    properties
        Model
        Color
    end
    properties (SetAccess = private)
        SerialNumber
    end
    methods
        function obj = NewCar(model,color)
obj.Model = model;
obj.Color = color;
obj.SerialNumber = datenum(datetime('now'));
end
end
end
A = NewCar('XGT7000','Red')
A =

NewCar with properties:

    Model: 'XGT7000'
    Color: 'Red'
    SerialNumber: 7.362456078531134e+05

See Also

Related Examples

•  “Ways to Use Properties” on page 8-2
•  “Validate Property Values” on page 8-24
Methods and Functions

In this section...

“Methods and Functions” on page 5-15
“Method Calling Syntax” on page 5-15
“Private Methods” on page 5-17
“More Detailed Information On Methods” on page 5-17
“Class-Related Functions” on page 5-17
“How to Overload Functions and Operators” on page 5-18
“Rules for Defining Methods in Separate Files” on page 5-18

The Methods Block

Define methods as MATLAB functions within a methods block, inside the classdef block. The constructor method has the same name as the class and returns an object. You can assign values to properties in the class constructor. Terminate all method functions with an end statement.

```
classdef ClassName
    properties
        PropertyName
    end
    methods
        function obj = ClassName(arg1,...)
            obj.PropertyName = arg1;
        ...
        end
        function ordinaryMethod(obj,arg1,...)
            ...
        end
    end
    methods (Static)
        function staticMethod(arg1,...)
            ...
        end
    end
end
```

Method Calling Syntax

MATLAB differs from languages like C++ and Java in that there is no special hidden class object passed to all methods. You must pass an object of the class explicitly to the method.
The left most argument does not need to be the class object, and the argument list can have multiple objects. MATLAB dispatches to the method defined by the class of the dominant argument. For more information, see “Method Invocation” on page 9-15.

Methods must be on the MATLAB path when called. For example, if you create an object and then change your current folder to a folder from which the method file is not visible, an error occurs when you call that method.

Always use case sensitive method names in your MATLAB code.

**Ordinary Methods**

Call ordinary methods using MATLAB function syntax or dot notation. For example, suppose you have a class that defines ordinaryMethod. Pass an object of the defining class and whatever arguments are required.

```matlab
classdef MyClass
    methods
        function out = ordinaryMethod(obj,arg1)
            ...
        end
    end
end
```

Call `ordinaryMethod` using the object `obj` of the class and either syntax:

```matlab
obj = MyClass;
r = ordinaryMethod(obj,arg1);
r = obj.ordinaryMethod(arg1);
```

**Static Methods**

Static methods do not require an object of the class. To call a static method, prefix the method name with the class name so that MATLAB can determine what class defines the method.

```matlab
classdef MyClass
    methods (Static)
        function out = staticMethod(arg1)
            ...
        end
    end
end
```
Call staticMethod using the syntax `classname.methodname`:

```
r = MyClass.staticMethod(arg1);
```

See “Static Methods” on page 9-31 for information on methods that do not require objects of their class.

**Private Methods**

Use the `Access` method attribute to create a private method. You do not need to use a private folder.

See “Method Attributes” on page 9-5 for a list of method attributes.

**More Detailed Information On Methods**

“Methods”

**Class-Related Functions**

You can define functions that are not class methods in the file that contains the class definition (`classdef`). Define local functions outside of the `classdef - end` block, but in the same file as the class definition. Functions defined in `classdef` files work like local functions. You can call these functions from anywhere in the same file, but they are not visible outside of the file in which you define them.

Local functions in `classdef` files are useful for utility functions that you use only within that file. These functions can take or return arguments that are instances of the class but, it is not necessary, as in the case of ordinary methods. For example, the following code defines `myUtilityFcn` outside the `classdef` block:

```matlab
classdef MyClass
  properties
    PropName
  end
  methods
    function obj = MyClass(arg1)
      obj.PropName = arg1;
    end
  end
end

% End of classdef
```
function myUtilityFcn

end

You also can create package functions, which require the use of the package name when calling these functions.

**How to Overload Functions and Operators**

Overload MATLAB functions for your class by defining a class method with the same name as the function that you want to overload. MATLAB dispatches to the class method when the function is called with an instance of the class.

You can overload MATLAB arithmetic, logical, relational, and indexing operators by defining class methods with the appropriate names.

See the `handle` class for a list of operations defined for that class. All classes deriving from `handle` inherit these methods.

**Rules for Defining Methods in Separate Files**

The following rules apply to methods defined in separate files:

- To specify attributes for a method defined in a separate file, declare this method in a methods block in the `classdef` file. Specify the attribute values with the methods block.
- The syntax declared in the methods block (if used) must match the method's `function` line.
- The separate file must be in the class folder.
- The class constructor method must be defined in the `classdef` file. The constructor cannot be in a separate file.
- Handle class `delete` methods must be defined in the `classdef` file. The `delete` method cannot be in a separate file.

All functions that use dots in their names must be defined in the `classdef` file, including:

- Converter methods that must use the package name as part of the class name because the class is contained in packages.
• Property set and get access methods

For more information on defining methods in separate files, see “Methods in Separate Files” on page 9-11

See Also

More About
• “Methods in Class Design” on page 9-2
**Events and Listeners**

*In this section...*

<table>
<thead>
<tr>
<th>“Define and Trigger Events” on page 5-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Listen for Events” on page 5-20</td>
</tr>
</tbody>
</table>

**Define and Trigger Events**

To define an event, declare a name for the event in an `events` block. Trigger the event using the `handle` class `notify` method. Only classes derived from the `handle` class can define events.

For example, `MyClass` class:

- Subclasses `handle`
- Defines an event named `StateChange`
- Triggers the event using the inherited `notify` method in its `upDateUI` method.

```matlab
classdef MyClass < handle
    events
        StateChange
    end
    ...
    methods
        function upDateUI(obj)
            ...
            notify(obj,'StateChange');
        end
    end
end
```

**Listen for Events**

Any number of objects can listen to the `StateChange` event. When `notify` executes, MATLAB calls all registered listener callbacks. MATLAB passes the handle of the object generating the event and event data to the callback functions. To create a listener, use the `addlistener` method of the `handle` class.

```matlab
addlistener(event_obj,'StateChange',@myCallback)
```

To control the lifecycle of the listener, use the `event.listener` constructor to create the listener object.
See Also
- event.hasListener
- event.listener
- event.proplistener

Related Examples
- “Overview Events and Listeners” on page 11-2
- “Events and Listeners Syntax” on page 11-22
Attribute Specification

In this section...

“Attribute Syntax” on page 5-22
“Attribute Descriptions” on page 5-22
“Attribute Values” on page 5-23
“Simpler Syntax for true/false Attributes” on page 5-23

Attribute Syntax

Attributes modify the behavior of classes and class components (properties, methods, and events). Attributes enable you to define useful behaviors without writing complicated code. For example, you can create a read-only property by setting its SetAccess attribute to private, but leaving its GetAccess attribute set to public:

properties (SetAccess = private)
    ScreenSize = getScreenSize
end

All class definition blocks (classdef, properties, methods, and events) support specific attributes. All attributes have default values. Specify attribute values only in cases where you want to change from the default value to another predefined value.

Note Specify the value of a particular attribute only once in any component block.

Attribute Descriptions

For lists of supported attributes, see:

• “Class Attributes” on page 6-6
• “Property Attributes” on page 8-9
• “Method Attributes” on page 9-5
• “Event Attributes” on page 11-19
**Attribute Values**

When you specify attribute values, those values affect all the components defined within the defining block. For example, the following property definition blocks set the:

- AccountBalance property SetObservable attribute to true
- SSNumber and CreditCardNumber properties' Hidden attribute to true and SetAccess attribute to private.

Defining properties with different attribute settings requires multiple properties blocks.

```plaintext
properties (SetObservable = true)
  AccountBalance
end
properties (SetAccess = private, Hidden = true)
  SSNumber
  CreditCardNumber
end
```

Specified multiple attributes in a comma-separated list, as shown in the previous example.

When specifying class attributes, place the attribute list directly after the `classdef` keyword:

```plaintext
classdef (AttributeName = attributeValue) ClassName
  ...
end
```

**Simpler Syntax for true/false Attributes**

You can use a simpler syntax for attributes whose values are true or false — the attribute name alone implies true and adding the not operator (~) to the name implies false. For example:

```plaintext
methods (Static)
  ...
end
```

Is the same as:

```plaintext
methods (Static = true)
  ...
end
```
Use the not operator before an attribute name to define it as false:

```plaintext
methods (~Static)
  ...
end
```

Is the same as:

```plaintext
methods (Static = false)
  ...
end
```

All attributes that take a logical value (that is, `true` or `false`) have a default value of `false`. Therefore, specify an attribute only if you want to set it to `true`.

**See Also**

**Related Examples**

- “Evaluation of Expressions in Class Definitions” on page 6-10
Call Superclass Methods on Subclass Objects

In this section...

“Superclass Relation to Subclass” on page 5-25
“How to Call Superclass Methods” on page 5-25
“How to Call Superclass Constructor” on page 5-26

Superclass Relation to Subclass

Subclasses can override superclass methods to support the greater specialization defined by the subclass. Because of the relationship that a subclass object is a superclass object, it is often useful to call the superclass version of the method before executing the specialized subclass code.

How to Call Superclass Methods

Subclass methods can call superclass methods if both methods have the same name.
From the subclass, reference the method name and superclass name with the @ symbol.

This diagram illustrates how to call the superMethod defined by MySuperClass.

```
superMethod@MySuperClass(obj,superMethodArguments)
```

Arguments passed to the superclass method

For example, a subclass can call a superclass disp method to implement the display of the superclass part of the object. Then the subclass adds code to display the subclass part of the object:
```matlab
classdef MySub < MySuperClass
    methods
        function disp(obj)
            disp@MySuperClass(obj)
            ...
        end
    end
end
```

**How to Call Superclass Constructor**

If you create a subclass object, MATLAB calls the superclass constructor to initialize the superclass part of the subclass object. By default, MATLAB calls the superclass constructor without arguments. If you want the superclass constructor called with specific arguments, explicitly call the superclass constructor from the subclass constructor. The call to the superclass constructor must come before any other references to the object.

The syntax for calling the superclass constructor uses an @ symbol:

```
obj = obj@MySuperClass(SuperClassArguments);
```

In the following class, the `MySub` object is initialized by the `MySuperClass` constructor. The superclass constructor constructs the `MySuperClass` part of the object using the specified arguments.

```matlab
classdef MySub < MySuperClass
    methods
        function obj = MySub(arg1,arg2,...)
            obj = obj@MySuperClass(SuperClassArguments);
            ...
    end
end
```
See “Subclass Constructors” on page 9-26 for more information.

**See Also**

**Related Examples**

- “Modify Inherited Methods” on page 12-16
Representative Class Code

In this section...

“Class Calculates Area” on page 5-28
“Description of Class Definition” on page 5-31

Class Calculates Area

The CircleArea class shows the syntax of a typical class definition. This class stores a value for the radius of a circle and calculates the area of the circle when you request this information. CircleArea also implements methods to graph, display, and create objects of the class.

To use the CircleArea class, copy this code into a file named CircleArea.m and save this file in a folder that is on the MATLAB path.

classdef CircleArea
    properties
        Radius
    end
    properties (Constant)
        P = pi
    end
    properties (Dependent)
        Area
    end
    methods
        function obj = CircleArea(r)
            if nargin > 0
                obj.Radius = r;
            end
        end
        function val = get.Area(obj)
            val = obj.P*obj.Radius^2;
        end
        function obj = set.Radius(obj,val)
            if val < 0
                error('Radius must be positive')
            end
            obj.Radius = val;
        end
end
function plot(obj)
    r = obj.Radius;
    d = r*2;
    pos = [0 0 d d];
    curv = [1 1];
    rectangle('Position',pos,'Curvature',curv,...
        'FaceColor',[.9 .9 .9])
    line([0,r],[r,r])
    text(r/2,r+.5,['r = ',num2str(r)])
    title([Area = ',num2str(obj.Area)])
    axis equal
end
function disp(obj)
    rad = obj.Radius;
    disp(['Circle with radius: ',num2str(rad)])
end
end
methods (Static)
function obj = createObj
    prompt = {'Enter the Radius'};
    dlgTitle = 'Radius';
    rad = inputdlg(prompt,dlgTitle);
    r = str2double(rad{:});
    obj = CircleArea(r);
end
end
end

Use the CircleArea Class

Create an object using the dialog box:

ca = CircleArea.createObj

Add a value for radius and click OK.
Query the area of the defined circle:

```matlab
ca.Area
ans =
    164.2202
```

Call the overloaded plot method:

```matlab
plot(ca)
```
Description of Class Definition

Class definition code begins with the \texttt{classdef} keyword followed by the class name:

\begin{verbatim}
classdef CircleArea
end
\end{verbatim}

Define the \texttt{Radius} property within the \texttt{properties-end} keywords. Use default attributes:

\begin{verbatim}
    properties
        Radius
    end
\end{verbatim}

Define the \texttt{P} property as \texttt{Constant} (“Define Class Properties with Constant Values” on page 15-2). Call the \texttt{pi} function only once when class is initialized.

\begin{verbatim}
    properties (Constant)
        P = pi
    end
\end{verbatim}

Define the \texttt{Area} property as \texttt{Dependent} because its value depends on the \texttt{Radius} property.

\begin{verbatim}
    properties (Dependent)
        Area
    end
\end{verbatim}

\texttt{methods} \% Begin defining methods

The \texttt{CircleArea} class constructor method has the same name as the class and accepts the value of the circle radius as an argument. This method also allows no input arguments. (“Class Constructor Methods” on page 9-21)

\begin{verbatim}
function obj = CircleArea(r)
    if nargin > 0
        obj.Radius = r;
    else
        obj.Radius = 0;
    end
end
\end{verbatim}

Because the \texttt{Area} property is \texttt{Dependent}, it does not store its value. The \texttt{get.Area} method calculates the value of the \texttt{Area} property whenever it is queried. (“Set and Get Methods for Dependent Properties” on page 8-61)

\begin{verbatim}
function val = get.Area(obj)
    val = obj.P*obj.Radius^2;
end
\end{verbatim}
The `set.Radius` method tests the value assigned to the `Radius` property to ensure the value is not less than zero. MATLAB calls `set.Radius` to assign a value to `Radius`. (“Property Set Methods” on page 8-55.

```
function obj = set.Radius(obj,val)
    if val < 0
        error('Radius must be positive')
    end
    obj.Radius = val;
end
```

The `CircleArea` class overloads the `plot` function. The `plot` method uses the `rectangle` function to create a circle and draws the radius. (“Overload Functions in Class Definitions” on page 9-33

```
function plot(obj)
    r = obj.Radius;
    d = r*2;
    pos = [0 0 d d];
    curv = [1 1];
    rectangle('Position',pos,'Curvature',curv)
    line([0,r],[r,r])
    text(r/2,r+.5,['r = ',num2str(r)])
    axis equal
end
```

The `CircleArea` class overloads the `disp` function to change the way MATLAB displays objects in the command window.

```
function disp(obj)
    rad = obj.Radius;
    disp(['Circle with radius: ',num2str(rad)])
end
```

```
methods (Static)
```

The `CircleArea` class defines a Static method that uses a dialog box to create an object. (“Static Methods” on page 9-31

```
function obj = createObj
    prompt = {'Enter the Radius'};
    dlgTitle = 'Radius';
    rad = inputdlg(prompt,dlgTitle);
    r = str2double(rad{:});
    obj = CircleArea(r);
end
```
End of Static methods block and end of classdef block.
    end
end
MATLAB Code Analyzer Warnings

Syntax Warnings and Property Names

The MATLAB Code Analyzer helps you optimize your code and avoid syntax errors while you write code. It is useful to understand some of the rules that the Code Analyzer applies in its analysis of class definition code. This understanding helps you avoid situations in which MATLAB allows code that is undesirable.

Variable/Property Name Conflict Warnings

The Code Analyzer warns about the use of variable names in methods that match the names of properties. For example, suppose a class defines a property called EmployeeName and in this class, there is a method that uses EmployeeName as a variable:

```matlab
properties
    EmployeeName
end
methods
    function someMethod(obj,n)
        EmployeeName = n;
    end
end
```

While the previous function is legal MATLAB code, it results in Code Analyzer warnings for two reasons:

- The value of EmployeeName is never used
- EmployeeName is the name of a property that is used as a variable

If the function someMethod contained the following statement instead:

```matlab
obj.EmployeeName = n;
```
The Code Analyzer generates no warnings.

If you change `someMethod` to:

```matlab
function EN = someMethod(obj)
    EN = EmployeeName;
end
```

The Code Analyzer returns only one warning, suggesting that you might actually want to refer to the `EmployeeName` property.

While this version of `someMethod` is legal MATLAB code, it is confusing to give a property the same name as a function. Therefore, the Code Analyzer provides a warning suggesting that you might have intended the statement to be:

```matlab
EN = obj.EmployeeName;
```

**Exception to Variable/Property Name Rule**

Suppose you define a method that returns a value of a property and uses the name of the property for the output variable name. For example:

```matlab
function EmployeeName = someMethod(obj)
    EmployeeName = obj.EmployeeName;
end
```

The Code Analyzer does not warn when a variable name is the same as a property name when the variable is:

- An input or output variable
- A global or persistent variable

In these particular cases, the Code Analyzer does not warn you that you are using a variable name that is also a property name. Therefore, a coding error like the following:

```matlab
function EmployeeName = someMethod(obj)
    EmployeeName = EmployeeName; % Forgot to include obj.
end
```

does not trigger a warning from the Code Analyzer.
See Also

Related Examples

• “Use of Editor and Debugger with Classes” on page 5-48
Objects In Conditional Statements

Enable Use of Objects In Conditional Statements

Enable the use of objects in conditional statements by defining relational operators for the class of the object. Classes that derive from the handle class inherit relational operators. Value classes can implement operators to support the use of conditional statements involving objects. For information on defining operators for your class, see “Operator Overloading” on page 17-47.

How MATLAB Evaluates Switch Statements

MATLAB enables you to use objects in switch statements when the object’s class defines an eq method. The eq method implements the == operation on objects of that class.

For objects, switch_expression == case_expression defines how MATLAB evaluates switch and cases statements.

Note  You do not need to define eq methods for enumeration classes. See “Enumerations in Switch Statements” on page 5-41.

Handle Objects in Switch Statements

All classes derived from the handle class inherit an eq method. The expression, h1 == h2
is true if h1 and h2 are handles for the same object.

For example, the BasicHandle class derives from handle:
**classdef** BasicHandle < handle
    **properties**
    Prop1
    end
    **methods**
    function obj = BasicHandle(val)
        if nargin > 0
            obj.Prop1 = val;
        end
    end
end
end

Create a BasicHandle object and use it in a switch statement:

```matlab
h1 = BasicHandle('Handle Object');
h2 = h1;
```

Here is the switch statement code:

```matlab
switch h1
    case h2
        disp('h2 is selected')
    otherwise
        disp('h2 not selected')
end
```

The result is:

```
h2 is selected
```

**Object Must Be Scalar**

The switch statements work only with scalar objects. For example:

```matlab
h1(1) = BasicHandle('Handle Object');
h1(2) = BasicHandle('Handle Object');
h1(3) = BasicHandle('Handle Object');
h2 = h1;

switch h1
    case h2
        disp('h2 is selected')
    otherwise
        disp('h2 not selected')
end
```
The result is:

SWITCH expression must be a scalar or string constant.

In this case, h1 is not scalar. Use isscalar to determine if an object is scalar before entering a switch statement.

**How to Define the eq Method**

To enable the use of value-class objects in switch statements, implement an eq method for the class. Use the eq method to determine what constitutes equality of two object of the class.

**Behave Like a Built-in Type**

Some MATLAB functions also use the built-in == operator in their implementation. Therefore, your implementation of eq should be replaceable with the built-in eq to enable objects of your class work like built-in types in MATLAB code.

**Design of eq**

Implement the eq method to returns a logical array representing the result of the == comparison.

For example, the SwitchOnVer class implements an eq method that returns true for the == operation if the value of the Version property is the same for both objects. In addition, eq works with arrays the same way as the built-in eq. For the following expression:

\[ \text{obj1} == \text{obj2} \]

The eq method works like this:

- If both \text{obj1} and \text{obj2} are scalar, eq returns a scalar value.
- If both \text{obj1} and \text{obj2} are nonscalar arrays, then these arrays must have the same dimensions, and eq returns an array of the same size.
- If one input argument is scalar and the other is a nonscalar array, then eq treats the scalar object as if it is an array having the same dimensions as the nonscalar array.
Implementation of eq

Here is a class that implements an eq method. Ensure your implementation contains appropriate error checking for the intended use.

```matlab
classdef SwitchOnVer
    properties
        Version
    end
    methods
        function obj = SwitchOnVer(ver)
            if nargin > 0
                obj.Version = ver;
            end
        end
        function bol = eq(obj1,obj2)
            if ~strcmp(class(obj1),class(obj2))
                error('Objects are not of the same class')
            end
            s1 = numel(obj1);
            s2 = numel(obj2);
            if s1 == s2
                bol = false(size(obj1));
                for k=1:s1
                    if obj1(k).Version == obj2(k).Version
                        bol(k) = true;
                    else
                        bol(k) = false;
                    end
                end
            elseif s1 == 1
                bol = scalarExpEq(obj2,obj1);
            elseif s2 == 1
                bol = scalarExpEq(obj1,obj2);
            else
                error('Dimension missmatch')
            end
        function ret = scalarExpEq(ns,s)
            % ns is nonscalar array
            % s is scalar array
            ret = false(size(ns));
            n = numel(ns);
            for kk=1:n
                if ns(kk).Version == s.Version
```
ret(kk) = true;
else
ret(kk) = false;
end
datestr
date
disp
end
end
end
end

Use SwitchOnVer objects in switch statements:

% Create known versions of objects
ov1 = SwitchOnVer(1.0);
ov2 = SwitchOnVer(2.0);
ov3 = SwitchOnVer(3.0);
...

... iscalar(objIn)
switch(objIn)
    case ov1
        disp('This is version 1.0')
    case ov2
        disp('This is version 2.0')
    case ov3
        disp('This is version 3.0')
    otherwise
        disp('There is no version')
    end
else
    error('Input object must be scalar')
end

Enumerations in Switch Statements

MATLAB enables you to use enumerations in switch statements without requiring an explicitly defined eq method for the enumeration class.

For example, the WeeklyPlanner class defines enumerations for five days of the week. The switch/case statements in the todaySchedule static method dispatch on the enumeration member corresponding to the current day of the week. The date and datestr functions return a char vector with the name of the current day.
classdef WeeklyPlanner
    enumeration
        Monday, Tuesday, Wednesday, Thursday, Friday
    end
    methods (Static)
        function todaySchedule
            dayName = datestr(date,'dddd');
            dayEnum = WeeklyPlanner.(dayName);
            switch dayEnum
                case WeeklyPlanner.Monday
                    disp('Monday schedule')
                case WeeklyPlanner.Tuesday
                    disp('Tuesday schedule')
                case WeeklyPlanner.Wednesday
                    disp('Wednesday schedule')
                case WeeklyPlanner.Thursday
                    disp('Thursday schedule')
                case WeeklyPlanner.Friday
                    disp('Friday schedule')
            end
        end
    end
end
end

Call todaySchedule to display today’s schedule:

WeeklyPlanner.todaySchedule

Enumerations Derived from Built-In Types

Enumeration classes that derived from built-in types inherit the superclass eq method. For example, the FlowRate class derives from int32:

classdef FlowRate < int32
    enumeration
        Low    (10)
        Medium (50)
        High   (100)
    end
end

The switchEnum function switches on the input argument, which can be a FlowRate enumeration value.

function switchEnum(inpt)
    switch inpt
        case FlowRate.Low:
            disp('Low flow rate')
        case FlowRate.Medium:
            disp('Medium flow rate')
        case FlowRate.High:
            disp('High flow rate')
    end
end
case 10
disp('Flow = 10 cfm')
end

Call switchEnum with an enumerated value:

switchEnum(FlowRate.Medium)

Flow = 50 cfm
Object Operations

A fundamental purpose of objects is to contain data and facilitate ways to manipulate that data. Objects often define their own version of ordinary MATLAB functions that work with the object. For example, you can create a timeseries object and pass the object to plot:

```matlab
ts = timeseries(rand(100,1),.01:.01:1,'Name','Data1'); plot(ts)
```
However, MATLAB does not call the standard plot function. MATLAB calls the timeseries plot method, which can extract the data from the timeseries object and create a customized graph.

**Help on Objects**

Suppose you use an audioplayer object to play audio with MATLAB. To do this, load audio data into MATLAB and create an audioplayer:

```matlab
load('handel','Fs','y')
chorus = audioplayer(y,Fs);
```
The `audioplayer` function creates an object that you access using the object variable `chorus`. MATLAB stores the audio source and other information in the object properties.

Here are the properties and values for the `chorus` instance of the `audioplayer`:

```
chorus
chorus =

Click the link to get the documentation on `audioplayer` objects.
```

```
audioplayer with properties:

    SampleRate: 8192
    BitsPerSample: 16
    NumberOfChannels: 1
    DeviceID: -1
    CurrentSample: 1
    TotalSamples: 73113
    Running: 'off'
    StartFcn: []
    StopFcn: []
    TimerFcn: []
    TimerPeriod: 0.0500
    Tag: ''
    UserData: []
    Type: 'audioplayer'
```

The object’s documentation discusses the purpose of the object and describes the properties and methods that you use when working with objects of that class.

You can also list the methods to see what operations you can perform. Pass the object to the `methods` function to see the list:

```
methods(chorus)
```

```
Methods for class audioplayer:

    audioplayer    getdisp    pause    resume    stop
```
To play the audio, use the `play` method:

```matlab
play(chorus)
```

## Functions to Test Objects

These functions provide logical tests, which are useful when using objects in ordinary functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isa</code></td>
<td>Determine whether an argument is an object of specific class.</td>
</tr>
<tr>
<td><code>isequal</code></td>
<td>Determine if two objects are equal, which means both objects are of the same class and size and their corresponding property values are equal.</td>
</tr>
<tr>
<td><code>a == b (eq)</code></td>
<td>Determine if handle variable <code>a</code> refers to the same object as handle variable <code>b</code>.</td>
</tr>
<tr>
<td><code>isobject</code></td>
<td>Determine whether input is a MATLAB object</td>
</tr>
</tbody>
</table>

## Functions to Query Class Components

These functions provide information about object class components.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class</code></td>
<td>Return class of object.</td>
</tr>
<tr>
<td><code>enumeration</code></td>
<td>Display class enumeration members and names.</td>
</tr>
<tr>
<td><code>events</code></td>
<td>List event names defined by the class.</td>
</tr>
<tr>
<td><code>methods</code></td>
<td>List methods implemented by the class.</td>
</tr>
<tr>
<td><code>methodview</code></td>
<td>List methods in separate window.</td>
</tr>
<tr>
<td><code>properties</code></td>
<td>List class property names.</td>
</tr>
</tbody>
</table>
Use of Editor and Debugger with Classes

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Write Class Code in the Editor” on page 5-48</td>
</tr>
<tr>
<td>“How to Refer to Class Files” on page 5-48</td>
</tr>
<tr>
<td>“How to Debug Class Files” on page 5-49</td>
</tr>
</tbody>
</table>

Write Class Code in the Editor

The MATLAB code editor provides an effective environment for class development. The Code Analyzer, which is built into the editor, check code for problems and provides information on fixing these problems. For information on editor use and features, see “Code Editor”.

How to Refer to Class Files

Define classes in files just like scripts and functions. To use the editor or debugger with a class file, use the full class name. For example, suppose the file for a class, myclass.m is in the following location:

```
+PackFld1/+PackFld2/@myclass/myclass.m
```

To open myclass.m in the MATLAB editor, you could reference the file using dot-separated package names:

```
edit PackFld1.PackFld2.myclass
```

You could also use path notation:

```
edit +PackFld1/+PackFld2/@myclass/myclass
```

If myclass.m is not in a class folder, then enter:

```
edit +PackFld1/+PackFld2/myclass
```

To refer to functions inside a package folder, use dot or path separators:

```
edit PackFld1.PackFld2.packFunction
edit +PackFld1/+PackFld2/packFunction
```

To refer to a function defined in its own file inside of a class folder, use:
edit +PackFld1/+PackFld2/@myclass/myMethod

**How to Debug Class Files**

For debugging, `dbstop` accepts any of the file specifications used by the `edit` command.

See “Automatic Updates for Modified Classes” on page 5-50 for information about clearing class.

**See Also**

**Related Examples**

- “MATLAB Code Analyzer Warnings” on page 5-34
Automatic Updates for Modified Classes

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“When MATLAB Loads Class Definitions” on page 5-50</td>
</tr>
<tr>
<td>“Consequences of Automatic Update” on page 5-51</td>
</tr>
<tr>
<td>“What Happens When Class Definitions Change” on page 5-51</td>
</tr>
<tr>
<td>“Actions That Do Not Trigger Updates” on page 5-52</td>
</tr>
<tr>
<td>“Multiple Updates to Class Definitions” on page 5-52</td>
</tr>
<tr>
<td>“Object Validity with Deleted Class File” on page 5-52</td>
</tr>
<tr>
<td>“When Updates Are Not Possible” on page 5-52</td>
</tr>
<tr>
<td>“Potential Consequences of Class Updates” on page 5-53</td>
</tr>
<tr>
<td>“Updates to Class Attributes” on page 5-53</td>
</tr>
<tr>
<td>“Updates to Property Definitions” on page 5-54</td>
</tr>
<tr>
<td>“Updates to Method Definitions” on page 5-55</td>
</tr>
<tr>
<td>“Updates to Event Definitions” on page 5-56</td>
</tr>
</tbody>
</table>

**When MATLAB Loads Class Definitions**

MATLAB loads a class definition:

- The first time the class is referenced, such as creating an instance, accessing a constant property, or calling a static method of the class.
- Whenever the definition of a loaded class changes and MATLAB returns to the command prompt.
- When you change the MATLAB path and cause a different definition of the class to be used.
- Whenever you access the class metadata.

MATLAB allows only one definition for a class to exist at any time. Therefore, MATLAB attempts to update all existing objects of a class automatically to conform to the new class definition. You do not need to call `clear classes` to remove existing objects when you change their defining class.
Using an editor other than the MATLAB editor or using MATLAB Online™ can result in delays to automatic updating.

**Consequences of Automatic Update**

MATLAB follows a set of basic rules when updating existing objects. An automatic update can result in:

- Existing objects being updated to the new class definition.
- An error if MATLAB cannot convert the objects to the new class definition or if there is an error in the class definition itself.

Here is an example of what happens when you create an instance of a concrete class edit the class definition to make the class abstract.

```matlab
a = MyClass;
% Edit MyClass to make it Abstract
a
```

```
Error using MyClass/display
Cannot update object because the class 'MyClass' is now abstract.
```

MATLAB does not update metaclass instances when you change the definition of a class. You must get new metaclass data after updating a class definition.

**What Happens When Class Definitions Change**

MATLAB updates existing objects when a class definition changes, including the following situations:

- Value change to handle — Existing objects become independent handles referring to different objects.
- Enumeration member added — Existing objects preserve the enumeration members they had previously, even if the underlying values have changed.
- Enumeration member removed — Existing objects that are not using the removed member have the same enumeration members that they had previously. Existing objects that use the removed member replace the removed member with the default member of the enumeration.
• Enumeration block removed — Enumeration members are taken out of use.
• Superclass definition changed — Changes applied to all subclasses in the hierarchy of that superclass.
• Superclass added or removed — Change of superclass applied to all existing objects.

**Actions That Do Not Trigger Updates**

These actions do not update existing objects:

• Calling the `class` function on an out-of-date object
• Assigning an out-of-date object to a variable
• Calling a method that does not access class data
• Changing property validation in the class definition ("Validate Property Values" on page 8-24)

Objects do not update until referenced in a way that exposes the change, such as invoking the object display or assigning to a property.

**Multiple Updates to Class Definitions**

Updates do not occur incrementally. Updates conform to the latest version of the class.

**Object Validity with Deleted Class File**

Deleting a class definition file does not make instances of that class invalid. However, you cannot call methods on existing objects of that class.

**When Updates Are Not Possible**

Some class updates result in an invalid class definition. In these cases, objects do not update until the error is resolved:

• Adding a superclass can result in a property or method being defined twice.
• Changing a superclass to be Sealed when objects of one of its subclasses exists results in an invalid subclass definition.
Some class updates cause situations in which MATLAB cannot update existing objects to conform to a modified class definition. These cases result in errors until you delete the objects:

- Adding an enumeration block to a non-enumeration class
- Redefining a class to be abstract
- Removing a class from a heterogeneous hierarchy that results in there being no default object to replace existing objects in a heterogeneous array
- Updating a class to restrict array formation behavior, such as overloading array indexing and concatenation.
- Inheriting a `subsref`, `subsasgn`, `cat`, `vertcat`, or `horzcat` method
- Redefining a handle class to be a value class.

**Potential Consequences of Class Updates**

- Following an update, existing objects can be incompatible with the new class definition. For example, a newly added property can require execution of the constructor to be valid.
- Removing or renaming properties can lose the data held in the property. For example, if a property holds the only reference to another object and you remove that property from the class, the MATLAB deletes the object because there are no longer any references to it.
- Removing a class from a heterogeneous class hierarchy can result in invalid heterogeneous array elements. In this case, the default object for the heterogeneous hierarchy replaces these array elements.

**Updates to Class Attributes**

Changing class attributes can change existing object behavior or make the objects invalid. MATLAB returns an error when you access the invalid objects.

<table>
<thead>
<tr>
<th>Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make Abstract = true</td>
<td>Accessing existing objects returns an error.</td>
</tr>
<tr>
<td>Change AllowedSubclasses</td>
<td>Newly created objects can inherit from different superclasses than existing objects.</td>
</tr>
<tr>
<td>Change</td>
<td>Effect</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Change ConstructOnLoad</td>
<td>Loading classes obeys the current value of ConstructOnLoad.</td>
</tr>
<tr>
<td>Change HandleCompatible</td>
<td>Newly created objects can have different class hierarchy than existing objects.</td>
</tr>
<tr>
<td>Change Hidden</td>
<td>Appearance of class in list of superclasses and access by help function can change.</td>
</tr>
<tr>
<td>Change InferiorClasses</td>
<td>Method dispatching for existing objects can change.</td>
</tr>
<tr>
<td>Make Sealed = true</td>
<td>Existing subclass objects return errors when accessed.</td>
</tr>
</tbody>
</table>

### Updates to Property Definitions

When you change the definition of class properties, MATLAB applies the changes to existing objects of the class.

<table>
<thead>
<tr>
<th>Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add property</td>
<td>Adds the new property to existing objects of the class.</td>
</tr>
<tr>
<td></td>
<td>Sets the property values to the default value (which is [] if the class definition does not specify a default).</td>
</tr>
<tr>
<td>Remove property</td>
<td>Removes the property from existing objects of the class.</td>
</tr>
<tr>
<td></td>
<td>Attempts to access the removed property fail.</td>
</tr>
<tr>
<td>Change property default value</td>
<td>Does not apply the new default value to existing objects of the class.</td>
</tr>
<tr>
<td>Move property between</td>
<td>Does not apply different default value when property definition moves between superclass and subclass.</td>
</tr>
<tr>
<td>subclass and superclass</td>
<td></td>
</tr>
</tbody>
</table>


Change property attribute value

<table>
<thead>
<tr>
<th>Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change property attribute value</td>
<td>Applies changes to existing objects of the class.</td>
</tr>
<tr>
<td></td>
<td>Some cases require transitional steps:</td>
</tr>
<tr>
<td></td>
<td>• Abstract — Existing objects of a class that becomes abstract cannot be updated. Delete these objects.</td>
</tr>
<tr>
<td></td>
<td>• Access — Changes to the public, protected, or private property access settings affect access to existing objects.</td>
</tr>
<tr>
<td></td>
<td>Changes to the access lists do not change existing objects. However, if you add classes to the access list, instances of those classes have access to this property. If you remove classes from the access list, objects of those classes no longer have access to this property.</td>
</tr>
<tr>
<td></td>
<td>• Dependent — If changed to true, existing objects no longer store property values. If you want to query the property value, add a property get method for the property.</td>
</tr>
<tr>
<td></td>
<td>• Transient — If changed to true, objects already saved, reload this property value. If changed to false, objects already saved reload this property using the default value.</td>
</tr>
</tbody>
</table>

**Updates to Method Definitions**

When you change the definition of class methods, MATLAB changes the affected class member in existing objects as follows.

<table>
<thead>
<tr>
<th>Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add method</td>
<td>You can call the new method on existing objects of the class.</td>
</tr>
<tr>
<td>Modify method</td>
<td>Modifications are available to existing objects.</td>
</tr>
<tr>
<td>Remove method</td>
<td>You can no longer call deleted method on existing objects.</td>
</tr>
</tbody>
</table>
### Change

<table>
<thead>
<tr>
<th>Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change method attribute value</td>
<td>Apply changes to existing objects of the class. Some cases require transitional steps:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Abstract</strong> — Existing objects of a class that becomes abstract cannot be updated. Delete these objects.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Access</strong> — Changes to method <code>public</code>, <code>protected</code>, or <code>private</code> access settings affect access to existing objects.</td>
</tr>
<tr>
<td></td>
<td>Changes to the access lists do not change existing instances. However, if you add classes to the access list, instances of those classes have access to this method. If you remove classes from the access list, objects of those classes no longer have access to this method.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Sealed</strong> — If changed to <code>true</code> and existing subclasses already have defined the method, MATLAB returns an error because the new class definition cannot be applied to existing subclasses.</td>
</tr>
</tbody>
</table>

### Updates to Event Definitions

<table>
<thead>
<tr>
<th>Change</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add event</td>
<td>Existing objects of the class support the new event.</td>
</tr>
<tr>
<td>Change event name</td>
<td>New event name is visible to existing objects of the class. MATLAB:</td>
</tr>
<tr>
<td></td>
<td>• Does not update existing metaclass objects</td>
</tr>
<tr>
<td></td>
<td>• Does update newly acquired metaclass objects</td>
</tr>
<tr>
<td></td>
<td>• Does not update listeners to use new event name</td>
</tr>
<tr>
<td>Remove event</td>
<td>Existing objects no longer support deleted event.</td>
</tr>
<tr>
<td>Change</td>
<td>Effect</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Change event attribute value</td>
<td>Apply changes to existing objects of the class.</td>
</tr>
<tr>
<td></td>
<td>Some cases require transitional steps:</td>
</tr>
<tr>
<td></td>
<td>• <strong>ListenAccess</strong> — Changes to event public, protected, or private listen access settings affect access to existing objects.</td>
</tr>
<tr>
<td></td>
<td>Changes to the access list do not change existing objects. However, if you add classes to the access list, objects of those classes can create listeners for this event. If you remove classes from the access list, objects of those classes are not allowed to create listeners for this event.</td>
</tr>
<tr>
<td></td>
<td>• <strong>NotifyAccess</strong> — Changes to event public, protected, or private notify access settings affect access to existing objects.</td>
</tr>
<tr>
<td></td>
<td>Changes to the access list do not change existing objects. However, if you add classes to the access list, instances of those classes can trigger this event. If you remove classes, objects of those classes are not able to trigger this event.</td>
</tr>
</tbody>
</table>

**See Also**

**Related Examples**

- “Use of Editor and Debugger with Classes” on page 5-48
Compatibility with Previous Versions

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“New Class-Definition Syntax Introduced with MATLAB Software Version 7.6” on page 5-58</td>
</tr>
<tr>
<td>“Changes to Class Constructors” on page 5-59</td>
</tr>
<tr>
<td>“New Features Introduced with Version 7.6” on page 5-60</td>
</tr>
<tr>
<td>“Examples of Old and New” on page 5-60</td>
</tr>
</tbody>
</table>

New Class-Definition Syntax Introduced with MATLAB Software Version 7.6

MATLAB software Version 7.6 introduces a new syntax for defining classes. This new syntax includes:

- The `classdef` keyword begins a block of class-definitions code. An `end` statement terminates the class definition.
- Within the `classdef` code block, `properties`, `methods`, and `events` are also keywords delineating where you define the respective class members.

Cannot Mix Class Hierarchy

It is not possible to create class hierarchies that mix classes defined before Version 7.6 and current class definitions that use `classdef`. Therefore, you cannot subclass an old class to create a version of the new class.

Only One “@” Class Folder per Class

For classes defined using the new `classdef` keyword, a class folder shadows all class folders that occur after it on the MATLAB path. Classes defined in class folders must locate all class files in that single folder. However, classes defined in class folders continue to take precedence over functions and scripts having the same name, even those functions and scripts that come before them on the path.

Private Methods

You do not need to define private folders in class folders in Version 7.6. You can set the method's `Access` attribute to `private` instead.
Changes to Class Constructors

Class constructor methods have two major differences. Class constructors:

- Do not use the class function.
- Must call the superclass constructor only if you want to pass arguments to its constructor. Otherwise, no call to the superclass constructor is necessary.

Example of Old and New Syntax

Compare the following two Stock constructor methods. The Stock class is a subclass of the Asset class, which requires arguments passed to its constructor.

Constructor Function Before Version 7.6

```matlab
function s = Stock(description,num_shares,share_price)
    s.NumShares = num_shares;
    s.SharePrice = share_price;
% Construct Asset object
    a = Asset(description,'stock',share_price*num_shares);
% Use the class function to define the stock object
    s = class(s,'Stock',a);
end
```

Write the same Stock class constructor as shown here. Define the inheritance on the classdef line and define the constructor within a methods block.

Constructor Function for Version 7.6

```matlab
classdef Stock < Asset
    ...
    methods
        function s = Stock(description,num_shares,share_price)
% Call superclass constructor to pass arguments
            s = s@Asset(description,'stock',share_price*num_shares);
            s.NumShares = num_shares;
            s.SharePrice = share_price;
        end % End of function
    end % End of methods block
end % End of classdef block
```
New Features Introduced with Version 7.6

• Properties: “Ways to Use Properties” on page 8-2
• Handle classes: “Comparison of Handle and Value Classes” on page 7-2
• Events and listeners: “Event and Listener Concepts” on page 11-14
• Class member attributes: “Attribute Specification” on page 5-22
• Abstract classes: “Abstract Classes” on page 12-91
• Dynamic properties: “Dynamic Properties — Adding Properties to an Instance” on page 8-68
• Ability to subclass MATLAB built-in classes: “Design Subclass Constructors” on page 12-9
• Packages for scoping functions and classes: “Packages Create Namespaces” on page 6-25. MATLAB does not support packages for classes created before MATLAB Version 7.6 (that is, classes that do not use classdef).
• The JIT/Accelerator supports objects defined only by classes using classdef.

Examples of Old and New

The MATLAB Version 7.6 implementation of classes uses different syntax from previous releases. However, classes written in previous versions continue to work. Most of the code you use to implement the methods is likely to remain the same, except where you take advantage of new features.

The following sections reimplement examples using the latest syntax. The original MATLAB Classes and Objects documentation implemented these same examples and provide a comparison of old and new syntax.

“Class Design for Polynomials” on page 19-2

“A Class Hierarchy for Heterogeneous Arrays” on page 20-2
Comparison of MATLAB and Other OO Languages

In this section...

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Some Differences from C++ and Java Code”</td>
<td>5-61</td>
</tr>
<tr>
<td>“Object Modification”</td>
<td>5-62</td>
</tr>
<tr>
<td>“Static Properties”</td>
<td>5-66</td>
</tr>
<tr>
<td>“Common Object-Oriented Techniques”</td>
<td>5-67</td>
</tr>
</tbody>
</table>

Some Differences from C++ and Java Code

The MATLAB programming language differs from other object-oriented languages, such as C++ or Java in some important ways.

Public Properties

Unlike fields in C++ or the Java language, you can use MATLAB properties to define a public interface separate from the implementation of data storage. You can provide public access to properties because you can define set and get access methods that execute automatically when assigning or querying property values. For example, the following statement:

```matlab
myobj.Material = 'plastic';
```

assigns the char vector plastic to the Material property of myobj. Before making the actual assignment, myobj executes a method called `set.Material` (assuming the class of myobj defines this method), which can perform any necessary operations. See “Property Access Methods” on page 8-49 for more information on property access methods.

You can also control access to properties by setting attributes, which enable public, protected, or private access. See “Property Attributes” on page 8-9 for a full list of property attributes.

No Implicit Parameters

In some languages, one object parameter to a method is always implicit. In MATLAB, objects are explicit parameters to the methods that act on them.
Dispatching

In MATLAB classes, method dispatching is not based on method signature, as it is in C++ and Java code. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.

However, if the class of an argument is superior to the class of the other arguments, MATLAB dispatches to the method of the superior argument, regardless of its position within the argument list.

See “Class Precedence” on page 6-23 for more information.

Calling Superclass Method

• In C++, you call a superclass method using the scoping operator: 
  `superclass::method`

• In Java code, you use: `superclass.method`

The equivalent MATLAB operation is `method@superclass`.

Other Differences

In MATLAB classes, there is no equivalent to C++ templates or Java generics. However, MATLAB is weakly typed and it is possible to write functions and classes that work with different types of data.

MATLAB classes do not support overloading functions using different signatures for the same function name.

Object Modification

MATLAB classes can define public properties, which you can modify by explicitly assigning values to those properties on a given instance of the class. However, only classes derived from the `handle` class exhibit reference behavior. Modifying a property value on an instance of a value classes (classes not derived from `handle`), changes the value only within the context in which the modification is made.

The sections that follow describe this behavior in more detail.
**Objects Passed to Functions**

MATLAB passes all variables by value. When you pass an object to a function, MATLAB copies the value from the caller into the parameter variable in the called function.

However, MATLAB supports two kinds of classes that behave differently when copied:

- **Handle classes** — a handle class instance variable refers to an object. A copy of a handle class instance variable refers to the same object as the original variable. If a function modifies a handle object passed as an input argument, the modification affects the object referenced by both the original and copied handles.

- **Value classes** — the property data in an instance of a value class are independent of the property data in copies of that instance (although, a value class property could contain a handle). A function can modify a value object that is passed as an input argument, but this modification does not affect the original object.

See “Comparison of Handle and Value Classes” on page 7-2 for more information on the behavior and use of both kinds of classes.

**Passing Value Objects**

When you pass a value object to a function, the function creates a local copy of the argument variable. The function can modify only the copy. If you want to modify the original object, return the modified object and assign it to the original variable name. For example, consider the value class, `SimpleClass`:

```matlab
classdef SimpleClass
    properties
        Color
    end
    methods
        function obj = SimpleClass(c)
            if nargin > 0
                obj.Color = c;
            end
        end
    end
end
```

Create an instance of `SimpleClass`, assigning a value of `red` to its `Color` property:

```matlab
obj = SimpleClass('red');
```

Pass the object to the function `g`, which assigns `blue` to the `Color` property:
function y = g(x)
    x.Color = 'blue';
    y = x;
end

y = g(obj);

The function $g$ modifies its copy of the input object and returns that copy, but does not change the original object.

```matlab
y.Color
ans =
    blue
```

```matlab
obj.Color
ans =
    red
```

If the function $g$ did not return a value, the modification of the object `Color` property would have occurred only on the copy of `obj` within the function workspace. This copy would have gone out of scope when the function execution ended.

Overwriting the original variable actually replaces it with a new object:

```matlab
obj = g(obj);
```

**Passing Handle Objects**

When you pass a handle to a function, the function makes a copy of the handle variable, just like when passing a value object. However, because a copy of a handle object refers to the same object as the original handle, the function can modify the object without having to return the modified object.

For example, suppose you modify the `SimpleClass` class definition to make a class derived from the `handle` class:

```matlab
classdef SimpleHandleClass < handle
    properties
        Color
    end
    methods
```

5-64
function obj = SimpleHandleClass(c)
    if nargin > 0
        obj.Color = c;
    end
end
end

Create an instance of SimpleHandleClass, assigning a value of red to its Color property:

obj = SimpleHandleClass('red');

Pass the object to the function g, which assigns blue to the Color property:

y = g(obj);

The function g sets the Color property of the object referred to by both the returned handle and the original handle:

y.Color
ans =
blue

obj.Color
ans =
blue

The variables y and obj refer to the same object:

y.Color = 'yellow';
obj.Color
ans =
yellow

The function g modified the object referred to by the input argument (obj) and returned a handle to that object in y.
**MATLAB Passes Handles by Value**

A handle variable is a reference to an object. MATLAB passes this reference by value.

Handles do not behave like references in C++. If you pass an object handle to a function and that function assigns a different object to that handle variable, the variable in the caller is not affected. For example, suppose you define a function g2:

```matlab
function y = g2(x)
    x = SimpleHandleClass('green');
    y = x;
end
```

Pass a handle object to g2:

```matlab
obj = SimpleHandleClass('red');
y = g2(obj);
y.Color
ans =
green
```

```matlab
obj.Color
ans =
red
```

The function overwrites the handle passed in as an argument, but does not overwrite the object referred to by the handle. The original handle `obj` still references the original object.

**Static Properties**

In MATLAB, classes can define constant properties, but not "static" properties in the sense of other languages like C++. You cannot change constant properties from the initial value specified in the class definition.

MATLAB has long-standing rules that variables always take precedence over the names of functions and classes. Assignment statements introduce a variable if one does not exist.

Expressions of this form
Introduce a new variable, \( A \), that is a struct containing a field \( B \) whose value is \( C \). If \( A.B = C \) could refer to a static property of class \( A \), then class \( A \) would take precedence over variable \( A \).

This behavior would be a significant incompatibility with prior releases of MATLAB. For example, the introduction of a class named \( A \) on the MATLAB path could change the meaning of an assignment statement like \( A.B = C \) inside a .m code file.

In other languages, classes rarely use static data, except as private data within the class or as public constants. In MATLAB, you can use constant properties the same way you use public final static fields in Java. To use data that is internal to a class in MATLAB, create persistent variables in private or protected methods or local functions used privately by the class.

Avoid static data in MATLAB. If a class has static data, using the same class in multiple applications causes conflicts among applications. Conflicts are less of an issue in some other languages. These languages compile applications into executables that run in different processes. Each process has its own copy of the class static data. MATLAB, frequently runs many different applications in the same process and environment with a single copy of each class.

For ways to define and use static data in MATLAB, see “Static Data” on page 4-2.

**Common Object-Oriented Techniques**

This table provides links to sections that discuss object-oriented techniques commonly used by other object-oriented languages.

<table>
<thead>
<tr>
<th>Technique</th>
<th>How to Use in MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator overloading</td>
<td>“Operator Overloading” on page 17-47</td>
</tr>
<tr>
<td>Multiple inheritance</td>
<td>“Subclassing Multiple Classes” on page 12-22</td>
</tr>
<tr>
<td>Subclassing</td>
<td>“Design Subclass Constructors” on page 12-9</td>
</tr>
<tr>
<td>Destructor</td>
<td>“Handle Class Destructor” on page 7-16</td>
</tr>
<tr>
<td>Data member scoping</td>
<td>“Property Attributes” on page 8-9</td>
</tr>
<tr>
<td>Packages (scoping classes)</td>
<td>“Packages Create Namespaces” on page 6-25</td>
</tr>
<tr>
<td>Technique</td>
<td>How to Use in MATLAB</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Named constants</td>
<td>See “Define Class Properties with Constant Values” on page 15-2 and “Named Values” on page 14-2</td>
</tr>
<tr>
<td>Enumerations</td>
<td>“Define Enumeration Classes” on page 14-5</td>
</tr>
<tr>
<td>Static methods</td>
<td>“Static Methods” on page 9-31</td>
</tr>
<tr>
<td>Static properties</td>
<td>Not supported. See persistent variables. For the equivalent of Java static final or C++ static const properties, use Constant properties. See “Define Class Properties with Constant Values” on page 15-2 For mutable static data, see “Static Data” on page 4-2</td>
</tr>
<tr>
<td>Constructor</td>
<td>“Class Constructor Methods” on page 9-21</td>
</tr>
<tr>
<td>Copy constructor</td>
<td>No direct equivalent</td>
</tr>
<tr>
<td>Reference/reference classes</td>
<td>“Comparison of Handle and Value Classes” on page 7-2</td>
</tr>
<tr>
<td>Abstract class/Interface</td>
<td>“Abstract Classes” on page 12-91</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>“Object Lifecycle” on page 7-20</td>
</tr>
<tr>
<td>Instance properties</td>
<td>“Dynamic Properties — Adding Properties to an Instance” on page 8-68</td>
</tr>
<tr>
<td>Importing classes</td>
<td>“Import Classes” on page 6-30</td>
</tr>
<tr>
<td>Events and Listeners</td>
<td>“Event and Listener Concepts” on page 11-14</td>
</tr>
</tbody>
</table>
Defining and Organizing Classes

- “User-Defined Classes” on page 6-2
- “Class Attributes” on page 6-6
- “Evaluation of Expressions in Class Definitions” on page 6-10
- “Folders Containing Class Definitions” on page 6-17
- “Class Precedence” on page 6-23
- “Packages Create Namespaces” on page 6-25
- “Import Classes“ on page 6-30
User-Defined Classes

In this section...

“What Is a Class Definition” on page 6-2
“Attributes for Class Members” on page 6-2
“Kinds of Classes” on page 6-3
“Constructing Objects” on page 6-3
“Class Hierarchies” on page 6-3
“classdef Syntax” on page 6-3
“Class Code” on page 6-4

What Is a Class Definition

A MATLAB class definition is a template whose purpose is to provide a description of all
the elements that are common to all instances of the class. Class members are the
properties, methods, and events that define the class.

Define MATLAB classes in code blocks, with subblocks delineating the definitions of
various class members. For syntax information on these blocks, see “Class Components”
on page 5-5.

Attributes for Class Members

Attributes modify the behavior of classes and the members defined in the class-definition
block. For example, you can specify that methods are static or that properties are private.
The following sections describe these attributes:

- “Class Attributes” on page 6-6
- “Method Attributes” on page 9-5
- “Property Attributes” on page 8-9
- “Event Attributes” on page 11-19

Class definitions can provide information, such as inheritance relationships or the names
of class members without actually constructing the class. See “Class Metadata” on page
16-2.
See “Specifying Attributes” on page 6-8 for more on attribute syntax.

**Kinds of Classes**

There are two kinds of MATLAB classes—handle classes and value classes.

- Value classes represent independent values. Value objects contain the object data and do not share this data with copies of the object. MATLAB numeric types are value classes. Values objects passed to and modified by functions must return a modified object to the caller.

- Handle classes create objects that reference the object data. Copies of the instance variable refer to the same object. Handle objects passed to and modified by functions affect the object in the caller’s workspace without returning the object.

For more information, see “Comparison of Handle and Value Classes” on page 7-2.

**Constructing Objects**

For information on class constructors, see “Class Constructor Methods” on page 9-21.

For information on creating arrays of objects, see “Construct Object Arrays” on page 10-2.

**Class Hierarchies**

For more information on how to define class hierarchies, see “Hierarchies of Classes — Concepts” on page 12-2.

**classdef Syntax**

Class definitions are blocks of code that are delineated by the `classdef` keyword at the beginning and the `end` keyword at the end. Files can contain only one class definition.

The following diagram shows the syntax of a `classdef` block. Only comments and blank lines can precede the `classdef` keyword.
**Class Code**

Here is a simple class definition with one property and a constructor method that sets the value of the property when there is an input argument supplied.

```matlab
classdef MyClass
    properties
        Prop
    end
    methods
        function obj = MyClass(val)
            if nargin > 0
                obj.Prop = val;
            end
        end
    end
end
```

To create an object of `MyClass`, save the class definition in a `.m` file having the same name as the class and call the constructor with any necessary arguments:

```matlab
d = datestr(now);
o = MyClass(d);
```
Use dot notation to access the property value:

```matlab
o.Prop
ans =
```

```
10-Nov-2005 10:38:14
```

The constructor should support a no argument syntax so MATLAB can create default objects. For more information, see “No Input Argument Constructor Requirement” on page 9-25.

For more information on the components of a class definition, see “Class Components” on page 5-5

**See Also**

**Related Examples**

- “Create a Simple Class” on page 2-2
- “Developing Classes — Typical Workflow” on page 3-8
- “Class to Represent Structured Data” on page 3-19
Class Attributes

Specifying Class Attributes

All classes support the attributes listed in the following table. Attributes enable you to modify the behavior of class. Attribute values apply to the class defined within the classdef block.

classdef (Attribute1 = value1, Attribute2 = value2,...) ClassName

end

For more information on attribute syntax, see “Attribute Specification” on page 5-22.
### Class Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>logical (default = false)</td>
<td>If specified as true, this class is an abstract class (cannot be instantiated). See “Abstract Classes” on page 12-91 for more information.</td>
</tr>
</tbody>
</table>
| AllowedSubclasses    | meta.class object or cell array of meta.class objects | List classes that can subclass this class. Specify subclasses as meta.class objects in the form:  
• A single meta.class object  
• A cell array of meta.class objects. An empty cell array, {}, is the same as a Sealed class (no subclasses).  
Specify meta.class objects using the ?ClassName syntax only.  
See “Specify Allowed Subclasses” on page 12-25 for more information. |
| ConstructOnLoad      | logical (default = false)           | If true, MATLAB calls the class constructor when loading an object from a MAT-file. Therefore, implement the constructor so it can be called with no arguments without producing an error.  
See “Initialize Objects When Loading” on page 13-30 for more information. |
<p>| HandleCompatible     | logical (default = false)           | If specified as true, this class can be used as a superclass for handle classes. All handle classes are HandleCompatible by definition. See “Handle Compatible Classes” on page 12-39 for more information. |
| Hidden               | logical (default = false)           | If true, this class does not appear in the output of the superclasses or help functions.                                                      |</p>
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InferiorClasses</td>
<td>meta.class object or cell array of meta.class objects</td>
<td>Use this attribute to establish a precedence relationship among classes. Specify a cell array of meta.class objects using the ? operator. The fundamental classes are always inferior to user-defined classes and do not show up in this list. See “Class Precedence” on page 6-23 and “Dominant Argument in Overloaded Graphics Functions” on page 9-48.</td>
</tr>
<tr>
<td>Sealed</td>
<td>logical (default = false)</td>
<td>If true, this class cannot be subclassed.</td>
</tr>
</tbody>
</table>

### Specifying Attributes

Attributes are specified for class members in the `classdef`, `properties`, `methods`, and `events` definition blocks. The particular attribute setting applies to all members defined within that particular block. You can use multiple `properties` definition blocks to apply different attribute setting to different properties.

### Superclass Attributes Are Not Inherited

Class attributes are not inherited, so superclass attributes do not affect subclasses.

### Attribute Syntax

Specify class attribute values in parentheses, separating each attribute name/attribute value pair with a comma. The attribute list always follows the `classdef` or class member keyword, as shown:

```matlab
classdef (attribute-name = expression, ...) ClassName
    properties (attribute-name = expression, ...)
    ...
    end
    methods (attribute-name = expression, ...)
    ...
    end
    events (attribute-name = expression, ...)
    ...
    end
end
```
See Also

More About
• “Expressions in Attribute Specifications” on page 6-11
Evaluation of Expressions in Class Definitions

In this section...

| "Why Use Expressions" on page 6-10 |
| "Where to Use Expressions in Class Definitions" on page 6-10 |
| "How MATLAB Evaluates Expressions" on page 6-13 |
| "When MATLAB Evaluates Expressions" on page 6-13 |
| "Expression Evaluation in Handle and Value Classes" on page 6-14 |

Why Use Expressions

An expression used in a class definition can be any valid MATLAB statement that evaluates to a single array. Use expressions to define property default values and in attribute specifications. Expressions are useful to derive values in terms of other values. For example, suppose you want to define a constant property with the full precision value of \(2\pi\). You can assign the property the value returned by the expression \(2\pi\). MATLAB evaluates the function the when first loading the class.

For information on assign property default values and attribute values, see the following topics:

- "Property Definition" on page 8-17
- "Attribute Specification" on page 5-22

Where to Use Expressions in Class Definitions

Here are some examples of expressions used in a class definition:

```matlab
classdef MyClass
  % Some attributes are set to logical values
  properties (Constant = true)
    CnstProp = 2*pi
  end
properties
  % Static method of this class
  Prop1 = MyClass.setupAccount
  % Constant property from this class
  Prop2 = MyClass.CnstProp
end
```
% Function that returns a value
Prop3 = datestr(now)

% A class constructor
Prop4 = AccountManager

end

methods (Static)
  function accNum = setupAccount
    accNum = randi(9,[1,12]);
  end
end
end

MATLAB does not call property set methods when assigning the result of default value expressions to properties. (See “Property Access Methods” on page 8-49 for information about these special methods.)

Enumerations that derived from MATLAB types can use expression to assign a value:

classdef FlowRate < int32
  enumeration
    Low    (10)
    Medium (FlowRate.Low*5)
    High   (FlowRate.Low*10)
  end
end

MATLAB evaluates these expressions only once when enumeration members are first accessed.

**Expressions in Attribute Specifications**

For attributes values that are logical true or false, class definitions can specify attribute values using expressions. For example, this assignment makes MyClass sealed (cannot be subclassed) for versions of MATLAB before R2014b (verLessThan)

classdef (Sealed = verLessThan('matlab','8.4')) MyClass

The expression on the right side of the equal sign (=) must evaluate to true or false. You cannot use any definitions from the class file in this expression, including any constant properties, static methods, and local functions.

While you can use conditional expression to set attribute values, doing so can cause the class definition to change based on external conditions. Ensure that this behavior is consistent with your class design.
Note The AllowedSubclasses and the InferiorClasses attributes require an explicit specification of a cell array of meta.class objects as their values. You cannot use expressions to return these values.

See “Attribute Specification” on page 5-22 for more information on attribute syntax.

Expressions That Specify Default Property Values

Property definitions allow you to specify default values for properties using any expression that has no reference to variables. For example, VectorAngle defines a constant property (Rad2Deg) and uses it in an expression that defines the default value of another property (Angle). The default value expression also uses a static method (getAngle) defined by the class:

```matlab
classdef VectorAngle
    properties (Constant)
        Rad2Deg = 180/pi
    end
    properties
        Angle = VectorAngle.Rad2Deg*VectorAngle.getAngle([1 0],[0 1])
    end
    methods
        function obj = VectorAngle(vx,vy)
            obj.Angle = VectorAngle.getAngle(vx,vy);
        end
    end
    methods (Static)
        function r = getAngle(vx,vy)
            % Calculate angle between 2D vectors
            cr = vx(1)*vy(1) + vx(2)*vy(2)/sqrt(vx(1)^2 + vx(2)^2) * ...
                sqrt(vy(1)^2 + vy(2)^2);
            r = acos(cr);
        end
    end
end
```

You cannot use the input variables to the constructor to define the default value of the Angle property. For example, this definition for the Angle property is not valid:

```matlab
properties
    Angle = VectorAngle.Rad2Deg*VectorAngle.getAngle(vx,vy)
end
```
Attempting to create an instance causes an error:

```matlab
a = VectorAngle([1,0],[0,1])
```

Error using VectorAngle
Unable to update the class 'VectorAngle' because the new definition contains an error:
Undefined function or variable 'vx'.

**Expressions in Class Methods**

Expression in class methods execute like expressions in any function. MATLAB evaluates an expression within the function workspace when the method executes. Therefore, expressions used in class methods are not considered part of the class definition and are not discussed in this section.

**How MATLAB Evaluates Expressions**

MATLAB evaluates the expressions used in the class definition without any workspace. Therefore, these expressions cannot reference variables of any kind.

MATLAB evaluates expressions in the context of the class file, so these expressions can access any functions, static methods, and constant properties of other classes that are on your path at the time MATLAB initializes the class. Expressions defining property default values can access constant properties defined in their own class.

**When MATLAB Evaluates Expressions**

MATLAB evaluates the expressions in class definitions only when initializing the class. Initialization occurs before the first use of the class.

After initialization, the values returned by these expressions are part of the class definition and are constant for all instances of the class. Each instance of the class uses the results of the initial evaluation of the expressions without re-evaluation.

If you clear a class, then MATLAB reinitializes the class by reevaluating the expressions that are part of the class definition. (see “Automatic Updates for Modified Classes” on page 5-50)
Expression Evaluation in Handle and Value Classes

The following example shows how value and handle object behave when assigned to properties as default values. Suppose that you have the following classes.

Expressions in Value Classes

The ClassExp class has a property that contains a ContClass object:

classdef ContClass
    properties
        % Assign current date and time
        TimeProp = datestr(now)
    end
end
classdef ClassExp
    properties
        ObjProp = ContClass
    end
end

When you first use the ClassExp class, MATLAB creates an instance of the ContClass class. MATLAB initializes both classes at this time. All instances of ClassExp include a copy of this same instance of ContClass.

    a = ClassExp;
    a.ObjProp.TimeProp

ans =

08-Oct-2003 17:16:08

The TimeProp property of the ContClass object contains the date and time when MATLAB initialized the class. Creating additional instances of the ClassExp class shows that the date string has not changed:

    b = ClassExp;
    b.ObjProp.TimeProp

ans =

08-Oct-2003 17:16:08
Because this example uses a value class for the contained object, each instance of the ClassExp has its own copy of the object. For example, suppose that you change the value of the TimeProp property on the object contained by ClassExp object b:

```matlab
b.ObjProp.TimeProp = datestr(now)
ans = 08-Oct-2003 17:22:49
```

The copy of the object contained by object a is unchanged:

```matlab
a.ObjProp.TimeProp
ans = 08-Oct-2003 17:16:08
```

**Expressions in Handle Classes**

Now consider the behavior if the contained object is a handle object:

```matlab
classdef ContClass < handle
    properties
        TimeProp = datestr(now)
    end
end
```

Creating two instances of the ClassExp class shows that MATLAB created an object when it initialized the ContClass. MATLAB used a copy of the object’s handle for each instance of the ClassExp class. Therefore, there is one ContClass object and the ObjProp property of each ClassExp object contains a copy of its handle.

Create an instance of the ClassExp class and note the time of creation:

```matlab
a = ClassExp;
a.ObjProp.TimeProp
ans = 08-Oct-2003 17:46:01
```

Create a second instance of the ClassExp class. The ObjProp contains the handle of the same object:
b = ClassExp;
b.ObjProp.TimeProp

ans =

08-Oct-2003 17:46:01

Reassign the value of the contained object TimeProp property:

b.ObjProp.TimeProp = datestr(now);
b.ObjProp.TimeProp

ans =

08-Oct-2003 17:47:34

The ObjProp property of object b contains a handle to the same object as the ObjProp property of object a. The value of the TimeProp property has changed on this object as well:

a.ObjProp.TimeProp

ans =

08-Oct-2003 17:47:34

**See Also**

**More About**

- “Comparison of Handle and Value Classes” on page 7-2
Folders Containing Class Definitions

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Class Definitions On the Path” on page 6-17</td>
</tr>
<tr>
<td>“Class and Path Folders” on page 6-17</td>
</tr>
<tr>
<td>“Using Path Folders” on page 6-17</td>
</tr>
<tr>
<td>“Using Class Folders” on page 6-18</td>
</tr>
<tr>
<td>“Functions in Private Folders Within Class Folders” on page 6-19</td>
</tr>
<tr>
<td>“Class Precedence and MATLAB Path” on page 6-19</td>
</tr>
<tr>
<td>“Changing Path to Update Class Definition” on page 6-21</td>
</tr>
</tbody>
</table>

Class Definitions On the Path

To call a class method, the class definition must be on the MATLAB path, as described in the next sections.

Class and Path Folders

There are two types of folders that can contain class definition files.

- Path folders — The folder is on the MATLAB path and the folder name does not begin with an @ character. Use this type of folder when you want multiple classes in one folder. However, the entire class definition must be contained in one .m file.

- Class folders — The folder name begins with an @ character followed by the class name. The folder is not on the MATLAB path, but its parent folder is on the path. Use this type of folder when you want to use multiple files for one class definition.

See the path function for information about the MATLAB path.

Using Path Folders

The class definition files in the folders are on the MATLAB path. Therefore, class definitions placed in path folders behave like any ordinary function with respect to precedence—the first occurrence of a name on the MATLAB path takes precedence over all subsequent occurrences of the same name.
The name of each class definition file must match the name of the class that is specified with the `classdef` keyword. Using a path folder eliminates the need to create a separate class folder for each class. However, the entire class definition, including all methods, must be contained within a single file.

Suppose that you have three classes defined in a single folder:

```plaintext
.../path_folder/MyClass1.m
.../path_folder/MyClass2.m
.../path_folder/MyClass3.m
```

To use these classes, add `path_folder` to your MATLAB path:

```matlab
addpath path_folder
```

### Using Class Folders

A class folder name always begins with the `@` character followed by the class name for the folder name. A class folder must be contained in a path folder, but the class folder is not on the MATLAB path. Place the class definition file inside the class folder, which also can contain separate method files. The class definition file must have the same name as the class folder (without the `@` character). The class definition (beginning with the `classdef` keyword) must appear in the file before any other code (white space and comments do not constitute code).

```plaintext
.../parent_folder/@MyClass/MyClass.m
.../parent_folder/@MyClass/myMethod1.m
.../parent_folder/@MyClass/myMethod2.m
```

Define only one class per folder. All files must have a `.m` extension.

Use a class folder when you want to use more than one file for your class definition. MATLAB treats any `.m` file in the class folder as a method of the class. Class files provide the advantage that MATLAB can explicitly identify any file in the folder as a method of that class. For more information, see “Changing Path to Update Class Definition” on page 6-21.

The base name of each file must be a valid MATLAB function name. Valid function names begin with an alphabetic character and can contain letters, numbers, or underscores. For more information, see “Methods in Separate Files” on page 9-11.
Functions in Private Folders Within Class Folders

Private folders contain functions that are accessible only from functions defined in folders immediately above the private folder. Any functions defined in a private folder inside a class folder become methods of the class that have private access.

If a class folder contains a private folder, only the class defined in that folder can access functions defined in the private folder. Subclasses do not have access to superclass private functions. For more information on private folders, see “Private Functions”.

If you want a subclass to have access to the private functions of the superclass, define the functions as protected methods of the superclass. Specify the methods with the Access attribute set to protected.

Dispatching to Methods in Private Folders

If a class defines functions in a private folder that is in a class folder, then MATLAB follows these precedence rules when dispatching to the private functions vs. a local function defined in the classdef file:

• Using dot notation (obj.methodName), a function in a private folder takes precedence over a local function defined in the classdef file.
• Using function notation (methodName(obj)), a local function defined in the classdef file takes precedence over the function in the private folder.

No Class Definitions in Private Folders

You cannot put class definitions (classdef file) in private folders because doing so would not meet the requirements for class or path folders.

Class Precedence and MATLAB Path

When there are multiple class definition files with the same name, the file location on the MATLAB path determines the file precedence. All class definition files before a class on the path take precedence, whether or not the definitions are contained in a class folder. The class takes precedence over all class definition files occurring later on the path.

For example, consider a path with the following folders, containing the files indicated.
Here is the logic that MATLAB applies to determine which version of Foo to call:

Class `fldr1/Foo.m` takes precedence over the class `fldr3/@Foo` because:

- `fldr1/Foo.m` is before `fldr3/@Foo` on the path

Class `fldr3/@Foo` takes precedence over function `fldr2/Foo.m` because:

- `fldr3/@Foo` is a class in a class folder
- `fldr2/Foo.m` is not a class
- Classes in class folders take precedence over functions

Function `fldr2/Foo.m` takes precedence over class `fldr5/Foo.m` because:

- `fldr2/Foo.m` comes before class `fldr5/Foo.m` on the path
- `fldr5/Foo.m` is not in a class folder
- Classes that are not defined in class folders obey the path order with respect to functions.

Class `fldr3/@Foo` takes precedence over class `fldr4/@Foo` because:

- The method `bar` is not recognized as part of the Foo class defined in `fldr3/@Foo`.

If `fldr3/@Foo/Foo.m` does not contain a `classdef` keyword (that is, it is a MATLAB class prior to Version 7.6), then `fldr4/@Foo/bar.m` becomes a method of the Foo class defined in `fldr3/@Foo`.

**Previous Behavior of Classes Defined in Class Folders**

In MATLAB Versions 5 through 7, class folders do not shadow other class folders having the same name, but residing in later path folders. Instead, the class uses the combination
of methods from all class folders having the same name to define the class. This behavior is no longer supported.

For backward compatibility, classes defined in class folders always take precedence over functions and scripts having the same name. This precedence applies to functions and scripts that come before these classes on the path.

### Changing Path to Update Class Definition

Changing your MATLAB path can change the class definition file for a class (see `path`). However, for classes that are defined in path folders (that is, not in class @ folders), you must clear the class before MATLAB recognizes the new folder as the current class definition.

#### Class Definitions in Class Folders

Suppose that you define two versions of a class named Foo in two folders, fldA and fldB.

```
fldA/+FooPkg/@Foo/Foo.m
fldB/+FooPkg/@Foo/Foo.m
```

Add folder fldA to the top of the path.

```
addpath fldA
```

Create an instance of class FooPkg.Foo. MATLAB uses fldA/+FooPkg/@Foo/Foo.m as the class definition.

```
a = FooPkg.Foo;
```

Change the current folder to fldB.

```
cd fldB
```

The current folder is always first on the path. Therefore, MATLAB finds fldB/+FooPkg/@Foo/Foo.m as the definition for class FooPkg.Foo.

```
b = FooPkg.Foo;
```

MATLAB automatically updates the existing instance, a, to use the new class definition in fldB.
Class Definitions in Path Folders

Suppose that you define two versions of a class named Foo in two folders, fldA and fldB, but do not use a class folder.

fldA/+FooPkg/Foo.m
fldB/+FooPkg/Foo.m

Add folder fldA to the top of the path.

addpath fldA

Create an instance of class FooPkg.Foo. MATLAB uses fldA/+FooPkg/@Foo/Foo.m as the class definition.

a = FooPkg.Foo;

Change the current folder to fldB.

cd fldB

The current folder is effectively the top of the path. However, MATLAB does not identify fldB/+FooPkg/Foo.m as the definition for class FooPkg.Foo. MATLAB continues to use the original class definition until you clear the class.

To use the definition of FooPkg.Foo in foldB, clear FooPkg.Foo.

clear FooPkg.Foo

MATLAB automatically updates the existing objects to conform to the class definition in fldB. In most cases, clearing instance variables is unnecessary.

See Also

More About

- "Packages Create Namespaces" on page 6-25
- "Automatic Updates for Modified Classes" on page 5-50
Class Precedence

In this section...

“Use of Class Precedence” on page 6-23
“Why Mark Classes as Inferior” on page 6-23
“InferiorClasses Attribute” on page 6-23

Use of Class Precedence

MATLAB uses class precedence to determine which method to call when multiple classes have the same method. You can specifying the relative precedence of user-defined classes with the class InferiorClasses attribute.

The material presented in this topic builds on an understanding of the following information:

• “Class Metadata” on page 16-2
• “Attribute Specification” on page 5-22

Why Mark Classes as Inferior

When more than one class defines methods with the same name or when classes overload functions, MATLAB determines which method or function to call based on the dominant argument. Here is how MATLAB determines the dominant argument:

1. Determine the dominant argument based on the class of arguments.
2. If there is a dominant argument, call the method of the dominant class.
3. If arguments are of equal precedence, use the leftmost argument as the dominant argument.
4. If the class of the dominant argument does not define a method with the name of the called function, call the first function on the path with that name.

InferiorClasses Attribute

Specify the relative precedence of user-defined classes using the class InferiorClasses attribute. To specify classes that are inferior to the class you are defining, assign a cell array of class meta.class objects to this attribute.
For example, the following `classdef` declares that `MyClass` is dominant over `ClassName1` and `ClassName2`.

```matlab
classdef (InferiorClasses = {?ClassName1,?ClassName2}) MyClass
    ...
end
```

The `?` operator combined with a class name creates a `meta.class` object. See `metaclass`.

MATLAB built-in classes are always inferior to user-defined classes and should not be used in this list.

The built-in classes include: `double`, `single`, `char`, `logical`, `int64`, `uint64`, `int32`, `uint32`, `int16`, `uint16`, `int8`, `uint8`, `cell`, `struct`, and `function_handle`.

**Dominant Class**

MATLAB uses class dominance when evaluating expressions involving objects of more than one class. The dominant class determines:

- Which class method to call when more than one class defines methods with the same names.
- The class of arrays that are formed by combining objects of different classes, assuming MATLAB can convert the inferior objects to the dominant class.

**No Attribute Inheritance**

Subclasses do not inherit a superclass `InferiorClasses` attribute. Only classes specified in the subclass `InferiorClasses` attribute are inferior to subclass objects.

**See Also**

**More About**

- “Class Precedence and MATLAB Path” on page 6-19
- “Dominant Argument in Overloaded Graphics Functions” on page 9-48
Packages Create Namespaces

In this section...

“Package Folders” on page 6-25
“Internal Packages” on page 6-26
“Referencing Package Members Within Packages” on page 6-26
“Referencing Package Members from Outside the Package” on page 6-27
“Packages and the MATLAB Path” on page 6-28

Package Folders

Packages are special folders that can contain class folders, function, and class definition files, and other packages. The names of classes and functions are scoped to the package folder. A package is a namespace within which names must be unique. Function and class names must be unique only within the package. Using a package provides a means to organize classes and functions. Packages also enable you to reuse the names of classes and functions in different packages.

Note Packages are not supported for classes created prior to MATLAB Version 7.6 (that is, classes that do not use classdef).

Package folders always begin with the + character. For example,

```
+mypack
+mypack/pkfcn.m  % a package function
+mypack/@myClass % class folder in a package
```

The parent of the top-level package folder must be on the MATLAB path.

Listing the Contents of a Package

List the contents of a package using the help command:

```
help event
```

Contents of event:

- EventData - event.EVENTDATA Base class for event data
- PropertyEvent - event.PROPERTYEVENT Event data for object property events
You can also use the `what` command:

```
what event
```

Classes in directory Y:xxx\matlab\toolbox\matlab\lang\+event

```
EventData       PropertyEvent  listener       proplistener
```

### Internal Packages

MathWorks® reserves the use of packages named `internal` for utility functions used by internal MATLAB code. Functions that belong to an `internal` package are intended for MathWorks use only. Using functions or classes that belong to an `internal` package is discouraged. These functions and classes are not guaranteed to work in a consistent manner from one release to the next. Any of these functions and classes might be removed from the MATLAB software in any subsequent release without notice and without documentation in the product release notes.

### Referencing Package Members Within Packages

All references to packages, functions, and classes in the package must use the package name prefix, unless you import the package. (See “Import Classes” on page 6-30.) For example, call this package function:

```
+mypack/pkfcn.m
```

With this syntax:

```
z = mypack.pkfcn(x,y);
```

Definitions do not use the package prefix. For example, the function definition line of the `pkfcn.m` function would include only the function name:

```
function z = pkfcn(x,y)
```

Define a package class with only the class name:

```
classdef myClass
```

but call it with the package prefix:
obj = mypack.myClass(arg1,arg2,...);

Calling class methods does not require the package name because you have an object of the class. You can use dot or function notation:

obj.myMethod(arg)
myMethod(obj,arg)

A static method requires the full class name, which includes the package name:

mypack.myClass.stMethod(arg)

**Referencing Package Members from Outside the Package**

Functions, classes, and other packages contained in a package are scoped to that package. To reference any of the package members, prefix the package name to the member name, separated by a dot. For example, the following statement creates an instance of MyClass, which is contained in mypack package.

obj = mypack.MyClass;

**Accessing Class Members — Various Scenarios**

This section shows you how to access various package members from outside a package. Suppose that you have a package mypack with the following contents:

+mypack
+mypack/myfcn.m
+mypack/@MyFirstClass
+mypack/@MyFirstClass/myFcn.m
+mypack/@MyFirstClass/otherFcn.m
+mypack/@MyFirstClass/MyFirstClass.m
+mypack/@MySecondClass
+mypack/@MySecondClass/MySecondClass.m
+mypack/+mysubpack
+mypack/+mysubpack/myFcn.m

Invoke the myFcn function in mypack:

mypack.myFcn(arg)

Create an instance of each class in mypack:
obj1 = mypack.MyFirstClass;
obj2 = mypack.MySecondClass(arg);

Invoke the myFcn function that is in the package mysubpack:

mypack.mysubpack.myFcn(arg1,arg2);

If mypack.MyFirstClass has a method called myFcn, call it like any method call on an object:

obj = mypack.MyFirstClass;
myFcn(obj,arg);

If mypack.MyFirstClass has a property called MyProp, assign it using dot notation and the object:

obj = mypack.MyFirstClass;
obj.MyProp = x;

**Packages and the MATLAB Path**

You cannot add package folders to the MATLAB path, but you must add the package parent folder to the MATLAB path. Package members are not accessible if the package parent folder is not on the MATLAB path, even if the package folder is the current folder. Making the package folder the current folder is not sufficient to add the package parent folder to the path.

Package members remain scoped to the package. Always refer to the package members using the package name. Alternatively, import the package into the function in which you call the package member, see “Import Classes” on page 6-30.

Package folders do not shadow other package folders that are positioned later on the path, unlike classes, which do shadow other classes. If two or more packages have the same name, MATLAB treats them all as one package. If redundantly named packages in different path folders define the same function name, then MATLAB finds only one of these functions.

**Resolving Redundant Names**

Suppose a package and a class have the same name. For example:

fldr_1/+foo
fldr_2/@foo/foo.m
A call to `which foo` returns the path to the executable class constructor:

```
>> which foo
fldr_2/@foo/foo.m
```

A function and a package can have the same name. However, a package name by itself is not an identifier. Therefore, if a redundant name occurs alone, it identifies the function. Executing a package name alone returns an error.

**Package Functions vs. Static Methods**

In cases where a package and a class have the same name, a static method takes precedence over a package function. For example, path folder `fldrA` contains a package function and path folder `fldrB` contains a class static method:

```
fldrA/+foo/bar.m % bar is a function in package foo
fldrB/@foo/bar.m % bar is a static method of class foo
```

A call to `which foo.bar` returns the path to the static method:

```
which foo.bar
fldrB/@foo/bar.m
```

In cases where the same path folder contains both package and class folders with the same name, the class static method takes precedence over the package function.

```
fldr/@foo/bar.m % bar is a static method of class foo
fldr/+foo/bar.m % bar is a function in package foo
```

A call to `which foo.bar` returns the path to the static method:

```
which foo.bar
fldr/@foo/bar.m
```

**See Also**

**More About**

- “Folders Containing Class Definitions” on page 6-17
- “Class Precedence” on page 6-23
Import Classes

In this section...

“Syntax for Importing Classes” on page 6-30
“Import Package Functions” on page 6-30
“Package Function and Class Method Name Conflict” on page 6-31
“Clearing Import List” on page 6-31

Syntax for Importing Classes

Import classes into a function to simplify access to class members. For example, suppose that there is a package that contains several classes and you need to use only one of these classes in your function, or even just a static method from that class. Use the import command to simplify code. Once you have imported the class, you do not need to reference the package name:

```matlab
function myFunc
    import pkg.cls1
    obj = cls1(arg,...); % call cls1 constructor
    obj.Prop = cls1.staticMethod(arg,...); % call cls1 static method
end
```

Import all classes in a package using the syntax pkg.*:

```matlab
function myFunc
    import pkg.*
    obj1 = cls1(arg,...); % call pkg.cls1 constructor
    obj2 = cls2(arg,...); % call pkg.cls2 constructor
    a = pkgFunction(); % call package function named pkgFunction
end
```

Import Package Functions

Use import to import package functions:

```matlab
function myFunc
    import pkg.pkfcn
    pkfcn(arg,...); % call imported package function
end
```
Package Function and Class Method Name Conflict

Avoid importing an entire package using the * wildcard syntax. Doing so imports an unspecified set of names into the local scope. For example, suppose that you have the following folder organization:

```
+pkg/timedata.m % package function
+pkg/@MyClass/MyClass.m % class definition file
+pkg/@MyClass/timedata.m % class method
```

Import the package and call `timedata` on an instance of `MyClass`:

```matlab
import pkg.*
myobj = pkg.MyClass;
timedata(myobj)
```

A call to `timedata` finds the package function, not the class method because MATLAB applies the `import` and finds `pkg.timedata` first. Do not use a package in cases where you have name conflicts and plan to import the package.

Clearing Import List

You cannot clear the import list from a function workspace. To clear the base workspace only, use:

```
clear import
```

See Also

More About

- "Packages Create Namespaces" on page 6-25
Value or Handle Class — Which to Use

- “Comparison of Handle and Value Classes” on page 7-2
- “Which Kind of Class to Use” on page 7-11
- “The Handle Superclass” on page 7-13
- “Handle Class Destructor” on page 7-16
- “Find Handle Objects and Properties” on page 7-25
- “Implement Set/Get Interface for Properties” on page 7-27
- “Implement Copy for Handle Classes” on page 7-34
Comparison of Handle and Value Classes

In this section...

| “Basic Difference” on page 7-2 |
| “Behavior of MATLAB Built-In Classes” on page 7-3 |
| “User-Defined Value Classes” on page 7-4 |
| “User-Defined Handle Classes” on page 7-5 |
| “Determining Equality of Objects” on page 7-8 |
| “Functionality Supported by Handle Classes” on page 7-10 |

Basic Difference

A value class constructor returns an object that is associated with the variable to which it is assigned. If you reassign this variable, MATLAB creates an independent copy of the original object. If you pass this variable to a function to modifying it, the function must return the modified object as an output argument. For information on value-class behavior, see “Avoid Unnecessary Copies of Data”.

A handle class constructor returns a handle object that is a reference to the object created. You can assign the handle object to multiple variables or pass it to functions without causing MATLAB to make a copy of the original object. A function that modifies a handle object passed as an input argument does not need to return the object.

All handle classes are derived from the abstract handle class.

Create a Value Class

By default, MATLAB classes are value classes. The following definition creates a value class named MyValueClass:

classdef MyValueClass
    ...
end

Create a Handle Class

To create a handle class, derive the class from the handle class.
classdef MyHandleClass < handle
    ...
end

**Behavior of MATLAB Built-In Classes**

MATLAB fundamental classes are value classes (numeric, logical, char, cell, struct, and function handle). For example, if you create an object of the class `int32` and make a copy of this object, the result is two independent objects. When you change the value of `a`, the value of `b` does not change. This behavior is typical of classes that represent values.

```matlab
a = int32(7);
b = a;
a = a^4;
b
```

`7`

MATLAB graphics objects are implemented as handle objects because they represent visual elements. For example, create a graphics line object and copy its handle to another variable. Both variables refer to the same line object.

```matlab
x = 1:10; y = sin(x);
l1 = line(x,y);
l2 = l1;

Set the properties of the line object using either copy of the handle.

```matlab
set(l2,'Color','red')
set(l1,'Color','green')

get(l2,'Color')
```

ans =

```
    0     1     0
```

Calling the `delete` function on the `l2` handle destroys the line object. If you attempt to set the `Color` property on the line `l1`, the `set` function returns an error.

```matlab
delete(l2)
set(l1,'Color','blue')
```

Error using `matlab.graphics.primitive.Line/set` Invalid or deleted object.
If you delete the object by deleting any one of the existing handles, all copies are now invalid because you deleted the single object to which all handles refer.

Deleting a handle object is not the same as clearing the handle variable. In the graphics object hierarchy, the parent of the object holds a reference to the object. For example, the parent axes hold a reference to the line object referred to by $l1$ and $l2$. If you clear both variables from the workspace, the object still exists.

For more information on the behavior of handle objects, see “Handle Object Behavior” on page 1-12.

**User-Defined Value Classes**

MATLAB associates objects of value classes with the variables to which you assign the object. When you copy a value object to another variable or pass a value object to a function, MATLAB creates an independent copy of the object and all the data contained by the object. The new object is independent of changes to the original object. Value objects behave like MATLAB numeric and `struct` classes. Each property behaves essentially like a MATLAB array.

Value objects are always associated with one workspace or temporary variable. Value objects go out of scope when their variable goes out of scope or is cleared. There are no references to value objects, only copies that are independent objects.

**Value Object Behavior**

Here is a value class that stores a value in its `Number` property. The default property value is the number 1.

```matlab
classdef NumValue
    properties
        Number = 1
    end
end
```

Create a `NumValue` object assigned to the variable `a`.

```matlab
a = NumValue
```

```matlab
a =
    NumValue with properties:
```
Assign the value of `a` to another variable, `b`.

```plaintext
b = a
```

The variables `a` and `b` are independent. Changing the value of the `Number` property of `a` does not affect the `Number` property of `b`.

```plaintext
a.Number = 7
```

```plaintext
b
```

```plaintext
b =
```

**Modifying Value Objects in Functions**

When you pass a value object to a function, MATLAB creates a copy of that object in the function workspace. Because copies of value objects are independent, the function does not modify the object in the caller’s workspace. Therefore, functions that modify value objects must return the modified object to be reassigned in the caller’s workspace.

For more information, see “Object Modification” on page 5-62.

**User-Defined Handle Classes**

Instances of classes that derive from the `handle` class are references to the underlying object data. When you copy a handle object, MATLAB copies the handle, but does not
copy the data stored in the object properties. The copy refers to the same object as the original handle. If you change a property value on the original object, the copied handle references the same change.

**Handle Object Behavior**

Here is a handle class that stores a value in its `Number` property. The default property value is the number 1.

```matlab
classdef NumHandle < handle
    properties
        Number = 1
    end
end
```

Create a `NumHandle` objects assigned to the variable `a`.

```matlab
a = NumHandle
```

```matlab
a =
    NumHandle with properties:
        Number: 1
```

Assign the value of `a` to another variable, `b`.

```matlab
b = a
```

```matlab
b =
    NumHandle with properties:
        Number: 1
```

The variables `a` and `b` refer to the same underlying object. Changing the value of the `Number` property of `a` also changes the `Number` property of `b`. That is, `a` and `b` refer to the same object.

```matlab
a.Number = 7
```

```matlab
a =
    NumHandle with properties:
        Number: 7
```
Number: 7

b

b =

NumHandle with properties:

Number: 7

Modifying Handle Objects in Functions

When you pass a handle object to a function, MATLAB creates a copy of the handle in the function workspace. Because copies of handles reference the same underlying object, functions that modify the handle object effectively modify the object in the caller’s workspace as well. Therefore, it is not necessary for functions that modify handle objects passed as input arguments to return the modified object to the caller.

For more information, see “Object Modification” on page 5-62.

Deleting Handles

You can destroy handle objects by explicitly calling the handle delete method. Deleting the handle of a handle class object makes all handles invalid. For example:

```
a = NumHandle;
b = a;
delete(a)
b.Number
```

Invalid or deleted object.

Calling delete on a handle object invokes the destructor function or functions for that object. See “Handle Class Destructor” on page 7-16 for more information.

Initialize Properties to Contain Handle Objects

For information on the differences between initializing properties to default values in the properties block and initializing properties from within the constructor, see “Initialize Property Values” on page 8-18 and “Initialize Arrays of Handle Objects” on page 10-11.
Determining Equality of Objects

Equality for value objects means that the objects are of the same class and have the same state.

Equality for handle objects means that the handle variables refer to the same object. You also can identify handle variables that refer to different objects of the same class that have the same state.

Equality of Value Objects

To determine if value objects are the same size and their contents are of equal value, use isequal. For example, use the previously defined NumValue class to create two instances and test for equality:

```matlab
a = NumValue;
b = NumValue;
isequal(a,b)
```

```matlab
ans =

1
```

a and b are independent and therefore are not the same object. However each represents the same value.

If you change the value represented by a value object, the objects are no longer equal.

```matlab
a = NumValue;
b = NumValue;
b.Number = 7;
isequal(a,b)
```

```matlab
ans =

0
```

Value classes do not have a default eq method to implement the == operation.

Equality of Handle Objects

Handle objects inherit an eq method from the handle base class. You can use == and isequal to test for two different relationships among handle objects:
• The handles refer to the same object: == and isequal return true.
• The handles refer to objects of the same class that have the same values, but are not
  the same objects — only isequal returns true.

Use the previously defined NumHandle class to create an object and copy the handle.

\[
a = \text{NumHandle};
b = a;
\]

Test for equality using == and isequal.

\[
a == b
\]
\[
\text{ans = 1}
\]
\[
\text{isequal(a,b)}
\]
\[
\text{ans = 1}
\]

Create two instances of the NumHandle class using the default values.

\[
a = \text{NumHandle};
b = \text{NumHandle};
\]

Determine if a and b refer to the same object.

\[
a == b
\]
\[
\text{ans = 0}
\]

Determine if a and b have the same values.

\[
\text{isequal(a,b)}
\]
\[
\text{ans = 1}
\]
Functionality Supported by Handle Classes

Deriving from the handle class enables your class to:

- Inherit several useful methods ("Handle Class Methods" on page 7-13)
- Define events and listeners ("Events and Listeners Syntax" on page 11-22)
- Define dynamic properties ("Dynamic Properties — Adding Properties to an Instance" on page 8-68)
- Implement set and get methods ("Implement Set/Get Interface for Properties" on page 7-27)
- Customize copy behavior ("Implement Copy for Handle Classes" on page 7-34)

See "The Handle Superclass" on page 7-13 for more information on the handle class and its methods.

See Also

Related Examples

- "Which Kind of Class to Use" on page 7-11
- "Implement Copy for Handle Classes" on page 7-34
- "Handle Object Behavior" on page 1-12
Which Kind of Class to Use

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Examples of Value and Handle Classes” on page 7-11</td>
</tr>
<tr>
<td>“When to Use Value Classes” on page 7-11</td>
</tr>
<tr>
<td>“When to Use Handle Classes” on page 7-12</td>
</tr>
</tbody>
</table>

Examples of Value and Handle Classes

Handle and value classes are useful in different situations. For example, value classes enable you to create new array classes that have the same behavior as MATLAB numeric classes.

“Class Design for Polynomials” on page 19-2 and “Class to Represent Structured Data” on page 3-19 provides examples of value classes.

Handle classes enable you to create objects that more than one function or object can share. Handle objects allow more complex interactions among objects because they allow objects to reference each other.

“Class to Implement Linked Lists” on page 3-31 and “Developing Classes — Typical Workflow” on page 3-8 provides examples of a handle class.

When to Use Value Classes

Value class objects behave like normal MATLAB variables. A typical use of value classes is to define data structures. For example, suppose that you want to define a class to represent polynomials. This class can define a property to contain a list of coefficients for the polynomial. It can implement methods that enable you to perform various operations on the polynomial object. For example, implement addition and multiplication without converting the object to another class.

A value class is suitable because you can copy a polynomial object and have two objects that are identical representations of the same polynomial. For an example of value classes, see “Subclasses of MATLAB Built-In Types” on page 12-50.

For information on MATLAB pass-by-value semantics, see “Avoid Unnecessary Copies of Data”. 

7-11
When to Use Handle Classes

Handle objects are useful in specialized circumstances where an object represents a physical object such as a graph or an external device rather than a mathematical object like a number or matrix. Handle objects are derivations of the handle class, which provides functionality such as events and listeners, destructor method, and support for dynamic properties.

Use a handle class when:

- No two instances of a class can have the same state, making it impossible to have exact copies. For example:
  - A copy of a graphics object (such as a line) has a different position in its parents list of children than the object from which it was copied. Therefore, the two objects are not identical.
  - Nodes in lists or trees having specific connectivity to other nodes—no 2 nodes can have the same connectivity.
- The class represents physical and unique objects like serial ports and printers.
- The class represents visible objects like graphics components.
- The class defines events and notifies listeners when an event occurs (`notify` is a handle class method).
- The class creates listeners by calling the `handle` class `addlistener` method.
- The class subclasses the `dynamicprops` class (a subclass of handle) so that instances can define dynamic properties.
- The class subclasses the `matlab.mixin.SetGet` class (a subclass of handle) so that it can implement a graphics object style set/get interface to access property values.
- You want to create a singleton class or a class in which you track the number of instances from within the constructor.
- Instances of a class cannot share state, such as nodes in a linked list.

See Also

Related Examples

- “Handle Compatible Classes” on page 12-39
The Handle Superclass

In this section...

“Building on the Handle Class” on page 7-13
“Handle Class Methods” on page 7-13
“Event and Listener Methods” on page 7-14
“Relational Methods” on page 7-14
“Test Handle Validity” on page 7-14
“When MATLAB Destroys Objects” on page 7-15

Building on the Handle Class

The handle class is an abstract class. Therefore, you cannot create objects of this class directly. Use the handle class as a superclass to implement subclasses that inherit handle behavior. MATLAB defines several classes that derive from the handle class. These classes provide specialized functionality to subclasses.

Specialized Handle Base Classes

To add both handle behavior and specific functionality to your class, derive your class from these handle classes:

- `matlab.mixin.SetGet` — Provides set and get methods to access property values.
- `dynamicprops` — Enables you to define properties that are associated with an object, but not the class in general.
- `matlab.mixin.Copyable` Provides a copy method that you can customize for your class.

For information on how to define subclasses, see “Design Subclass Constructors” on page 12-9.

Handle Class Methods

When you derive a class from the handle class, the subclass inherits methods that enable you to work more effectively with handle objects.

List the methods of a class by passing the class name to the methods function:
methods('handle')

Methods for class handle:

addlistener  findobj      gt           lt
delete       findprop     isvalid      ne
eq           ge           le           notify

**Event and Listener Methods**

For information on how to use the `notify` and `addlistener` methods, see “Events and Listeners Syntax” on page 11-22.

**Relational Methods**

TF = eq(H1,H2)
TF = ne(H1,H2)
TF = lt(H1,H2)
TF = le(H1,H2)
TF = gt(H1,H2)
TF = ge(H1,H2)

The handle class overloads these functions to support equality tests and sorting on handles. For each pair of input arrays, these functions return a logical array of the same size. Each element is an element-wise equality or comparison test result. The input arrays must be the same size or one (or both) can be scalar. The method performs scalar expansion as required. For more information on handle class relational methods, see `relationaloperators`.

**Test Handle Validity**

Use the `isvalid` handle class method to determine if a variable is a valid handle object. For example, in the statement:

B = isvalid(H)

B is a logical array in which each element is `true` if, and only if, the corresponding element of H is a valid handle. B is always the same size as H.
When MATLAB Destroys Objects

MATLAB destroys objects in the workspace of a function when the function:

- Reassigns an object variable to a new value
- Does not use an object variable for the remainder of a function
- Function execution ends

When MATLAB destroys an object, it also destroys values stored in the properties of the object. MATLAB frees computer memory associated with the object for use by MATLAB or the operating system.

You do not need to free memory in handle classes. However, there can be other operations that you want to perform when destroying an object. For example, closing a file or shutting down an external program that the object constructor started. Define a delete method in your handle subclass for these purposes.

See “Handle Class Destructor” on page 7-16 for more information.

See Also

Related Examples

- “Comparison of Handle and Value Classes” on page 7-2
Handle Class Destructor

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Basic Knowledge” on page 7-16</td>
</tr>
<tr>
<td>“Syntax of Handle Class Destructor Method” on page 7-16</td>
</tr>
<tr>
<td>“Support Destruction of Partially Constructed Objects” on page 7-17</td>
</tr>
<tr>
<td>“When to Define a Destructor Method” on page 7-19</td>
</tr>
<tr>
<td>“Destructors in Class Hierarchies” on page 7-19</td>
</tr>
<tr>
<td>“Object Lifecycle” on page 7-20</td>
</tr>
<tr>
<td>“Restrict Access to Object Delete Method” on page 7-21</td>
</tr>
<tr>
<td>“Nondestructor Delete Methods” on page 7-22</td>
</tr>
<tr>
<td>“Java Objects Referencing MATLAB Objects” on page 7-22</td>
</tr>
</tbody>
</table>

Basic Knowledge

Terms and Concepts

Class destructor - a method named delete that MATLAB calls implicitly before destroying an object of a handle class. Also, user-defined code can call delete explicitly to destroy an object.

Nondestructor - a method named delete that does not meet the syntax requirements of a valid destructor. Therefore, MATLAB does not call this method implicitly when destroying handle objects. A method named delete in a value class is not a destructor.

“Object Lifecycle” on page 7-20

“Method Attributes” on page 9-5

Syntax of Handle Class Destructor Method

MATLAB calls the destructor of a handle class when destroying objects of the class. MATLAB recognizes a method named delete as the class destructor only if you define delete as an ordinary method with the appropriate syntax.

To be a valid class destructor, the delete method:
• Must define one, scalar input argument, which is an object of the class.
• Must not define output arguments
• Cannot be Sealed, Static, or Abstract

In addition, the delete method should not:
• Throw errors, even if the object is invalid.
• Create new handles to the object being destroyed
• Call methods or access properties of subclasses

MATLAB does not call a noncompliant delete method when destroying objects of the class. A delete method that is not a valid destructor shadows the handle base class delete method. A noncompliant delete method can prevent the destruction of the object by shadowing the handle class delete method.

Declare delete as an ordinary method:

```matlab
methods
  function delete(obj)
      % obj is always scalar
      ...
  end
end
```

**delete Called Element-Wise on Array**

MATLAB calls the delete method separately for each element in an array. Therefore, a delete method is passed only one scalar argument with each invocation.

Calling delete on a deleted handle should not error and can take no action. This design enables delete to work on object arrays containing a mix of valid and invalid objects.

**Support Destruction of Partially Constructed Objects**

Errors that occur while constructing an object can result in a call to delete before the object is fully created. Therefore, class delete methods must be able to work with partially constructed objects.

For example, the PartialObject class delete method determines if the Data property is empty before accessing the data this property contains. If an error occurs while
assigning the constructor argument to the Name property, MATLAB passes the partially constructed object to delete.

classdef PartialObject < handle
    properties
        % Restrict the Name property
        % to a cell array
        Name cell
        Data
    end
    methods
        function h = PartialObject(name)
            if nargin > 0
                h.Name = name;
                h.Data.a = rand(10,1);
            end
        end
        function delete(h)
            % Protect against accessing properties
            % of partially constructed objects
            if ~isempty(h.Data)
                t = h.Data.a;
                disp(t)
            else
                disp('Data is empty')
            end
        end
    end
end

An error occurs if you call the constructor with a char vector, instead of the required cell array:

obj = PartialObject('Test')

MATLAB passes the partially constructed object to the delete method. The constructor did not set the value of the Data property because the error occurred when setting the Name property.

Data is empty
Error setting 'Name' property of 'PartialObject' class:
...
When to Define a Destructor Method

Use a delete method to perform cleanup operations before MATLAB destroys the object. MATLAB calls the delete method reliably, even if execution is interrupted with Ctrl-c or an error.

If an error occurs during the construction of a handle class, MATLAB calls the class destructor on the object along with the destructors for any objects contained in properties and any initialized base classes.

For example, suppose that a method opens a file for writing and you want to close the file in your delete method. The delete method can call fclose on a file identifier that the object stores in its FileID property:

```matlab
function delete(obj)
    fclose(obj.FileID);
end
```

Destructors in Class Hierarchies

If you create a hierarchy of classes, each class can define its own delete method. When destroying an object, MATLAB calls the delete method of each class in the hierarchy. Defining a delete method in a handle subclass does not override the handle class delete method. Subclass delete methods augment the superclass delete methods.

Inheriting a Sealed Delete Method

Classes cannot define a valid destructor that is Sealed. MATLAB returns an error when you attempt to instantiate a class that defines a Sealed delete method.

Normally, declaring a method as Sealed prevents subclasses from overriding that method. However, a Sealed method named delete that is not a valid destructor does not prevent a subclass from defining its own destructor.

For example, if a superclass defines a method named delete that is not a valid destructor, but is Sealed, then subclasses:

- Can define valid destructors (which are always named delete).
- Cannot define methods named delete that are not valid destructors.
Destructors in Heterogeneous Hierarchies

Heterogeneous class hierarchies require that all methods to which heterogeneous arrays are passed must be sealed. However, the rule does not apply to class destructor methods. Because destructor methods cannot be sealed, you can define a valid destructor in a heterogeneous hierarchy that is not sealed, but does function as a destructor.

For information on heterogeneous hierarchies, see “Designing Heterogeneous Class Hierarchies” on page 10-24

Object Lifecycle

MATLAB invokes the delete method when the lifecycle of an object ends. The lifecycle of an object ends when the object is:

- No longer referenced anywhere
- Explicitly deleted by calling delete on the handle

Inside a Function

The lifecycle of an object referenced by a local variable or input argument exists from the time the variable is assigned until the time it is reassigned, cleared, or no longer referenced within that function or in any handle array.

A variable goes out of scope when you explicitly clear it or when its function ends. When a variable goes out of scope and its value belongs to a handle class that defines a delete method, MATLAB calls that method. MATLAB defines no ordering among variables in a function. Do not assume that MATLAB destroys one value before another value when the same function contains multiple values.

Sequence During Handle Object Destruction

MATLAB invokes the delete methods in the following sequence when destroying an object:

1. The delete method for the class of the object
2. The delete method of each superclass class, starting with the immediate superclasses and working up the hierarchy to the most general superclasses

MATLAB invokes the delete methods of superclasses at the same level in the hierarchy in the order specified in the class definition. For example, the following class definition
specifies `supclass1` before `supclass2`. MATLAB calls the `delete` method of `supclass1` before the `delete` method of `supclass2`.

```matlab
classdef myClass < supclass1 & supclass2

After calling each `delete` method, MATLAB destroys the property values belonging exclusively to the class whose method was called. The destruction of property values that contain other handle objects can cause a call the `delete` methods for those objects when there are no other references to those objects.

Superclass `delete` methods cannot call methods or access properties belonging to a subclass.

**Destruction of Objects with Cyclic References**

Consider a set of objects that reference other objects of the set such that the references form a cyclic graph. In this case, MATLAB:

- Destroys the objects if they are referenced only within the cycle
- Does not destroy the objects as long as there is an external reference to any of the objects from a MATLAB variable outside the cycle

MATLAB destroys the objects in the reverse of the order of construction.

**Restrict Access to Object Delete Method**

Destroy handle objects by explicitly calling `delete` on the object:

```matlab
delete(obj)
```

A class can prevent explicit destruction of an object by setting its `delete` method `Access` attribute to `private`. However, a method of the class can call the `private` `delete` method.

If the class `delete` method `Access` attribute is `protected`, only methods of the class and of subclasses can explicitly delete objects of that class.

However, when an object lifecycle ends, MATLAB calls the object’s `delete` method when destroying the object regardless of the method’s `Access` attribute.
Inherited Private Delete Methods

Class destructor behavior differs from the normal behavior of an overridden method. MATLAB executes each delete method of each superclass upon destruction, even if that delete method is not public.

When you explicitly call an object’s delete method, MATLAB checks the delete method Access attribute in the class defining the object, but not in the superclasses of the object. A superclass with a private delete method cannot prevent the destruction of subclass objects.

Declaring a private delete method makes most sense for sealed classes. In the case where classes are not sealed, subclasses can define their own delete methods with public access. MATLAB calls a private superclass delete method as a result of an explicit call to a public subclass delete method.

Nondestructor Delete Methods

A class can implement a method named delete that is not a valid class destructor. MATLAB does not call this method implicitly when destroying an object. In this case, delete behaves like an ordinary method.

For example, if the superclass implements a Sealed method named delete that is not a valid destructor, then MATLAB does not allow subclasses to override this method.

A delete method defined by a value class cannot be a class destructor.

Java Objects Referencing MATLAB Objects

Java does not support the object destructors that MATLAB objects use. Therefore, it is important to manage the lifecycle of all objects used in applications that include both Java and MATLAB objects.

References Can Prevent Destructor Execution

Java objects that hold references to MATLAB objects can prevent deletion of the MATLAB objects. In these cases, MATLAB does not call the handle object delete method even when there is no handle variable referring to that object. To ensure your delete method executes, call delete on the object explicitly before the handle variable goes out of scope.
Problems can occur when you define callbacks for Java objects that reference MATLAB objects.

For example, the `CallbackWithJava` class creates a Java `com.mathworks.jmi.Callback` object and assigns a class method as the callback function. The result is a Java object that has a reference to a handle object via the function-handle callback.

```matlab
classdef CallbackWithJava < handle
    methods
        function obj = CallbackWithJava
            jo = com.mathworks.jmi.Callback;
            set(jo,'DelayedCallback',@obj.cbFunc); % Assign method as callback
            jo.postCallback
        end
        function cbFunc(obj,varargin)
            c = class(obj);
            disp(['Java object callback on class ',c])
        end
        function delete(obj)
            c = class(obj);
            disp(['ML object destructor called for class ',c])
        end
    end
end
```

Suppose that you create a `CallbackWithJava` object from within a function:

```matlab
function testDestructor
    cwj = CallbackWithJava
    ...
end
```

Creating an instance of the `CallbackWithJava` class creates the `com.mathworks.jmi.Callback` object and executes the callback function:

```matlab
testDestructor

cwj =
    CallbackWithJava with no properties.

Java object callback on class CallbackWithJava
```

The handle variable, `cwj`, exists only in the function workspace. However, MATLAB does not call the class `delete` method when the function ends. The `com.mathworks.jmi.Callback` object still exists and holds a reference to the object of the `CallbackWithJava` class, which prevents destruction of the MATLAB object.
clear classes

Warning: Objects of 'CallbackWithJava' class exist. Cannot clear this class or any of its superclasses.

To avoid causing inaccessible objects, call the delete explicitly before losing the handle to the MATLAB object.

function testDestructor
    cwj = CallbackWithJava
    ...
    delete(cwj)
end

**Manage Object Lifecycle in Applications**

MATLAB applications that use Java objects should manage the lifecycle of the objects involved. A typical user-interface application references Java objects from MATLAB objects and creates callbacks on Java objects that reference MATLAB objects.

You can break these cyclic references in various ways:

- Explicitly call delete on the MATLAB objects when they are no longer needed
- Unregister the Java object callbacks that reference MATLAB objects
- Use intermediate handle objects that reference both the Java callbacks and the MATLAB objects.

**See Also**

**More About**

- “Handle Object Behavior” on page 1-12
Find Handle Objects and Properties

In this section...

“Find Handle Objects” on page 7-25
“Find Handle Object Properties” on page 7-25

Find Handle Objects

The `findobj` method enables you to locate handle objects that meet certain conditions.

```matlab
function HM = findobj(H,<conditions>)
```

The `findobj` method returns an array of handles matching the conditions specified. You can use regular expressions with `findobj`. For more information, see `regexp`.

Find Handle Object Properties

The `findprop` method returns the `meta.property` object for the specified object and property.

```matlab
function mp = findprop(h,'PropertyName')
```

The property can also be a dynamic property created by the `addprop` method of the `dynamicprops` class.

Use the returned `meta.property` object to obtain information about the property, such as the settings of any of its attributes. For example, the following statements determine that the setting of the `AccountStatus` property `Dependent` attribute is false.

```matlab
ba = BankAccount(007,50,'open');
mp = findprop(ba,'AccountStatus');
mp.Dependent
```

ans =

0

See Also

`handle`
Related Examples

• “Class Metadata” on page 16-2
Implement Set/Get Interface for Properties

In this section...

“The Standard Set/Get Interface” on page 7-27
“Subclass Syntax” on page 7-27
“Get Method Syntax” on page 7-28
“Set Method Syntax” on page 7-28
“Class Derived from matlab.mixin.SetGet” on page 7-29

The Standard Set/Get Interface

Some MATLAB objects, such as graphics objects, implement an interface based on set and get functions. These functions enable access to multiple properties on arrays of objects in a single function call.

You can add set and get functionality to your class by deriving from one of these classes:

- `matlab.mixin.SetGet` — use when you want support for case-insensitive, partial property name matching. Deriving from `matlab.mixin.SetGet` does not affect the exact property name required by the use of dot notation reference to properties.
- `matlab.mixin.SetGetExactNames` — use when you want to support only case-sensitive full property name matching.

Note The set and get methods referred to in this section are different from property set access and property get access methods. See “Property Access Methods” on page 8-49 for information on property access methods.

Subclass Syntax

Use the abstract class `matlab.mixin.SetGet` or `matlab.mixin.SetGetExactNames` as a superclass:

```matlab
classdef MyClass < matlab.mixin.SetGet
    ...
end
```
Because `matlab.mixin.SetGet` and `matlab.mixin.SetGetExactNames` derive from the handle class, your subclass is also a handle class.

**Get Method Syntax**

The `get` method returns the value of an object property using the object handle and the property name. For example, assume `H` is the handle to an object:

\[ v = \text{get}(H, 'Property\text{Name}'); \]

If you specify an array of handles with a single property name, `get` returns the property value for each object as a cell array of values:

\[ CV = \text{get}(H, 'Property\text{Name}'); \]

The `CV` array is always a column regardless of the shape of `H`.

If you specify a cell array of char vector property names and an array of handles, `get` returns a cell array of property values. Each row in the cell corresponds to an object in the handle array. Each column in the cell corresponds to a property name.

\[ \text{props} = \{'\text{PropertyName1}', '\text{PropertyName2}'\}; \]
\[ CV = \text{get}(H, \text{props}); \]

`get` returns an m-by-n cell array, where \( m = \text{length}(H) \) and \( n = \text{length(props)} \).

If you specify a handle array, but no property names, `get` returns an array of type `struct` in which each structure in the array corresponds to an object in `H`. Each field in each structure corresponds to a property defined by the class of `H`. The value of each field is the value of the corresponding property.

\[ SV = \text{get}(H); \]

If you do not assign an output variable, then `H` must be scalar.

For an example, see “Using `get` with Arrays of Handles” on page 7-31.

**Set Method Syntax**

The `set` method assigns the specified value to the specified property for the object with handle `H`. If `H` is an array of handles, MATLAB assigns the value to the property for each object in the array `H`. 
You can pass a cell array of property names and a cell array of property values to `set`:

```matlab
def = {'PropertyName1', 'PropertyName2'};
data = {{Property1Value, Property2Value}; {Property11Value, Property12Value}; {Property21Value, Property22Value}};
def = dfe = (def, data)
```

If `length(H)` is greater than one, then the property value cell array (`data`) can have values for each property in each object. For example, suppose `length(H)` is 2 (two object handles). You want to assign two property values on each object:

```matlab
def = {'PropertyName1', 'PropertyName2'};
data = {{Property1Value, Property12Value}; {Property21Value, Property22Value}};
def = def = (def, data)
```

The preceding statement is equivalent to the follow two statements:

```matlab
def(H(1), 'PropertyName1', Property11Value, 'PropertyName2', Property12Value)
def(H(2), 'PropertyName1', Property21Value, 'PropertyName2', Property22Value)
```

If you specify a scalar handle, but no property names, `set` returns a struct with one field for each property in the class of `H`. Each field contains an empty cell array.

```matlab
SV = set(h);
```

**Class Derived from matlab.mixin.SetGet**

This sample class defines a set/get interface and illustrates the behavior of the inherited methods:

```matlab
classdef LineType < matlab.mixin.SetGet
    properties
        Style = '-'
        Marker = 'o'
    end
    properties (SetAccess = protected)
        Units = 'points'
    end
    methods
        function obj = LineType(s,m)
            if nargin > 0
                obj.Style = s;
                obj.Marker = m;
            end
        end
end
```
function obj = set.Style(obj,val)
    if ~(strcmpi(val,'-') ||...
        strcmpi(val,'--') ||...
        strcmpi(val,'..'))
        error('Invalid line style ')
    end
    obj.Style = val;
end
function obj = set.Marker(obj,val)
    if ~isstrprop(val,'graphic')
        error('Marker must be a visible character')
    end
    obj.Marker = val;
end
end
end

Create an instance of the class and save its handle:

h = LineType('--','*');

Query the value of any object property using the inherited get method:

get(h,'Marker')

ans =

*

Set the value of any property using the inherited set method:

set(h,'Marker','Q')

**Property Access Methods Called with set and get**

MATLAB calls property access methods (set.Style or set.Marker in the LineType class) when you use the set and get methods.

set(h,'Style','-.-')

Error using LineType>LineType.set.Style
Invalid line style

For more information on property access methods, see “Property Access Methods” on page 8-49
**List All Properties**

Return a struct containing object properties and their current values using get:

```matlab
h = LineType('--','*');
SV = get(h)
```

```matlab
SV =
    Style: '--'
    Marker: '*'
    Units: 'points'
```

Return a struct containing the properties that have public SetAccess using set:

```matlab
S = set(h)
```

```matlab
S =
    Style: {}
    Marker: {}
```

The `LineType` class defines the `Units` property with SetAccess = protected. Therefore, `S = set(h)` does not create a field for `Units` in `S`.

`set` cannot return possible values for properties that have nonpublic set access.

**Using get with Arrays of Handles**

Suppose that you create an array of `LineType` objects:

```matlab
H = [LineType('..','z'),LineType('--','q')]
```

```matlab
H =
    1x2 LineType with properties:
    Style
    Marker
    Units
```

When `H` is an array of handles, `get` returns a `(length(H)-1)` by 1 cell array of property values:

```matlab
CV = get(H,'Style')
```
CV =
  '..'
  '-'

When H is an array of handles and you do not specify a property name, get returns a struct array containing fields with names corresponding to property-names. Assign the output of get to a variable when H is not scalar.

SV = get(H)

SV =

2x1 struct array with fields:
  Style
  Marker
  Units

Get the value of the Marker property from the second array element in the SV array of structures:

SV(2).Marker

ans =
 q

Arrays of Handles, Names, and Values

You can pass an array of handles, a cell array of property names, and a cell array of property values to set. The property value cell array must have one row of property values for each object in H. Each row must have a value for each property in the property name array:

H = [LineType('..','z'),LineType('--','q')];
set(H,{'Style','Marker'},{'..','o';'--','x'})

The results of this call to set is:

H(1)

ans =

  LineType with properties:
Style: '..'
Marker: 'o'
Units: 'points'

H(2)
ans =

LineType with properties:

Style: '--'
Marker: 'x'
Units: 'points'

Customize the Property List

Customize the way property lists display by redefining the following methods in your subclass:

- `setdisp` — When you call `set` with no output argument and a single scalar handle input, `set` calls `setdisp` to determine how to display the property list.
- `getdisp` — When you call `get` with no output argument and a single scalar handle input, `get` calls `getdisp` to determine how to display the property list.

See Also

get | set

More About

- “Ways to Use Properties” on page 8-2
Implement Copy for Handle Classes

In this section...

“Copy Method for Handle Classes” on page 7-34
“Customize Copy Operation” on page 7-35
“Copy Properties That Contain Handles” on page 7-36
“Exclude Properties from Copy” on page 7-39

Copy Method for Handle Classes

Copying a handle variable results in another handle variable that refers to the same object. You can add copy functionality to your handle class by subclassing `matlab.mixin.Copyable`. The inherited copy method enables you to make shallow copies of objects of the class. The `CopyObj` class shows the behavior of copy operations.

```plaintext
classdef CopyObj < matlab.mixin.Copyable
    properties
        Prop
    end
end
```

Create an object of the `CopyObj` class and assign the handle of a `line` object to the property `Prop`.

```plaintext
a = CopyObj;
a.Prop = line;
```

Copy the object.

```plaintext
b = copy(a);
```

Confirm that the handle variables `a` and `b` refer to different objects.

```plaintext
a == b
ans =
   logical
   0
```
However, the line object referred to by \( \text{a}.\text{Prop} \) has not been copied. The handle contained in \( \text{a}.\text{Prop} \) refers to the same object as the handle contained in \( \text{b}.\text{Prop} \).

\[
\text{a}.\text{Prop} == \text{b}.\text{Prop}
\]

\[
\text{ans} =
\]

\[
\text{logical}
\]

\[
1
\]

For more detailed information on the behavior of the copy operation, see \texttt{matlab.mixin.Copyable.copy}.

**Customize Copy Operation**

Customize handle object copy behavior by deriving your class from \texttt{matlab.mixin.Copyable}. The \texttt{matlab.mixin.Copyable} class is an abstract base class that derives from the handle class. \texttt{matlab.mixin.Copyable} provides a template for customizing object copy operations by defining:

- \texttt{matlab.mixin.Copyable.copy} — Sealed method that defines the interface for copying objects
- \texttt{matlab.mixin.Copyable.copyElement} — Protected method that subclasses can override to customize object copy operations for the subclass

The \texttt{matlab.mixin.Copyable.copy} method, calls the \texttt{copyElement} method. Your subclass customizes the copy operation by defining its own version of \texttt{copyElement}.

The default implementation of \texttt{copyElement} makes shallow copies of all the nondependent properties. \texttt{copyElement} copies each property value and assigns it to the new (copied) property. If a property value is a handle object, \texttt{copyElement} copies the handle, but not the underlying data.

To implement different copy behavior for different properties, override \texttt{copyElement}. For example, the \texttt{copyElement} method of the \texttt{SpecializedCopy} class:

- Creates a new class object
- Copies the value of \texttt{Prop1} to the new object
- Reinitializes the default value of \texttt{Prop2} by adding a timestamp when the copy is made
classdef SpecializedCopy < matlab.mixin.Copyable
  properties
    Prop1
    Prop2 = datestr(now)
  end
  methods(Access = protected)
    function cp = copyElement(obj)
      cp = SpecializedCopy;
      cp.Prop1 = obj.Prop1;
      cp.Prop2 = datestr(now);
    end
  end
end

Create an object of the class and assign a value to Prop1:

```matlab
a  = SpecializedCopy;
a.Prop1 = 7
```

```matlab
a =

SpecializedCopy with properties:

  Prop1: 7
  Prop2: '17-Feb-2015 17:51:23'
```

Use the inherited copy method to create a copy of a:

```matlab
b = copy(a)
```

```matlab
b =

SpecializedCopy with properties:

  Prop1: 7
  Prop2: '17-Feb-2015 17:51:58'
```

The copy (object b) has the same value for Prop1, but the subclass copyElement method assigned a new value to Prop2. Notice the different timestamp.

**Copy Properties That Contain Handles**

Copying an object also copies the values of object properties. Object properties can contain other objects, including handle objects. If you simply copy the value of a property
that contains a handle object, you are actually copying the handle, not the object itself. Therefore, your copy references the same object as the original object. Classes that derive from the `matlab.mixin.Copyable` class can customize the way the copy method copies objects of the class.

**Class to Support Handle Copying**

Suppose that you define a class that stores a handle in an object property. You want to be able to copy objects of the class and want each copy of an object to refer to a new handle object. Customize the class copy behavior using these steps:

- Create a subclass of `matlab.mixin.Copyable`.
- Override `copyElement` to control how the property containing the handle is copied.
- Because the property value is a handle, create a new default object of the same class.
- Copy property values from the original handle object to the new handle object.

The “HandleCopy” on page 7-38 class customizes copy operations for the property that contains a handle object. The “ColorProp” on page 7-38 class defines the handle object to assign to `Prop2`:

Create an object and assign property values:

```matlab
a = HandleCopy;
a.Prop1 = 7;
a.Prop2 = ColorProp;
```

Make a copy of the object using the `copy` method inherited from `matlab.mixin.Copyable`:

```matlab
b = copy(a);
```

Demonstrate that the handle object contained by object `a` and `b` are independent. Changing the value on object `a` does not affect object `b`:

```matlab
a.Prop2.Color = 'red';
b.Prop2.Color
```

```matlab
ans =
blue
```
### HandleCopy

The HandleCopy class customizes the copy operation for objects of this class.

```matlab
classdef HandleCopy < matlab.mixin.Copyable
    properties
        Prop1 % Shallow copy
        Prop2 % Handle copy
    end
    methods (Access = protected)
        function cp = copyElement(obj)
            % Shallow copy object
            cp = copyElement@matlab.mixin.Copyable(obj);
            % Get handle from Prop2
            hobj = obj.Prop2;
            % Create default object
            new_hobj = eval(class(hobj));
            % Add public property values from orig object
            HandleCopy.propValues(new_hobj,hobj);
            % Assign the new object to property
            cp.Prop2 = new_hobj;
        end
    end
    methods (Static)
        function propValues(newObj,orgObj)
            pl = properties(orgObj);
            for k = 1:length(pl)
                if isprop(newObj,pl{k})
                    newObj.(pl{k}) = orgObj.(pl{k});
                end
            end
        end
    end
end
```

### ColorProp

The ColorProp class defines a color by assigning an RGB value to its Color property.

```matlab
classdef ColorProp < handle
    properties
        Color = 'blue';
    end
end
```
Exclude Properties from Copy

Use the NonCopyable property attribute to indicate that you do not want a copy operation to copy a particular property value. By default, NonCopyable is false, indicating that the property value is copyable. You can set NonCopyable to true only on properties of handle classes.

For classes that derive from matlab.mixin.Copyable, the default implementation of copyElement honors the NonCopyable attribute. Therefore, if a property has its NonCopyable attribute set to true, then copyElement does not copy the value of that property. If you override copyElement in your subclass, you can choose how to use the NonCopyable attribute.

Set the Attribute to Not Copy

Set NonCopyable to true in a property block:

```matlab
properties (NonCopyable)
    Prop1
end
```

Default Values

If a property that is not copyable has a default value assigned in the class definition, the copy operation assigns the default value to the property. For example, the CopiedClass assigns a default value to Prop2.

```matlab
classdef CopiedClass < matlab.mixin.Copyable
    properties (NonCopyable)
        Prop1
        Prop2 = datestr(now) % Assign current time
    end
end
```

Create an object to copy and assign a value to Prop1:

```matlab
a = CopiedClass;
a.Prop1 = 7
```

```
CopiedClass with properties:
```
prop1: 7
prop2: '17-Feb-2015 15:19:34'

Copy a to b using the copy method inherited from matlab.mixin.Copyable:

b = copy(a)

b =

CopiedClass with properties:

    Prop1: []
    Prop2: '17-Feb-2015 15:19:34'

In the copy b, the value of prop1 is not copied. The value of prop2 is set to its default value, which MATLAB determined when first loading the class. The timestamp does not change.

**Objects with Dynamic Properties**

Subclasses of the dynamicprops class allow you to add properties to an object of the class. When a class derived from dynamicprops is also a subclass of matlab.mixin.Copyable, the default implementation of copyElement does not copy dynamic properties. The default value of NonCopyable is true for dynamic properties.

The default implementation of copyElement honors the value of a dynamic property NonCopyable attribute. If you want to allow copying of a dynamic property, set its NonCopyable attribute to false. Copying a dynamic property copies the property value and the values of the property attributes.

For example, this copy operation copies the dynamic property, DynoProp, because its NonCopyable attribute is set to false. The object obj must be an instance of a class that derives from both dynamicprops and matlab.mixin.Copyable:

```matlab
obj = MyDynamicClass;
p = addprop(obj,'DynoProp');
p.NonCopyable = false;
obj2 = copy(obj);
```

**See Also**

matlab.mixin.Copyable
Related Examples

- “Dynamic Properties — Adding Properties to an Instance” on page 8-68
Properties — Storing Class Data

- “Ways to Use Properties” on page 8-2
- “Property Syntax” on page 8-5
- “Property Attributes” on page 8-9
- “Property Definition” on page 8-17
- “Mutable and Immutable Properties” on page 8-22
- “Validate Property Values” on page 8-24
- “Property Class and Size Validation” on page 8-31
- “Property Validation Functions” on page 8-39
- “Metadata Interface to Property Validation” on page 8-47
- “Property Access Methods” on page 8-49
- “Property Set Methods” on page 8-55
- “Property Get Methods” on page 8-59
- “Set and Get Methods for Dependent Properties” on page 8-61
- “Properties Containing Objects” on page 8-65
- “Dynamic Properties — Adding Properties to an Instance” on page 8-68
- “Set and Get Methods for Dynamic Properties” on page 8-72
- “Dynamic Property Events” on page 8-74
- “Dynamic Properties and ConstructOnLoad” on page 8-79
Ways to Use Properties

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“What Are Properties” on page 8-2</td>
</tr>
<tr>
<td>“Types of Properties” on page 8-3</td>
</tr>
</tbody>
</table>

What Are Properties

Properties encapsulate the data that belongs to instances of classes. Data contained in properties can be public, protected, or private. This data can be a fixed set of constant values, or it can depend on other values and calculated only when queried. You control these aspects of property behaviors by setting property attributes and by defining property-specific access methods.

Flexibility of Object Properties

In some ways, properties are like fields of a struct object. However, storing data in an object property provides more flexibility. Properties can:

- Define a constant value that you cannot change outside the class definition. See “Define Class Properties with Constant Values” on page 15-2.
- Calculate its value based on the current value of other data. See “Property Get Methods” on page 8-59.
- Execute a function to determine if an attempt to assign a value meets a certain criteria. See “Property Set Methods” on page 8-55.
- Trigger an event notification when any attempt is made to get or set its value. See “Property-Set and Query Events” on page 11-17.
- Control access by code to the property values. See the SetAccess and GetAccess attributes “Property Attributes” on page 8-9.
- Control whether its value is saved with the object in a MAT-file. See “Save and Load Objects” on page 13-2.

For an example of a class that defines and uses a class, see “Create a Simple Class” on page 2-2.
Types of Properties

There are two types of properties:

- Stored properties — Use memory and are part of the object
- Dependent properties — No allocated memory and the get access method calculates the value when queried

Features of Stored Properties

- Can assign an initial value in the class definition
- Property value is stored when you save the object to a MAT-file
- Can use a set access method to control possible values, but you are not required to use such methods.

When to Use Stored Properties

- You want to be able to save the property value in a MAT-file
- The property value is not dependent on other property values

Features of Dependent Properties

Dependent properties save memory because property values that depend on other values are calculated only when needed.

When to Use Dependent Properties

Define properties as dependent when you want to:

- Compute the value of a property from other values (for example, you can compute area from Width and Height properties).
- Provide a value in different formats depending on other values. For example, the size of a push button in values determined by the current setting of its Units property.
- Provide a standard interface where a particular property is or is not used, depending on other values. For example, different computer platforms can have different components on a toolbar).

For examples of classes that use dependent properties, see “Calculate Data on Demand” on page 3-23 and “A Class Hierarchy for Heterogeneous Arrays” on page 20-2.
See Also

Related Examples

- “Property Attributes” on page 8-9
- “Property Access Methods” on page 8-49
- “Static Properties” on page 5-66
Property Syntax

In this section...

“Property Definition Block” on page 8-5
“Access Property Values” on page 8-6
“Inheritance of Properties” on page 8-7
“Specify Property Attributes” on page 8-7

Property Definition Block

The following illustration shows a typical property specification. The properties and end keywords delineate a block of code that defines properties having the same attribute settings.

properties keyword begins definition block. Attribute specification

properties (SetAccess = protected)

Coefficients = [0 0 1]

end

properties block

Property name Default value

end keyword terminates definition block.

Note Properties cannot have the same name as the class.
For an example, see “Create a Simple Class” on page 2-2.

Assigning a Default Value

The preceding example shows the Coefficients property specified as having a default value of $[0 \ 0 \ 1]$.

You can initialize property values with MATLAB expressions. However, these expressions cannot refer to the class that you are defining in any way, except to call class static methods. MATLAB executes expressions that create initial property values only when initializing the class, which occurs just before first using the class. See “Property Default Values” on page 8-18 for more information about how MATLAB evaluates default value expressions.

Define One Property Per Line

Property names must be listed on separate lines. MATLAB interprets a name following a property name as the name of a class.

Restrict Property Values

You can restrict property values by associating a class with the property in the property definition. For example, the definition of MyData requires that values assigned to this property must be of type int32 or types that are compatible with int32.

```matlab
properties
    MyData int32
end
```

For more information, see “Validate Property Values” on page 8-24.

Access Property Values

Property access syntax is like MATLAB structure field syntax. For example, if obj is an object of a class, then you can get the value of a property by referencing the property name:

```matlab
val = obj.PropertyName
```

Assign values to properties by putting the property reference on the left side of the equal sign:

```matlab
obj.PropertyName = val
```
When you access a property, MATLAB executes any property set or get access method and triggering any enabled property events.

**Inheritance of Properties**

When you derive one class from another class, the derived (subclass) class inherits all the properties of the superclass. In general, subclasses define only properties that are unique to that particular class. Superclasses define properties that more than one subclass use.

**Specify Property Attributes**

Attributes specified with the `properties` keyword apply to all property definitions that follow in that block. If you want to apply attribute settings to certain properties only, reuse the `properties` keyword and create another property block for those properties.

For example, the following code shows the `SetAccess` attribute set to `private` for the `IndependentVar` and `Order` properties, but not for the `Coefficients` property:

```matlab
properties
    Coefficients = [0 0 1];
end
properties (SetAccess = private)
    IndependentVar
    Order = 0
end
```

These properties (and any others placed in this block) have private set access

For information about the properties of a specific class, use the `properties` function.

**See Also**

**Related Examples**

- “Validate Property Values” on page 8-24
• “Property Definition” on page 8-17
• “Property Attributes” on page 8-9
Property Attributes

In this section...

“Purpose of Property Attributes” on page 8-9
“Specifying Property Attributes” on page 8-9
“Table of Property Attributes” on page 8-9

Purpose of Property Attributes

Specifying attributes in the class definition enables you to customize the behavior of properties for specific purposes. Control characteristics like access, data storage, and visibility of properties by setting attributes. Subclasses do not inherit superclass member attributes.

Specifying Property Attributes

Assign property attributes on the same line as the properties keyword:

properties (Attribute1 = value1, Attribute2 = value2,...)
...
end

For example, give the Data property private access:

properties (Access = private)
    Data
end

For more information on attribute syntax, see “Attribute Specification” on page 5-22.

Table of Property Attributes

All properties support the attributes listed in the following table. Attribute values apply to all properties defined within the properties...end code block that specifies the nondefault values.
### Property Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbortSet</td>
<td>logical</td>
<td>If true, MATLAB does not set the property value if the new value is the same as the current value. MATLAB does not call the property set method, if one exists.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For handle classes, setting <strong>AbortSet</strong> to true also prevent the triggering of property <strong>PreSet</strong> and <strong>PostSet</strong> events.</td>
</tr>
<tr>
<td>Abstract</td>
<td>logical</td>
<td>If true, the property has no implementation, but a concrete subclass must redefine this property without <strong>Abstract</strong> being set to true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Assignment When Property Value Unchanged” on page 11-44.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Abstract properties cannot define set or get access methods. See “Property Access Methods” on page 8-49.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Abstract properties cannot define initial values. See “Assigning a Default Value” on page 8-6.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• All subclasses must specify the same values as the superclass for the property <strong>SetAccess</strong> and <strong>GetAccess</strong> attributes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Abstract=true</strong> use with the class attribute <strong>Sealed=false</strong> (the default).</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>Access</td>
<td>• enumeration, default = public&lt;br&gt;• meta.class object&lt;br&gt;• cell array of meta.class objects</td>
<td>Use Access to set both SetAccess and GetAccess to the same value. Query the values of SetAccess and GetAccess directly (not Access). public - unrestricted access protected - access from class or subclasses private - access by class members only (not subclasses) List of classes that have get and set access to this property. Specify classes as meta.class objects in the form: • A single meta.class object • A cell array of meta.class objects. An empty cell array, {}, is the same as private access.</td>
</tr>
<tr>
<td>Constant</td>
<td>logical&lt;br&gt;default = false</td>
<td>Set to true if you want only one value for this property in all instances of the class: • Subclasses inherit constant properties, but cannot change them. • Constant properties cannot be Dependent. • SetAccess is ignored. See “Define Class Properties with Constant Values” on page 15-2 for more information.</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Dependent</td>
<td>logical</td>
<td>If false, property value is stored in object. If true, property value is not stored in object. The set and get functions cannot access the property by indexing into the object using the property name. MATLAB does not display in the command window the names and values of Dependent properties that do not define a get method (scalar object display only). Values returned by dependent property get methods are not considered when testing for object equality using isequal.</td>
</tr>
</tbody>
</table>

- “Calculate Data on Demand” on page 3-23
- “Property Get Methods” on page 8-59
- “Avoid Property Initialization Order Dependency” on page 13-12
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetAccess</td>
<td>enumeration</td>
<td>public — unrestricted access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>protected — access from class or subclasses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>private — access by class members only (not from subclasses)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>List classes that have get access to this property. Specify classes as <code>meta.class</code> objects in the form:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A single <code>meta.class</code> object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A cell array of <code>meta.class</code> objects. An empty cell array, <code>{}</code>, is the same as private access.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Class Members Access” on page 12-28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MATLAB does not display in the command window the names and values of properties having protected or private GetAccess or properties whose Hidden attribute is true.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The <code>struct</code> function defines fields for all properties when converting objects to structs.</td>
</tr>
<tr>
<td>GetObservable</td>
<td>logical</td>
<td>If true, and it is a handle class property, then you can create listeners for access to this property. The listeners are called whenever property values are queried. See “Property-Set and Query Events” on page 11-17</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Hidden</td>
<td>logical</td>
<td>Determines if the property can be shown in a property list (e.g., Property Inspector, call to <code>set</code> or <code>get</code>, etc.). MATLAB does not display in the command window the names and values of properties whose <code>Hidden</code> attribute is true or properties having protected or private <code>GetAccess</code>.</td>
</tr>
<tr>
<td>NonCopyable</td>
<td>logical</td>
<td>Determine if property value can be copied when object is copied. You can set <code>NonCopyable</code> to true only in handle classes. For more information, see “Exclude Properties from Copy” on page 7-39.</td>
</tr>
<tr>
<td>Attribute Name</td>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| **SetAccess**  | enumeration | public — unrestricted access  
|                | default = public | protected — access from class or subclasses  
|                |                    | private — access by class members only (not from subclasses)  
|                |                    | immutable — property can be set only in the constructor.  
|                |                    | See “Mutable and Immutable Properties” on page 8-22  
|                |                    | List classes that have set access to this property. Specify classes as `meta.class` objects in the form:  
|                |                    | • A single `meta.class` object  
|                |                    | • A cell array of `meta.class` objects.  
|                |                    | An empty cell array, `{}`, is the same as private access.  
|                |                    | See “Class Members Access” on page 12-28  
| **SetObservable** | logical | If true, and it is a handle class property, then you can create listeners for access to this property. The listeners are called whenever property values are modified. See “Property-Set and Query Events” on page 11-17  
|                | default = false |  
| **Transient**  | logical | If true, property value is not saved when object is saved to a file. See “Save and Load Process for Objects” on page 13-2 for more about saving objects.  
|                | default = false |  


See Also

Related Examples
•  “Property Definition” on page 8-17
Property Definition

In this section...

“What You Can Define” on page 8-17
 “Initialize Property Values” on page 8-18
 “Property Default Values” on page 8-18
 “Assign Property Values from Constructor” on page 8-19
 “Initialize Properties to Unique Values” on page 8-19
 “Property Attributes” on page 8-19
 “Property Access Methods” on page 8-20
 “Reference Object Properties Using Variables” on page 8-21

What You Can Define

Control aspects of property definitions in the following ways:

• Specify a default value for each property individually, see “Property Default Values” on page 8-18.
• Assign property values in a class constructor, see “Assign Property Values from Constructor” on page 8-19.
• Define properties with constant values, see “Named Values” on page 14-2.
• Assign property attribute values on a per block basis, see “Property Attributes” on page 8-19.
• Define methods that execute when the property is set or queried, see “Property Access Methods” on page 8-20.
• Define the class and size of property values, see “Validate Property Values” on page 8-24.
• Define properties that do not store values, but whose values depend on other properties, see “Types of Properties” on page 8-3.

Note Always use case-sensitive property names in your MATLAB code. Properties cannot have the same name as the class.
Initialize Property Values

There are two basic approaches to initializing property values:

- In the property definition — MATLAB evaluates the expression only once and assigns the same value to the property of every instance.
- In a class constructor — MATLAB evaluates the assignment expression for each instance, which ensures that each instance has a unique value.

For more information on the evaluation of expressions that you assign as property default values, see “When MATLAB Evaluates Expressions” on page 6-13.

Property Default Values

Within a properties block, you can control an individual property's default value. Assign default values as a value or MATLAB expressions. Expressions cannot reference variables. For example:

- Prop1 — No assignment results in empty [] default value
- Prop2 — Assign character array as default value
- Prop3 — Assign result of expression as default value

```matlab
classdef ClassName
    properties
        Prop1
        Prop2 = 'some text'
        Prop3 = sin(pi/12)
    end
end
```

If the class definition does not specify a default property value, MATLAB sets the property value to empty `double([])`.

**Note** Evaluation of property default values occurs only when the value is first needed, and only once when MATLAB first initializes the class. MATLAB does not reevaluate the expression each time you create an instance of the class.

For more information on the evaluation of expressions that you assign as property default values, see “Evaluation of Expressions in Class Definitions” on page 6-10 and “Properties Containing Objects” on page 8-65.
Assign Property Values from Constructor

To assign values to a property from within the class constructor, reference the object that the constructor returns (the output variable `obj`):

```matlab
classdef MyClass
    properties
        Prop1
    end
    methods
        function obj = MyClass(intval)
            obj.Prop1 = intval;
        end
    end
end
```

When you assign a property in the class constructor, MATLAB evaluates the assignment statement for each object you create. Assign property values in the constructor if you want each object to contain a unique value for that property.

For example, suppose that you want to assign a unique handle object to the property of another object each time you create one of those objects. Assign the handle object to the property in the constructor. Call the handle object constructor to create a unique handle object with each instance of your class.

For more information on constructor methods, see “Referencing the Object in a Constructor” on page 9-25.

Initialize Properties to Unique Values

MATLAB assigns properties to the specified default values only once when MATLAB loads the class definition. Therefore, if you initialize a property value with a handle-class constructor, MATLAB calls this constructor only once and every instance references the same handle object. If you want a property value to be initialized to a new instance of a handle object each time you create an object, assign the property value in the constructor.

Property Attributes

All properties have attributes that modify certain aspects of the property's behavior. Specified attributes apply to all properties in a particular properties block. For example:
classdef ClassName
    properties (PropertyAttribute = value)
        Prop1
        Prop2
    end
end

For example, only methods in the same class definition can modify and query the Salary and Password properties.

classdef EmployeeInfo
    properties (Access = private)
        Salary
        Password
    end
end

This restriction exists because the class defines these properties in a properties block with the Access attribute set to private.

Property Attributes

For a description of property attributes you can specify, see, “Property Attributes” on page 8-9.

Property Access Methods

MATLAB calls whenever setting or querying a property value. Define property set access or get access methods in methods blocks that specify no attributes and have the following syntax:

methods
    function obj = set.PropertyName(obj,value)
        ...  
    end
    function value = get.PropertyName(obj)
        ...  
    end
end

MATLAB does not call the property set access method when assigning the default value specified in the property's definition block.
For example, the `set.Password` method tests the length of the character array assigned to a property named `Password`. If there are fewer than seven characters in the value assigned to the property, MATLAB returns the error. Otherwise, MATLAB assigns the specified value to the property.

```matlab
function obj = set.Password(obj,pw)
    if numel(pw) < 7
        error('Password must have at least 7 characters')
    else
        obj.Password = pw;
    end
```

For more information on property access methods, see “Property Access Methods” on page 8-49.

**Reference Object Properties Using Variables**

MATLAB can resolve a property name from a `char` variable using an expression of the form:

```matlab
object.(PropertyNameVar)
```

where `PropertyNameVar` is a variable containing the name of a valid object property. Use this syntax when passing property names as arguments. For example, the `getPropValue` function returns the value of the `KeyType` property:

```matlab
PropName = 'KeyType';
function o = getPropValue(obj,PropName)
    o = obj.(PropName);
end
```

**See Also**

**Related Examples**

- “Evaluation of Expressions in Class Definitions” on page 6-10
- “Ways to Use Properties” on page 8-2
- “Validate Property Values” on page 8-24
Mutable and Immutable Properties

**In this section...**

“Set Access to Property Values” on page 8-22  
“Define Immutable Property” on page 8-22

**Set Access to Property Values**

The property `SetAccess` attribute enables you to determine under what conditions code can modify object property values. There are four levels of set access that provide varying degrees of access to object property values:

- **SetAccess = public** — Any code with access to an object can set public property values. There are differences between the behavior of handle and value classes with respect to modifying object properties.

- **SetAccess = protected** — Only code executing from within class methods or methods of subclasses can set property values. You cannot change the value of an object property unless the class or any of its subclasses defines a method to do so.

- **SetAccess = private** — Only the defining class can set property values. You can change the value of an object property only if the class defines a method that sets the property.

- **SetAccess = immutable** — Property value is set during construction. You cannot change the value of an immutable property after the object is created. Set the value of the property as a default or in the class constructor.

**Define Immutable Property**

In this class definition, only the `Immute` class constructor can set the value of the `CurrentDate` property:

```matlab
classdef Immute
    properties (SetAccess = immutable)
        CurrentDate
    end
    methods
        function obj = Immute
            obj.CurrentDate = date;
        end
    end
end
```

---

8-22
end
end

a = Immute

a =

  Immute with properties:

    CurrentDate: '19-Oct-2005'

See Also

Related Examples

- "Property Attributes" on page 8-9
- "Object Modification" on page 5-62
Validate Property Values

### In this section...

| “Property Validation in Class Definitions” on page 8-24 |
| “Sample Class Using Property Validation” on page 8-25 |
| “Order of Validation” on page 8-27 |
| “Abstract Property Validation” on page 8-28 |
| “Objects Not Updated When Changing Validation” on page 8-28 |
| “Validation During Load Operation” on page 8-28 |

### Property Validation in Class Definitions

MATLAB property validation enables you to place specific restrictions on property values. You can use validation to constrain the class and size of property values. Also, you can use functions to establish criteria that the property value must conform to. MATLAB defines a set of validation functions and you can write your own validation functions.

The use of property validation is optional in class definitions.

**Validation Syntax**

The highlighted area in the following code shows the syntax for property validation.

```matlab
classdef MyClass
    properties
        Prop(dim1,dim2,...) ClassName {fcn1,fcn2,...} = defaultValue
    end
end
```

Property validations any of the of the following:

- **Size** — The length of each dimension, specified as a positive integer or a colon. A colon indicates that any length is allowed in that dimension. The value assigned to the
property must conform to the specified size or be compatible with the specified size. For more information, see “Property Size Validation” on page 8-31.

- Class — The name of a single MATLAB class. The value assigned to the property must be of the specified class or convertible to the specified class. Use any MATLAB class or externally defined class that is supported by MATLAB, except for Java and COM classes. For more information, see “Property Class Validation” on page 8-33.

- Functions — A comma-separated list of validation function names. MATLAB passes the value assigned to the property to each the validation functions after applying any possible class and size conversions. Validator functions throw errors if the validation fails, but do not return values. For more information, see “Property Validation Functions” on page 8-39.

For a list of MATLAB validation functions, see “MATLAB Validation Functions” on page 8-42.

Using Property Validation

Use property validation for public properties to control the values user code assigns to the properties.

If you want to restrict property values to a fixed set of identifiers, create an enumeration class for these identifiers and constrain the property to this class. For information on enumeration classes, see “Define Enumeration Classes” on page 14-5.

MATLAB type conversion rules apply to property validation. For example, MATLAB can coerce among numeric types in most cases. Therefore, restricting a property value to a specific numeric type, such as double does not prevent the assignment of other numeric types to the property. To ensure that a property can be assigned only a specific type, define a class that supports only the desired type conversions or use a validation function to specify the exact type allowed for the property.

Ensure that any default value assigned to the property meets the restrictions imposed by the specified validation. If you do not specify a default value, MATLAB creates a default value by assigning an empty object of the specified class or by calling the default constructor if size restriction does not allow the use of an empty default value. The default constructor must return an object of the correct size.

Sample Class Using Property Validation

The ValidateProps class defines three properties with validation.
classdef ValidateProps
    properties
        Location(1,3) double {mustBeReal, mustBeFinite}
        Label(1,:) char {mustBeMember(Label,{'High','Medium','Low'})} = 'Low'
        State(1,1) matlab.lang.OnOffSwitchState
    end
end

- **Location** must be a 1-by-3 array of class double whose values are real, finite numbers.
- **Label** must be a char vector that is either 'High', 'Medium', or 'Low'.
- **State** must be an enumeration member of the matlab.lang.OnOffSwitchState class (off or on).

**Validation at Instantiation**

Creating an object of the ValidateProps class performs the validation on implicit and explicit default values:

```matlab
a = ValidateProps
```

```matlab
a = 

ValidateProps with properties:

    Location: [0 0 0]
    Label: 'Low'
    State: off
```

When creating the object, MATLAB:

- Initializes the **Location** property value to [0 0 0] to satisfy the size and class requirements.
- Sets the **Label** property to its default value, 'Low'. The default value must be a member of the allowed set of values. The empty char implicit default value would cause an error.
- Sets the **State** property to the off enumeration member defined by the matlab.lang.OnOffSwitchState class.

For information on how MATLAB selects default values, see “Default Values Per Size and Class” on page 8-37.
Order of Validation

When a value is assigned to the property, including default values that are specified in the class definition, MATLAB performs validation in this order:

- **Class validation** — This validation can cause conversion to a different class, such as conversion of a `char` to `string`. Assignment to properties follows MATLAB coercion rules for arrays.
- **Size validation** — This validation can cause size conversion, such as scalar expansion or conversion of a column vector to a row vector. Assignment to a property that specifies a size validation behaves the same as assignment to any MATLAB array. For information on indexed assignment, see “Array Indexing”.
- **Validator functions** — MATLAB passes the result of the class and size validation to each validation function, in left to right order. An error can occur before all validation functions have been called, which ends the validation process.
- **Set method** — MATLAB performs property validation before calling a property set method, if one is defined for that property. Assignment to the property within a property set or get method does not apply the validation again. Often, you can replace property set methods using property validation.

Property Validation Errors

The `ValueProp` class uses size, class, and function validation to ensure that an assignment to the `Value` property is a double scalar that is not negative.

```matlab
classdef ValueProp
    properties
        Value(1,1) double {mustBeNonnegative} = 0
    end
end
```

This statement attempts to assign a cell array to the property. This assignment violates the class validation.

```matlab
a.Value = {10,20};
```

Error setting property 'Value' of class 'ValueProp':
Invalid data type. Value must be double or be convertible to double.

This statement attempts to assign a 1-by-2 double array to the property. This assignment violates the size validation.
a.Value = [10 20];

Error setting property 'Value' of class 'ValueProp':
Size of value must be scalar.

This statement attempts to assign a scalar double to the property. This assignment fails
the function validation, which requires a nonnegative number.

a.Value = -10;

Error setting property 'Value' of class 'ValueProp':
Value must be nonnegative.

The validation process ends with the first error encountered.

Abstract Property Validation

You can define property validation for abstract properties. The validation applies to all
subclasses that implement the property. However, subclasses cannot use any validation on
their implementation of the property. When inheriting validation for a property from
multiple classes, only a single Abstract property in one superclass can define the
validation. None of the superclasses can define the property as nonAbstract.

Objects Not Updated When Changing Validation

If you change the property validation while objects of the class exist, MATLAB does not
attempt to apply the new validation to existing property values. However, MATLAB does
apply the new validation when you make assignments to the properties of existing
objects.

Validation During Load Operation

When saving an object to a MAT file, MATLAB saves all nondefault property values with
the object. When loading the object, MATLAB restores these property values in the newly
created object.

If a class definition changes the property validation such that the loaded property value is
no longer valid, MATLAB substitutes the currently defined default value for that property.
However, the load function suppresses the validation errors that occur before assigning
the default value from the current class definition. Therefore, validation errors are
silently ignored during load operations.
To illustrate this behavior, this example creates, saves, and loads an object of the MonthTemp class. This class restricts the AveTemp property to a cell array.

```matlab
classdef MonthTemp
    properties
        AveTemp cell
    end
end
```

Create a MonthTemp object and assign a value to the AveTemp property.

```matlab
a = MonthTemp;
a.AveTemp = {'May',70};
```

Save the object using save.

```matlab
save TemperatureFile a
```

Edit the property definition to change the validation class for the AveTemp property from cell array to containers.Map.

```matlab
classdef MonthTemp
    properties
        AveTemp containers.Map
    end
end
```

Load the saved object with the new class definition on the MATLAB path. MATLAB cannot assign the saved value to the AveTemp property because the cell array, {'May',70}, is not compatible with the current requirement that the property value be a containers.Map object. MATLAB cannot convert a cell array to a containers.Map.

To address the incompatibility, MATLAB sets the AveTemp property of the loaded object to the current default value, which is an empty containers.Map object.

```matlab
load TemperatureFile a
a.AveTemp
```

ans =

```
Map with properties:

    Count: 0
    KeyType: char
    ValueType: any
```
The loaded object has a different value assigned to the AveTemp property because the saved value is now invalid. However, the load process suppresses the validation error.

To prevent loss of data when changing class definitions and reloading objects, implement a loadobj method or class converter method that enables the saved values to satisfy the current property validation.

For more information on saving and loading objects, see “Save and Load Process for Objects” on page 13-2.

See Also

Related Examples

•  “Property Class and Size Validation” on page 8-31
•  “Property Validation Functions” on page 8-39
Property Class and Size Validation

**In this section...**

- “Property Class and Size” on page 8-31
- “Property Size Validation” on page 8-31
- “Property Class Validation” on page 8-33
- “Default Values Per Size and Class” on page 8-37

**Property Class and Size**

MATLAB applies any class and size validation defined for a property before calling validation functions. Assignment to a property that defines size or class validation is analogous to assignment to a MATLAB object array. MATLAB can apply class and size conversions to the right side of the assignment to satisfy class and size validation.

For more information, see “Order of Validation” on page 8-27 and “Property Validation Functions” on page 8-39.

**Property Size Validation**

Specify the property size as the row, column, and additional dimension following the property name. Size validation does not support scientific notation.

```matlab
classdef MyClass
    properties
        Prop(dim1,dim2,...) = defaultValue
    end
end
```

**Assignment and Size Validation**

This class defines the size of the `Location` property as 1-by-3. Any value assigned to this property must conform to that size or must be convertible to that size.

```matlab
classdef ValidateProps
    properties
        Location(1,3)
    end
end
```
The implicit default value assigned by MATLAB, [0 0 0], conforms to the specified size:

```matlab
a = ValidateProps
a =

ValidateProps with properties:
    Location: [0 0 0]
```

MATLAB applies scalar expansion when you assign a scalar the Location property.

```matlab
a = ValidateProps;
a.Location = 1
a =

ValidateProps with properties:
    Location: [1 1 1]
```

MATLAB converts columns to rows to match the size specification:

```matlab
col = [1;1;1]
col =
    1
    1
    1
a.Location = col
a =

ValidateProps with properties:
    Location: [1 1 1]
```

**Colon in Size Specification**

A colon in the size specification indicates that the corresponding dimension can have any length. For example, you can assign a value of any length to the Label property in this class.

```matlab
classdef ValidateProps
    properties
```
Assignment to a property that defines size validation follows the same rules as the equivalent indexed array assignment. For information on indexing behavior of multidimensional arrays, see “Compatible Array Sizes for Basic Operations”.

**Property Class Validation**

Defining the class of a property can reduce the need to test the values assigned to the property in your code. Any value assigned to the property must be of the specified class or convertible to the specified class.

You can specify only one class per property. Use validation functions like `mustBeNumeric` or `mustBeInteger` to restrict properties to a broader category of classes. For more information on validation functions, see “Property Validation Functions” on page 8-39.

You can use any MATLAB class or externally defined class that is supported by MATLAB, except Java and COM classes.

Place the name of the class in the property definition block following the property name and optional size specification.

```matlab
classdef MyClass
    properties
        Prop ClassName = defaultValue
    end
end
```

If you do not specify a default value, MATLAB assigns an empty object of the specified class to the property. If you define a size and a class, MATLAB attempts to create a default value for the property that satisfies both the size and class requirement.
MATLAB creates the default value by calling the class constructor with no arguments. The class must have a constructor that returns an object of the specified size when called with no input arguments or you must specify a default value for the property that satisfies the property size restriction. For more information, see “Default Values Per Size and Class” on page 8-37.

**Using Class Validation**

The PropsWithClass class defines two properties with class definitions:

- **Number** — Values must be of class double or convertible to double.
- **Today** — Values must be of class char or convertible to char. The default value is the char vector returned by the date function.

```matlab
classdef PropsWithClass
    properties
        Number double
        Today char = date
    end
end
```

Create an object of the PropsWithClass class.

```matlab
p = PropsWithClass
```

```matlab
p =

    PropsWithClass with properties:

        Number: []
        Today: '10-Sep-2016'
```

MATLAB performs conversions from any compatible class to the property class. For example, assign a datetime array to the Today property.

```matlab
p.Today = [datetime('now'),datetime('tomorrow')];
disp(class(p.Today))
```

```matlab
ans =

    char
```

Because the datetime class has a char converter, you can assign a datetime array to the Today property.
Assigning an incompatible value to a property that uses class validation causes an error.

```matlab
p.Number = datetime('now');
```

Error setting property 'Number' of class 'PropsWithClass':
Invalid data type. Value must be double or be convertible to double.

**User-Defined Class for Validation**

You can define a class to control the values assigned to a property. Enumeration classes enable users to set property values to character vectors or string scalars with inexact name matching.

For example, suppose that there is a class that represents a three-speed mechanical pump. You can define an enumeration class to represent the three flow rates.

```matlab
classdef FlowRate < int32
    enumeration
        Low    (10)
        Medium (50)
        High   (100)
    end
end
```

The Pump class has a method to return the current flow rate in gallons per minute. Define the Speed property as a FlowRate class.

```matlab
classdef Pump
    properties
        Speed FlowRate
    end
    methods
        function getGPM(p)
            if isempty(p.Speed)
                gpm = 0;
            else
                gpm = int32(p.Speed);
            end
            fprintf('Flow rate is: %i GPM\n',gpm);
        end
    end
end
```

Users can set the Speed property using inexact text.
p = Pump;
p.Speed = 'm'

p =
    Pump with properties:
        Speed: Medium

The numerical value is available from the property.

getGPM(p)
Flow rate is: 50 GPM

For information about enumeration classes, see “Define Enumeration Classes” on page 14-5.

**Integer Class Validation**

MATLAB supports several integer classes (see “Integers”). However, restricting a property to an integer class can result in integer overflow. The resulting value can saturate at the maximum or minimum value in the integer’s range.

When integer overflow occurs, the value that is assigned to a property might not be the value from the right side of the assignment statement.

For example, suppose that you want to restrict a property value to a scalar uint8.

```matlab
classdef IntProperty
    properties
        Value(1,1) uint8
    end
end
```

Assigning a numeric value to the Value property effectively casts the numeric value to uint8, but does not result in an error for out-of-range values.

```matlab
a = IntProperty;
a.Value = -10;
disp(a.Value)
```

0
Assignment to the Value property is equivalent to indexed assignment of an array. If the assigned value is out of the range of values that uint8 can represent, MATLAB sets the value to the closest value that it can represent using uint8.

```matlab
a = uint8.empty;
a(1) = -10
```

```matlab
a =
    uint8
     0
```

To avoid the potential for integer overflow, use a combination of validation functions that restrict the value to the intended range instead of an integer class.

```matlab
classdef IntProperty
    properties
        Value(1,1) {mustBeInteger, mustBeNonnegative,...
               mustBeLessThan(Value,256)}
    end
end
```

Because there is no conversion of the assigned value by the uint8 class, the validators catch out of range values and throw an appropriate error.

```matlab
a = IntProperty;
a.Value = -10;
```

Error setting property 'Value' of class 'IntProperty':
Value must be nonnegative.

**Default Values Per Size and Class**

Any default property value that you assign in the class definition must conform to the specified validation.

**Implicit Default Values**

MATLAB defines a default value implicitly if you do not specify a default value in the class definition. This table shows how size and class determine the implicit default value of MATLAB classes.
To determine the implicit default value, MATLAB calls the default class constructor and builds an array of the specified size using the instance returned by the constructor call. If the class does not support a default constructor (that is, a constructor called with no arguments), then MATLAB throws an error when instantiating the class containing the validation.

For heterogeneous arrays, MATLAB calls the `getDefaultScalarElement` method to get the default object.

### See Also

### Related Examples

- “Validate Property Values” on page 8-24
- “Property Validation Functions” on page 8-39
- “Enumerations for Property Values” on page 14-17
Property Validation Functions

In this section...

| “Validate Property Using Functions” on page 8-39 |
| “MATLAB Validation Functions” on page 8-42 |
| “Define Validation Functions” on page 8-44 |
| “Add Support for Validation Functions” on page 8-45 |

Validate Property Using Functions

Use property validation functions in class definitions to impose specific restrictions on property values. A validation function accepts a potential property value as an argument and issues an error if the value does not meet the specific requirement imposed by the function.

During the validation process, MATLAB passes the value to each validation function listed in the class definition. MATLAB calls each function from left to right and throws the first error encountered. The value passed to the validation functions is the result of any conversion applied by the class and size specifications. For more information on class and size validation, see “Property Class and Size Validation” on page 8-31.

For a list of MATLAB validation functions, see “MATLAB Validation Functions” on page 8-42

Validation Function Syntax

Specify validation functions as a comma-separated list of function names or function calls with arguments, enclosed in braces.

```matlab
classdef MyClass
    properties
        Prop {fcn1,fcn2,...} = defaultValue
    end
end
```

MATLAB passes the potential property value to the validation function implicitly. However, if the validation function requires input arguments in addition to the potential property value, then you must include both the property and the additional arguments. Additional arguments must be literal values and cannot reference variables. Literal values are nonsymbolic representations, such as numbers and text.
For example, consider the function `mustBeGreaterThan`. It requires a limiting value as an input parameter. This validation function requires that a property value must be greater than this limiting value.

Pass the property as the first argument. Use the property name, but do not enclose the name in quotation marks. This property definition restricts Prop to values greater than 10.

```matlab
properties
    Prop {mustBeGreaterThan(Prop, 10)}
end
```

**Using Validation Functions**

The following class specifies validation functions for each property.

- **Data** must be numeric and finite.
- **Interp** must be one of the three options listed. Specify a default value for this property to satisfy this requirement.

```matlab
classdef ValidatorFunction
    properties
        Data {mustBeNumeric, mustBeFinite}
        Interp {mustBeMember(Interp, {'linear', 'cubic', 'spline'})} = 'linear'
    end
end
```

Creating a default object of the class shows the initial values.

```matlab
a = ValidatorFunction
```

```matlab
a =
    ValidatorFunction with properties:
        Data: []
        Interp: 'linear'
```

Assigning values to properties calls the validation functions.

```matlab
a.Data = 'cubic'
```

Error setting property 'Data' of class 'ValidatorFunction':
Value must be numeric.

Because the `Data` property validation does not include a numeric class, there is no conversion of the char vector to a numeric value. If you change the validation of the
Data property to specify the class as double, MATLAB converts the char vector to a double array.

properties
    Data double {mustBeNumeric, mustBeFinite}
end

The assignment to the char vector does not produce an error because MATLAB converts the char vector to class double.

a.Data = 'cubic'

a =

    ValidatorFunction with properties:

        Data: [99 117 98 105 99]
        Interp: 'linear'

Assignment to the Interp property requires an exact match.

a = ValidatorFunction;
a.Interp = 'cu'

Error setting property 'Interp' of class 'ValidatorFunction':
Value must be a member of this set
    linear
    cubic
    spline

Using an enumeration class provides inexact matching and case insensitivity.

**Enumeration Class for Inexact Matching**

Property validation using an enumeration class provides these advantages:

- Inexact, case-insensitive matching for unambiguous char vectors or string scalars
- Conversion of inexact matches to correct values

For example, suppose that you define the InterpMethod enumeration class for the Interp property validation.

classdef InterpMethod
    enumeration
        linear

8-41
cubic
spline
end
end

Change the Interp property validation to use the InterpMethod class.

```matlab
classdef ValidatorFunction
    properties
        Data {mustBeNumeric, mustBeFinite}
        Interp InterpMethod
    end
end
```

Assign a value matching the first few letters of 'cubic'.

```matlab
a = ValidatorFunction;
a.Interp = 'cu'
```

```
a =

    ValidatorFunction with properties:

        Data: []
        Interp: cubic
```

## MATLAB Validation Functions

MATLAB defines functions for use in property validation. These functions support common use patterns for property validation and provide descriptive error messages. This table lists the MATLAB validation functions, their meanings, and the MATLAB functions used by the validation functions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mustBePositive(A)</code></td>
<td>A &gt; 0</td>
<td><code>gt</code>, <code>isreal</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>isnumeric</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>islogical</code></td>
</tr>
<tr>
<td><code>mustBeNonpositive(A)</code></td>
<td>A &lt;= 0</td>
<td><code>ge</code>, <code>isreal</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>isnumeric</code>,</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>islogical</code></td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Dependencies</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>mustBeFinite(A)</td>
<td>A has no NaN and no Inf elements.</td>
<td>isfinite</td>
</tr>
<tr>
<td>mustBeNonNan(A)</td>
<td>A has no NaN elements.</td>
<td>isnan</td>
</tr>
<tr>
<td>mustBeNonnegative(A)</td>
<td>A &gt;= 0</td>
<td>ge, isreal, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeNegative(A)</td>
<td>A &lt; 0</td>
<td>lt, isreal, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeNonzero(A)</td>
<td>A ~= 0</td>
<td>eq, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeGreaterThan(A,B)</td>
<td>A &gt; B</td>
<td>gt, isscalar, isreal, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeLessThan(A,B)</td>
<td>A &lt; B</td>
<td>lt, isreal, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeGreaterThanOrEqual(A,B)</td>
<td>A &gt;= B</td>
<td>ge, isreal, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeLessThanOrEqual(A,B)</td>
<td>A &lt;= B</td>
<td>le, isreal, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeNonempty(A)</td>
<td>A is not empty</td>
<td>isempty</td>
</tr>
<tr>
<td>mustBeNonsparse(A)</td>
<td>A has no sparse elements.</td>
<td>issparse</td>
</tr>
<tr>
<td>mustBeNumeric(A)</td>
<td>A is numeric.</td>
<td>isnumeric</td>
</tr>
<tr>
<td>mustBeNumericOrLogical(A)</td>
<td>A is numeric or logical.</td>
<td>isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeReal(A)</td>
<td>A has no imaginary part.</td>
<td>isreal</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Dependencies</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>mustBeInteger(A)</td>
<td>A==floor(A)</td>
<td>isreal, isfinite, floor, isnumeric, islogical</td>
</tr>
<tr>
<td>mustBeMember(A,B)</td>
<td>A is an exact match for a member of B.</td>
<td>ismember</td>
</tr>
</tbody>
</table>

### Define Validation Functions

Validator functions are ordinary MATLAB functions that are designed for the specific purpose of validating property values. Functions used as property validators:

- Accept the potential property value as an input argument
- Do not return values
- Throw errors if the validation fails

Creating your own validation function is useful when you want to provide specific validation that is not available using the MATLAB validation functions. You can create local functions within the class file or place the function on the MATLAB path to be available for use in any class.

For example, the `ImData` class uses a local function to define a validator that restricts the `Data` property to a specific range of numeric values.

```matlab
classdef ImData
    properties
        Data {mustBeNumeric, mustBeInRange(Data,[0,255])} = 0
    end
end
function mustBeInRange(a,b)
    if any(a(:) < b(1)) || any(a(:) > b(2))
        error(['Value assigned to Data property is not in range ',... num2str(b(1)),'...',num2str(b(2))])
    end
end
end
```

When you create an instance of the `ImData` class, MATLAB validates that the default value is numeric, in the range 0...255, and not empty.

```matlab
a = ImData
```
a =

ImData with properties:

Data: 0

Property assignment invokes the validators in left-to-right order. Assigning a char vector to the Data property causes an error thrown by mustBeNumeric.

a.Data = 'red'

Error setting property 'Data' of class 'ImData':
Value must be numeric.

Assigning a numeric value that is out of range causes an error thrown by mustBeInRange.

a.Data = -1

Error setting property 'Data' of class 'ImData':
Value assigned to Data property is not in range 0...255

### Add Support for Validation Functions

Support MATLAB validation functions for objects of your class by implementing the dependent functions as methods of your class. To determine which methods to implement for each function, see the validation function reference pages listed in this table “MATLAB Validation Functions” on page 8-42.

For example, suppose that you want your class to support the mustBeGreaterThan validation function. Overload these MATLAB functions as methods in your class:

- `isreal` — Always return logical `true` because mustBeGreaterThan does not support complex numbers.
- `gt` — The second object in the comparison must be scalar, as required by mustBeGreaterThan.

The SupportmBGT class implements support for mustBeGreaterThan.

```matlab
classdef SupportmBGT
    properties
        Prop(1,1) double {mustBeReal}
    end
```
methods
  function obj = SupportmBGT(data)
    if nargin > 0
      obj.Prop = data;
    end
  end
  function tf = isreal(obj)
    tf = true;
  end
  function tf = gt(obj1, obj2)
    tf = [obj1(:).Prop] > obj2.Prop;
  end
end

Use `mustBeGreaterThan` with objects of this class:

```matlab
a = SupportmBGT(10);
b = SupportmBGT(12);
mustBeGreaterThan(a,b)
```

Error using `mustBeGreaterThan` (line 19)
Value must be greater than the comparison value.

See Also

Related Examples

- “Validate Property Values” on page 8-24
- “Property Class and Size Validation” on page 8-31
Metadata Interface to Property Validation

For information on property validation, see “Validate Property Values” on page 8-24.

You can determine what validation applies to a property by accessing the validation metadata. Instances of the meta.Validation class provide the following information about property validation.

- Class requirement of the property specified as a meta.class object
- Size requirements of the property value specified as an array of meta.FixedDimension and meta.UnrestrictedDimension objects
- Function handles referencing validation functions applied to property values specified as a cell array of function handles.

For example, the ValidationExample class defines a property that must be an array of doubles that is 1-by-any number of elements and must be a real number that is greater than 10.

```matlab
classdef ValidationExample
    properties
        Prop (1,:) double {mustBeReal, mustBeGreaterThan(Prop, 10)} = 200;
    end
end
```

Access the meta.Validation object from the property's meta.property object. Get the validation information from the meta.Validation object properties. Collection this information into a cell array.

- Get the size information from the Size property
- Get the class name from the Class property
- Get a cell array of function handles for the validation functions from the ValidatorFunctions property.

```matlab
mc = ?ValidationExample;
mp = findobj(mc.PropertyList,'Name','Prop');
sz = mp.Validation.Size;
len = length(sz);
dim = cell(1:len);
for k = 1:len
    switch class(sz(k))
        case 'meta.FixedDimension'
```
dim{k} = sz(k).Length;
case 'meta.UnrestrictedDimension'
dim{k} = ':';
end
end
dim{end+1} = mp.Validation.Class.Name;
dim{end+1} = mp.Validation.ValidatorFunctions;

See Also
meta.Validation | meta.property

Related Examples
- “Validate Property Values” on page 8-24
- “Property Class and Size Validation” on page 8-31
- “Property Validation Functions” on page 8-39
Property Access Methods

In this section...

“Properties Provide Access to Class Data” on page 8-49
“Property Setter and Getter Methods” on page 8-50
“Set and Get Method Execution and Property Events” on page 8-52
“Access Methods and Properties Containing Arrays” on page 8-53
“Access Methods and Arrays of Objects” on page 8-53
“Modify Property Values with Access Methods” on page 8-53

Properties Provide Access to Class Data

In MATLAB, properties can have public access. Therefore, properties can provide access to data that the class design exposes to users.

Use property access methods to provide error checking or to implement side effects resulting from property access. Examples of access methods include functions that update other property values when setting the property or translate the format of a property value before returning the value.

You can use property validation to restrict the size, class, and other aspects of property values. For information on property validation, see “Validate Property Values” on page 8-24.

Performance Considerations with Access Methods

Property access methods do add the overhead of a function call whenever accessing property values. If performance-critical access to properties occurs inside methods of the class, define private properties to store values. Use these values inside methods without any error checking. For less frequent access from outside the class, define public Dependent properties that use access methods for error checking.

For information on access methods used with Dependent properties, see “Set and Get Methods for Dependent Properties” on page 8-61.
Property Setter and Getter Methods

Property access methods execute specific code whenever the property value is queried or assigned a value. These methods enable you to perform various operations:

- Execute code before assigning property values to perform actions such as:
  - Impose value range restrictions ("Validate Property Values" on page 8-24)
  - Check for proper types and dimensions
  - Provide error handling
- Execute code before returning the current values of properties to perform actions such as:
  - Calculate the value of properties that do not store values (see “Calculate Data on Demand” on page 3-23)
  - Change the value of other properties
  - Trigger events (see “Overview Events and Listeners” on page 11-2)

To control what code can access properties, see “Property Attributes” on page 8-9.

MATLAB Calls Access Methods

**Note** You cannot call property access methods directly. MATLAB calls these methods when you access property values.

Property access methods execute automatically whenever you set or query the corresponding property values from outside the access method. MATLAB does not call access methods recursively. That is, MATLAB does not call the set method when setting the property from within its set method. Similarly, MATLAB does not call the get method when querying the property value from within its get method.

Obtain the function handle for the set and get access methods from the property meta.property object. The meta.property SetMethod and GetMethod properties contain the function handles that refer to these methods.

Restrictions on Access Methods

Define property access methods only:
• For concrete properties (that is, properties that are not abstract)
• Within the class that defines the property (unless the property is abstract in that class, in which case the concrete subclass must define the access method).

MATLAB has no default set or get property access methods. Therefore, if you do not define property access methods, MATLAB software does not invoke any methods before assigning or returning property values.

Once defined, only the set and get methods can set and query the actual property values. See “When Set Method Is Called” on page 8-56 for information on cases where MATLAB does not call property set methods.

**Note** Property set and get access methods are not equivalent to user-callable set and get methods used to set and query property values from an instance of the class. See “Implement Set/Get Interface for Properties” on page 7-27 for information on user-callable set and get methods.

**Access Methods Cannot Call Functions to Access Properties**

You can set and get property values only from within your property set or get access method. You cannot call another function from the set or get method and attempt to access the property value from that function.

For example, an anonymous function that calls another function to do the actual work cannot access the property value. Similarly, an access function cannot call another function to access the property value.

**Defining Access Methods**

Access methods have special names that include the property name. Therefore, `getPropertyName` executes whenever `PropertyName` is referenced and `setPropertyName` executes whenever `PropertyName` is assigned a value.

Define property access methods in a methods block that specifies no attributes. You cannot call these methods directly. MATLAB calls these methods when any code accesses the properties.

Property access methods do not appear in the list of class methods returned by the `methods` command and are not included in the `meta.class` object `Methods` property.
Access Method Function Handles

The property `meta.property` object contains function handles to the property set and get methods. `SetMethod` contains a function handle to the set method. `GetMethod` contains a function handle to the get method.

Obtain these handles from the `meta.property` object:

```matlab
mc = ClassName;
mp = findobj(mc.PropertyList,'Name','PropertyName');
fh = mp.GetMethod;
```

For example, if the class `MyClass` defines a get method for its `Text` property, you can obtain a function handle to this function from the `meta.class` object:

```matlab
mc = MyClass;
mp = findobj(mc.PropertyList,'Name','Text');
fh = mp.GetMethod;
```

The returned value, `fh`, contains a function handle to the get method defined for the specified property name for the specified class.

For information on defining function handles, see “Create Function Handle”

Set and Get Method Execution and Property Events

MATLAB software generates events before and after set and get operations. You can use these events to inform listeners that property values have been referenced or assigned. The timing of event generation is as follows:

- **PreGet** — Triggered before calling the property get method
- **PostGet** — Triggered after the property get method has returned its value

If a class computes a property value (`Dependent = true`), then the behaviors of its set events are like the get events:

- **PreSet** — Triggered before calling the property set method
- **PostSet** — Triggered after calling the property set method

If a property is not computed (`Dependent = false`, the default), then the assignment statement with the set method generates the events:
• **PreSet** — Triggered before assigning the new property value within the set method
• **PostSet** — Triggered after assigning the new property value within the set method

For information about using property events, see “Create Property Listeners” on page 11-40.

### Access Methods and Properties Containing Arrays

You can use array indexing with properties that contain arrays without interfering with property set and get methods.

For indexed reference:

```
val = obj. PropName(n);
```

MATLAB calls the get method to get the referenced value.

For indexed assignment:

```
obj. PropName(n) = val;
```

MATLAB:

• Invokes the get method to get the property value
• Performs the indexed assignment on the returned property
• Passes the new property value to the set method

### Access Methods and Arrays of Objects

When reference or assignment occurs on an object array, MATLAB calls the set and get methods in a loop. In this loop, MATLAB always passes scalar objects to set and get methods.

### Modify Property Values with Access Methods

Property access methods are useful in cases where you want to perform some additional steps before assigning or returning a property value. For example, the Testpoint class uses a property set method to check the range of a value. It then applies scaling if it is within a particular range, and set it to NaN if it is not.
The property get methods applies a scale factor before returning its current value:

```matlab
classdef Testpoint
    properties
        expectedResult = []
    end
    properties(Constant)
        scalingFactor = 0.001
    end
    methods
        function obj = set.expectedResult(obj,erIn)
            if erIn >= 0 && erIn <= 100
                erIn = erIn.*obj.scalingFactor;
                obj.expectedResult = erIn;
            else
                obj.expectedResult = NaN;
            end
        end
        function er = get.expectedResult(obj)
            er = obj.expectedResult/obj.scalingFactor;
        end
    end
end
```

See Also

More About

- “Properties Containing Objects” on page 8-65
Property Set Methods

Overview of Property Access Methods

For an overview of property access methods, see “Property Access Methods” on page 8-49

Property Set Method Syntax

MATLAB calls a property's set method whenever a value is assigned to the property.

Note You cannot call property access methods directly. MATLAB calls these methods when you access property values.

Property set methods have the following syntax, where PropertyName is the name of the property.

For a value class:

methods
  function obj = set.PropertyName(obj,value)
  ...
end

• obj — Object whose property is being assigned a value
• value — The new value that is assigned to the property

Value class set functions must return the modified object to the calling function. Handle classes do not need to return the modified object.

For a handle class:
methods
    function set.PropertyName(obj,value)
        ...
    end

Use default method attributes for property set methods. The methods block defining the
set method cannot specify attributes.

**Validate Property Set Value**

Use the property set method to validate the value being assigned to the property. The
property set method can perform actions like error checking on the input value before
taking whatever action is necessary to store the new property value.

classdef MyClass
    properties
        Prop1
    end
    methods
        function obj = set.Prop1(obj,value)
            if (value > 0)
                obj.Prop1 = value;
            else
                error('Property value must be positive')
            end
        end
    end
end

For an example of a property set method, see “Restrict Properties to Specific Values” on
page 3-21.

**When Set Method Is Called**

If a property set method exists, MATLAB calls it whenever a value is assigned to that
property. However, MATLAB does NOT call property set methods in the following cases:

- A value is assigned to a property from within its own property set method, to prevent
  recursive calling of the set method. However, property assignments made from
  functions called by a set method do call the set method.
- MATLAB assigns a default value to the property during initialization of an object
  before calling object constructor functions.
• When MATLAB copies a value object (any object that is not a handle), MATLAB does not call the set or get method when copying property values from one object to another.

• Any assignment made to a property value that is the same as the current value when the property's AbortSet attribute is true. See “Assignment When Property Value Unchanged” on page 11-44 for more information on this attribute.

**Setting Property Value in Constructor**

Setting a property value in the constructor causes the property set method to be called. For example, the PropertySetMethod class defines a property set method for the Prop1 property.

```matlab
classdef PropertySetMethod
    properties
        Prop1 = "Default String"
    end
    methods
        function obj = PropertySetMethod( str )
            if nargin > 0
                obj.Prop1 = str;
            end
        end

        function obj = set.Prop1(obj,str)
            obj.Prop1 = str;
            fprintf( 'set.Prop1 method called. Prop1 = %s\n', obj.Prop1 );
        end
    end
end
```

If you call the class constructor with no input arguments, MATLAB does not call the set.Prop1 method.

```matlab
>> o = PropertySetMethod

o =

    PropertySetMethod with properties:

        Prop1: "Default String"
```
Setting the property value in the constructor results in a call to the property set method.

```matlab
>> o = PropertySetMethod("New string")
```

> set.Propl method called. Propl = New string

```matlab
o =

    PropertySetMethod with properties:

        Propl: "New string"
```

If you copy the object to another variable, MATLAB does not call the property set method even though the right side object in the assignment uses a nondefault value for the property:

```matlab
a = o;
a.Propl
```

```matlab
a.Propl
```

```matlab
ans =

    "New String"
```

### See Also

### Related Examples

- “Property Get Methods” on page 8-59
- “Property Access Methods” on page 8-20
- “Validate Property Values” on page 8-24
Property Get Methods

Overview of Property Access Methods

For an overview of property access methods, see “Property Access Methods” on page 8-49.

Property Get Method Syntax

MATLAB calls a property's get method whenever the property value is queried.

Note You cannot call property access methods directly. MATLAB calls these methods when you access property values.

Property get methods have the following syntax, where PropertyName is the name of the property. The function must return the property value.

```matlab
methods
    function value = get.PropertyName(obj)
    ...
end
```

Calculate Value for Dependent Property

The SquareArea class defines a dependent property Area. MATLAB does not store a value for the dependent Area property. When you query the value of the Area property, MATLAB calls the get.Area method calculates the value based on the Width and Height properties.
% classdef SquareArea
properties
    Width
    Height
end
properties (Dependent)
    Area
end
methods
    function a = get.Area(obj)
        a = obj.Width * obj.Height;
    end
end

Errors Not Returned from Get Method

The MATLAB default object display suppresses error messages returned from property get methods. MATLAB does not allow an error issued by a property get method to prevent the display of the entire object.

Use the property set method to validate the property value. Validating the value when setting a property ensures that the object is in a valid state. Use the property get method only to return the value that the set method has validated.

Get Method Behavior

MATLAB does NOT call property get methods in the following cases:

- Getting a property value from within its own property get method, which prevents recursive calling of the get method
- Copying a value object (that is, not derived from the handle class). Neither the set or get method is called when copying property values from one object to another.

See Also

Related Examples

- “Set and Get Methods for Dependent Properties” on page 8-61
Set and Get Methods for Dependent Properties

In this section...

“Calculate Dependent Property Value” on page 8-62
“When to Use Set Methods with Dependent Properties” on page 8-62
“Private Set Access with Dependent Properties” on page 8-63

Dependent properties do not store data. The value of a dependent property depends on some other value, such as the value of a nondependent property.

Dependent properties must define get-access methods (get.\textit{PropertyName}) to determine a value for the property when the property is queried.

The values returned by dependent property get methods are not considered when testing for object equality using \texttt{isequal} and \texttt{isequaln}.

To be able to set the value of a dependent property, the property must define a set access method (set.\textit{PropertyName}). The property set access method usually assigns the value to another, nondependent property for storage of the value.

For example, the Account class returns a value for the dependent Balance property that depends on the value of the Currency property. The get.Balance method queries the Currency property before calculating a value for the Balance property.

MATLAB calls the get.Balance method when the Balance property is queried. You cannot call get.Balance explicitly.

Here is a partial listing of the class showing a dependent property and its get method:

```matlab
classdef Account
    properties
        Currency
        DollarAmount
    end
    properties (Dependent)
        Balance
    end
...,
methods
    function value = get.Balance(obj)
        c = obj.Currency;
```
switch c
    case 'E'
        v = obj.DollarAmount / 1.1;
    case 'P'
        v = obj.DollarAmount / 1.5;
    otherwise
        v = obj.DollarAmount;
    end
end
format bank
value = v;
end
end

Calculate Dependent Property Value

One application of a property get method is to determine the value of a property only when you need it, and avoid storing the value. To use this approach, set the property Dependent attribute to true:

properties (Dependent = true)
    Prop
end

The get method for the Prop property determines the value of that property and assigns it to the object from within the method:

function value = get.Prop(obj)
    value = calculateValue;
    ...
end

This get method calls a function or static method called calculateValue to calculate the property value and returns value as a result. The property get method can take whatever action is necessary within the method to produce the output value.

For an example of a property get method, see “Calculate Data on Demand” on page 3-23.

When to Use Set Methods with Dependent Properties

Although a dependent property does not store its value, you can define a set method for a dependent property to enable code to set the property.
For example, suppose that you have a class that changes the name of a property from OldPropName to NewPropName. You can continue to allow the use of the old name without exposing it to new users. To support the old property name, define OldPropName a dependent property with set and get methods:

```matlab
properties
    NewPropName
end
properties (Dependent, Hidden)
    OldPropName
end
methods
    function obj = set.OldPropName(obj,val)
        obj.NewPropName = val;
    end
    function value = get.OldPropName(obj)
        value = obj.NewPropName;
    end
end
```

There is no memory wasted by storing both old and new property values. Code that accesses OldPropName continues to work as expected. Setting the Hidden attribute of OldPropName prevents new users from seeing the property.

Assignments made from property set methods cause the execution of any set methods defined for properties being set. See “Calculate Data on Demand” on page 3-23 for an example.

**Private Set Access with Dependent Properties**

If you use a dependent property only to return a value, then do not define a set access method for the dependent property. Instead, set the SetAccess attribute of the dependent property to private. For example, consider the following get method for the MaxValue property:

```matlab
methods
    function mval = get.MaxValue(obj)
        mval = max(obj.BigArray(:));
    end
end
```

This example uses the MaxValue property to return a value that it calculates only when queried. For this application, define the MaxValue property as dependent and private:
properties (Dependent, SetAccess = private)
    MaxValue
end

See Also

Related Examples

• “Property Attributes” on page 8-9
Properties Containing Objects

In this section...

“Assigning Objects as Default Property Values” on page 8-65
“Assigning to Read-Only Properties Containing Objects” on page 8-65
“Assignment Behavior” on page 8-65

Assigning Objects as Default Property Values

MATLAB evaluates property default values only once when loading the class. MATLAB does not reevaluate the assignment each time you create an object of that class. If you assign an object as a default property value in the class definition, MATLAB calls the constructor for that object only once when loading the class.

Note  Evaluation of property default values occurs only when the value is first needed, and only once when MATLAB first initializes the class. MATLAB does not reevaluate the expression each time you create an instance of the class.

For more information on the evaluation of expressions that you assign as property default values, see “When MATLAB Evaluates Expressions” on page 6-13.

Assigning to Read-Only Properties Containing Objects

When a class defines a property with private or protected SetAccess, and that property contains an object which itself has properties, assignment behavior depends on whether the property contains a handle or a value object:

- Handle object - you can set properties on handle objects contained in read-only properties
- Value object - you cannot set properties on value object contained in read-only properties.

Assignment Behavior

These classes illustrate the assignment behavior:
• **ReadOnlyProps** – class with two read-only properties. The class constructor assigns a handle object of type HanClass to the PropHandle property and a value object of type ValClass to the PropValue property.

• **HanClass** – handle class with public property

• **ValClass** – value class with public property

```matlab
classdef ReadOnlyProps
    properties(SetAccess = private)
        PropHandle
        PropValue
    end
    methods
        function obj = ReadOnlyProps
            obj.PropHandle = HanClass;
            obj.PropValue = ValClass;
        end
    end
end

classdef HanClass < handle
    properties
        Hprop
    end
end

classdef ValClass
    properties
        Vprop
    end
end
```

Create an instance of the ReadOnlyProps class:

```matlab
a = ReadOnlyProps
```

Create an instance of the ReadOnlyProps class:

```matlab
a =
```

```matlab
ReadOnlyProps with properties:

    PropHandle: [1x1 HanClass]
    PropValue: [1x1 ValClass]
```

Use the private PropHandle property to set the property of the HanClass object it contains:
class(a.PropHandle.Hprop)
ans =
double
a.PropHandle.Hprop = 7;

Attempting to make an assignment to the value class object property is not allowed:

a.PropValue.Vprop = 11;

You cannot set the read-only property 'PropValue' of ReadOnlyProps.

See Also

More About

• “Mutable and Immutable Properties” on page 8-22
What Are Dynamic Properties

You can add properties to instances of classes that derive from the `dynamicprops` class. These dynamic properties are sometimes referred to as instance properties. Use dynamic properties to attach temporary data to objects or to assign data that you want to associate with an instance of a class, but not all objects of that class.

It is possible for more than one program to define dynamic properties on the same object. In these cases, avoid name conflicts. Dynamic property names must be valid MATLAB identifiers (see “Variable Names”) and cannot be the same name as a method of the class.

Characteristics of Dynamic Properties

Once defined, dynamic properties behave much like class-defined properties:

- Set and query the values of dynamic properties using dot notation. (See “Assign Data to the Dynamic Property” on page 8-70.)
- MATLAB saves and loads dynamic properties when you save and load the objects to which they are attached. (See “Dynamic Properties and ConstructOnLoad” on page 8-79.)
- Define attributes for dynamic property. (See “Set Dynamic Property Attributes” on page 8-69).
- MATLAB does not save property attribute values when you save objects that have dynamic properties. Therefore, any nondefault attribute settings are not retained when you load the object. (See “Save and Load Dynamic Properties” on page 13-39.)
- By default, dynamic properties have their `NonCopyable` attribute set to `true`. If you copy an object containing a dynamic property, the dynamic property is not copied. (See “Objects with Dynamic Properties” on page 7-40)
- Add property set and get access methods. (See “Set and Get Methods for Dependent Properties” on page 8-61.)
- Listen for dynamic property events. (See “Dynamic Property Events” on page 8-74.)
• Access dynamic property values from object arrays, with restricted syntax. (See “Accessing Dynamic Properties in Arrays” on page 10-14.)
• The isequal function always returns false when comparing objects that have dynamic properties, even if the properties have the same name and value. To compare objects that contain dynamic properties, overload isequal for your class.

Define Dynamic Properties

Any class that is a subclass of the dynamicprops class (which is itself a subclass of the handle class) can define dynamic properties using the addprop method. The syntax is:

P = addprop(H,'PropertyName')

where:

P is an array of meta.DynamicProperty objects
H is an array of handles

PropertyName is the name of the dynamic property you are adding to each object

Name Dynamic Properties

Use only valid names when naming dynamic properties (see “Variable Names”). In addition, do not use names that:

• Are the same as the name of a class method
• Are the same as the name of a class event
• Contain a period (.)

Set Dynamic Property Attributes

To set property attributes, use the meta.DynamicProperty object associated with the dynamic property. For example, if P is the object returned by addprop, this statement sets the property’s Hidden attribute to true:

P.Hidden = true;

The property attributes Constant and Abstract have no meaning for dynamic properties. Setting the value of these attributes to true has no effect.
**Remove a Dynamic Property**

Remove the dynamic property by deleting its `meta.DynamicProperty` object:

```matlab
delete(P);
```

**Assign Data to the Dynamic Property**

Suppose, you are using a predefined set of user interface widget classes (buttons, sliders, check boxes, etc.). You want to store the location of each instance of the widget class. Assume that the widget classes are not designed to store location data for your particular layout scheme. You want to avoid creating a map or hash table to maintain this information separately.

Assuming the `button` class is a subclass of `dynamicprops`, add a dynamic property to store your layout data. Here is a simple class to create a `uicontrol` button:

```matlab
classdef button < dynamicprops
    properties
        UiHandle
    end
    methods
        function obj = button(pos)
            if nargin > 0
                if length(pos) == 4
                    obj.UiHandle = uicontrol('Position',pos,'Style','pushbutton');
                else
                    error('Improper position')
                end
            end
        end
    end
end
```

Create an instance of the `button` class, add a dynamic property, and set its value:

```matlab
b1 = button([20 40 80 20]);
b1.addprop('myCoord');
b1.myCoord = [2,3];
```

Access the dynamic property just like any other property, but only on the object on which you defined it:

```matlab
b1.myCoord
```
Access Attribute for Dynamic Properties

Using nonpublic Access with dynamic properties is not recommended because these properties belong to specific instances that are often created outside of class methods. The Access attribute of a dynamic property applies to the class of the instance that contains the dynamic property. The dynamic property Access attribute does not necessarily apply to the class whose method adds the dynamic property.

For example, if a base class method adds a dynamic property with private access to an instance, the private access applies only to the class of the instance.

For more information on dynamic property attributes, see meta.DynamicProperty. Use the handle findprop method to get the meta.DynamicProperty object.

See Also

Related Examples

• “Set and Get Methods for Dynamic Properties” on page 8-72
• “Dynamic Property Events” on page 8-74
• “Dynamic Properties and ConstructOnLoad” on page 8-79
Set and Get Methods for Dynamic Properties

In this section...

“Create Access Methods for Dynamic Properties” on page 8-72
“Shared Set and Get Methods” on page 8-73

You can define property set access or get access methods for dynamic properties without creating additional class methods. For general information on the use of access methods, see “Property Access Methods” on page 8-49.

Create Access Methods for Dynamic Properties

Use these steps to create a property access method:

- Define a function that implements the operations you want to perform before the property set or get occurs. These methods must have the following signatures:
  
  mySet(obj, val) or val = myGet(obj)

- Obtain the dynamic property's corresponding meta.DynamicProperty object.

- Assign a function handle referencing your set or get property function to the meta.DynamicProperty object's GetMethod or SetMethod property. This function does not need to be a method of the class. You cannot use a naming scheme like set.PropertyName. Instead, use any other valid function name.

Suppose that you want to create a property set function for the myCoord dynamic property of the button class created in “Define Dynamic Properties” on page 8-69.

Write the function as follows.

```matlab
function set_myCoord(obj, val)
    if ~ (length(val) == 2)
        error('myCoords require two values')
    end
    obj.myCoord = val;
end
```

Because button is a handle class, the property set function does not need to return the object as an output argument.

To get the meta.DynamicProperty object, use the handle class findprop method:
mb1 = b1.findprop('myCoord');
mb1.SetMethod = @set_myCoord;

MATLAB calls the property set function whenever you set this property:

b1.myCoord = [1 2 3] % length must be two

Error using button.set_myCoord
myCoords require two values

You can set and get the property values only from within your property access methods. You cannot call another function from the set or get method, and then attempt to access the property value from that function.

**Shared Set and Get Methods**

You can assign the same function handle for the set or get method of multiple dynamic properties. MATLAB passes only the object and the value to the assigned set function.

Reference or assignment to a property from within its set or get method does not invoke the set or get method again. Therefore, if you use a handle to the same function for multiple dynamic properties, that function is not invoked when accessing any of those properties from within that function.

**See Also**

**Related Examples**

- “Dynamic Properties — Adding Properties to an Instance” on page 8-68
Dynamic Property Events

Dynamic Properties and Ordinary Property Events

Dynamic properties support property set and get events so you can define listeners for these properties. Listeners are bound to the particular dynamic property for which they are defined.

If you delete a dynamic property, and then create another dynamic property with the same name, the listeners do not respond to events generated by the new property. A listener defined for a dynamic property that has been deleted does not cause an error, but the listener callback is never executed.

“Property-Set and Query Events” on page 11-17 provides more information on how to define listeners for these events.

Dynamic-Property Events

To respond to the addition and removal of dynamic properties, attach listeners to objects containing the dynamic properties. The dynamicprops class defines events for this purpose:

• PropertyAdded — Triggered when you add a dynamic property to an object derived from the dynamicprops class.

• PropertyRemoved — Triggered when you delete the object or the meta.DynamicProperty object associated with a dynamic property.

• ObjectBeingDestroyed — Triggered when the object is destroyed. This event is inherited from the handle class.
These events have public listen access (ListenAccess attribute) and private notify access (NotifyAccess attribute).

The PropertyAdded and PropertyRemoved events pass an event.DynamicPropertyEventArgs object to listener callbacks. The event data object has three properties:

- **PropertyName** — Name of the dynamic property that is added or removed
- **Source** — Handle to the object that is the source of the event
- **EventName** — Name of the event (PropertyAdded, PropertyRemoved, or ObjectBeingDestroyed)

### Listen for a Specific Property Name

Suppose that you have an application that creates a dynamic property under certain conditions. You want to:

- Set the value of a hidden property to true when a property named SpecialProp is added.
- Set the value of the hidden property to false when SpecialProp is removed.

Use the event.DynamicPropertyEventArgs event data to determine the name of the property and whether it is added or deleted.

The DynamTest class derives from dynamicprops. It defines a hidden property, HiddenProp.

```matlab
classdef DynamTest < dynamicprops
    properties (Hidden)
    HiddenProp
end
end
```

Define a callback function that uses the EventName property of the event data to determine if a property is added or removed. Obtain the name of the property from the PropertyName property of the event data. If a dynamic property is named SpecialProp, change the value of the hidden property.

```matlab
function DyPropEvtCb(src,evt)
    switch evt.EventName
        case 'PropertyAdded'
            ...
        case 'PropertyRemoved'
            ...
    end
end
```
switch evt.PropertyName
    case 'SpecialProp'
        % Take action based on the addition of this property
        %...
        %...
        src.HiddenProp = true;
        disp('SpecialProp added')
    otherwise
        % Other property added
        %...
        disp([evt.PropertyName,' added'])
end

case 'PropertyRemoved'
    switch evt.PropertyName
        case 'SpecialProp'
            % Take action based on the removal of this property
            %...
            %...
            src.HiddenProp = false;
            disp('SpecialProp removed')
        otherwise
            % Other property removed
            %...
            disp([evt.PropertyName,' removed'])
    end
end
end

Create an object of the DynamTest class.

dt = DynamTest;

Add a listener for both PropertyAdded and PropertyRemoved events.

lad = addlistener(dt,'PropertyAdded',@DyPropEvtCb);
lrm = addlistener(dt,'PropertyRemoved',@DyPropEvtCb);

**PropertyAdded Event Callback Execution**

Adding a dynamic property triggers the PropertyAdded event. This statement adds a
dynamic property to the object and saves the returned meta.DynamicProperty object.

ad = addprop(dt,'SpecialProp');
The addition of the dynamic property causes the listener to execute its callback function, DyPropEvtCb. The callback function assigns a value of true to the HiddenProp property.

dt.HiddenProp
ans =
    1

**PropertyRemoved Event Callback Execution**

Remove a dynamic property by calling `delete` on the `meta.DynamicProperty` object that is returned by the `addprop` method. Removing the `meta.DynamicProperty` object triggers the `PropertyRemoved` event.

Delete the `meta.DynamicProperty` object returned when adding the dynamic property SpecialProp.

delete(ad)

The callback executes:

SpecialProp removed

The value of HiddenProp is now false.

dt.HiddenProp
ans =
    0

**How to Find meta.DynamicProperty Objects**

You can obtain the `meta.DynamicProperty` object for a dynamic property using `findprop`. Use `findprop` if you do not have the object returned by `addprop`.

ad = findprop(dt,'SpecialProp');
See Also

Related Examples

- “Dynamic Properties — Adding Properties to an Instance” on page 8-68
Dynamic Properties and ConstructOnLoad

Setting the class ConstructOnLoad attribute to true causes MATLAB to call the class constructor when loading the class. MATLAB saves and restores dynamic properties when loading an object.

If you create dynamic properties from the class constructor, you can cause a conflict if you also set the class ConstructOnLoad attribute to true. Here is the sequence:

- A saved object saves the names and values of properties, including dynamic properties
- When loaded, a new object is created and all properties are restored to the values at the time the object was saved
- Then, the ConstructOnLoad attribute causes a call to the class constructor, which would create another dynamic property with the same name as the loaded property. See “Save and Load Objects” on page 13-2 for more on the load sequence.
- MATLAB prevents a conflict by loading the saved dynamic property, and does not execute addprop when calling the constructor.

If you use ConstructOnLoad, add dynamic properties from the class constructor, and want the constructor to call addprop at load time, then set the dynamic property Transient attribute to true. This setting prevents the property from being saved. For example:

```matlab
classdef (ConstructOnLoad) MyClass < dynamicprops
    function obj = MyClass
        P = addprop(obj,'DynProp');
        P.Transient = true;
        ...
    end
end
```

See Also

Related Examples
- “Dynamic Properties — Adding Properties to an Instance” on page 8-68
Methods — Defining Class Operations

- “Methods in Class Design” on page 9-2
- “Method Attributes” on page 9-5
- “Ordinary Methods” on page 9-8
- “Methods in Separate Files” on page 9-11
- “Method Invocation” on page 9-15
- “Class Constructor Methods” on page 9-21
- “Static Methods” on page 9-31
- “Overload Functions in Class Definitions” on page 9-33
- “Class Support for Array-Creation Functions” on page 9-37
- “Object Precedence in Method Invocation” on page 9-46
- “Dominant Argument in Overloaded Graphics Functions” on page 9-48
- “Class Methods for Graphics Callbacks” on page 9-51
Methods in Class Design

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Class Methods” on page 9-2</td>
</tr>
<tr>
<td>“Examples and Syntax” on page 9-2</td>
</tr>
<tr>
<td>“Kinds of Methods” on page 9-3</td>
</tr>
<tr>
<td>“Method Naming” on page 9-3</td>
</tr>
</tbody>
</table>

**Class Methods**

*Methods* are functions that implement the operations performed on objects of a class. Methods, along with other class members support the concept of encapsulation—class instances contain data in properties and class methods operate on that data. This design allows the internal workings of classes to be hidden from code outside of the class, and thereby enabling the class implementation to change without affecting code that is external to the class.

Methods have access to private members of their class including other methods and properties. This encapsulation enables you to hide data and create special interfaces that must be used to access the data stored in objects.

**Examples and Syntax**

For an example to get started writing classes, see “Create a Simple Class” on page 2-2

For sample code and syntax, see “Methods and Functions” on page 5-15

For a discussion of how to create classes that modify standard MATLAB behavior, see “Methods That Modify Default Behavior” on page 17-2.

For information on the use of @ and path directors and packages to organize your class files, see “Class Files and Folders” on page 5-2

For the syntax to use when defining classes in more than one file, see “Methods in Separate Files” on page 9-11
Kinds of Methods

There are specialized kinds of methods that perform certain functions or behave in particular ways:

- **Ordinary methods** are functions that act on one or more objects and return some new object or some computed value. These methods are like ordinary MATLAB functions that cannot modify input arguments. Ordinary methods enable classes to implement arithmetic operators and computational functions. These methods require an object of the class on which to operate. See “Ordinary Methods” on page 9-8.

- **Constructor methods** are specialized methods that create objects of the class. A constructor method must have the same name as the class and typically initializes property values with data obtained from input arguments. The class constructor method must return the object it creates. See “Class Constructor Methods” on page 9-21.

- **Destructor methods** are called automatically when the object is destroyed, for example if you call `delete(object)` or there are no longer any references to the object. See “Handle Class Destructor” on page 7-16.

- **Property access methods** enable a class to define code to execute whenever a property value is queried or set. See “Property Access Methods” on page 8-49.

- **Static methods** are functions that are associated with a class, but do not necessarily operate on class objects. These methods do not require an instance of the class to be referenced during invocation of the method, but typically perform operations in a way specific to the class. See “Static Methods” on page 9-31.

- **Conversion methods** are overloaded constructor methods from other classes that enable your class to convert its own objects to the class of the overloaded constructor. For example, if your class implements a `double` method, then this method is called instead of the double class constructor to convert your class object to a MATLAB double object. See “Object Converters” on page 17-12 for more information.

- **Abstract methods** define a class that cannot be instantiated itself, but serves as a way to define a common interface used by numerous subclasses. Classes that contain abstract methods are often referred to as interfaces. See “Abstract Classes” on page 12-91 for more information and examples.

Method Naming

The name of a function that implements a method can contain dots (for example, `set.PropertyName`) only if the method is one of the following:
• Property set/get access method (see “Property Access Methods” on page 8-49)
• Conversion method that converts to a package-qualified class, which requires the use of the package name (see “Packages Create Namespaces” on page 6-25)

You cannot define property access or conversion methods as local functions, nested functions, or separately in their own files. Class constructors and package-scoped functions must use the unqualified name in the function definition; do not include the package name in the function definition statement.

See Also

Related Examples
• “Method Attributes” on page 9-5
• “Rules for Naming to Avoid Conflicts” on page 9-35
Method Attributes

In this section...

“Purpose of Method Attributes” on page 9-5
“Specifying Method Attributes” on page 9-5
“Table of Method Attributes” on page 9-5

Purpose of Method Attributes

Specifying attributes in the class definition enables you to customize the behavior of methods for specific purposes. Control characteristics like access, visibility, and implementation by setting method attributes. Subclasses do not inherit superclass member attributes.

Specifying Method Attributes

Assign method attributes on the same line as the methods keyword:

methods (Attribute1 = value1, Attribute2 = value2,...)

... end

For more information on attribute syntax, see “Attribute Specification” on page 5-22.

Table of Method Attributes

Attributes enable you to modify the behavior of methods. All methods support the attributes listed in the following table.

Attribute values apply to all methods defined within the methods...end code block that specifies the nondefault values.
## Method Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
</table>
| Abstract      | logical Default = false | If true, the method has no implementation. The method has a syntax line that can include arguments, which subclasses use when implementing the method:  
- Subclasses are not required to define the same number of input and output arguments. However, subclasses generally use the same signature when implementing their version of the method.  
- The method can have comments after the function line.  
- The method does not contain function or end keywords, only the function syntax (e.g., \([a,b] = \text{myMethod}(x,y)\)) |
| Access        | • enumeration, default = public  
• meta.class object  
• cell array of meta.class objects | Determines what code can call this method:  
- public — Unrestricted access  
- protected — Access from methods in class or subclasses  
- private — Access by class methods only (not from subclasses)  
- List classes that have access to this method. Specify classes as meta.class objects in the form:  
  - A single meta.class object  
  - A cell array of meta.class objects. An empty cell array, \(\{}\), is the same as private access.  
  See “Class Members Access” on page 12-28 |
<p>| Hidden        | logical Default = false | When false, the method name shows in the list of methods displayed using the methods or methodsviwer commands. If set to true, the method name is not included in these listings and ismethod does not return true for this method name. |
| Sealed        | logical Default = false | If true, the method cannot be redefined in a subclass. Attempting to define a method with the same name in a subclass causes an error. |</p>
<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
</table>
| Static        | logical Default = false | Specify as true to define a method that does not depend on an object of the class and does not require an object argument. Use the class name to call the method: \( \text{classname}.\text{methodname} \) or an instance of the class: \( \text{obj}.\text{methodname} \)  

“Static Methods” on page 9-31 provides more information.
Ordinary Methods

Ordinary Methods Operate on Objects

Ordinary methods define functions that operate on objects of the class. Therefore, one of the input arguments must be an object or array of objects of the defining class. These methods can compute values based on object data, can overload MATLAB built-in functions, and can call other methods and functions. Ordinary methods can return modified objects.

Methods Inside classdef Block

This example shows the definition of a method \( \text{methodName} \) within the \text{classdef} and \text{methods} blocks:

\[
\text{classdef ClassName}
\]
\[
\quad \text{methods (AttributeName = value,...)}
\]
\[
\quad \quad \text{function methodName(obj,args)}
\]
\[
\quad \quad \quad \% \text{method code}
\]
\[
\quad \quad \quad \ldots
\]
\[
\quad \quad \quad \text{end}
\]
\[
\quad \quad \quad \ldots
\]
\[
\quad \quad \quad \text{end} \% \text{end of method block}
\]
\[
\quad \quad \ldots
\]
\[
\quad \text{end}
\]

Method attributes apply only to that particular methods block, which is terminated by the \text{end} statement.

\textbf{Note} Nonstatic methods must include an explicit object variable as a function argument. The MATLAB language does not support an implicit reference in the method function definition.
Sample Method

The `addData` method adds a value to the `Data` property of `MyData` objects. The `Data` property has a default value of 0.

```matlab
classdef MyData
    properties
        Data = 0
    end
    methods
        function obj = addData(obj,val)
            newData = obj.Data + val;
            obj.Data = newData;
        end
    end
end
```

```matlab
a = MyData;
a = addData(a,75)
a =
    MyData with properties:
        Data: 75
```

Calling Methods

Either of the following statements is correct syntax for calling a method, where `obj` is an object of the class defining the `methodName` method:

```matlab
obj.methodName(arg)
methodName(arg)
```

Method Files

You can define methods:

- Inside the class definition block
- In a separate file in the class folder (that is, `@ClassName` folder)

For more information on class folders, see “Folders Containing Class Definitions” on page 6-17.
See Also

More About

- “Methods in Separate Files” on page 9-11
- “Determining Which Method Is Invoked” on page 9-15
- “Operator Overloading” on page 17-47
Methods in Separate Files

In this section...

“Class Folders” on page 9-11
“Define Method in Function File” on page 9-12
“Specify Method Attributes in classdef File” on page 9-12
“Methods You Must Define in the classdef File” on page 9-13

Class Folders

You can define class methods in files that are separate from the class definition file, with certain exceptions (see “Methods You Must Define in the classdef File” on page 9-13).

To use multiple files for class definitions, put the class files in a folder having a name beginning with the @ character followed by the name of the class (this is called a class folder). Ensure that the parent folder of the class folder is on the MATLAB path.

If the class folder is contained in one or more package folders, then the top-level package folder must be on the MATLAB path.

For example, the folder @MyClass must contain the file MyClass.m (which contains the classdef block) and can contain other methods and function defined in files having a .m extension. The folder @MyClass can contain a number of files:

@MyClass/MyClass.m
@MyClass/subsref.m
@MyClass/subsasgn.m
@MyClass/horzcat.m
@MyClass/vertcat.m
@MyClass/myFunc.m

Note MATLAB treats any .m file in the class folder as a method of the class. The base name of the file must be a valid MATLAB function name. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.
Define Method in Function File

To define a method in a separate file in the class folder, create the function in a file with the .m extension. Do not use the method-end keywords in that file. Name the file with the function name, as with any function.

In the myFunc.m file, implement the method:

```matlab
function output = myFunc(obj,arg1,arg2)
    ...% code here
end
```

It is a good practice to declare the function signature in the classdef file in a methods block:

```matlab
classdef MyClass
    methods
        output = myFunc(obj,arg1,arg2)
        end
    end
end
```

Specify Method Attributes in classdef File

If you specify method attributes for a method that you define in a separate function file, include the method signature in a methods block in the classdef file. This methods block specifies the attributes that apply to the method.

For example, the following code shows a method with Access set to private in the methods block. The method implementation resides in a separate file. Do not include the function or end keywords in the methods block. Include only the function signature showing input and output arguments.

```matlab
classdef MyClass
    methods (Access = private)
        output = myFunc(obj,arg1,arg2)
        end
    end
end
```

In a file named myFunc.m, in the @MyClass folder, define the function:
function output = myFunc(obj, arg1, arg2)
...
end

Static Methods in Separate Files

To create a static method, set the method Static attribute to true and list the function signature in a static methods block in the classdef file. Include the input and output arguments with the function name. For example:

classdef MyClass
...
    methods (Static)
        output = staticFunc1(arg1, arg2)
        staticFunc2
    end
...
end

Define the functions in separate files using the same function signature. For example, in the file @MyClass/staticFunc1.m:

function output = staticFunc1(arg1, arg2)
...
end

and in @MyClass/staticFunc2.m:

function staticFunc2
...
end

Methods You Must Define in the classdef File

Define the following methods in the classdef file. You cannot define these methods in separate files:

- Class constructor
- All functions that use dots in their names, including:
  - Converter methods that must use the package name as part of the class name because the class is contained in packages
• Property set and get access methods

Related Information
• “Converters for Package Classes” on page 17-12
• “Property Access Methods” on page 8-49

See Also

Related Examples
• “Folders Containing Class Definitions” on page 6-17
Method Invocation

Determining Which Method Is Invoked

When MATLAB invokes an ordinary method that has an argument list, it uses the following criteria to determine which method to call:

- The class of the leftmost argument whose class is not specified as inferior to any other argument's class is chosen as the dominant class and its method is invoked.
- If this class does not define the named method, then a function with that name on the MATLAB path is invoked.
- If no such function exists, MATLAB issues an error indicating that the dominant class does not define the named method.

Dominant Argument

The dominant argument in a method's argument list determines which version of the method or function that the MATLAB runtime calls. Dominance is determined by the relative precedences of the classes of the arguments. In general, user-defined classes take precedence over built-in MATLAB classes. Therefore, the leftmost argument determines which method to call. However, user-defined classes can specify the relative dominance of specific classes.

For example, suppose classA defines classB as inferior and suppose that both classes define a method called combine.

Calling the method with an object of classB and classA:

\[ \text{combine}(B, A) \]

actually calls the combine method of classA because A is the dominant argument.
Dot Notation vs. Function Notation

MATLAB classes support both function and dot notation syntax for calling methods. For example, if `setColor` is a method of the class of object `X`, then calling `setColor` with function notation would be:

```matlab
X = setColor(X,'red');
```

The equivalent method call using dot notation is:

```matlab
X = X.setColor('red')
```

However, in certain cases, the results for dot notation can differ with respect to how MATLAB dispatching works:

- If there is an overloaded `subsref`, it is invoked whenever using dot notation. That is, the statement is first tested to see if it is subscripted assignment.
- If there is no overloaded `subsref`, then `setColor` must be a method of `X`. An ordinary function or a class constructor is never called using this notation.
- Only the argument `X` (to the left of the dot) is used for dispatching. No other arguments, even if dominant, are considered. Therefore dot notation can call only methods of `X`; methods of other argument are never called.

Case Where Result Is Different

Here is an example of a case where dot and function notation can give different results. Suppose that you have the following classes:

- `classA` defines a method called `methodA` that requires an object of `classB` as one of its arguments
- `classB` defines `classA` as inferior to `classB`

```matlab
classdef (InferiorClasses = {?classA}) classB
    ...
end
```

The `methodA` method is defined with two input arguments, one of which is an object of `classB`:

```matlab
classdef classA
    methods
        function methodA(obj,obj_classB)
            ...
```

9-16
classB does not define a method with the same name as methodA. Therefore, the following syntax causes the MATLAB runtime to search the path for a function with the same name as methodA because the second argument is an object of a dominant class. If a function with that name exists on the path, then MATLAB attempts to call this function instead of the method of classA and most likely returns a syntax error.

obj = classA(...);
methodA(obj,obj_classB)

Dot notation is stricter in its behavior. For example, this call to methodA:

obj = classA(...);
obj.methodA(obj_classB)

can call only methodA of the class of obj.

**Referencing Names with Expressions—Dynamic Reference**

You can reference an object’s properties or methods using an expression in dot-parentheses syntax:

obj.(expression)

The expression must evaluate to a char vector that is the name of a property or a method. For example, the following statements are equivalent:

obj.Property1
obj.(’Property1’)

In this case, obj is an object of a class that defines a property called Property1. Therefore, you can pass a char variable in the parentheses to reference to property:

propName = ’Property1’;
obj.(propName)

You can call a method and pass input arguments to the method using another set of parentheses:

obj.(expression)(arg1,arg2,...)
Using this notation, you can make dynamic references to properties and methods in the same way you can create dynamic references to the fields of structs.

As an example, suppose that an object has methods corresponding to each day of the week. These methods have the same names as the days of the week (Monday, Tuesday, and so on). Also, the methods take as char vector input arguments, the current day of the month (the date). Now suppose that you write a function in which you want to call the correct method for the current day.

Use an expression created with the date and datestr functions:

\[
\text{obj.}(\text{datestr(date,}'ddd\text{'})\text{(datestr(date,}'dd\text{'})}
\]

The expression datestr(date, 'ddd') returns the current day as a char vector. For example:

\[
\text{datestr(date,}'ddd\text{'})
\]

\[
\text{ans =}
\]

Tuesday

The expression datestr(date, 'dd') returns the current date as a char vector. For example:

\[
\text{datestr(date,}'dd\text{'})
\]

\[
\text{ans =}
\]

11

Therefore, the expression using dot-parentheses (called on Tuesday the 11th) is the equivalent of:

\[
\text{obj.Tuesday('11')}
\]

**Controlling Access to Methods**

There can be situations where you want to create methods for internal computation within the class, but do not want to publish these methods as part of the public interface to the class. In these cases, you can use the Access attribute to set the access to one of the following options:
• public — Any code having access to an object of the class can access this method (the default).

• private — Restricts method access to the defining class, excluding subclasses. Subclasses do not inherit private methods.

• protected — Restricts method access to the defining class and subclasses derived from the defining class. Subclasses inherit this method.

• Access list — Restricts method access to classes in access list. For more information, see “Class Members Access” on page 12-28

Local and nested functions inside the method files have the same access as the method. Local functions inside a class-definition file have private access to the class defined in the same file.

**Invoking Superclass Methods in Subclass Methods**

A subclass can override the implementation of a method defined in a superclass. Sometimes, the subclass method needs to execute some additional code instead of completely replacing the superclass method. MATLAB classes can use a special syntax for invocation of superclass methods from a subclass implementation for the same-named method.

The syntax to call a superclass method in a subclass class uses the @ symbol:

\[ MethodName@SuperclassName \]

For example, the following disp method is defined for a Stock class that is derived from an Asset class. The method first calls the Asset class disp method, passing the Stock object so that the Asset components of the Stock object can be displayed. After the Asset disp method returns, the Stock disp method displays the two Stock properties:

```java
classdef Stock < Asset
    methods
        function disp(s)
            disp@Asset(s) % Call base class disp method first
            fprintf(1,'Number of shares: %g
nShare price: %3.2f
','

s.NumShares,s.SharePrice);
        end % disp
    end
end
```
Limitations of Use

The following restrictions apply to calling superclass methods. You can use this notation only within:

- A method having the same name as the superclass method you are invoking
- A class that is a subclass of the superclass whose method you are invoking

Invoking Built-In Functions

The MATLAB `builtin` function enables you to call the built-in version of a function that has been overloaded by a method.

See Also

More About

- “Object Precedence in Method Invocation” on page 9-46
- “Class Precedence” on page 6-23
Class Constructor Methods

Purpose of Class Constructor Methods

A constructor method is a special function that creates an instance of the class. Typically, constructor methods accept input arguments to assign the data stored in properties and return an initialized object.

For a basic example, see “Create a Simple Class” on page 2-2.

All MATLAB classes have a default constructor method. This method returns an object of the class that is created with no input arguments. A class can define a constructor method that overrides the default constructor. An explicitly defined constructor can accept input arguments, initialize property values, call other methods, and perform other operations necessary to create objects of the class.

Basic Structure of Constructor Methods

Constructor methods can be structured into three basic sections:

- Pre-initialization — Compute arguments for superclass constructors.

In this section...

“Purpose of Class Constructor Methods” on page 9-21
“Basic Structure of Constructor Methods” on page 9-21
“Guidelines for Constructors” on page 9-23
“Default Constructor” on page 9-23
“When to Define Constructors” on page 9-24
“Related Information” on page 9-24
“Initializing Objects in Constructor” on page 9-24
“No Input Argument Constructor Requirement” on page 9-25
“Subclass Constructors” on page 9-26
“Implicit Call to Inherited Constructor” on page 9-28
“Errors During Class Construction” on page 9-29
“Output Object Suppressed” on page 9-30
• Object initialization — Call superclass constructors.
• Post initialization — Perform any operations related to the subclass, including referencing and assigning to the object, call class methods, passing the object to functions, and so on.

This code illustrates the basic operations performed in each section:

classdef ConstructorDesign < BaseClass1
    properties
        ComputedValue
    end
    methods
        function obj = ConstructorDesign(a,b,c)

            %% Pre Initialization %%
            % Any code not using output argument (obj)
            if nargin == 0
                % Provide values for superclass constructor
                % and initialize other inputs
                a = someDefaultValue;
                args{1} = someDefaultValue;
                args{2} = someDefaultValue;
            else
                % When nargin ~= 0, assign to cell array,
                % which is passed to supclass constructor
                args{1} = b;
                args{2} = c;
            end
            compvalue = myClass.staticMethod(a);

            %% Object Initialization %%
            % Call superclass constructor before accessing object
            % You cannot conditionalize this statement
            obj = obj@BaseClass1(args{:});

            %% Post Initialization %%
            % Any code, including access to object
            obj.classMethod(arg);
            obj.ComputedValue = compvalue;
            ...
        end
        ...
    end
end
Call the constructor like any function, passing arguments and returning an object of the class.

```matlab
obj = ConstructorDesign(a,b,c);
```

**Guidelines for Constructors**

- The constructor has the same name as the class.
- The only output argument from a constructor is the object constructed.
- If you do not want to assign the output argument, you can clear the object variable in the constructor (see “Output Object Suppressed” on page 9-30).
- If you create a class constructor, provide support for no input arguments. See “No Input Argument Constructor Requirement” on page 9-25.
- If your constructor makes an explicit call to a superclass constructor, this call must occur before any other reference to the constructed object and cannot occur after a `return` statement.
- Calls to superclass constructors cannot be conditional. You cannot place superclass construction calls in loops, conditions, switches, `try/catch`, or nested functions. See “No Conditional Calls to Superclass Constructors” on page 9-27 for more information.

**Default Constructor**

If a class does not define a constructor, MATLAB supplies a constructor that takes no arguments and returns a scalar object whose properties are initialized to property default values. The default constructor supplied by MATLAB also calls all superclass constructors with no arguments or with any argument passed to the default subclass constructor.

When a subclass does not define a constructor, the default constructor passes its inputs to the direct superclass constructor. This behavior is useful when there is not need for a subclass to define a constructor, but the superclass constructor does require input arguments.
When to Define Constructors

Define a constructor method to perform object initialization that a default constructor cannot perform. For example, when creating an object of the class requires:

- Input arguments
- Initializing object state, such as property values, for each object of the class
- Calling the superclass constructor with values that are determined by the subclass constructor

Related Information

For information specific to constructing enumerations, see “Enumeration Class Constructor Calling Sequence” on page 14-8.

For information on creating object arrays in the constructor, see “Construct Object Arrays” on page 10-2.

If the class being created is a subclass, MATLAB calls the constructor of each superclass class to initialize the object. Implicit calls to the superclass constructor are made with no arguments. If superclass constructors require arguments, call them from the subclass constructor explicitly. See “Control Sequence of Constructor Calls” on page 12-14

Initializing Objects in Constructor

Constructor methods must return an initialized object as the only output argument. The output argument is created when the constructor executes, before executing the first line of code.

For example, the following constructor function can assign the value of the object's property A as the first statement because the object obj has already been assigned to an instance of myClass.

```matlab
function obj = myClass(a,b,c)
    obj.A = a;
    ...
end
```

You can call other class methods from the constructor because the object is already initialized.
The constructor also creates an object whose properties have their default values—either empty ([]), or the default value specified in the property definition block.

For example, the following code calls the class method CalculateValue to assign the value of the property Value.

```matlab
function obj = myClass(a,b,c)
    obj.Value = obj.CalculateValue(a,b);
    ...
end
```

**Referencing the Object in a Constructor**

When initializing the object, for example, by assigning values to properties, use the name of the output argument to refer to the object within the constructor. For example, in the following code the output argument is `obj` and the object is reference as `obj`:

```matlab
% obj is the object being constructed
function obj = myClass(arg)
    obj.propert1 = arg*10;
    obj.method1;
    ...
end
```

For more information on defining default property values, see “Property Default Values” on page 8-18.

**No Input Argument Constructor Requirement**

There are cases where the constructor must be able to be called with no input argument:

- When loading objects into the workspace. If the class `ConstructOnLoad` attribute is set to `true`, the `load` function calls the class constructor with no arguments.
- When creating or expanding an object array such that not all elements are given specific values, the class constructor is called with no arguments to fill in unspecified elements (for example, `x(10,1) = myclass(a,b,c);`). In this case, the constructor is called once with no arguments to populate the empty array elements with copies of this one object.

If there are no input arguments, the constructor creates an object using only default properties values. A good practice is to add a check for zero arguments to the class constructor to prevent an error if either of these two cases occur:
function obj = myClass(a,b,c)
    if nargin > 0
        obj.A = a;
        obj.B = b;
        obj.C = c;
        ...
    end
end

For ways to handle superclass constructors, see “Basic Structure of Constructor Methods” on page 9-21.

Subclass Constructors

Subclass constructor methods can explicitly call superclass constructors to pass arguments to the superclass constructor. The subclass constructor must specify these arguments in the call to the superclass constructor using the constructor output argument. Here is the syntax:

classdef MyClass < SuperClass
    function obj = MyClass(arg)
        obj@SuperClass(SuperArgList);
        ...
    end
end

The class constructor must make all calls to superclass constructors before any other references to the object (obj). This restriction include assigning property values or calling ordinary class methods. Also, a subclass constructor can call a superclass constructor only once.

Reference Only Specified Superclasses

If the classdef does not specify the class as a superclass, the constructor cannot call a superclass constructor with this syntax. That is, subclass constructor can call only direct superclass constructors.

classdef MyClass < SuperClass

MATLAB calls any uncalled constructors in the left-to-right order in which they are specified in the classdef line. MATLAB passes no arguments to these functions.
No Conditional Calls to Superclass Constructors

Calls to superclass constructors must be unconditional. There can be only one call for a given superclass. Initialize the superclass portion of the object by calling the superclass constructors before using the object (for example, to assign property values or call class methods).

If you must call superclass constructors with different arguments that depend on some condition, you can build a cell array of arguments and provide one call to the constructor.

For example, the Cube class constructor calls the superclass Shape constructor using default values when the Cube constructor is called with no arguments. If the Cube constructor is called with four input arguments, then pass upvector and viewangle to the superclass constructor:

```matlab
classdef Cube < Shape
    properties
        SideLength = 0
        Color = [0 0 0]
    end
    methods
        function cubeObj = Cube(length,color,upvector,viewangle)
            if nargin == 0
                super_args{1} = [0 0 1];
                super_args{2} = 10;
            elseif nargin == 4
                super_args{1} = upvector;
                super_args{2} = viewangle;
            else
                error('Wrong number of input arguments')
            end
            cubeObj@Shape(super_args{:});
            if nargin > 0 % Use value if provided
                cubeObj.SideLength = length;
                cubeObj.Color = color;
            end
        end
    end
end
```
Zero or More Superclass Arguments

To support a syntax that calls the superclass constructor with no arguments, provide this syntax explicitly.

Suppose in the case of the Cube class example, all property values in the Shape superclass and the Cube subclass have default values specified in the class definitions. Then you can create an instance of Cube without specifying any arguments for the superclass or subclass constructors.

Here is how you can implement this behavior in the Cube constructor:

```matlab
methods
    function cubeObj = Cube(length,color,upvector,viewangle)
        if nargin == 0
            super_args = {}; 
        elseif nargin == 4
            super_args{1} = upvector;
            super_args{2} = viewangle;
        else
            error('Wrong number of input arguments')
        end
        cubeObj@Shape(super_args{:});
        if nargin > 0
            cubeObj.SideLength = length;
            cubeObj.Color = color;
        end
    ...
end
end
```

More on Subclasses

See “Design Subclass Constructors“ on page 12-9 for information on creating subclasses.

Implicit Call to Inherited Constructor

MATLAB passes arguments implicitly from a default subclass constructor to the superclass constructor. This behavior eliminates the need to implement a constructor method for a subclass only to pass arguments to the superclass constructor.
For example, the following class constructor requires one input argument (a `datetime` object), which the constructor assigns to the `CurrentDate` property.

```matlab
classdef BaseClassWithConstr
    properties
        CurrentDate datetime
    end
    methods
        function obj = BaseClassWithConstr(dt)
            obj.CurrentDate = dt;
        end
    end
end
```

Suppose you create a subclass of `BaseClassWithConstr`, but your subclass does not require an explicit constructor method.

```matlab
classdef SubclassDefaultConstr < BaseClassWithConstr
    ...
end
```

You can construct an object of the `SubclassDefaultConstr` by calling its default constructor with the superclass argument:

```matlab
obj = SubclassDefaultConstr(datetime);
```

For information on subclass constructors, see “Subclass Constructors” on page 9-26 and “Default Constructor” on page 9-23.

**Errors During Class Construction**

For handle classes, MATLAB calls the `delete` method when an error occurs under the following conditions:

- A reference to the object is present in the code prior to the error.
- An early `return` statement is present in the code before the error.

MATLAB calls the `delete` method on the object, the `delete` methods for any objects contained in properties, and the `delete` methods for any initialized base classes.

Depending on when the error occurs, MATLAB can call the class destructor before the object is fully constructed. Therefore class `delete` methods must be able to operate on
partially constructed objects that might not have values for all properties. For more
information, see “Support Destruction of Partially Constructed Objects” on page 7-17.

For information on how objects are destroyed, see “Handle Class Destructor” on page 7-
16.

**Output Object Suppressed**

You can suppress the assignment of the class instance to the ans variable when no output
variable is assigned in a call to the constructor. This technique is useful for apps that
creates graphical interface windows that hold onto the constructed objects. These apps
do not need to return the object.

Use nargout to determine if the constructor has been called with an output argument.
For example, the class constructor for the MyApp class clears the object variable, obj, if
called with no output assigned:

```matlab
classdef MyApp
    methods
        function obj = MyApp
            ... 
            if nargout == 0
                clear obj
            end
        end
    end
end
```

When a class constructor does not return an object, MATLAB does not trigger the
meta.class InstanceCreated event.

**See Also**

**Related Examples**

- “Simplifying the Interface with a Constructor” on page 3-22
- “Subclass Constructor Implementation” on page 12-11
Static Methods

In this section...

“What Are Static Methods” on page 9-31
“Why Define Static Methods” on page 9-31
“Defining Static Methods” on page 9-31
“Calling Static Methods” on page 9-32
“Inheriting Static Methods” on page 9-32

What Are Static Methods

Static methods are associated with a class, but not with specific instances of that class. These methods do not require an object of the class as an input argument. Therefore, you can call static methods without creating an object of the class.

Why Define Static Methods

Static methods are useful when you do not want to create an instance of the class before executing some code. For example, you might want to set up the MATLAB environment or use the static method to calculate data required to create class instances.

Suppose that a class needs a value for pi calculated to particular tolerances. The class could define its own version of the built-in pi function for use within the class. This approach maintains the encapsulation of the class's internal workings, but does not require an instance of the class to return a value.

Defining Static Methods

To define a method as static, set the methods block Static attribute to true. For example:

```matlab
classdef MyClass
    ...
    methods(Static)
        function p = pi(tol)
            [n d] = rat(pi,tol);
            p = n/d;
    end
end
```
Calling Static Methods

Invoke static methods using the name of the class followed by dot (.), then the name of the method:

`classname.staticMethodName(args,...)`

Calling the `pi` method of `MyClass` in the previous section would require this statement:

```java
value = MyClass.pi(.001);
```

You can also invoke static methods using an instance of the class, like any method:

```java
obj = MyClass;
value = obj.pi(.001);
```

Inheriting Static Methods

Subclasses can redefine static methods unless the method's `Sealed` attribute is also set to `true` in the superclass.

See Also

Related Examples
- “Implementing the AccountManager Class” on page 3-15
Overload Functions in Class Definitions

**Why Overload Functions**

Classes can redefine MATLAB functions by implementing methods having the same name. Overloading is useful when defining specialized types that you want to behave like existing MATLAB types. For example, you can implement relational operations, plotting functions, and other commonly used MATLAB functions to work with objects of your class.

You can also modify default behaviors by implementing specific functions that control these behaviors. For more information on functions that modify default behaviors, see “Methods That Modify Default Behavior” on page 17-2.

**Implementing Overloaded MATLAB Functions**

Class methods can provide implementations of MATLAB functions that operate only on instances of the class. This restriction is possible because MATLAB can always identify to which class an object belongs.

MATLAB uses the dominant argument to determine which version of a function to call. If the dominant argument is an object, then MATLAB calls the method defined by the object's class, if one exists.

In cases where a class defines a method with the same name as a global function, the class's implementation of the function is said to overload the original global implementation.

To overload a MATLAB function:

- Define a method with the same name as the function you want to overload.
- Ensure that the method argument list accepts an object of the class, which MATLAB uses to determine which version to call.
• Perform the necessary steps in the method to implement the function. For example, access the object properties to manipulate data.

Generally, the method that overloads a function produces results similar to the MATLAB function. However, there are no requirements regarding how you implement the overloading method. The overloading method does not need to match the signature of the overloaded function.

**Note** MATLAB does not support overloading functions using different signatures for the same function name.

**Overload the bar Function**

It is convenient to overload commonly used functions to work with objects of your class. For example, suppose that a class defines a property that stores data that you often graph. The `MyData` class overrides the `bar` function and adds a title to the graph:

```matlab
classdef MyData
    properties
        Data
    end
    methods
        function obj = MyData(d)
            if nargin > 0
                obj.Data = d;
            end
        end
        function bar(obj)
            y = obj.Data;
            bar(y,'EdgeColor','r');
            title('My Data Graph')
        end
    end
end
```

The `MyData bar` method has the same name as the MATLAB `bar` function. However, the `MyData bar` method requires a `MyData` object as input. Because the method is specialized for `MyData` objects, it can extract the data from the `Data` property and create a specialized graph.

To use the `bar` method, create an object:
y = rand(1,10);
md = MyData(y);

Call the method using the object:

bar(md)

You can also use dot notation:

md.bar

**Implementing MATLAB Operators**

Classes designed to implement new MATLAB data types typically define certain operators, such as addition, subtraction, or equality.

For example, standard MATLAB addition (+) cannot add two polynomials because this operation is not defined by simple addition. However, a polynomial class can define its own `plus` method that the MATLAB language calls to perform addition of polynomial objects when you use the + symbol:

```matlab
p1 + p2
```

For information on overloading operators, see “Operator Overloading” on page 17-47.

**Rules for Naming to Avoid Conflicts**

The names of methods, properties, and events are scoped to the class. Therefore, adhere to the following rules to avoid naming conflicts:

- You can reuse names that you have used in unrelated classes.
- You can reuse names in subclasses if the member does not have public or protected access. These names then refer to entirely different methods, properties, and events without affecting the superclass definitions.
- Within a class, all names exist in the same name space and must be unique. A class cannot define two methods with the same name and a class cannot define a local function with the same name as a method.
- The name of a static method is considered without its class prefix. Thus, a static method name without its class prefix cannot match the name of any other method.
See Also

Related Examples

- “Dominant Argument in Overloaded Graphics Functions” on page 9-48
- “Class Support for Array-Creation Functions” on page 9-37
Class Support for Array-Creation Functions

In this section...

“Extend Array-Creation Functions for Your Class” on page 9-37
“Which Syntax to Use” on page 9-38
“Implement Support for Array-Creation Functions” on page 9-39

Extend Array-Creation Functions for Your Class

There are several MATLAB functions that create arrays of a specific size and type, such as `ones` and `zeros`. User-defined classes can add support for array-creation functions without requiring the use of overloaded method syntax.

Class support for any of the array-creation functions enables you to develop code that you can share with built-in and user-defined data types. For example, the class of the variable `x` in the following code can be a built-in type during initial development, and then be replaced by a user-defined class that transparently overloads `zeros`:

```
cls = class(x);
zArray = zeros(m,n,cls);
```

Array-creation functions create arrays of a specific type in two ways:

- Class name syntax — Specify class name that determines the type of array elements.
- Prototype object syntax — Provide a prototype object that the function uses to determine the type and other characteristics of the array elements.

For example:

```
zArray = zeros(2,3,'uint8');
p = uint8([1 3 5; 2 4 6]);
zArray = zeros(2,3,'like',p);
```

After adding support for these functions to a class named `MyClass`, you can use similar syntax with that class:

```
zArray = zeros(2,3,'MyClass');
```

Or pass an object of your class:
\begin{verbatim}
p = MyClass(...);
zArray = zeros(size(p),'like',p);
\end{verbatim}

MATLAB uses these arguments to dispatch to the appropriate method in your class.

**Array-Creation Functions That Support Overloading**

The following functions support this kind of overloading.

<table>
<thead>
<tr>
<th>Array-Creation Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ones</td>
</tr>
<tr>
<td>zeros</td>
</tr>
<tr>
<td>eye</td>
</tr>
<tr>
<td>nan (lowercase)</td>
</tr>
<tr>
<td>inf</td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>cast</td>
</tr>
<tr>
<td>rand</td>
</tr>
<tr>
<td>randn</td>
</tr>
<tr>
<td>randi</td>
</tr>
</tbody>
</table>

**Which Syntax to Use**

To create an array of default objects, which require no input arguments for the constructor, then use the class name syntax.

To create an array of objects with specific property values or if the constructor needs other inputs, use the prototype object to provide this information.

Classes can support both the class name and the prototype object syntax.

You can implement a class name syntax with the `true` and `false` functions even though these functions do not support that syntax by default.
**Class Name Method Called If Prototype Method Does Not Exist**

If your class implements a class name syntax, but does not implement a prototype object syntax for a particular function, you can still call both syntaxes. For example, if you implement a static `zeros` method only, you can call:

```matlab
zeros(...,'like',MyClass(...))
```

In the case in which you call the prototype object syntax, MATLAB first searches for a method named `zerosLike`. If MATLAB cannot find this method, it calls for the `zeros` static method.

This feature is useful if you only need the class name to create the array. You do not need to implement both methods to support the complete array-creation function syntax. When you implement only the class name syntax, a call to a prototype object syntax is the same as the call to the class name syntax.

**Implement Support for Array-Creation Functions**

Use two separate methods to support an array-creation function. One method implements the class name syntax and the other implements the prototype object syntax.

For example, to support the `zeros` function:

- Implement the class name syntax:
  ```matlab
  zeros(...,'ClassName')
  ```

  As a Static method:
  ```matlab
  methods (Static)
     function z = zeros(varargin)
       ...
   end
  end
  ```

- Implement the prototype object syntax:
  ```matlab
  zeros(...,'like',obj)
  ```

  As a Hidden method with the char vector 'Like' appended to the name.

  ```matlab
  methods (Hidden)
     function z = zerosLike(obj,varargin)
  ```
How MATLAB Interprets the Function Call

The special support for array-creation functions results from the interpretation of the syntax.

- A call to the `zeros` function of this form:
  \[ \text{zeros}(\ldots,'\text{ClassName}') \]
  Calls the class static method with this syntax:
  \[ \text{ClassName}.\text{zeros}(\text{varargin}{1:\text{end}-1}) \]
- A call to the `zeros` function of this form:
  \[ \text{zeros}(\ldots,'\text{like}',\text{obj}) \]
  Calls the class method with this syntax:
  \[ \text{zerosLike}(\text{obj},\text{varargin}{1:\text{end}-2}) \]

Support All Function Inputs

The input arguments to an array-creation function can include the dimensions of the array the function returns and possibly other arguments. In general, there are three cases that your methods must support:

- No dimension input arguments resulting in the return of a scalar. For example:
  \[ z = \text{zeros}('\text{MyClass}') \]
- One or more dimensions equal to or less than zero, resulting in an empty array. For example:
  \[ z = \text{zeros}(2,0,'\text{MyClass}') \]
- Any number of valid array dimensions specifying the size of the array. For example:
  \[ z = \text{zeros}(2,3,5,'\text{MyClass}') \]

When the array-creation function calls your class method, it passes the input arguments, excluding the class name or the literal 'like' and the object variable to your method. You can implement your methods with these signatures:
• zeros(varargin) for “class name” methods
• zeros(obj,varargin) for “like prototype object” methods

Sample Class

The Color class represents a color in a specific color space, such as, RGB, HSV, and so on. The discussions in “Class Name Method Implementations” on page 9-41 and “Prototype Object Method Implementation” on page 9-43 use this class as a basis for the overloaded method implementations.

classdef Color
    properties
        ColorValues = [0,0,0]
        ColorSpace = 'RGB'
    end
    methods
        function obj = Color(cSpace,values)
            if nargin > 0
                obj.ColorSpace = cSpace;
                obj.ColorValues = values;
            end
        end
    end
end

Class Name Method Implementations

The zeros function strips the final ClassName char vector and uses it to form the call to the static method in the Color class. The arguments passed to the static method are the array dimension arguments.

Here is an implementation of a zeros method for the Color class. This implementation:

• Defines the zeros method as Static (required)
• Returns a scalar Color object if the call to zeros has no dimension arguments
• Returns an empty array if the call to zeros has any dimensions arguments equal to 0.
• Returns an array of default Color objects. Use repmat to create an array of the dimensions specified by the call to zeros.

classdef Color
    ...
    methods (Static)
function z = zeros(varargin)
    if (nargin == 0)
        % For zeros('Color')
        z = Color;
    elseif any([varargin{:}] <= 0)
        % For zeros with any dimension <= 0
        z = Color.empty(varargin{:});
    else
        % For zeros(m,n,...,'Color')
        % Use property default values
        z = repmat(Color,varargin{:});
    end
end
end
end
end
end
end
end
end
end

The zeros method uses default values for the ColorValues property because these values are appropriate for this application. An implementation of a ones method can set the ColorValues property to [1,1,1], for example.

Suppose that you want to overload the randi function to achieve the following objectives:

• Define each ColorValue property as a 1-by-3 array in the range of 1 to a specified maximum value (for example, 1-255).
• Accommodate scalar, empty, and multidimensional array sizes.
• Return an array of Color objects of the specified dimensions, each with random ColorValues.

classdef Color
    ...
    methods (Static)
        function r = randi(varargin)
            if (nargin == 0)
                % For randi('ClassName')
                r = Color('RGB',randi(255,[1,3]));
            elseif any([varargin{2:end}] <= 0)
                % For randi with any dimension <= 0
                r = Color.empty(varargin{2:end});
            else
                % For randi(max,m,n,...,'ClassName')
                if numel([varargin{:}]) < 2
                    error('Not enough input arguments')
                end
            end
    end
end
Prototype Object Method Implementation

The objective of a method that returns an array of objects that are “like a prototype object” depends on the requirements of the class. For the Color class, the zeroLike method creates objects that have the ColorSpace property value of the prototype object, but the ColorValues are all zero.

Here is an implementation of a zerosLike method for the Color class. This implementation:

- Defines the zerosLike method as Hidden
- Returns a scalar Color object if the call to the zeros function has no dimension arguments
- Returns an empty array if the call to the zeros function has any dimension arguments that are negative or equal to 0.
- Returns an array of Color objects of the dimensions specified by the call to the zeros function.

classdef Color
 ...
 methods (Hidden)
 function z = zerosLike(obj,varargin)
   if nargin == 1
     % For zeros('like',obj)
     cSpace = obj.ColorSpace;
     z = Color;
     z.ColorSpace = cSpace;
   elseif any([varargin{:}] <= 0)
     % For zeros with any dimension <= 0
     z = Color.empty(varargin{:});
   else
     % For zeros(m,n,...,'like',obj)
     if ~isscalar(obj)
       error('Prototype object must be scalar')
     end
     obj = Color(obj.ColorSpace,zeros(1,3,'like',obj.ColorValues));
   end
 end
end
end
end
\begin{verbatim}
z = repmat(obj,varargin{:});
end
end
end
end

Full Class Listing

Here is the \texttt{Color} class definition with the overloaded methods.

\textbf{Note} In actual practice, the \texttt{Color} class requires error checking, color space conversions, and so on. This overly simplified version illustrates the implementation of the overloaded methods.

\texttt{classdef Color
  properties
    ColorValues = \[0,0,0\]
    ColorSpace = 'RGB'
  end
  methods
    function obj = Color(cSpace,values)
      if nargin > 0
        obj.ColorSpace = cSpace;
        obj.ColorValues = values;
      end
    end
  end
  methods (Static)
    function z = zeros(varargin)
      if (nargin == 0)
        % For zeros('ClassName')
        z = Color;
      elseif any([varargin{:}] <= 0)
        % For zeros with any dimension <= 0
        z = Color.empty(varargin{:});
      else
        % For zeros(m,n,...,'ClassName')
        % Use property default values
        z = repmat(Color,varargin{:});
      end
    end
    function r = randi(varargin)
      if (nargin == 0)
        % For randi('ClassName')
        r = Color('RGB',randi(255,[1,3]));
      elseif any([varargin{2:end}] <= 0)
        % For randi with any dimension <= 0
        r = Color.empty(varargin{2:end});
      else
        % For randi(max,m,n,...,'ClassName')
        if numel([varargin{:}]) < 2
          %
        end
      end
    end
  end
end}

9 Methods — Defining Class Operations
error('Not enough input arguments')
end
dims = [varargin{2:end}];
r = zeros(dims,'Color');
for k = 1:prod(dims)
    r(k) = Color('RGB',randi(varargin{1},[1,3]),randi(varargin{1},[1,3]));
end
end
end
methods (Hidden)
function z = zerosLike(obj,varargin)
if nargin == 1
    % For zeros('like',obj)
    cSpace = obj.ColorSpace;
    z = Color;
    z.ColorSpace = cSpace;
elseif any(varargin{:} <= 0)
    % For zeros with any dimension <= 0
    z = Color.empty(varargin{:});
else
    % For zeros(m,n,...,'like',obj)
    if ~isscalar(obj)
        error('Prototype object must be scalar')
    end
    obj = Color(obj.ColorSpace,zeros(1,3,'like',obj.ColorValues));
    z = repmat(obj,varargin{:});
end
end
end

See Also

Related Examples
• “Construct Object Arrays” on page 10-2
Object Precedence in Method Invocation

In this section...

<table>
<thead>
<tr>
<th>“Object Precedence” on page 9-46</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Specifying Precedence of User-Defined Classes” on page 9-47</td>
</tr>
</tbody>
</table>

Object Precedence

Establishing an object precedence enables MATLAB to determine which of possibly many versions of an operator or function to call in a given situation.

For example, consider the expression

\[ \text{objectA + objectB} \]

Ordinarily, objects have equal precedence and the method associated with the leftmost object is called. However, there are two exceptions:

- User-defined classes have precedence over MATLAB fundamental classes (numeric, logical, char, cell, struct, and function handle) and certain built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using the InferiorClasses attribute.

In “Class Design for Polynomials” on page 19-2, the polynom class defines a plus method that enables the addition of DocPolynom objects. Given the object \( p \):

\[
\begin{align*}
p &= \text{DocPolynom}([1 \ 0 \ -2 \ -5]) \\
p &= x^3 - 2x - 5
\end{align*}
\]

the expression:

\[
\begin{align*}
1 + p \\
\text{ans} &= x^3 - 2x - 5
\end{align*}
\]

calls the \text{DocPolynom} plus method (which converts the double, 1, to a DocPolynom object and then implements the addition of two polynomials). The user-defined \text{DocPolynom} class has precedence over the built-in double class.
Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by listing inferior classes in a class attribute. The InferiorClasses property places a class below other classes in the precedence hierarchy. Define the InferiorClasses property in the classdef statement:

classdef (InferiorClasses = {?class1,?class2}) myClass

This attribute establishes a relative priority of the class being defined with the order of the classes listed.

Location in the Hierarchy

If objectA is above objectB in the precedence hierarchy, then the expression

objectA + objectB

calls @classA/plus.m. Conversely, if objectB is above objectA in the precedence hierarchy, then the MATLAB runtime calls @classB/plus.m.

See Also

More About
• “Dominant Argument in Overloaded Graphics Functions” on page 9-48
• “Class Precedence” on page 6-23
Dominant Argument in Overloaded Graphics Functions

In this section...

“Graphics Object Precedence” on page 9-48  
“Dominant Argument” on page 9-48  
“Defining Class Precedence” on page 9-48  
“Calls to Inferior-Class Methods” on page 9-50

Graphics Object Precedence

MATLAB graphics objects have the same precedence as user-defined objects. If you want to implement a method that accepts a graphics object as its first argument (for example, an axes handle), define the MATLAB graphics class as inferior to your class.

Dominant Argument

When evaluating expression involving objects of more than one class, MATLAB uses the dominant argument to determine which method or function to call.

Here is how MATLAB dispatches in response to a function call:

• Determine the dominant argument based on the class of arguments.
• If there is a dominant argument, call the method of the dominant class.
• If arguments are of equal precedence, use the leftmost argument as the dominant argument.
• If the class of the dominant argument does not define a method with the name of the called function, call the first function on the path with that name.

Defining Class Precedence

Specify the relative precedence of MATLAB classes using the InferiorClasses class attribute. Here is the basic syntax:

classdef (InferiorClasses = {?class1,?class2}) ClassName
The following definition of the TemperatureData class implements a specialized version of plot to graph temperature data. The class plot method supports a variable number of input arguments to allow an axes handle as the first argument:

```
plot(obj)
plot(ax,obj)
```

obj is an instance of the TemperatureData class and ax is an axes handle.

MATLAB calls the plot method in both cases because the TemperatureData class specifies the matlab.graphics.axis.Axes as inferior.

```
classdef (InferiorClasses = {?matlab.graphics.axis.Axes}) TemperatureData
    properties
        Time
        Temperature
    end
    methods
        function obj = TemperatureData(x,y)
            obj.Time = x;
            obj.Temperature = y;
        end
        function plot(varargin)
            if nargin == 1
                obj = varargin{1};
                plot(obj.Time,obj.Temperature)
            elseif nargin == 2
                ax = varargin{1};
                obj = varargin{2};
                plot(ax,obj.Time,obj.Temperature)
            elseif nargin > 2
                ax = varargin{1};
                obj = varargin{2};
                plot(ax,obj.Time,obj.Temperature,varargin{3:end})
            end
            datetick('x')
xlabel('Time')
ylabel('Temperature')
        end
    end
end
```

The following call to plot dispatches to the TemperatureData plot method, not the built-in plot function, because the TemperatureData object is dominant over the axes object.

```
x = 1:10;
y = rand(1,10)*100;
ax = axes;
td = TemperatureData(x,y);
plot(ax<td)
```
Calls to Inferior-Class Methods

When you declare a class as inferior to your class, and both classes define a method with the same name, MATLAB dispatches to your class method regardless of argument order.

Suppose the TemperatureData class that is described in the previous section defines a set method. If you attempt to assign an object of the TemperatureData class to the UserData property of an axes object:

```matlab
td = TemperatureData(x,y);
set(gca,'UserData',td)
```

The result is a call to the TemperatureData set method. MATLAB does not call the built-in set function.

To support the use of a set function with inferior classes, implement a set method in your class that calls the built-in set function when the first argument is an object of the inferior class.

```matlab
function set(varargin)
    if isa(varargin{1},'matlab.graphics.axis.Axes')
        builtin('set',varargin{:})
    else
        ...
    end
end
```

See Also

More About

- “Object Precedence in Method Invocation” on page 9-46
Class Methods for Graphics Callbacks

Referencing the Method

To use an ordinary class method as callback for a graphics object, specify the callback property as a function handle referencing the method. For example,

```matlab
uicontrol('Style','slider','Callback',@obj.sliderCallback)
```

Where your class defines a method called `sliderCallback` and `obj` is an instance of your class.

To use a static methods as a callback, specify the callback property as a function handle that includes the class name that is required to refer to a static method:

```matlab
uicontrol('Style','slider','Callback',@MyClass.sliderCallback)
```

Syntax for Method Callbacks

For ordinary methods, use dot notation to pass an instance of the class defining the callback as the first argument:

```matlab
@obj.methodName
```

Define the callback method with the following input arguments:

- An instance of the defining class as the first argument
- The event source handle
- The event data

The function signature would be of this form:

```matlab
function methodName(obj,src,eventData)
...
end
```
For static methods, the required class name ensures MATLAB dispatches to the method of the specified class:

@MyClass.methodName

Define the static callback method with two input arguments — the event source handle and the event data

The function signature would be of this form:

function methodName(src, eventData)

**Passing Extra Arguments**

If you want to pass arguments to your callback in addition to the source and event data arguments passed by MATLAB, you can use an anonymous function. The basic syntax for an anonymous function that you assign to the graphic object's Callback property includes the object as the first argument:

@((src, event) callbackMethod(object, src, eventData, arg1, ...argn))

The function signature would be of this form:

function methodName(obj, src, eventData, varargin)

... end

**Use a Class Method for a Slider Callback**

This example shows how to use a method of your class as a callback for a uicontrol slider.

The SeaLevelSlider class creates a slider that varies the color limits of an indexed image to give the illusion of varying the sea level.

**Class Definition**

Define SeaLevelSlider as a handle class with the following members:

- The class properties store figure and axes handles and the calculated color limits.
- The class constructor creates the graphics objects and assigns the slider callback.
- The callback function for the slider accepts the three required arguments — a class instance, the handle of the event source, and the event data. The event data argument is empty and not used.
• The uicontrol callback uses dot notation to reference the callback method: ...'Callback',@obj.sliderCallback.

```matlab
classdef SeaLevelSlider < handle
    properties
        Figure
        Axes
        CLimit
    end
    methods
        function obj = SeaLevelSlider(x,map)
            obj.Figure = figure('Colormap',map,...
                'Position',[100,100,560,580],...
                'Resize','off');
            obj.Axes = axes('DataAspectRatio',[1,1,1],...
                'XLimMode','manual','YLimMode','manual',...
                'Parent',obj.Figure);
            image(x,'CDataMapping','scaled',...
                'Parent',obj.Axes);
            obj.CLimit = get(obj.Axes,'CLim');
            uicontrol('Style','slider',...
                'Parent',obj.Figure,...
                'Max',obj.CLimit(2)-10,...
                'Min',obj.CLimit(1)-1,...
                'Value',obj.CLimit(1),...
                'Units','normalized',...
                'Position',[0.9286,0.1724,0.0357,0.6897],...
                'SliderStep',[0.003,0.005],...
                'Callback',@obj.sliderCallback);
        end
        function sliderCallback(obj,src,~)
            minVal = get(src,'Value');
            maxVal = obj.CLimit(2);
            obj.Axes.CLim = [minVal maxVal];
        end
    end
end
```

Using the SeaLevelAdjuster Class

The class uses the cape image that is included with the MATLAB product. To obtain the image data, use the load command:
load `cape X map`

After loading the data, create a `SeaLevelSlider` object for the image:

```matlab
slaObj = SeaLevelSlider(X,map);
```

Move the slider to change the color mapping and visualize a rise in sea level.
See Also

More About
• “Listener Callback Syntax” on page 11-30
Object Arrays

- “Construct Object Arrays” on page 10-2
- “Initialize Object Arrays” on page 10-5
- “Empty Arrays” on page 10-8
- “Initialize Arrays of Handle Objects” on page 10-11
- “Accessing Dynamic Properties in Arrays” on page 10-14
- “Implicit Class Conversion” on page 10-16
- “Concatenating Objects of Different Classes” on page 10-18
- “Designing Heterogeneous Class Hierarchies” on page 10-24
- “Heterogeneous Array Constructors” on page 10-34
Construct Object Arrays

In this section...

“Build Arrays in the Constructor” on page 10-2
“Referencing Property Values in Object Arrays” on page 10-3

Build Arrays in the Constructor

A class constructor can create an array by building the array and returning it as the output argument.

For example, the ObjectArray class creates an object array that is the same size as the input array. Then it initializes the Value property of each object to the corresponding input array value.

```matlab
classdef ObjectArray
    properties
        Value
    end
    methods
        function obj = ObjectArray(F)
            if nargin ~= 0
                m = size(F,1);
                n = size(F,2);
                obj(m,n) = obj;
                for i = 1:m
                    for j = 1:n
                        obj(i,j).Value = F(i,j);
                    end
                end
            end
        end
    end
end
```

To preallocate the object array, assign the last element of the array first. MATLAB fills the first to penultimate array elements with the ObjectArray object.

After preallocating the array, assign each object Value property to the corresponding value in the input array F. To use the class:
• Create 5-by-5 array of magic square numbers
• Create a 5-by-5 object array

\[
F = \text{magic}(5);
A = \text{ObjectArray}(F);
\]

\[
\text{whos}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5x5</td>
<td>304</td>
<td>ObjectArray</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>5x5</td>
<td>200</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

**Referencing Property Values in Object Arrays**

Given an object array `objArray` in which each object has a property `PropName`:

• Reference the property values of specific objects using array indexing:

\[
\text{objArray}(ix).\text{PropName}
\]

• Reference all values of the same property in an object array using dot notation. MATLAB returns a comma-separated list of property values.

\[
\text{objArray.}\text{PropName}
\]

• To assign the comma-separated list to a variable, enclose the right-side expression in brackets:

\[
\text{values} = [\text{objArray.}\text{PropName}]
\]

For example, given the `ObjProp` class:

```MATLAB
classdef ObjProp
    properties
        RegProp
    end
    methods
        function obj = ObjProp
            obj.RegProp = randi(100);
        end
    end
end
```

Create an array of `ObjProp` objects:
for k = 1:5
    objArray(k) = ObjProp;
end

Access the RegProp property of the second element of the object array using array indexing:

objArray(2).RegProp

ans =

    91

Assign the values of all RegProp properties to a numeric array:

propValues = [objArray.RegProp]

propValues =

    82    91    13    92    64

Use standard indexing operations to access the values of the numeric array. For more information on numeric arrays, see “Matrices and Arrays”.

See Also

Related Examples

- “Initialize Object Arrays” on page 10-5
- “Initialize Arrays of Handle Objects” on page 10-11
- “Class Constructor Methods” on page 9-21
Initialize Object Arrays

Calls to Constructor

During the creation of object arrays, MATLAB can call the class constructor with no arguments, even if the constructor does not build an object array. For example, suppose that you define the following class:

```matlab
classdef SimpleValue
    properties
        Value
    end
    methods
        function obj = SimpleValue(v)
            obj.Value = v;
        end
    end
end
```

Execute the following statement to create an array:

```matlab
a(1,7) = SimpleValue(7)
```

Error using SimpleValue (line 7)
Not enough input arguments.

This error occurs because MATLAB calls the constructor with no arguments to initialize elements 1 through 6 in the array.

Your class must support the no input argument constructor syntax. A simple solution is to test nargin and let the case when nargin == 0 execute no code, but not error:

```matlab
classdef SimpleValue
    properties
        Value
    end
    methods
        function obj = SimpleValue(v)
            obj.Value = v;
        end
    end
end
```
if nargin > 0
    obj.Value = v;
end
end
end
end

Using the revised class definition, the previous array assignment statement executes without error:

```
a(1,7) = SimpleValue(7)
a =

1x7 SimpleValue array with properties:
   Value
```

The object assigned to array element `a(1,7)` uses the input argument passed to the constructor as the value assigned to the property:

```
a(1,7)
ans =
    SimpleValue with properties:
       Value: 7
```

MATLAB created the objects contained in elements `a(1,1:6)` with no input argument. The default value for properties empty `[]`. For example:

```
a(1,1)
ans =
    SimpleValue with properties:
       Value: []
```

MATLAB calls the `SimpleValue` constructor once and copies the returned object to each element of the array.

**Initial Value of Object Properties**

When MATLAB calls a constructor with no arguments to initialize an object array, one of the following assignments occurs:
• If property definitions specify default values, MATLAB assigns these values.
• If the constructor assigns values in the absence of input arguments, MATLAB assigns these values.
• If neither of the preceding situations apply, MATLAB assigns the value of empty double (that is, [ ]) to the property.

See Also

Related Examples
• “Initialize Arrays of Handle Objects” on page 10-11
Empty Arrays

In this section...

“Creating Empty Arrays” on page 10-8
“Assigning Values to an Empty Array” on page 10-8

Creating Empty Arrays

Empty arrays have no elements, but are of a certain class. All nonabstract classes have a static method named empty that creates an empty array of the same class. The empty method enables you to specify the dimensions of the output array. However, at least one of the dimensions must be 0. For example, define the SimpleValue class:

```matlab
classdef SimpleValue
    properties
        Value
    end
    methods
        function obj = SimpleValue(v)
            if nargin > 0
                obj.Value = v;
            end
        end
    end
end
```

Create a 5-by-0 empty array of class SimpleValue.

```matlab
ary = SimpleValue.empty(5,0)
```

```
ary =

5x0 SimpleValue array with properties:

    Value
```

Calling empty with no arguments returns a 0-by-0 empty array.

Assigning Values to an Empty Array

An empty object defines the class of an array. To assign nonempty objects to an empty array, MATLAB calls the class constructor to create default instances of the class for
every other array element. Once you assign a nonempty object to an array, all array elements must be nonempty objects.

**Note** A class constructor must avoid returning empty objects by default.

For example, using the `SimpleValue` defined in the “Initialize Object Arrays” on page 10-5 section, create an empty array:

```matlab
ary = SimpleValue.empty(5,0);
class(ary)

ans =
SimpleValue

ary is an array of class `SimpleValue`. However, it is an empty array:

```matlab
erary(1)
Index exceeds matrix dimensions.
```

If you make an assignment to a property value, MATLAB calls the `SimpleClass` constructor to grow the array to the require size:

```matlab
ary(5).Value = 7;
ary(5).Value

ans =
7

ary(1).Value

ans =
[]
```

MATLAB populates array elements one through five with `SimpleValue` objects created by calling the class constructor with no arguments. Then MATLAB assigns the property value 7 to the object at `ary(5)`.
See Also

Related Examples

• “Initialize Arrays of Handle Objects” on page 10-11
Initialize Arrays of Handle Objects

When initializing an array of handle objects, MATLAB fills in the empty elements of an array with a default object. To create the default object, MATLAB:

- Calls the class constructor once to obtain an object
- Creates unique handles for each element in the array
- Copies the property values from the constructed default object without calling the constructor again.

The `InitHandleArray` class illustrates this behavior.

```matlab
classdef InitHandleArray < handle
    properties
        RandNumb
    end
    methods
        function obj = InitHandleArray
            obj.RandNumb = randi(100);
        end
    end
end
```

The property `RandNumb` contains a random number that the `InitHandleArray` constructor assigns.

Consider what happens when MATLAB initialize an array created by assigning to the last element in the array. (The last element is the one with the highest index values). Suppose the value of the `RandNumb` property of the `InitHandleArray` object assigned to the element `A(4,5)` is 59:

```matlab
A(4,5) = InitHandleArray;
A(4,5).RandNumb
```

```matlab
ans =
    59
```

The element in the index location `A(4,5)` is an instance of the `InitHandleArray` class. The default object used for element `A(1,1)` is also an instance of the `InitHandleArray` class, but its `RandNumb` property is set to a different random number.
To fill in the preceding array elements, MATLAB calls the class constructor to create a single object. MATLAB copies this object to all the remaining array elements. Calling the constructor to create the default object resulted in another call to the `randi` function, which returns a new random number:

```
A(1,1).RandNumb
ans =
    10
```

MATLAB copies this second instance to all remaining array elements:

```
A(2,2).RandNumb
ans =
    10
A(2,3).RandNumb
ans =
    10
```

When initializing an object array, MATLAB assigns a copy of a single object to the empty elements in the array. MATLAB gives each object a unique handle so that later you can assign different property values to each object. The objects are not equivalent:

```
A(1,1) == A(2,2)
ans =
    0
```

That is, the handle `A(1,1)` does not refer to the same object as `A(2,2)`. The creation of an array with a statement such as:

```
A(4,5) = InitHandleArray;
```

results in two calls to the class constructor. The first creates the object for array element `A(4,5)`. The second creates a default object that MATLAB copies to all remaining empty array elements.
Related Information

For information on array manipulation, see “Multidimensional Arrays”

See “Initialize Properties to Unique Values” on page 8-19 for information on assigning values to properties.

See “Object Array Indexing” on page 17-15 for information on implementing subsasgn methods for your class.
Accessing Dynamic Properties in Arrays

You cannot reference all the dynamic properties in an object array using a single statement, as you can with ordinary properties. For example, the ObjectArrayDynamic class subclasses the dynamicprops class.

```matlab
classdef ObjectArrayDynamic < dynamicprops
    properties
        RegProp
    end
    methods
        function obj = ObjectArrayDynamic
            obj.RegProp = randi(100);
        end
    end
end
```

You can add dynamic properties to objects of the ObjectArrayDynamic class. Create an object array and add dynamic properties to each member of the array. Define elements 1 and 2 as ObjectArrayDynamic objects:

```matlab
a(1) = ObjectArrayDynamic;
a(2) = ObjectArrayDynamic;
```

Add dynamic properties to each object and assign a value.

```matlab
a(1).addprop('DynoProp');
a(1).DynoProp = 1;
a(2).addprop('DynoProp');
a(2).DynoProp = 2;
```

Get the values of the ordinary properties, as with any array.

```matlab
a.RegProp
```

```matlab
ans =
   4
```

```matlab
ans =
   85
```

However, MATLAB returns an error if you try to access the dynamic properties of all array elements using this syntax.
a.DynoProp

No appropriate method, property, or field 'DynoProp' for class 'ObjectArrayDynamic'.

Refer to each object individually to access dynamic property values:

a(1).DynoProp

ans =

    1

a(2).DynoProp

ans =

    2

For information about classes that can define dynamic properties, see “Dynamic Properties — Adding Properties to an Instance” on page 8-68.
Implicit Class Conversion

**In this section...**
- “Class Conversion Mechanism” on page 10-16
- “Concatenation” on page 10-16
- “Subscripted Assignment” on page 10-17

**Class Conversion Mechanism**

When you create or modify object arrays using concatenation or subscripted assignment, MATLAB attempts to convert unlike types to conform to the class of the array.

To perform the conversion, MATLAB attempts to call a converter method defined by the class to be converted. A converter method has the same name as the destination class. For example, if a class defines a method named `double`, this method converts an object of the class to an object of class `double`.

If no converter exists, MATLAB passes the object to be converted to the constructor of the destination class.

Both concatenation and subscripted assignment can cause MATLAB to apply this class conversion mechanism. The conversion can be successful or can result in an error if the conversion is not possible.

**Concatenation**

In concatenation operations, the dominant object determines the class of the resulting array. MATLAB determines the dominant object as follows:

- User-defined classes are dominant over built-in classes like `double`.
- If there is no defined dominance relationship between any two objects, then the left-most object dominates

For example, in the statement `C = [A, B]`, if A is the dominant object, MATLAB attempts to convert B to the class of A.
Subscripted Assignment

In subscripted assignment, the left side of the assignment statement defines the class of the array. If you assign array elements when the right side is a different class than the left side, MATLAB attempts to convert to the class of the left side.

For example, assigning an object of ClassB to an element of array A requires conversion.

```matlab
A = ClassA;
B = ClassB;
A(2) = B;
```

MATLAB first looks for a converter method defined by the class of the source object B. This converter method must have the name ClassA. The subscripted assignment is effectively a call to the converter defined by ClassB:

```matlab
A(2) = B.ClassA % Call method of ClassB
```

If no converter method exists, MATLAB passes the source object to the destination class constructor:

```matlab
A(2) = ClassA(B) % Call ClassA constructor
```

See Also

Related Examples

- “Valid Combinations of Unlike Classes”
- “Concatenating Objects of Different Classes” on page 10-18
- “Object Converters” on page 17-12
Concatenating Objects of Different Classes

In this section...

| “Basic Knowledge” on page 10-18 |
| “MATLAB Concatenation Rules” on page 10-18 |
| “Concatenating Objects” on page 10-19 |
| “Calling the Dominant-Class Constructor” on page 10-19 |
| “Converter Methods” on page 10-21 |

Basic Knowledge

The material presented in this section builds on an understanding of the information presented in the following sections.

- “Construct Object Arrays” on page 10-2
- “Valid Combinations of Unlike Classes”

MATLAB Concatenation Rules

MATLAB follows these rules for concatenating objects:

- MATLAB always attempts to convert all objects to the dominant class.
- User-defined classes take precedence over built-in classes like double.
- If there is no defined dominance relationship between any two objects, then the left-most object dominates (see “Class Precedence” on page 6-23).

When converting to a dominant class during concatenation or subscripted assignment, MATLAB searches the non-dominant class for a conversion method that is the same name as the dominant class. If such a conversion method exists, MATLAB calls it. If a conversion method does not exist, MATLAB calls the dominant class constructor on the non-dominant object.

It is possible for the dominant class to define horzcat, vertcat, or cat methods that modify the default concatenation process.
MATLAB does not convert objects to a common superclass unless those objects are part of a heterogeneous hierarchy. For more information, see “Designing Heterogeneous Class Hierarchies” on page 10-24.

Concatenating Objects

Concatenation combines objects into arrays:

\[
ary = [obj1, obj2, obj3, \ldots, objn];
\]

The size of \( ary \) is 1-by-n.

\[
ary = [obj1; obj2; obj3; \ldots; objn];
\]

The size of \( ary \) is n-by-1.

The class of the arrays is the same as the class of the objects being concatenated. Concatenating objects of different classes is possible if MATLAB can convert objects to the dominant class. MATLAB attempts to convert unlike objects by:

- Calling the inferior object converter method, if one exists.
- Passing an inferior object to the dominant class constructor to create an object of the dominant class.

If conversion of the inferior object is successful, MATLAB returns an array that is of the dominant class. If conversion is not possible, MATLAB returns an error.

Calling the Dominant-Class Constructor

MATLAB calls the dominant class constructor to convert an object of an inferior class to the dominant class. MATLAB passes the inferior object to the constructor as an argument. If the class design enables the dominant class constructor to accept objects of inferior classes as input arguments, then concatenation is possible without implementing a separate converter method.

If the constructor simply assigns this argument to a property, the result is an object of the dominant class with an object of an inferior class stored in a property. If this assignment is not a desired result, then ensure that class constructors include adequate error checking.
For example, consider the class `ColorClass` and two subclasses, `RGBColor` and `HSVColor`:

```matlab
classdef ColorClass
    properties
        Color
    end
end
```

The class `RGBColor` inherits the `Color` property from `ColorClass`. `RGBColor` stores a color value defined as a three-element vector of red, green, and blue (RGB) values. The constructor does not restrict the value of the input argument. It assigns this value directly to the `Color` property.

```matlab
classdef RGBColor < ColorClass
    methods
        function obj = RGBColor(rgb)
            if nargin > 0
                obj.Color = rgb;
            end
        end
    end
end
```

The class `HSVColor` also inherits the `Color` property from `ColorClass`. `HSVColor` stores a color value defined as a three-element vector of hue, saturation, brightness value (HSV) values.

```matlab
classdef HSVColor < ColorClass
    methods
        function obj = HSVColor(hsv)
            if nargin > 0
                obj.Color = hsv;
            end
        end
    end
end
```

Create an instance of each class and concatenate them into an array. The `RGBColor` object is dominant because it is the leftmost object and neither class defines a dominance relationship:

```matlab
crgb = RGBColor([1 0 0]);
chsv = HSVColor([0 1 1]);
```
ary = [crgb,chsv];
class(ary)

ans =
RGBColor

You can combine these objects into an array because MATLAB can pass the inferior object of class HSVColor to the constructor of the dominant class. However, notice that the Color property of the second RGBColor object in the array actually contains an HSVColor object, not an RGB color specification:

ary(2).Color

ans =

HSVColor with properties:

    Color: [0 1 1]

Avoid this undesirable behavior by:

- Implementing converter methods
- Performing argument checking in class constructors before assigning values to properties

**Converter Methods**

If your class design requires object conversion, implement converter methods for this purpose.

The ColorClass class defines converter methods for RGBColor and HSVColor objects:

classdef ColorClass
    properties
        Color
    end
    methods
        function rgbObj = RGBColor(obj)
            if isa(obj,'HSVColor')
                rgbObj = RGBColor(hsv2rgb(obj.Color));
            end
        end
    end
end
function hsvObj = HSVColor(obj)
    if isa(obj, 'RGBColor')
        hsvObj = HSVColor(rgb2hsv(obj.Color));
    end
end
end

Create an array of RGBColor and HSVColor objects with the revised superclass:

crgb = RGBColor([1 0 0]);
chsv = HSVColor([0 1 1]);
ary = [crgb, chsv];
class(ary)

ans = RGBColor

MATLAB calls the converter method for the HSVColor object, which it inherits from the superclass. The second array element is now an RGBColor object with an RGB color specification assigned to the Color property:

ary(2)

ans =

    RGBColor with properties:
        Color: [1 0 0]

ary(2).Color

ans =

    1     0     0

If the leftmost object is of class HSVColor, the array ary is also of class HSVColor, and MATLAB converts the Color property data to HSV color specification.

ary = [chsv crgb]

ary =

    1x2 HSVColor
Properties:
    Color

ary(2).Color

ans =
    0   1   1

Defining a converter method in the superclass and adding better argument checking in the subclass constructors produces more predictable results. Here is the RGBColor class constructor with argument checking:

classdef RGBColor < ColorClass
    methods
        function obj = RGBColor(rgb)
            if nargin == 0
                rgb = [0 0 0];
            else
                if ~(isa(rgb,'double')... 
                    && size(rgb,2) == 3 ... 
                    && max(rgb) <= 1 && min(rgb) >= 0)
                    error('Specify color as RGB values')
                end
                end
                obj.Color = rgb;
            end
        end
    end
end

Your applications can require additional error checking and other coding techniques. The classes in these examples are designed only to demonstrate concepts.

See Also

More About
- “Implicit Class Conversion” on page 10-16
- “Object Converters” on page 17-12
- “Hierarchies of Classes — Concepts” on page 12-2
Designing Heterogeneous Class Hierarchies

In this section...

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Creating Classes That Support Heterogeneous Arrays&quot;</td>
<td>10-24</td>
</tr>
<tr>
<td>&quot;MATLAB Arrays&quot;</td>
<td>10-24</td>
</tr>
<tr>
<td>&quot;Heterogeneous Hierarchies&quot;</td>
<td>10-25</td>
</tr>
<tr>
<td>&quot;Heterogeneous Arrays&quot;</td>
<td>10-25</td>
</tr>
<tr>
<td>&quot;Heterogeneous Array Concepts&quot;</td>
<td>10-26</td>
</tr>
<tr>
<td>&quot;Nature of Heterogeneous Arrays&quot;</td>
<td>10-26</td>
</tr>
<tr>
<td>&quot;Unsupported Hierarchies&quot;</td>
<td>10-29</td>
</tr>
<tr>
<td>&quot;Default Object&quot;</td>
<td>10-31</td>
</tr>
<tr>
<td>&quot;Conversion During Assignment and Concatenation&quot;</td>
<td>10-32</td>
</tr>
<tr>
<td>&quot;Empty Arrays of Heterogeneous Abstract Classes&quot;</td>
<td>10-32</td>
</tr>
</tbody>
</table>

Creating Classes That Support Heterogeneous Arrays

This topic describes the concepts involved in defining classes that support the formation of heterogeneous arrays. For information on the concatenation of existing MATLAB objects, see these topics.

- "Concatenating Objects of Different Classes" on page 10-18
- "Valid Combinations of Unlike Classes"

For an example that uses heterogeneous arrays, see "A Class Hierarchy for Heterogeneous Arrays" on page 20-2.

MATLAB Arrays

MATLAB determines the class of an array by the class of the objects contained in the array. MATLAB is unlike some languages in which you define an array of object pointers or references. In these other languages, the type of the array is different from the type of an object in the array. You can access the elements of the array and dispatch to methods on those elements, but you cannot call an object method on the whole array, as you can in MATLAB.
Object arrays in MATLAB are homogeneous in class. Because of this homogeneity, you can perform operations on whole arrays, such as multiplying numeric matrices. You can form heterogeneous array by defining a hierarchy of classes that derive from a common superclass. Cell arrays provide an option for an array type that can hold different kinds of unrelated objects.

Heterogeneous Hierarchies

You can form arrays of objects that are subclasses of a common superclass when these classes are part of a heterogeneous hierarchy. A MATLAB heterogeneous class hierarchy:

- Derives from `matlab.mixin.Heterogeneous`
- Defines a single root superclass that derives directly from `matlab.mixin.Heterogeneous`
- Seals methods that are inherited by subclasses.

For example, in the following diagram, Shape is the root of the heterogeneous hierarchy.

Heterogeneous Arrays

A heterogeneous array is an array of objects that differ in their specific class, but all objects derive from or are instances of a common superclass. The common superclass forms the root of the hierarchy of classes that you can combine into heterogeneous arrays.

The common superclass class must derive from `matlab.mixin.Heterogeneous`. Methods that you can call on the array as a whole must have the same definitions for all subclasses.
Heterogeneous hierarchies are useful to:

- Create arrays of objects that are of different classes, but part of a related hierarchy.
- Call methods of the most specific common superclass on the array as a whole
- Access properties of the most specific common superclass using dot notation with the array
- Use common operators that are supported for object arrays
- Support array indexing (scalar or nonscalar) that returns arrays of the most specific class

**Heterogeneous Array Concepts**

- Heterogeneous array — An array in which two or more elements belong to different specific classes. All elements derive from the same root superclass.
- Root superclass — Class derived directly from `matlab.mixin.Heterogeneous`. The root superclass can be abstract or concrete. Only concrete subclasses of the root superclass can form heterogeneous arrays.
- Most specific common superclass — The most specific class in the inheritance hierarchy from which all the objects in a heterogeneous array derive. The most specific common superclass can be the root superclass or a more specific superclass shared by the objects currently in the array.
- Class of a heterogeneous array — The most specific common superclass from which all objects in the heterogeneous array derive. Adding and removing objects from a heterogeneous array can change the most specific superclass shared by the instances. This change results in a change in the class of a heterogeneous array. The most specific common superclass can be abstract.

**Nature of Heterogeneous Arrays**

The heterogeneous hierarchy in this diagram illustrates the characteristics of heterogeneous arrays concerning:

- Array class
- Property access
- Method invocation
Class of Heterogeneous Arrays

The class of a heterogeneous array is that of the most specific superclass shared by the objects of the array.

If the following conditions are true, the concatenation and subscripted assignment operations return a heterogeneous array:

- The objects on the right side of the assignment statement are of different classes
- All objects on the right side of the assignment statement derive from a common subclass of `matlab.mixin.Heterogeneous`

For example, form an array by concatenating objects of these classes. The class of `a1` is `ClassA`:

```matlab
a1 = [SpecificA,SpecificB];
class(a1)
```
ans =  
ClassA

If the array includes an object of the class SpecificC, the class of a2 is RootSuperclass:

```matlab
a2 = [SpecificA, SpecificB, SpecificC];
class(a2)
ans = RootSuperclass
```

If you assigned an object of the class SpecificC to array a1 using indexing, the class of a1 becomes RootSuperclass:

```matlab
a1(3) = SpecificC;
class(a1)
ans = RootSuperclass
```

If the array contains objects of only one class, then the array is not heterogeneous. For example, the class of a is SpecificA.

```matlab
a = [SpecificA, SpecificA];
class(a)
ans = SpecificA
```

**Property Access**

Access array properties with dot notation when the class of the array defines the properties. The class of the array is the most specific common superclass, which ensures all objects inherit the same properties.

For example, suppose ClassA defines a property called Prop1.

```matlab
a1 = [SpecificA, SpecificB];
a1.Prop1
```
Referring to Prop1 using dot notation returns the value of Prop1 for each object in the array.

**Method Invocation**

To invoke a method on a heterogeneous array, the class of the array must define or inherit the method as Sealed. For example, suppose RootSuperclass defines a Sealed method called superMethod.

Call the method on all objects in the array a2:

```java
a2 = [SpecificA, SpecificB, SpecificC];
a2.superMethod
```

Sealing the method (so that it cannot be overridden in a subclass) ensures that the same method definition exists for all elements of the array. Calling that method on a single element of the array invokes the same method implementation as calling the method on the whole array.

**Unsupported Hierarchies**

Heterogeneous hierarchies cannot have ambiguities when obtaining default objects, determining the class of the array, and converting class objects to other types. Members of the hierarchy can derive from only one root superclass (that is, from only one direct subclass of matlab.mixin.Heterogeneous).

This diagram shows a hierarchy that is not allowed:
ClassA derives from two classes that are subclasses of `matlab.mixin.Heterogeneous`.

The next diagram shows two separate heterogeneous hierarchies. ClassA has only one root superclass (called `OtherBaseClass`). The heterogeneous hierarchy is no longer ambiguous:
Default Object

A default object is the object returned by calling the class constructor with no arguments. MATLAB uses default objects in these situations:

- Indexed assignment creates an array with gaps in array elements. For example, assign the first element of array h to index 5:

  \[ h(5) = \text{ClassA(arg1, arg2)}; \]

  MATLAB fills the unassigned positions with default objects.

- Loading a heterogeneous array from a MAT-file when the class definition of a specific object in the array is not available. MATLAB replaces the object with the default object.
Heterogeneous hierarchies enable you to define the default object for that hierarchy. The matlab.mixin.Heterogeneous class provides a default implementation of a method called getDefaultScalarElement. This method returns an instance of the root class of the heterogeneous hierarchy, unless the root superclass is abstract.

If the root superclass is abstract or is not appropriate for a default object, override the getDefaultScalarElement method. Implement the getDefaultScalarElement override in the root superclass, which derives directly from matlab.mixin.Heterogeneous.

getDefaultScalarElement must return a scalar object that is derived from the root superclass. For specific information on how to implement this method, see getDefaultScalarElement.

**Conversion During Assignment and Concatenation**

If you create a heterogeneous array that contains objects that are not derived from the same root superclass, MATLAB attempts to call a method called convertObject. Implement convertObject to convert objects to the appropriate class. There is no default implementation of this method.

To support the formation of heterogeneous arrays using objects that are not part of the heterogeneous hierarchy, implement a convertObject method in the root superclass. The convertObject method must convert the nonmember object to a valid member of the heterogeneous hierarchy.

For details on implementing the convertObject method, see matlab.mixin.Heterogeneous.

**Empty Arrays of Heterogeneous Abstract Classes**

For homogeneous arrays, MATLAB does not allow you to initialize an empty array of an abstract class. However, if the class is part of a heterogeneous hierarchy, you can initialize empty arrays of an abstract class. Initializing an empty heterogeneous array is useful in cases in which you do not know the class of the concrete elements in advance.

For example, suppose RootSuperclass is an abstract class that is the root of a heterogeneous hierarchy. Initialize an array using the empty static method:

```matlab
ary = RootSuperclass.empty;
```
See Also

Related Examples

- “A Class Hierarchy for Heterogeneous Arrays” on page 20-2
- “Handle-Compatible Classes and Heterogeneous Arrays” on page 12-48
Heterogeneous Array Constructors

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Building Arrays in Superclass Constructors” on page 10-34</td>
</tr>
<tr>
<td>“When Errors Can Occur” on page 10-34</td>
</tr>
<tr>
<td>“Initialize Array in Superclass Constructor” on page 10-35</td>
</tr>
<tr>
<td>“Sample Implementation” on page 10-36</td>
</tr>
<tr>
<td>“Potential Error” on page 10-38</td>
</tr>
</tbody>
</table>

**Building Arrays in Superclass Constructors**

When a subclass in a heterogeneous class hierarchy calls its superclass to construct an array of objects, you must ensure that the superclass constructor does not return a heterogeneous array to the subclass. The following programming patterns show how to avoid the errors caused by returning the wrong class to the subclass constructor.

**When Errors Can Occur**

Constructors must return objects that are the same class as the defining class. When working with objects from a heterogeneous class hierarchy, the class of an object array can change as you add array elements of different classes. As a result, heterogeneous superclass constructors can change the class of object arrays when the class design requires all the following techniques:

- Building object arrays in subclass constructors
- Calling superclass constructors from subclass constructors to pass arguments
- Creating object arrays in the superclass constructor

In addition, either of the following is true:

- The root superclass is not abstract and does not implement a `getDefaultScalarElement` method.
- The root superclass implements a `getDefaultScalarElement` method that returns an object that is not the same class as the subclass.

When assigning to object arrays, MATLAB uses the default object to fill in unassigned array elements. In a heterogeneous hierarchy, the default object can be the superclass...
that is called by the subclass constructor. Therefore, building an array in the superclass constructor can create a heterogeneous array.

If a superclass constructor returns a heterogeneous array to the subclass constructor, MATLAB generates an error (see “Potential Error” on page 10-38).

**Initialize Array in Superclass Constructor**

To avoid errors, initialize the object array explicitly in the superclass constructor. For example, use `repelem` in the superclass constructor to initialize the array before initializing the superclass part of the objects. Initializing the array ensures that all elements assigned into the array are of the same class as the `obj` argument.

In this code, the superclass constructor creates one object for each element in the input argument, `arg`:

```matlab
method
    function obj = SuperClass(arg)
        ...
        n = numel(arg);
        obj = repelem(obj,1,n);
        for k = 1:n
            obj(k).SuperProp = arg(k);
        end
        ...
    end
end
```

The subclass constructor calls the superclass constructor to pass the required argument array, `a`:

```matlab
method
    function obj = SubClass(a)
        obj = obj@SuperClass(a);
        for k = 1:numel(a)
            obj(k).SubProp = a(k);
        end
    end
end
```
Sample Implementation

The following class hierarchy defines a subclass that builds object arrays in its constructor. The root superclass of the hierarchy initializes the superclass part of the objects in the array.

This class hierarchy represents members of an engineering team. The classes in the hierarchy include:

- **TeamMembers** — Superclass for specific team member classes, like ProjectEngineer. TeamMembers defines the Name and PhoneX properties and derives from matlab.mixin.Heterogeneous.
- **ProjectEngineer** — Team members that are engineers. Each instance inherits a Name and PhoneX property and defines a billing Rate property.
- **Other members** — Other types of team members not implemented for this example for simplicity.
The `TeamMembers` class is the root of the heterogeneous hierarchy and is a concrete class. Before assigning values to the `Name` and `PhoneX` properties, the constructor initializes an array of subclass (`ProjectEngineer`) objects.

The `ProjectEngineer` constructor provides the `obj` argument for the call to `repelem` with this statement:

```matlab
obj = obj@TeamMembers(varargin{1:2});
```

Here is the `TeamMembers` class:

```matlab
classdef TeamMembers < matlab.mixin.Heterogeneous
    properties
        Name
        PhoneX
    end
    methods
        function obj = TeamMembers(nme,ext)
            if nargin > 0
                n = numel(nme);
                obj = repelem(obj,1,n);
                for k = 1:n
                    obj(k).Name = nme{k};
                    obj(k).PhoneX = ext(k);
                end
            else
                obj.Name = '';  
            end
        end
    end
end
```

The `ProjectEngineer` class represents one type of team member. This class supports array inputs and returns an array of objects.

```matlab
classdef ProjectEngineer < TeamMembers
    % Inputs: {Name}, [PhoneX], {Rate}
    properties
        Rate
    end
    methods
        function obj = ProjectEngineer(varargin)
            obj = obj@TeamMembers(varargin{1:2});
            for k = 1:numel(varargin{1})
                obj(k).Rate = varargin{3}{k};
            end
        end
end
```
The `ProjectEngineer` class requires a cell array of names, a numeric array of phone extensions, and a cell array of billing rates for each engineer in the team.

```matlab
nm = {'Fred','Nancy','Claudette'};
px = [8112,8113,8114];
rt = {'C2','B1','A2'};
tm = ProjectEngineer(nm,px,rt)
```

```matlab
tm =

1x3 ProjectEngineer array with properties:
    Rate
    Name
    PhoneX
```

### Potential Error

The `TeamMembers` constructor initializes the object array with this statement:

```matlab
obj = repelem(obj,1,n);
```

Because the `obj` argument to `repelem` is a `ProjectEngineer` object, the array returned is of the same class.

Without this statement, the `TeamMembers` constructor would create default objects to fill in array elements in the `for` loop. The resulting heterogeneous array would be of the class of the common superclass (`TeamMembers` in this case). If the superclass returns this heterogeneous array to the subclass constructor, it is a violation of the rule that class constructors must preserve the class of the returned object.

MATLAB issues this error:

```
When constructing an instance of class 'ProjectEngineer', the constructor must preserve the class of the returned object.
```

```
Error in ProjectEngineer (line 8)
    obj = obj@TeamMembers(varargin{1:2});
```
See Also

More About

• “Designing Heterogeneous Class Hierarchies” on page 10-24
Events — Sending and Responding to Messages

- “Overview Events and Listeners” on page 11-2
- “Define Custom Event Data” on page 11-6
- “Observe Changes to Property Values” on page 11-9
- “Implement Property Set Listener” on page 11-11
- “Event and Listener Concepts” on page 11-14
- “Event Attributes” on page 11-19
- “Events and Listeners Syntax” on page 11-22
- “Listener Lifecycle” on page 11-28
- “Listener Callback Syntax” on page 11-30
- “Callback Execution” on page 11-34
- “Determine If Event Has Listeners” on page 11-37
- “Listen for Changes to Property Values” on page 11-40
- “Assignment When Property Value Unchanged” on page 11-44
- “Techniques for Using Events and Listeners” on page 11-48
Overview Events and Listeners

In this section...

| “Why Use Events and Listeners” on page 11-2 |
| “Events and Listeners Basics” on page 11-2 |
| “Events and Listeners Syntax” on page 11-3 |
| “Create Listener” on page 11-4 |

Why Use Events and Listeners

Events are notices that objects broadcast in response to something that happens, such as a property value changing or a user interaction with an application program. Listeners execute functions when notified that the event of interest occurs. Use events to communicate changes to objects. Listeners respond by executing the callback function.

For more information, see “Event and Listener Concepts” on page 11-14.

Events and Listeners Basics

When using events and listeners:

- Only handle classes can define events and listeners.
- Define event names in the events block of a class definition (“Events and Listeners Syntax” on page 11-22).
- Use event attributes to specify access to the event (“Event Attributes” on page 11-19).
- Call the handle handle.notify method to trigger the event. The event notification broadcasts the named event to all listeners registered for this event.
- Use the handle handle.addlistener method to couple a listener to the event source object. MATLAB destroys the listener when the source of the event is destroyed.
- Use the handle handle.listener method to create listeners that are not coupled to the lifecycle of the event source object. This approach is useful when the event source and the listeners are defined in different components that you want to be able to add, remove, or modify independently. Your application code controls the listener object lifecycle.
Listener callback functions must define at least two input arguments — the event source object handle and the event data (See “Listener Callback Syntax” on page 11-30 for more information).

Modify the data passed to each listener callback by subclassing the `event.EventData` class.

Predefined Events

MATLAB defines events for listening to property sets and queries. For more information, see “Listen for Changes to Property Values” on page 11-40.

All handle objects define an event named `ObjectBeingDestroyed`. MATLAB triggers this event before calling the class destructor.

Events and Listeners Syntax

Define an event name in the `events` code block:

```matlab
classdef ClassName < handle
    ...
    events
        EventName
    end
    ...
end
```

For example, `MyClass` defines an event named `StateChange`:

```matlab
classdef MyClass < handle
    events
        StateChange
    end
end
```

Trigger an event using the `handle` class `notify` method:

```matlab
classdef ClassName < handle
    ...
    events
        EventName
    end
    ...
    methods
```
Any function or method can trigger the event for a specific instance of the class defining the event. For example, the `triggerEvent` method calls `notify` to trigger the `StateChange` event:

```matlab
classdef MyClass < handle
    events
        StateChange
    end
    methods
        function triggerEvent(obj)
            notify(obj,'StateChange')
        end
    end
end
```

Trigger the `StateChange` event with the `triggerEvent` method:

```matlab
obj = MyClass;
obj.triggerEvent
```

**Create Listener**

Define a listener using the `handle` class `handle.addlistener` or `handle.listener` method. Pass a function handle for the listener callback function using one of these syntaxes:

- `addlistener(SourceOfEvent,'EventName',@functionName)` — for an ordinary function.
- `addlistener(SourceOfEvent,'EventName',@Obj.methodName)` — for a method of `Obj`.
- `addlistener(SourceOfEvent,'EventName',@ClassName.methodName)` — for a static method of the class `ClassName`.

```matlab
ListenerObject = addlistener(SourceOfEvent,'EventName',@listenerCallback);
```

`addlistener` returns the listener object. The input arguments are:
• **SourceOfEvent** — An object of the class that defines the event. The event is triggered on this object.
• **EventName** — The name of the event defined in the class `events` code block.
• **@listenerCallback** — a function handle referencing the function that executes in response to the event.

For example, create a listener object for the `StateChange` event:

```matlab
function lh = createListener(src)
    lh = addlistener(src,'StateChange',@handleStateChange)
end
```

Define the callback function for the listener. The callback function must accept as the first two arguments the event source object and an event data object:

```matlab
function handleStateChange(src,eventData)
    ...
end
```

### See Also

**Related Examples**

- “Listener Lifecycle” on page 11-28
- “Implement Property Set Listener” on page 11-11
Define Custom Event Data

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Class Event Data Requirements” on page 11-6</td>
</tr>
<tr>
<td>“Define and Trigger Event” on page 11-6</td>
</tr>
<tr>
<td>“Define Event Data” on page 11-7</td>
</tr>
<tr>
<td>“Create Listener for Overflow Event” on page 11-8</td>
</tr>
</tbody>
</table>

Class Event Data Requirements

Suppose that you want to create a listener callback function that has access to specific information when the event occurs. This example shows how by creating custom event data.

Events provide information to listener callback functions by passing an event data argument to the specified function. By default, MATLAB passes an `event.EventData` object to the listener callback. This object has two properties:

- **EventName** — Name of the event triggered by this object.
- **Source** — Handle of the object triggering the event.

Provide additional information to the listener callback by subclassing the `event.EventData` class.

- Define properties in the subclass to contain the additional data.
- Define a constructor that accepts the additional data as arguments.
- Set the `ConstructOnLoad` class attribute.
- Use the subclass constructor as an argument to the `notify` method to trigger the event.

Define and Trigger Event

The `SimpleEventClass` defines a property set method (see “Property Set Methods” on page 8-55) from which it triggers an event if the property is set to a value exceeding a certain limit. The property set method performs these operations:
• Saves the original property value
• Sets the property to the specified value
• If the specified value is greater than 10, the set method triggers an Overflow event
• Passes the original property value, and other event data, in a SpecialEventDataClass object to the notify method.

```matlab
classdef SimpleEventClass < handle
    properties
        Prop1 = 0
    end
    events
        Overflow
    end
    methods
        function set.Prop1(obj,value)
            orgvalue = obj.Prop1;
            obj.Prop1 = value;
            if (obj.Prop1 > 10)
                % Trigger the event using custom event data
                notify(obj,'Overflow',SpecialEventDataClass(orgvalue));
            end
        end
    end
end
```

### Define Event Data

Event data is always contained in an `event.EventData` object. The `SpecialEventDataClass` adds the original property value to the event data by subclassing `event.EventData`:

```matlab
classdef (ConstructOnLoad) SpecialEventDataClass < event.EventData
    properties
        OrgValue = 0
    end
    methods
        function eventData = SpecialEventDataClass(value)
            eventData.OrgValue = value;
        end
    end
end
```
Create Listener for Overflow Event

To listen for the Overflow event, attach a listener to an instance of the SimpleEventClass class. Use the addlistener method to create the listener. Also, you must define a callback function for the listener to execute when the event is triggered.

The function setupSEC instantiates the SimpleEventClass class and adds a listener to the object. In this example, the listener callback function displays information that is contained in the eventData argument (which is a SpecialEventDataClass object).

```matlab
function sec = setupSEC
    sec = SimpleEventClass;
    addlistener(sec,'Overflow',@overflowHandler)
    function overflowHandler(eventSrc,eventData)
        disp('The value of Prop1 is overflowing!')
        disp(['Its value was: ' num2str(eventData.OrgValue)])
        disp(['Its current value is: ' num2str(eventSrc.Prop1)])
    end
end
```

Create the SimpleEventClass object and add the listener:

```matlab
sec = setupSEC;
sec.Prop1 = 5;
sec.Prop1 = 15; % listener triggers callback
```

The value of Prop1 is overflowing!
Its value was: 5
Its current value is: 15

See Also

Related Examples

- “Observe Changes to Property Values” on page 11-9
Observe Changes to Property Values

This example shows how to listen for changes to a property value. This example uses:

- `PostSet` event predefined by MATLAB
- `SetObservable` property attribute to enable triggering the property `PostSet` event.
- `addlistener` handle class method to create the listener

```matlab
classdef PropLis < handle
    % Define a property that is SetObservable
    properties (SetObservable)
        ObservedProp = 1
    end
    methods
        function attachListener(obj)
            %Attach a listener to a PropListener object
            addlistener(obj,'ObservedProp','PostSet',@PropLis.propChange);
        end
    end
    methods (Static)
        function propChange(metaProp,eventData)
            % Callback for PostSet event
            % Inputs: meta.property object, event.PropertyEvent
            h = eventData.AffectedObject;
            propName = metaProp.Name;
            disp(['The ',propName,' property has changed.'])
            disp(['The new value is: ',num2str(h.ObservedProp)])
            disp(['Its default value is: ',num2str(metaProp.DefaultValue)])
        end
    end
end
```

The `PropLis` class uses an ordinary method (`attachListener`) to add the listener for the `ObservedProp` property. If the `PropLis` class defines a constructor, the constructor can contain the call to `addlistener`.

The listener callback is a static method (`propChange`). MATLAB passes two arguments when calling this function:

- `metaProp` — a `meta.property` object for `ObservedProp`
- `eventData` — an `event.PropertyEvent` object contain event-specific data.

These arguments provide information about the property and the event.
Use the PropLis class by creating an instance and calling its attachListener method:

```matlab
plObj = PropLis;
plObj.ObservedProp

ans =

1

plObj.attachListener
plObj.ObservedProp = 2;
```

The ObservedProp property has changed.
The new value is: 2
Its default value is: 1

See Also

- `event.proplistener`
- `handle.addlistener`
- `handle.listener`

Related Examples

- “Listener Lifecycle” on page 11-28
- “Implement Property Set Listener” on page 11-11
Implement Property Set Listener

This example shows how to define a listener for a property set event. The listener callback triggers when the value of a specific property changes. The class defined for this example uses a method for a push-button callback and a static method for the listener callback. When the push-button callback changes the value of a property, the listener executes its callback on the PreSet event.

This example defines a class (PushButton) with these design elements:

• ResultNumber - Observable property
• uicontrol pushbutton - Push-button object used to generate a new graph when its callback executes
• A listener that responds to a change in the observable property

PushButton Class Design

The PushButton class creates figure, uicontrol, axes graphics objects, and a listener object in the class constructor.

The push button's callback is a class method (named pressed). When the push button is activated, the following sequence occurs:

1 MATLAB executes the pressed method, which graphs a new set of data and increments the ResultNumber property.
2 Attempting to set the value of the ResultNumber property triggers the PreSet event, which executes the listener callback before setting the property value.
3 The listener callback uses the event data to obtain the handle of the callback object (an instance of the PushButton class), which then provides the handle of the axes object that is stored in its AxHandle property.
4 The listener callback updates the axes Title property, after the callback completes execution, MATLAB sets the ResultsNumber property to its new value.

```matlab
classdef PushButton < handle
    properties (SetObservable)
        ResultNumber = 1
    end
    properties
        AxHandle
```

11-11
methods
    function buttonObj = PushButton
        myFig = figure;
        buttonObj.AxHandle = axes('Parent',myFig);
        uicontrol('Parent',myFig,...
            'Style','pushbutton',...
            'String','Plot Data',...
            'Callback',@(src,evnt)pressed(buttonObj));
        addlistener(buttonObj,'ResultNumber','PreSet',...
            @PushButton.updateTitle);
    end
end
methods
    function pressed(obj)
        scatter(obj.AxHandle,randn(1,20),randn(1,20),'p')
        obj.ResultNumber = obj.ResultNumber + 1;
    end
end
methods (Static)
    function updateTitle(~,eventData)
        h = eventData.AffectedObject;
        set(get(h.AxHandle,'Title'),'String',
            [num2str(h.ResultNumber)])
    end
end
end

The scatter graph looks similar to this graph after three push-button clicks.

buttonObj = PushButton;
See Also

Related Examples
• “Listen for Changes to Property Values” on page 11-40
Event and Listener Concepts

In this section...

| “The Event Model” on page 11-14 |
| “Limitations” on page 11-15 |
| “Default Event Data” on page 11-16 |
| “Events Only in Handle Classes” on page 11-16 |
| “Property-Set and Query Events” on page 11-17 |
| “Listeners” on page 11-17 |

The Event Model

Events represent changes or actions that occur within objects. For example,

- Modification of class data
- Execution of a method
- Querying or setting a property value
- Destruction of an object

Basically, any activity that you can detect programmatically can generate an event and communicate information to other objects.

MATLAB classes define a process that communicates the occurrence of events to other objects that respond to the events. The event model works this way:

- A handle class declares a name used to represent an event. “Name Events” on page 11-22
- After creating an object of the event-declaring class, attach listener to that object. “Control Listener Lifecycle” on page 11-28
- A call to the handle class notify method broadcasts a notice of the event to listeners. The class user determines when to trigger the event. “Trigger Events” on page 11-23
- Listeners execute a callback function when notified that the event has occurred. “Specifying Listener Callbacks” on page 11-30
- You can bind listeners to the lifecycle of the object that defines the event, or limit listeners to the existence and scope of the listener object. “Control Listener Lifecycle” on page 11-28
The following diagram illustrates the event model.

1. The **withdraw** method is called.

   ```
   if AccountBalance <= 0
     notify(obj,'InsufficientFunds');
   end
   ```

2. The **notify** method triggers an event, and a message is broadcast.

3. Listeners awaiting message execute their callbacks.
   (The broadcasting object does not necessarily know who is listening.)

**Limitations**

There are certain limitations to the use of events:

- The event source cannot guarantee that listeners exist when triggering the event.
- A listener cannot prevent other listeners from being notified that the event occurred.
• The order in which listeners execute is not defined.
• Listeners should not modify the event data object passed to the listener callback, because other listeners are passed this same handle object.

Default Event Data

Events provide information to listener callbacks by passing an event data argument to the callback function. By default, MATLAB passes an `event.EventData` object to the listener callback. This object has two properties:

• **EventName** — The event name as defined in the class `event` block
• **Source** — The object that is the source of the event

MATLAB passes the source object to the listener callback in the required event data argument. Use the source object to access any of the object's public properties from within your listener callback function.

Customize Event Data

You can create a subclass of the `event.EventData` class to provide additional information to listener callback functions. The subclass would define properties to contain the additional data and provide a method to construct the derived event data object so it can be passed to the `notify` method.

“Define Event-Specific Data” on page 11-26 provides an example showing how to customize this data.

Events Only in Handle Classes

You can define events only in handle classes. This restriction exists because a value class is visible only in a single MATLAB workspace so no callback or listener can have access to the object that triggered the event. The callback could have access to a copy of the object. However, accessing a copy is not useful because the callback cannot access the current state of the object that triggered the event or effect any changes in that object.

“Comparison of Handle and Value Classes” on page 7-2 provides general information on handle classes.

“Events and Listeners Syntax” on page 11-22 shows the syntax for defining a handle class and events.
Property-Set and Query Events

There are four predefined events related to properties:

- **PreSet** — Triggered just before the property value is set, before calling its set access method
- **PostSet** — Triggered just after the property value is set
- **PreGet** — Triggered just before a property value query is serviced, before calling its get access method
- **PostGet** — Triggered just after returning the property value to the query

These events are predefined and do not need to be listed in the class `events` block.

When a property event occurs, the callback is passed an `event.PropertyEvent` object. This object has three properties:

- **EventName** — The name of the event described by this data object
- **Source** — The source object whose class defines the event described by the data object
- **AffectedObject** — The object whose property is the source for this event (that is, `AffectedObject` contains the object whose property was either accessed or modified).

You can define your own property-change event data by subclassing the `event.EventData` class. The `event.PropertyEvent` class is a sealed subclass of `event.EventData`.

See “Listen for Changes to Property Values” on page 11-40 for a description of the process for creating property listeners.

See “The PostSet Event Listener” on page 11-58 for an example.

See “Property Access Methods” on page 8-49 for information on methods that control access to property values.

Listeners

Listeners encapsulate the response to an event. Listener objects belong to the `event.listener` class, which is a handle class that defines the following properties:
• **Source** — Handle or array of handles of the object that generated the event
• **EventName** — Name of the event
• **Callback** — Function to execute when an enabled listener receives event notification
• **Enabled** — Callback function executes only when Enabled is true. See “Enable and Disable Listeners” on page 11-61 for an example.
• **Recursive** — Allow listener to trigger the same event that caused execution of the callback.

Recursive is false by default. If the callback triggers the event for which it is defined as the callback, the listener cannot execute recursively. Therefore, set Recursive to false if the callback must trigger its own event. Setting the Recursive property to true can create a situation where infinite recursion reaches the recursion limit and triggers an error.

“Control Listener Lifecycle” on page 11-28 provides more specific information.
Event Attributes

Specify Event Attributes

The following table lists the attributes you can set for events. To specify a value for an attribute, assign the attribute value on the same line as the event keyword. For example, all the events defined in the following events block have protected ListenAccess and private NotifyAccess.

```
events (ListenAccess = protected, NotifyAccess = private)
    EventName1
    EventName2
end
```

To define other events in the same class definition that have different attribute settings, create another events block.
### Event Attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hidden</td>
<td>logical, Default = false</td>
<td>If true, event does not appear in list of events returned by events function (or other event listing functions or viewers).</td>
</tr>
</tbody>
</table>

**ListenAccess**

- enumeration, default = public
- meta.class object
- cell array of meta.class objects

Determines where you can create listeners for the event.

- public — Unrestricted access
- protected — Access from methods in class or subclasses
- private — Access by class methods only (not from subclasses)

List classes that have listen access to this event. Specify classes as meta.class objects in the form:

- A single meta.class object
- A cell array of meta.class objects. An empty cell array, {}, is the same as private access.

See “Class Members Access” on page 12-28

**NotifyAccess**

- enumeration, default = public
- meta.class object
- cell array of meta.class objects

Determines where code can trigger the event

- public — Any code can trigger event
- protected — Can trigger event from methods in class or derived classes
- private — Can trigger event by class methods only (not from derived classes)

List classes that have notify access to this event. Specify classes as meta.class objects in the form:

- A single meta.class object
- A cell array of meta.class objects. An empty cell array, {}, is the same as private access.

See “Class Members Access” on page 12-28
See Also

Related Examples

- “Events and Listeners Syntax” on page 11-22
Components to Implement

Implementation of events and listeners involves these components:

- Specification of the name of an event in a handle class — “Name Events” on page 11-22.
- A function or method to trigger the event when the action occurs — “Trigger Events” on page 11-23.
- Listener objects to execute callback functions in response to the triggered event — “Listen to Events” on page 11-23.
- Default or custom event data that the event passes to the callback functions — “Define Event-Specific Data” on page 11-26.

Name Events

Define an event by declaring an event name inside an events block. For example, this class creates an event called ToggledState:

```matlab
classdef ToggleButton < handle
    properties
        State = false
    end
    events
        ToggledState
    end
end```
Trigger Events

The `OnStateChange` method calls `notify` to trigger the `ToggledState` event. Pass the handle of the object that is the source of the event and the name of the event to `notify`.

```matlab
classdef ToggleButton < handle
    properties
        State = false
    end
    events
        ToggledState
    end
    methods
        function OnStateChange(obj,newState)
            if newState ~= obj.State
                obj.State = newState;
                notify(obj,'ToggledState');
            end
        end
    end
end
```

Listen to Events

After the call to `notify` triggers an event, MATLAB broadcasts a message to all listeners that are defined for that event and source object. There are two ways to create listeners: using the handle class `handle.addlistener` or `handle.listener` method.

Use `addlistener` for Persistent Listeners

If you want the listener to persist beyond the normal variable scope, use `addlistener` to create it. The event source object holds a reference to the listener object. When the event source object is destroyed, MATLAB destroys the listener.

This code defines a listener for the `ToggleState` event:

```matlab
lh = addlistener(obj,'ToggleState',@RespondToToggle.handleEvnt);
```

`addlistener` has these arguments:

- `obj` — The object that is the source of the event
- `ToggleState` — The event name passed as a char vector
@RespondToToggle.handleEvnt — A function handle to the callback function (see the following definition “Define Listener” on page 11-24).

Use handle.listener to Decouple Listener and Source

Use the handle.listener method to create listeners when you want to manage the lifecycle of the listener and do not want a coupling between the event source and listener object. MATLAB does not destroy listeners created with handle.listener when the event source is destroyed. However, your code must keep the listener object handle in scope when creating listeners using handle.listener.

The handle.listener method requires the same arguments as handle.addlistener: the event-naming object, the event name, and a function handle to the callback. handle.listener returns the handle to the listener object.

\[ \text{lh} = \text{listener(obj,'EventName',@callbackFunction)} \]

For example, this code uses the ToggleState event discussed previously:

\[ \text{lh} = \text{listener(obj,'ToggleState',@RespondToToggle.handleEvnt)} \]

Callback Function

The listener callback function must accept a minimum of two arguments, which MATLAB automatically passes to the callback. Here are the required arguments:

- The source of the event — that is, obj in the call to addlistener or event.listener.
- An event.EventData object or a subclass of event.EventData, such as the ToggleEventData object described in, “Define Event-Specific Data” on page 11-26.

Define the callback function to accept the source object and event data arguments.

\[
\text{function callbackFunction(src,evtdata)} \\
\text{...} \\
\text{end}
\]

For more information on callback syntax, see “Listener Callback Syntax” on page 11-30.

Define Listener

The RespondToToggle class defines objects that listen for the ToggleState event defined in the ToggleButton class.
classdef RespondToToggle < handle
  methods
    function obj = RespondToToggle(toggle_button_obj)
      addlistener(toggle_button_obj,'ToggledState',@RespondToToggle.handleEvnt);
    end
  end
  methods (Static)
    function handleEvnt(src,~)
      if src.State
        disp('ToggledState is true')
      else
        disp('ToggledState is false')
      end
    end
  end
end

The class RespondToToggle adds the listener in its constructor. In this case, the class defines the callback (handleEvnt) as a static method that accepts the two required arguments:

- **src** — The handle of the object triggering the event (that is, a ToggleButton object)
- **evtdata** — An event.EventData object

For example, this code creates objects of both classes:

```matlab
tb = ToggleButton;
rtt = RespondToToggle(tb);
```

Whenever you call the OnStateChange method of the ToggleButton object, notify triggers the event. For this example, the callback displays the value of the State property:

```matlab
tb.OnStateChange(true)
ToggledState is true

tb.OnStateChange(false)
ToggledState is false
```

**Remove Listeners**

Remove a listener object by calling delete on its handle. For example, if the class RespondToToggle saved the listener handle as a property, you could delete the listener.

```matlab
classdef RespondToToggle < handle
    properties
```
ListenerHandle % Property for listener handle
end
methods
    function obj = RespondToToggle(toggle_button_obj)
        hl = addlistener(toggle_button_obj,'ToggledState',@RespondToToggle.handleEvnt);
        obj.ListenerHandle = hl; % Save listener handle
    end
end
methods (Static)
    function handleEvnt(src,~)
        if src.State
            disp('ToggledState is true')
        else
            disp('ToggledState is false')
        end
    end
end
end

With this code change, you can remove the listener from an instance of the RespondToToggle class. For example:

tb = ToggleButton;
rtt = RespondToToggle(tb);

The object rtt is listening for the ToggledState event triggered by object tb. To remove the listener, call delete on the property containing the listener handle.

delete(rtt.ListenerHandle)

To deactivate a listener temporarily, see “Temporarily Deactivate Listeners” on page 11-28.

Define Event-Specific Data

Suppose that you want to pass the state of the toggle button as a result of the event to the listener callback. You can add more data to the default event data by subclassing the event.EventData class and adding a property to contain this information. Then you can pass this object to the handle.notify method.

Note To save and load objects that are subclasses of event.EventData, such as ToggleEventData, enable the ConstructOnLoad class attribute for the subclass.

classdef (ConstructOnLoad) ToggleEventData < event.EventData
    properties
function data = ToggleEventData(newState)
    data.NewState = newState;
end
end

The call to notify can use the ToggleEventData constructor to create the necessary argument.

evtdata = ToggleEventData(newState);
notify(obj,'ToggledState',evtdata);

See Also

Related Examples

• “Listener Callback Syntax” on page 11-30
• “Listen for Changes to Property Values” on page 11-40
• “Techniques for Using Events and Listeners” on page 11-48
Listener Lifecycle

In this section...

| “Control Listener Lifecycle” on page 11-28 |
| “Temporarily Deactivate Listeners” on page 11-28 |
| “Permanently Delete Listeners” on page 11-28 |

Control Listener Lifecycle

There are two ways to create listeners:

- `handle.addlistener` creates a coupling between the listener and event source object. The listener object persists until you delete it or until the event object is destroyed. When the event source object is destroyed, MATLAB automatically destroys the listener object.

- `handle.listener` constructs listener objects that are not coupled to the lifecycle of the event source object. The listener is active as long as the listener object remains in scope and is not explicitly deleted. Therefore, your application must maintain a reference to the listener object by storing the listener handle. The advantage of uncoupling the listener and event objects is that you can define and destroy each independently.

For more information, see “Events and Listeners Syntax” on page 11-22.

Temporarily Deactivate Listeners

The `addlistener` method returns the listener object so that you can set its properties. For example, you can temporarily disable a listener by setting its `Enabled` property to `false`:

```matlab
ListenerHandle.Enabled = false;
```

To reenable the listener, set `Enabled` to `true`:

```matlab
ListenerHandle.Enabled = true;
```

Permanently Delete Listeners

Calling `delete` on a listener object destroys it and permanently removes the listener:
delete(ListenerHandle)

**Note** Do not use the `pack` command with objects that define events and listeners. The `pack` command causes the destructor of any listeners defined for the objects in the workspace. For information on restoring listeners when saving objects, see “Restore Listeners” on page 13-36.

**See Also**

**Related Examples**
- “Enable and Disable Listeners” on page 11-61
Listener Callback Syntax

**Specifying Listener Callbacks**

Callbacks are functions that execute when the listener receives notification of the event. Pass a function handle referencing the callback function to `handle.addlistener` or `handle.listener` when creating the listener.

All callback functions must accept at least two arguments:

- The handle of the object that is the source of the event
- An `event.EventData` object or an object that is derived from the `event.EventData` class.

**Syntax to Reference Callback**

For a function: `functionName`

```matlab
lh = addlistener(eventSourceObj,'EventName',@functionName)
```

For an ordinary method called with an object of the class: `obj.methodName`

```matlab
lh = addlistener(eventSourceObj,'EventName',@obj.methodName)
```

For a static method: `ClassName.methodName`

```matlab
lh = addlistener(eventSourceObj,'EventName',@ClassName.methodName)
```

For a function in a package: `PackageName.functionName`

```matlab
lh = addlistener(eventSourceObj,'EventName',@PackageName.functionName)
```

**Input Arguments for Callback Function**

Define the callback function to accept the required arguments:
function callbackFunction(src,evnt)
    ...
end

If you do not use the event source and event data arguments, you can define the function to ignore these inputs:

function callbackFunction(~,~)
    ...
end

For a method:

function callbackMethod(obj,src,evnt)
    ...
end

### Additional Arguments for Callback Function

To pass arguments to your callback in addition to the source and event data arguments passed by MATLAB, use an anonymous function. Anonymous functions can use any variables that are available in the current workspace.

#### Syntax Using Anonymous Function

Here is the syntax for an ordinary method. The input arguments \( \text{arg1, \ldots argn} \) must be defined in the context in which you call `addlistener`.

\[
\text{lh} = \text{addlistener(src, 'EventName', @(src,evnt)obj.callbackMethod(src,evnt,arg1,...argn)}
\]

Use `varargin` to define the callback function.

function callbackMethod(src,evnt,varargin)
    arg1 = varargin{1};
    ...
    argn = varargin{n};
    ...
end

For general information on anonymous function, see “Anonymous Functions”.
Using Methods for Callbacks

The TestAnonyFcn class shows the use of an anonymous function with an additional argument. The listener callback displays the inputs arguments to show how MATLAB calls the callback method.

```matlab
classdef TestAnonyFcn < handle
    events
        Update
    end
    methods
        function obj = TestAnonyFcn
            t = datestr(now);
            addlistener(obj,'Update',@(src,evnt)obj.evntCb(src,evnt,t));
        end
        function triggerEvnt(obj)
            notify(obj,'Update')
        end
    end
    methods (Access = private)
        function evntCb(~,~,evnt,varargin)
            disp(['Number of inputs: ',num2str(nargin)])
            disp(evnt.EventName)
            disp(varargin{:})
        end
    end
end

Create an object and trigger the event by calling the triggerEvnt method:

```matlab
obj = TestAnonyFcn;
obj.triggerEvnt;
```

Number of inputs: 4
Update
01-Jul-2008 17:19:36

See Also

Related Examples

  •  “Callback Execution” on page 11-34
• “Create Function Handle”
Callback Execution

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“When Callbacks Execute” on page 11-34</td>
</tr>
<tr>
<td>“Listener Order of Execution” on page 11-34</td>
</tr>
<tr>
<td>“Callbacks That Call notify” on page 11-34</td>
</tr>
<tr>
<td>“Manage Callback Errors” on page 11-35</td>
</tr>
<tr>
<td>“Invoke Functions from Function Handles” on page 11-35</td>
</tr>
</tbody>
</table>

**When Callbacks Execute**

Listeners execute their callback function when notified that the event has occurred. Listeners are passive observers in the sense that errors in the execution of a listener callback do not prevent the execution of other listeners responding to the same event, or execution of the function that triggered the event.

Callback function execution continues until the function completes. If an error occurs in a callback function, execution stops and control returns to the calling function. Then any remaining listener callback functions execute.

**Listener Order of Execution**

The order in which listeners callback functions execute after the firing of an event is undefined. However, all listener callbacks execute synchronously with the event firing.

The handle class `handle.notify` method calls all listeners before returning execution to the function that called `notify`.

**Callbacks That Call notify**

Do not modify and reuse or copy and reuse the event data object that you pass to `notify`, which is then passed to the listener callback.

Listener callbacks can call `notify` to trigger events, including the same event that invoked the callback. When a function calls `notify`, MATLAB sets the property values of the event data object that is passed to callback functions. To ensure that these properties
have appropriate values for subsequently called callbacks, always create a new event data object if you call notify with custom event data.

Manage Callback Errors

If you want to control how your program responds to errors, use a try/catch statement in your listener callback function to handle errors.

Invoke Functions from Function Handles

When you create a function handle inside a class method, the context of the method determines the context in which the function executes. This context gives the function access to private and protected methods that are accessible to that class.

For example, the UpdateEvt class defines an event named Update and a listener for that event. The listener callback is the private method evtCb.

classdef UpdateEvt < handle
    events
        Update
    end
    methods
        function obj = UpdateEvt
            addlistener(obj,'Update',@evtCb);
        end
    end
    methods (Access = private)
        function obj = evtCb(obj,varargin)
            disp('Updated Event Triggered')
        end
    end
end

Private methods are normally accessible only by class methods. However, because the function handle is created in a class method, notify can execute the callback from outside of the class:

a = UpdateEvt;
a.notify('Update')

Updated Event Triggered
See Also

Related Examples
•  “Listener Callback Syntax” on page 11-30
Determine If Event Has Listeners

In this section...

“Do Listeners Exist for This Event?” on page 11-37
“Why Test for Listeners” on page 11-37
“Coding Patterns” on page 11-37
“Listeners in Heterogeneous Arrays” on page 11-38

Do Listeners Exist for This Event?

Use the event.hasListener function to determine if a specific event has listeners. event.hasListener accepts an array of event source objects and an event name as input arguments. It returns an array of logical values indicating if listeners exist for the specified event on each object in the array.

Note When called, event.hasListener must have NotifyAccess for the event. That is, call event.hasListener in a context in which you can call notify for the event in question.

Why Test for Listeners

Use event.hasListener to avoid sending event notifications when there are no listeners for the event. For example, if creating custom event data consumes significant resources, or if events are triggered repeatedly, use event.hasListener to test for listeners before performing these steps.

Coding Patterns

• Conditionalize the creation of event data and the call to notify using event.hasListener. For an object array a, determine if there are listeners before creating event data and triggering the event:

```javascript
if any(event.hasListener(a,"NameOfEvent"))
    evt = MyCustomEventData(...);
    notify(a,"NameOfEvent",evt)
end
```
- Trigger events selectively using logical indexing with the values returned by `event.hasListener`. Send event notifications only for array elements that have listeners:

```java
ind = event.hasListeners(a,'NameOfEvent');
notify(a(ind),'NameOfEvent',evt)
```

**Listeners in Heterogeneous Arrays**

If the input object array is heterogeneous, the class of the array must define the specified event. You can query the listeners only for events that all objects in the array define.

For example, in the following diagram, the class of a heterogeneous array formed with objects of classes `SpecificA`, `SpecificB`, and `SpecificC` is `RootSuperclass`. Therefore, `event.hasListener` can find listeners only for the `RootEvent` event, because it is the only event common to all array elements.

Create a heterogeneous array with the three most specific classes:
het = [SpecificA, SpecificB, SpecificC];
class(het)
ans
RootSuperclass
events(het)
Events for class RootSuperclass
     RootEvent

event.hasListener cannot determine if there are listeners for events that are defined by some but not all objects in the array:
event.hasListener(het,'ClassAEvent')
Error using event.hasListener
Event 'ClassAEvent' is not defined for class 'RootSuperclass'.

Determine if individual objects in the heterogeneous array have listeners defined for their specific events, by indexing into the array:
event.hasListener(het(1), 'ClassAEvent')

For more information about determining the class of heterogeneous arrays, see “Designing Heterogeneous Class Hierarchies” on page 10-24.

See Also

Related Examples
•   "Listener Lifecycle" on page 11-28
Listen for Changes to Property Values

Create Property Listeners

For handle classes, you can define listeners for the predeclared property events (named: PreSet, PostSet, PreGet, and PostGet). To create listeners for those named events:

- Specify the SetObservable and/or GetObservable property attributes.
- Define callback functions
- Create the property listener by including the name of the property and the event in the call to handle.addlistener or handle.listener.
- If necessary, subclass event.data to create a specialized event data object to pass to the callback function.
- Prevent execution of the callback if the new value is the same as the current value (see “Assignment When Property Value Unchanged” on page 11-44).

Set Property Attributes to Enable Property Events

In the properties block, enable the SetObservable attribute. You can define PreSet and PostSet listeners for the properties defined in this block:

```
properties (SetObservable)
    PropOne
    PropTwo
...
end
```

Define Callback Function for Property Event

The listener executes the callback function when MATLAB triggers the property event. Define the callback function to have two specific arguments, which are passed to the function automatically when called by the listener:

- Event source — a meta.property object describing the object that is the source of the property event
**Event data** — a `event.PropertyEvent` object containing information about the event

You can pass additional arguments if necessary. It is often simple to define this method as Static because these two arguments contain most necessary information in their properties.

For example, suppose the `handlePropEvents` function is a static method of the class creating listeners for two properties of an object of another class:

```plaintext
methods (Static)
    function handlePropEvents(src,evnt)
        switch src.Name
        case 'PropOne'
            % PropOne has triggered an event
            ...
        case 'PropTwo'
            % PropTwo has triggered an event
            ...
        end
    end
end
```

Another possibility is to use the `event.PropertyEvent` object's `EventName` property in the `switch` statement to key off the event name (`PreSet` or `PostSet` in this case).

“Class Metadata” on page 16-2 provides more information about the `meta.property` class.

**Add Listener to Property**

The `handle.addlistener` handle class method enables you to attach a listener to a property without storing the listener object as a persistent variable. For a property event, use the four-argument version of `addlistener`.

Here is a call to `addlistener`:

```plaintext
addlistener(EventObject,'PropOne','PostSet',@ClassName.handlePropertyEvents);
```

The arguments are:

- `EventObject` — handle of the object generating the event
- `PropOne` — name of the property to which you want to listen
- `PostSet` — name of the event for which you want to listen
- `@ClassName.handlePropertyEvents` — function handle referencing a static method, which requires the use of the class name
If your listener callback is an ordinary method and not a static method, the syntax is:
addlistener(EventObject,'PropOne','PostSet',@obj.handlePropertyEvents);

where \textit{obj} is the handle of the object defining the callback method.

If the listener callback is a function that is not a class method, you pass a function handle to that function. Suppose that the callback function is a package function:
addlistener(EventObject,'PropOne','PostSet',@package.handlePropertyEvents);

For more information on passing functions as arguments, see “Create Function Handle”.

\textbf{Property Event and Listener Classes}

The following two classes show how to create PostSet property listeners for two properties — PropOne and PropTwo.

\textbf{Class Generating the Event}

The PropEvent class enables property PreSet and PostSet event triggering by specifying the SetObservable property attribute. These properties also enable the AbortSet attribute, which prevents the triggering of the property events if the properties are set to a value that is the same as their current value (see “Assignment When Property Value Unchanged” on page 11-44).

\begin{verbatim}
classdef PropEvent < handle
    properties (SetObservable, AbortSet)
        PropOne
        PropTwo
    end
methods
    function obj = PropEvent(p1,p2)
        if nargin > 0
            obj.PropOne = p1;
            obj.PropTwo = p2;
        end
    end
end
\end{verbatim}

\textbf{Class Defining the Listeners}

The PropListener class defines two listeners:
• Property PropOne PostSet event
• Property PropTwo PostSet event

You can define listeners for other events or other properties using a similar approach. It is not necessary to use the same callback function for each listener. See the meta.property and event.PropertyEvent reference pages for more on the information contained in the arguments passed to the listener callback function.

classdef PropListener < handle
% Define property listeners
methods
    function obj = PropListener(evtobj)
      if nargin > 0
        addlistener(evtobj,'PropOne','PostSet',@PropListener.handlePropEvents);
        addlistener(evtobj,'PropTwo','PostSet',@PropListener.handlePropEvents);
      end
    end
end
methods (Static)
    function handlePropEvents(src,evnt)
      switch src.Name
        case 'PropOne'
          sprintf('PropOne is %s
',num2str(evnt.AffectedObject.PropOne))
        case 'PropTwo'
          sprintf('PropTwo is %s
',num2str(evnt.AffectedObject.PropTwo))
      end
    end
end
end

See Also

Related Examples
• “Assignment When Property Value Unchanged” on page 11-44
Assignment When Property Value Unchanged

AbortSet When Value Does Not Change

When you set a property value, MATLAB triggers the property PreSet and PostSet events, invokes the property set method (if one is defined), and sets the property value. These actions occur even when the current value of the property is the same as the new value.

You can prevent these actions by setting the property's AbortSet attribute to true. When AbortSet is enabled, MATLAB compares the current property value to the new value being assigned to the property. If the new value is the same as the current value, MATLAB does not:

- Set the property value.
- Trigger the PreSet and PostSet events.
- Call the property set method, if one exists.

To compare values, MATLAB must get the current value of the property. Getting the current value causes the property get method (get.Property) to execute, if one exists. Any errors that occur when calling the property get method are visible to the user, even if MATLAB does not change the current value.

How MATLAB Compares Values

MATLAB uses the isequal function to determine if the current value of the property is the same as the new value. To determine if specific values evaluate as equal when using the AbortSet attribute, see the isequal function documentation or any isequal method overloaded for the class of the property value.
When to Use AbortSet

Use of the AbortSet attribute does incur some overhead in the comparison of the current and new property values. However, enabling AbortSet is useful in these cases.

- The cost of setting a property value is greater than the cost of comparing the current value of the property with the new value being assigned.
- You want to prevent notification of the PreSet and PostSet events and execution of the listener callbacks when the property value does not change.

Implement AbortSet

The following example shows how the AbortSet attribute works. The AbortTheSet class defines a property, PropOne, that has listeners for the PreGet and PreSet events and enables the AbortSet attribute. The behavior of the post set/get events is equivalent so only the pre set/get events are used for simplicity:

Note To use this class, save the AbortTheSet class in a file with the same name in a folder on your MATLAB path.

classdef AbortTheSet < handle
    properties (SetObservable, GetObservable, AbortSet)
        PropOne = 7
    end
    methods
        function obj = AbortTheSet(val)
            obj.PropOne = val;
            addlistener(obj, 'PropOne', 'PreGet', @obj.getPropEvt);
            addlistener(obj, 'PropOne', 'PreSet', @obj.setPropEvt);
        end
        function propval = get.PropOne(obj)
            disp('get.PropOne called')
            propval = obj.PropOne;
        end
        function set.PropOne(obj, val)
            disp('set.PropOne called')
            obj.PropOne = val;
        end
        function getPropEvt(obj, src, evnt)
            disp('Pre-get event triggered')
        end
    end
end
function setPropEvt(obj,src,evnt)
    disp ('Pre-set event triggered')
end
function disp(obj)
    % Overload disp to avoid accessing property
    disp (class(obj))
end
end

The class specifies an initial value of 7 for the PropOne property. Therefore, if you create an object with the property value of 7, there is no need to trigger the PreSet event:

ats = AbortTheSet(7);
get.PropOne called

If you specify a value other than 7, then MATLAB triggers the PreSet event:

ats = AbortTheSet(9);
get.PropOne called
set.PropOne called
get.PropOne called

Similarly, if you set the PropOne property to the value 9, the AbortSet attribute prevents the property assignment and the triggering of the PreSet event. Notice also, that there is no PreGet event generated. Only the property get method is called:

ats.PropOne = 9;
get.PropOne called

If you query the property value, the PreGet event is triggered:

a = ats.PropOne

Pre-get event triggered
get.PropOne called
a =
    9
If you set the `PropOne` property to a different value, MATLAB:

- Calls the property get method to determine if the value is changing
- Triggers the `PreSet` event
- Calls the property set method to set the new value
- Calls the property get method again to determine if the result of calling the set method changed the value.

```matlab
ats.PropOne = 11;
```

get.PropOne called
Pre-set event triggered
set.PropOne called
get.PropOne called

Because a property set method can modify the value that is assigned to a property, MATLAB must query the property value that would result from an assignment after a call to the property set method. The result is multiple calls to a property get method, if one is defined for that property.

**See Also**

**Related Examples**

- “Property Access Methods” on page 8-49
- “Determine If Event Has Listeners” on page 11-37
Techniques for Using Events and Listeners

In this section...

“Example Overview” on page 11-48
“Techniques Demonstrated in This Example” on page 11-49
“Summary of fcneval Class” on page 11-49
“Summary of fcnview Class” on page 11-50
“Methods Inherited from Handle Class” on page 11-52
“Using the fcneval and fcnview Classes” on page 11-52
“Implement UpdateGraph Event and Listener” on page 11-54
“The PostSet Event Listener” on page 11-58
“Enable and Disable Listeners” on page 11-61
“@fcneval/fcneval.m Class Code” on page 11-62
“@fcnview/fcnview.m Class Code” on page 11-63

Example Overview

This example defines two classes:

- fcneval — The function evaluator class contains a MATLAB expression and evaluates this expression over a specified range
- fcnview — The function viewer class contains a fcneval object and displays surface graphs of the evaluated expression using the data contained in fcneval.

This class defines two events:

- A class-defined event that occurs when a new value is specified for the MATLAB function
- A property event that occurs when the property containing the limits is changed

The following diagram shows the relationship between the two objects. The fcnview object contains a fcneval object and creates graphs from the data it contains. fcnview creates listeners to change the graphs if any of the data in the fcneval object change.
Techniques Demonstrated in This Example

- Naming an event in the class definition
- Triggering an event by calling `notify`
- Enabling a property event via the `SetObservable` attribute
- Creating listeners for class-defined events and property `PostSet` events
- Defining listener callback functions that accept additional arguments
- Enabling and disabling listeners

Summary of fcneval Class

The `fcneval` class evaluates a MATLAB expression over a specified range of two variables. The `fcneval` is the source of the data that objects of the `fcnview` class graph as a surface. `fcneval` is the source of the events used in this example. For a listing of the class definition, see “@fcneval/fcneval.m Class Code” on page 11-62
<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>FofXY</td>
<td>function handle</td>
<td>MATLAB expression (function of two variables).</td>
</tr>
<tr>
<td>Lm</td>
<td>two-element vector</td>
<td>Limits over which function is evaluated in both variables. SetObservable attribute set to true to enable property event listeners.</td>
</tr>
<tr>
<td>Data</td>
<td>structure with x, y, and z matrices</td>
<td>Data resulting from evaluating the function. Used for surface graph. Dependent attribute set to true, which means the get.Data method is called to determine property value when queried and no data is stored.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>When Triggered</th>
</tr>
</thead>
<tbody>
<tr>
<td>UpdateGraph</td>
<td>FofXY property set function (set.FofXY) calls the notify method when a new value is specified for the MATLAB expression on an object of this class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcneval</td>
<td>Class constructor. Inputs are function handle and two-element vector specifying the limits over which to evaluate the function.</td>
</tr>
<tr>
<td>set.FofXY</td>
<td>FofXY property set function. Called whenever property value is set, including during object construction.</td>
</tr>
<tr>
<td>set.Lm</td>
<td>Lm property set function. Used to test for valid limits.</td>
</tr>
<tr>
<td>get.Data</td>
<td>Data property get function. This method calculates the values for the Data property whenever that data is queried (by class members or externally).</td>
</tr>
<tr>
<td>grid</td>
<td>A static method (Static attribute set to true) used in the calculation of the data.</td>
</tr>
</tbody>
</table>

**Summary of fcnview Class**

Objects of the fcnview class contain fcneval objects as the source of data for the four surface graphs created in a function view. fcnview creates the listeners and callback functions that respond to changes in the data contained in fcneval objects. For a listing of the class definition, see “@fcnview/fcnview.m Class Code” on page 11-63.
### Property

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>FcnObject</td>
<td>fcneval object</td>
<td>This object contains the data that is used to create the function graphs.</td>
</tr>
<tr>
<td>HAxes</td>
<td>axes handle</td>
<td>Each instance of a fcnview object stores the handle of the axes containing its subplot.</td>
</tr>
<tr>
<td>HLUpdateGraph</td>
<td>event.listener object for UpdateGraph event</td>
<td>Setting the event.listener object's Enabled property to true enables the listener; false disables listener.</td>
</tr>
<tr>
<td>HLLm</td>
<td>event.listener object for Lm property event</td>
<td>Setting the event.listener object's Enabled property to true enables the listener; false disables listener.</td>
</tr>
<tr>
<td>HEnableCm</td>
<td>uimenu handle</td>
<td>Item on context menu used to enable listeners (used to handle checked behavior)</td>
</tr>
<tr>
<td>HDisableCm</td>
<td>uimenu handle</td>
<td>Item on context menu used to disable listeners (used to manage checked behavior)</td>
</tr>
<tr>
<td>HSurface</td>
<td>surface handle</td>
<td>Used by event callbacks to update surface data.</td>
</tr>
</tbody>
</table>

### Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fcnview</td>
<td>Class constructor. Input is fcneval object.</td>
</tr>
<tr>
<td>createLisn</td>
<td>Calls addlistener to create listeners for UpdateGraph and Lm property PostSet listeners.</td>
</tr>
<tr>
<td>lims</td>
<td>Sets axes limits to current value of fcneval object's Lm property. Used by event handlers.</td>
</tr>
<tr>
<td>updateSurfaceData</td>
<td>Updates the surface data without creating a new object. Used by event handlers.</td>
</tr>
<tr>
<td>listenUpdateGraph</td>
<td>Callback for UpdateGraph event.</td>
</tr>
<tr>
<td>listenLm</td>
<td>Callback for Lm property PostSet event</td>
</tr>
<tr>
<td>delete</td>
<td>Delete method for fcnview class.</td>
</tr>
<tr>
<td>createViews</td>
<td>Static method that creates an instance of the fcnview class for each subplot, defines the context menus that enable/disable listeners, and creates the subplots</td>
</tr>
</tbody>
</table>
**Methods Inherited from Handle Class**

Both the `fcneval` and `fcnview` classes inherit methods from the `handle` class. The following table lists only those inherited methods used in this example.

“Handle Class Methods” on page 7-13 provides a complete list of methods that are inherited when you subclass the `handle` class.

<table>
<thead>
<tr>
<th>Methods Inherited from Handle Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>addlistener</td>
<td>Register a listener for a specific event and attach listener to event-defining object.</td>
</tr>
<tr>
<td>notify</td>
<td>Trigger an event and notify all registered listeners.</td>
</tr>
</tbody>
</table>

**Using the fcneval and fcnview Classes**

This section explains how to use the classes.

- Create an instance of the `fcneval` class to contain the MATLAB expression of a function of two variables and the range over which you want to evaluate this function
- Use the `fcnview` class static function `createView` to visualize the function
- Change the MATLAB expression or the limits contained by the `fcneval` object and all the `fcnview` objects respond to the events generated.

You create a `fcneval` object by calling its constructor with two arguments—an anonymous function and a two-element, monotonically increasing vector. For example:

```matlab
feobject = fcneval(@(x,y) x.*exp(-x.^2-y.^2),[-2 2]);
```

Use the `createView` static method to create the graphs of the function. Use the class name to call a static function:

```matlab
fcnview.createViews(feobject);
```

The `createView` method generates four views of the function contained in the `fcneval` object.
Each subplot defines a context menu that can enable and disable the listeners associated with that graph. For example, if you disable the listeners on subplot 221 (upper left) and change the MATLAB expression contained by the fcneval object, only the remaining three subplots update when the UpdateGraph event is triggered:

```matlab
tfeobject.FofXY = @(x,y) x.*exp(-x.^0.5-y.^0.5);
```

Similarly, if you change the limits by assigning a value to the `feobject.Lm` property, the `feobject` triggers a `PostSet` property event and the listener callbacks update the graph.

```matlab
feobject.Lm = [-8 3];
```

In this figure, the listeners are reenabled via the context menu for subplot 221. Because the listener callback for the property `PostSet` event also updates the surface data, all views are now synchronized.

**Implement UpdateGraph Event and Listener**

The `UpdateGraph` event occurs when the MATLAB representation of the mathematical function contained in the `fcneval` object is changed. The `fcnview` objects that contain the surface graphs are listening for this event, so they can update the graphs to represent the new function.

**Define and Trigger UpdateGraph Event**

The `UpdateGraph` event is a class-defined event. The `fcneval` class names the event and calls `notify` when the event occurs.
1. A property is assigned a new value.
   \[ \text{obj.FofXY} = @(x,y)x^2 + y^2 \]

2. Setting the property runs a set access method, which, in turn, executes \texttt{notify}.

```matlab
function set.FofXY(obj,func)
    obj.FofXY = func;
    notify(obj,'UpdateGraph');
end
```

3. The \texttt{notify} method triggers an event, and a message is broadcast.

4. A listener awaiting the message executes its callback.

5. The callback function is executed.

The \texttt{fcnview} class defines a listener for this event. When \texttt{fneval} triggers the event, the \texttt{fncview} listener executes a callback function that performs the follow actions:

- Determines if the handle of the surface object stored by the \texttt{fncview} object is still valid (that is, does the object still exist)
- Updates the surface \texttt{XData}, \texttt{YData}, and \texttt{ZData} by querying the \texttt{fncval} object's \texttt{Data} property.

The \texttt{fncval} class defines an event name in an event block:
**Determine When to Trigger Event**

The `fcneval` class defines a property set method for the `FofXY` property. `FofXY` is the property that stores the MATLAB expression for the mathematical function. This expression must be a valid MATLAB expression for a function of two variables.

The `set.FofXY` method:

- Determines the suitability of the expression
- If the expression is suitable:
  - Assigns the expression to the `FofXY` property
  - Triggers the `UpdateGraph` event

If `fcneval.isSuitable` does not return an `MException` object, the `set.FofXY` method assigns the value to the property and triggers the `UpdateGraph` event.

```matlab
function set.FofXY(obj,func)
    % Determine if function is suitable to create a surface
    me = fcneval.isSuitable(func);
    if ~isempty(me)
        throw(me)
    end
    % Assign property value
    obj.FofXY = func;
    % Trigger UpdateGraph event
    notify(obj,'UpdateGraph');
end
```

**Determine Suitability of Expression**

The `set.FofXY` method calls a static method (`fcneval.isSuitable`) to determine the suitability of the specified expression. `fcneval.isSuitable` returns an `MException` object if it determines that the expression is unsuitable. `fcneval.isSuitable` calls the `MException` constructor directly to create more useful error messages for the user.

`set.FofXY` issues the exception using the `MException.throw` method. Issuing the exception terminates execution of `set.FofXY` and prevents the method from making an assignment to the property or triggering the `UpdateGraph` event.
Here is the `fcneval.isSuitable` method:

```matlab
function isOk = isSuitable(funcH)
    v = [1 1;1 1];
    % Can the expression except 2 numeric inputs
    try
        funcH(v,v);
    catch %#ok<CTCH>
        me = MException('DocExample:fcneval','...
             ['The function ',func2str(funcH),' Is not a suitable F(x,y)']);
        isOk = me;
        return
    end
    % Does the expression return non-scalar data
    if isscalar(funcH(v,v));
        me = MException('DocExample:fcneval','...
             ['The function ',func2str(funcH),' Returns a scalar when evaluated']);
        isOk = me;
        return
    end
    isOk = [];
end
```

The `fcneval.isSuitable` method could provide additional test to ensure that the expression assigned to the FofXY property meets the criteria required by the class design.

**Other Approaches**

The class could have implemented a property set event for the FofXY property and would, therefore, not need to call notify (see “Listen for Changes to Property Values” on page 11-40). Defining a class event provides more flexibility in this case because you can better control event triggering.

For example, suppose that you wanted to update the graph only if the new data is different. If the new expression produced the same data within some tolerance, the `set.FofXY` method could not trigger the event and avoid updating the graph. However, the method could still set the property to the new value.

**Listener and Callback for UpdateGraph Event**

The `fcnview` class creates a listener for the UpdateGraph event using the `addlistener` method:

```matlab
obj.HLUpdateGraph = addlistener(obj.FcnObject,'UpdateGraph',...
    @(src,evnt)listenUpdateGraph(obj,src,evnt)); % Add obj to argument list
```

11-57
The `fcnview` object stores a handle to the `event.listener` object in its `HLUpdateGraph` property, which is used to enable/disable the listener by a context menu (see “Enable and Disable Listeners” on page 11-61).

The `fcnview` object (`obj`) is added to the two default arguments (`src`, `evnt`) passed to the listener callback. Keep in mind, the source of the event (`src`) is the `fcneval` object, but the `fcnview` object contains the handle of the surface object that the callback updates.

The `listenUpdateGraph` function is defined as follows:

```matlab
function listenUpdateGraph(obj, src, evnt)
    if ishandle(obj.HSurface) % If surface exists
        obj.updateSurfaceData % Update surface data
    end
end
```

The `updateSurfaceData` function is a class method that updates the surface data when a different mathematical function is assigned to the `fcneval` object. Updating a graphics object data is more efficient than creating a new object using the new data:

```matlab
function updateSurfaceData(obj)
    % Get data from fcneval object and set surface data
    set(obj.HSurface,...
        'XData', obj.FcnObject.Data.X,...
        'YData', obj.FcnObject.Data.Y,...
        'ZData', obj.FcnObject.Data.Matrix);
end
```

**The PostSet Event Listener**

All properties support the predefined `PostSet` event (See “Property-Set and Query Events” on page 11-17 for more information on property events). This example uses the `PostSet` event for the `fcneval Lm` property. This property contains a two-element vector specifying the range over which the mathematical function is evaluated. Just after this property is changed (by a statement like `obj.Lm = [-3 5];`), the `fcnview` objects listening for this event update the graph to reflect the new data.
1. New limits are assigned.
   \[ \text{obj.Lm} = [-3 \ 5]; \]

2. The `SetObservable` attribute of `Properties` is set to `True`, so setting the property automatically triggers a `PostSet` event.
   Note that methods and events did not have to be declared in `myfunceval`.

3. A message is broadcast.

4. A listener awaiting the message executes its callback.

5. The callback function is executed.

**Sequence During the Lm Property Assignment**

The `fcneval` class defines a set function for the `Lm` property. When a value is assigned to this property during object construction or property reassignment, the following sequence occurs:

1. An attempt is made to assign argument value to `Lm` property.
2. The `set.Lm` method executes to check whether the value is in appropriate range — if yes, it makes assignment, if no, it generates an error.
3. If the value of `Lm` is set successfully, MATLAB triggers a `PostSet` event.
4. All listeners execute their callbacks, but the order is nondeterministic.
The PostSet event does not occur until an actual assignment of the property occurs. The property set function provides an opportunity to deal with potential assignment errors before the PostSet event occurs.

**Enable PostSet Property Event**

To create a listener for the PostSet event, you must set the property's SetObservable attribute to `true`:

```matlab
properties (SetObservable = true)
    Lm = [-2*pi 2*pi]; % specifies default value
end
```

MATLAB automatically triggers the event so it is not necessary to call `notify`.

“Specify Property Attributes” on page 8-7 provides a list of all property attributes.

**Listener and Callback for PostSet Event**

The `fcnview` class creates a listener for the PostSet event using the `addlistener` method:

```matlab
obj.HLLm = addlistener(obj.FcnObject,'Lm','PostSet',
    @(src,evnt)listenLm(obj,src,evnt)); % Add obj to argument list
```

The `fcnview` object stores a handle to the `event.listener` object in its HLLm property, which is used to enable/disable the listener by a context menu (see “Enable and Disable Listeners” on page 11-61).

The `fcnview` object (obj) is added to the two default arguments (src, evnt) passed to the listener callback. Keep in mind, the source of the event (src) is the `fcneval` object, but the `fcnview` object contains the handle of the surface object that the callback updates.

The callback sets the axes limits and updates the surface data because changing the limits causes the mathematical function to be evaluated over a different range:

```matlab
function listenLm(obj,src,evnt)
    if ishandle(obj.HAxes) % If there is an axes
        lims(obj); % Update its limits
    end
    if ishandle(obj.HSurface) % If there is a surface
        obj.updateSurfaceData % Update its data
    end
end
```
Enable and Disable Listeners

Each `fcnview` object stores the handle of the listener objects it creates so that the listeners can be enabled or disabled via a context menu after the graphs are created. All listeners are instances of the `event.listener` class, which defines a property called `Enabled`. By default, this property has a value of `true`, which enables the listener. If you set this property to `false`, the listener still exists, but is disabled. This example creates a context menu active on the axes of each graph that provides a way to change the value of the `Enabled` property.

Context Menu Callback

There are two callbacks used by the context menu corresponding to the two items on the menu:

- **Listen** — Sets the `Enabled` property for both the `UpdateGraph` and `PostSet` listeners to `true` and adds a check mark next to the **Listen** menu item.
- **Don't Listen** — Sets the `Enabled` property for both the `UpdateGraph` and `PostSet` listeners to `false` and adds a check mark next to the **Don't Listen** menu item.

Both callbacks include the `fcnview` object as an argument (in addition to the required source and event data arguments) to provide access to the handle of the listener objects.

The `enableLisn` function is called when the user selects **Listen** from the context menu.

```matlab
function enableLisn(obj,src,evnt)
    obj.HLUpdateGraph.Enabled = true; % Enable listener
    obj.HLLm.Enabled = true; % Enable listener
    set(obj.HEnableCm,'Checked','on') % Check Listen
    set(obj.HDisableCm,'Checked','off') % Uncheck Don't Listen
end
```

The `disableLisn` function is called when the user selects **Don't Listen** from the context menu.

```matlab
function disableLisn(obj,src,evnt)
    obj.HLUpdateGraph.Enabled = false; % Disable listener
    obj.HLLm.Enabled = false; % Disable listener
    set(obj.HEnableCm,'Checked','off') % Uncheck Listen
    set(obj.HDisableCm,'Checked','on') % Check Don't Listen
end
```
@fcneval/fcneval.m Class Code

classdef fcneval < handle
    properties
        FofXY
    end

    properties (SetObservable = true)
        Lm = [-2*pi 2*pi]
    end % properties SetObservable = true

    properties (Dependent = true)
        Data
    end

    events
        UpdateGraph
    end

    methods
        function obj = fcneval(fcn_handle,limits)  % Constructor returns object
            if nargin > 0
                obj.FofXY = fcn_handle;                 % Assign property values
                obj.Lm = limits;
            end
        end

        function set.FofXY(obj,func)
            me = fcneval.isSuitable(func);
            if ~isempty(me)
                throw(me)
            end
            obj.FofXY = func;
            notify(obj,'UpdateGraph');
        end

        function set.Lm(obj,lim)
            if ~(lim(1) < lim(2))
                error('Limits must be monotonically increasing')
            else
                obj.Lm = lim;
            end
        end

        function data = get.Data(obj)
            [x,y] = fcneval.grid(obj.Lm);
            matrix = obj.FofXY(x,y);
            data.X = x;
            data.Y = y;
            data.Matrix = real(matrix);
        end
    end % methods
methods (Static = true)
  function [x,y] = grid(lim)
      inc = (lim(2)-lim(1))/20;
      [x,y] = meshgrid(lim(1):inc:lim(2));
  end % grid

function isOk = isSuitable(funcH)
    v = [1 1;1 1];
    try
      funcH(v,v);
    catch %#ok<CTCH>
      me = MException('DocExample:fcneval','... 
                     ['The function ',func2str(funcH),' Is not a suitable F(x,y)']);
      isOk = me;
      return
    end
    if isscalar(funcH(v,v));
      me = MException('DocExample:fcneval','... 
                      ['The function ',func2str(funcH),' Returns a scalar when evaluated']);
      isOk = me;
      return
    end
    isOk = [];
  end
end

@end

@fcnview/fcnview.m Class Code

classdef fcnview < handle
  properties
    FcnObject    % fcneval object
    HAxes        % subplot axes handle
    HLUUpdateGraph % UpdateGraph listener handle
    HLLm         % Lm property PostSet listener handle
    HEnableCm    % "Listen" context menu handle
    HDisableCm   % "Don't Listen" context menu handle
    HSurface     % Surface object handle
  end

  methods
    function obj = fcnview(fcnobj)
      if nargin > 0
        obj.FcnObject = fcnobj;
        obj.createListn;
      end
    end
function createLisn(obj)
    obj.HLUpdateGraph = addlistener(obj.FcnObject,'UpdateGraph',...
        @(src,evnt)listenUpdateGraph(obj,src,evnt));
    obj.HLLm = addlistener(obj.FcnObject,'Lm','PostSet',...
        @(src,evnt)listenLm(obj,src,evnt));
end

function lims(obj)
    lmts = obj.FcnObject.Lm;
    set(obj.HAxes,'XLim',lmts);
    set(obj.HAxes,'Ylim',lmts);
end

function updateSurfaceData(obj)
    data = obj.FcnObject.Data;
    set(obj.HSurface,...
        'XData',data.X,...
        'YData',data.Y,...
        'ZData',data.Matrix);
end

function listenUpdateGraph(obj,~,~)
    if ishandle(obj.HSurface)
        obj.updateSurfaceData
    else
        return
    end
end

function listenLm(obj,~,~)
    if ishandle(obj.HAxes)
        lims(obj);
        if ishandle(obj.HSurface)
            obj.updateSurfaceData
        end
    end
end

function delete(obj)
    if ishandle(obj.HAxes)
        delete(obj.HAxes);
    else
        return
    end
methods (Static)
createViews(a)
end
end

@fcnview/createViews

function createViews(fcnevalobj)
p = pi; deg = 180/p;
hfig = figure('Visible','off',...
'Toolbar','none');
for k=4:-1:1
    fcnviewobj(k) = fcnview(fcnevalobj);
    axh = subplot(2,2,k);
    fcnviewobj(k).HAxes = axh;
    hcm(k) = uicontextmenu;
    set(axh,'Parent',hfig,...
        'FontSize',8,...
        'UIContextMenu',hcm(k))
    fcnviewobj(k).HEnableCm = uimenu(hcm(k),...
        'Label','Listen',...
        'Checked','on',...
        'Callback',@(src,evnt)enableLisn(fcnviewobj(k),src,evnt));
    fcnviewobj(k).HDisableCm = uimenu(hcm(k),...
        'Label','Don''t Listen',...
        'Checked','off',...
        'Callback',@(src,evnt)disableLisn(fcnviewobj(k),src,evnt));
    az = p/k*deg;
    view(axh,az,30)
    title(axh,['View: ',num2str(az),' 30'])
    fcnviewobj(k).lims;
    surfLight(fcnviewobj(k),axh)
end
set(hfig,'Visible','on')
end

function surfLight(obj,axh)
    obj.HSurface = surface(obj.FcnObject.Data.X,...
        obj.FcnObject.Data.Y,...
        obj.FcnObject.Data.Matrix,...
        'FaceColor',[.8 .8 0],'EdgeColor',[.3 .3 .2],...
        'FaceLighting','phong',...
        'FaceAlpha',.3,...
'HitTest','off',...
'Parent',axh);
lims(obj)
camlight left; material shiny; grid off
colormap copper
end

function enableLisn(obj,~,~)
    obj.HLUpdateGraph.Enabled = true;
    obj.HLLm.Enabled = true;
    set(obj.HEnableCm,'Checked','on')
    set(obj.HDisableCm,'Checked','off')
end

function disableLisn(obj,~,~)
    obj.HLUpdateGraph.Enabled = false;
    obj.HLLm.Enabled = false;
    set(obj.HEnableCm,'Checked','off')
    set(obj.HDisableCm,'Checked','on')
end
How to Build on Other Classes

- “Hierarchies of Classes — Concepts” on page 12-2
- “Subclass Syntax” on page 12-7
- “Design Subclass Constructors” on page 12-9
- “Control Sequence of Constructor Calls” on page 12-14
- “Modify Inherited Methods” on page 12-16
- “Modify Inherited Properties” on page 12-20
- “Subclassing Multiple Classes” on page 12-22
- “Specify Allowed Subclasses” on page 12-25
- “Class Members Access” on page 12-28
- “Property Access List” on page 12-36
- “Method Access List” on page 12-37
- “Event Access List” on page 12-38
- “Handle Compatible Classes” on page 12-39
- “How to Define Handle-Compatible Classes” on page 12-41
- “Methods for Handle Compatible Classes” on page 12-46
- “Handle-Compatible Classes and Heterogeneous Arrays” on page 12-48
- “Subclasses of MATLAB Built-In Types” on page 12-50
- “Behavior of Inherited Built-In Methods” on page 12-54
- “Subclasses of Built-In Types Without Properties” on page 12-60
- “Subclasses of Built-In Types with Properties” on page 12-68
- “Use of size and numel with Classes” on page 12-77
- “Class to Represent Hardware” on page 12-83
- “Determine Array Class” on page 12-87
- “Abstract Classes” on page 12-91
- “Define an Interface Superclass” on page 12-96
Hierarchies of Classes — Concepts

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Classification” on page 12-2</td>
</tr>
<tr>
<td>“Develop the Abstraction” on page 12-3</td>
</tr>
<tr>
<td>“Design of Class Hierarchies” on page 12-4</td>
</tr>
<tr>
<td>“Super and Subclass Behavior” on page 12-4</td>
</tr>
<tr>
<td>“Implementation and Interface Inheritance” on page 12-5</td>
</tr>
</tbody>
</table>

Classification

Organizing classes into hierarchies facilitates the reuse of code and the reuse of solutions to design problems that have already been solved. You can think of class hierarchies as sets — supersets (referred to as superclasses or base classes), and subsets (referred to as subclasses or derived classes). For example, the following picture shows how you could represent an employee database with classes.
The root of the hierarchy is the Employees class. It contains data and operations that apply to the set of all employees. Contained in the set of employees are subsets whose members, while still employees, are also members of sets that more specifically define the type of employee. Subclasses like TestEngineer are examples of these subsets.

**Develop the Abstraction**

Classes are representations of real world concepts or things. When designing a class, form an abstraction of what the class represents. Consider an abstraction of an employee and what are the essential aspects of employees for the intended use of the class. Name, address, and department can be what all employees have in common.
When designing classes, your abstraction contains only those elements that are necessary. For example, the employee hair color and shoe size certainly characterize the employee, but are probably not relevant to the design of this employee class. Their sales region is relevant only to some employee so this characteristic belongs in a subclass.

**Design of Class Hierarchies**

As you design a system of classes, put common data and functionality in a superclass, which you then use to derive subclasses. The subclasses inherit the data and functionality of the superclass and define only aspects that are unique to their particular purposes. This approach provides advantages:

- Avoid duplicating code that is common to all classes.
- Add or change subclasses at any time without modifying the superclass or affecting other subclasses.
- If the superclass changes (for example, all employees are assigned a number), then the subclass automatically get these changes.

**Super and Subclass Behavior**

Subclass objects behave like objects of the superclass because they are specializations of the superclass. This fact facilitates the development of related classes that behave similarly, but are implemented differently.

**A Subclass Object “Is A” Superclass Object**

You can usually describe the relationship between an object of a subclass and an object of its superclass with a statement like:

The subclass is a superclass. For example: An Engineer is an Employee.

This relationship implies that objects belonging to a subclass have the same properties, methods, and events as the superclass. Subclass objects also have any new features defined by the subclass. Test this relationship with the isa function.

**Treat Subclass Objects like Superclass Objects**

You can pass a subclass object to a superclass method, but you can access only those properties that the superclass defines. This behavior enables you to modify the subclasses without affecting the superclass.
Two points about super and subclass behavior to keep in mind are:

- Methods defined in the superclass can operate on subclass objects.
- Methods defined in the subclass cannot operate on superclass objects.

Therefore, you can treat an Engineer object like any other Employees object, but an Employee object cannot pass for an Engineer object.

**Limitations to Object Substitution**

MATLAB determines the class of an object based on its most specific class. Therefore, an Engineer object is of class Engineer, while it is also an Employees object, as using the `isa` function reveals.

Generally, MATLAB does not allow you to create arrays containing a mix of superclass and subclass objects because an array can be of only one class. If you attempt to concatenate objects of different classes, MATLAB looks for a converter method defined by the less dominant class.

See “Concatenating Objects of Different Classes” on page 10-18 for more information.

See `matlab.mixin.Heterogeneous` for information on defining heterogeneous class hierarchies.

See “Object Converters” on page 17-12 for information on defining converter methods.

**Implementation and Interface Inheritance**

MATLAB classes support both the inheritance of implemented methods from a superclass and the inheritance of interfaces defined by abstract methods in the superclass.

Implementation inheritance enables code reuse by subclasses. For example, an employee class can have a `submitStatus` method that all employee subclasses can use. Subclasses can extend an inherited method to provide specialized functionality, while reusing the common aspects. See “Modify Inherited Methods” on page 12-16 for more information on this process.

Interface inheritance is useful in these cases:

- You want a group of classes to provide a common interface.
- Subclasses create specialized implementations of methods and properties.
Create an interface using an abstract class as the superclass. This class defines the methods and properties that you must implement in the subclasses, but does not provide an implementation.

The subclasses must provide their own implementation of the abstract members of the superclass. To create an interface, define methods and properties as abstract using their Abstract attribute. See "Abstract Classes" on page 12-91 for more information and an example.

See Also

Related Examples
• “Design Subclass Constructors” on page 12-9
Subclass Syntax

In this section...

“Subclass Definition Syntax” on page 12-7
“Subclass double” on page 12-7

Subclass Definition Syntax

To define a class that is a subclass of another class, add the superclass to the classdef line after a < character:

```matlab
classdef ClassName < SuperClass
```

When inheriting from multiple classes, use the & character to indicate the combination of the superclasses:

```matlab
classdef ClassName < SuperClass1 & SuperClass2
```

See “Class Member Compatibility” on page 12-22 for more information on deriving from multiple superclasses.

Class Attributes

Subclasses do not inherit superclass attributes.

Subclass double

Suppose you want to define a class that derived from double and restricts values to be positive numbers. The PositiveDouble class:

- Supports a default constructor (no input arguments). See “No Input Argument Constructor Requirement” on page 9-25
- Restricts the inputs to positive values using `mustBePositive`.
- Calls the superclass constructor with the input value to create the double numeric value.

```matlab
classdef PositiveDouble < double
    methods
        function obj = PositiveDouble(data)
```
if nargin == 0
    data = 1;
else
    mustBePositive(data)
end
obj = obj@double(data);
end
end
end

Create an object of the PositiveDouble class using a 1-by-5 array of numbers:

a = PositiveDouble(1:5);

You can perform operations on objects of this class like any double.

sum(a)

ans =

15

Objects of the PositiveDouble class must be positive values.

a = PositiveDouble(0:5);

Error using mustBePositive (line 19)
Value must be positive.

Error in PositiveDouble (line 7)
    mustBePositive(data)

See Also

Related Examples

• "Design Subclass Constructors" on page 12-9
• "Subclasses of MATLAB Built-In Types" on page 12-50
Design Subclass Constructors

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Call Superclass Constructor Explicitly” on page 12-9</td>
</tr>
<tr>
<td>“Call Superclass Constructor from Subclass” on page 12-9</td>
</tr>
<tr>
<td>“Subclass Constructor Implementation” on page 12-11</td>
</tr>
<tr>
<td>“Call Only Direct Superclass from Constructor” on page 12-12</td>
</tr>
</tbody>
</table>

**Call Superclass Constructor Explicitly**

Explicitly calling each superclass constructor from a subclass constructor enables you to:

- Pass arguments to superclass constructors
- Control the order in which MATLAB calls the superclass constructors

If you do not explicitly call the superclass constructors from the subclass constructor, MATLAB implicitly calls these constructors with no arguments. The superclass constructors must support the no argument syntax to support implicit calls.

MATLAB does not guarantee any specific calling order when there are multiple superclasses. If the order in which MATLAB calls the superclass constructors is important, call the superclass constructors explicitly from the subclass constructor.

If you do not define a subclass constructor, you can call the default constructor with superclass arguments. For more information, see “Default Constructor” on page 9-23 and “Implicit Call to Inherited Constructor” on page 9-28.

**Call Superclass Constructor from Subclass**

To call the constructor for each superclass within the subclass constructor, use the following syntax:

```
obj@SuperClass1(args,...);
```

```
...
```

```
obj@SuperclassN(args,...);
```
Where \textit{obj} is the output of the subclass constructor, \textit{SuperClass}... is the name of a superclass, and \textit{args} are any arguments required by the respective superclass constructor.

For example, the following segment of a class definition shows that a class called \texttt{Stocks} that is a subclass of a class called \texttt{Assets}.

\begin{verbatim}
classdef Stocks < Assets  
    methods  
        function s = Stocks(asset_args,...)
            if nargin == 0
                % Assign values to asset_args
            end
            % Call asset constructor
            s@Assets(asset_args);
            ...
        end
    end
end
\end{verbatim}

“Subclass Constructors” on page 9-26 provides more information on creating subclass constructor methods.

**Reference Superclasses Contained in Packages**

If a superclass is contained in a package, include the package name. For example, the \texttt{Assets} class is in the \texttt{finance} package:

\begin{verbatim}
classdef Stocks < finance.Assets  
    methods  
        function s = Stocks(asset_args,...)
            if nargin == 0
                ...
            end
            % Call asset constructor
            s@finance.Assets(asset_args);
            ...
        end
    end
end
\end{verbatim}

**Initialize Objects Using Multiple Superclasses**

To derive a class from multiple superclasses, initialize the subclass object with calls to each superclass constructor:
classdef Stocks < finance.Assets & Taxable
definition
function s = Stocks(asset_args,tax_args,...)
  if nargin == 0
    ... % Call asset and member class constructors
  s@finance.Assets(asset_args)
  s@Taxable(tax_args)
  ... end
end
end

Subclass Constructor Implementation

To ensure that your class constructor supports the zero arguments syntax, assign default values to input argument variables before calling the superclass constructor. You cannot conditionalize a subclass call to the superclass constructor. Locate calls to superclass constructors outside any conditional code blocks.

For example, the Stocks class constructor supports the no argument case with the if statement, but calls the superclass constructor outside of the if code block.

classdef Stocks < finance.Assets
  properties
    NumShares
    Symbol
  end
definition
function s = Stocks(description,numshares,symbol)
  if nargin == 0
    description = ''; numshares = 0; symbol = '';
  s@finance.Assets(description);
  s.NumShares = numshares;
  s.Symbol = symbol;
end
end
end
Call Only Direct Superclass from Constructor

Call only direct superclass constructors from a subclass constructor. For example, suppose class B derives from class A and class C derives from class B. The constructor for class C cannot call the constructor for class A to initialize properties. Class B must initialize class A properties.

The following implementations of classes A, B, and C show how to design this relationship in each class.

Class A defines properties x and y, but assigns a value only to x:

```matlab
classdef A
    properties
        x
        y
    end
    methods
        function obj = A(x)
            ...
            obj.x = x;
        end
    end
end
```

Class B inherits properties x and y from class A. The class B constructor calls the class A constructor to initialize x and then assigns a value to y.

```matlab
classdef B < A
    methods
        function obj = B(x,y)
            ...
            obj@A(x);
            obj.y = y;
        end
    end
end
```

Class C accepts values for the properties x and y, and passes these values to the class B constructor, which in turn calls the class A constructor:

```matlab
classdef C < B
    methods
        function obj = C(x,y)
```

12-12
See Also

Related Examples
• “No Input Argument Constructor Requirement” on page 9-25
Control Sequence of Constructor Calls

MATLAB does not guarantee the sequence in which superclass constructors are called when constructing a subclass object. However, you can control this order by calling superclass constructors explicitly from the subclass constructor.

If you explicitly call a superclass constructor from the most specific subclass constructor (ClassC in the following diagram), then MATLAB calls the most specific subclass constructor first. If you do not make an explicit call to a superclass constructor from the subclass constructor, MATLAB makes the implicit call when accessing the object.

Suppose that you have a hierarchy of classes in which ClassC derives from ClassB, which derives from ClassA. The constructor for a subclass can call only direct superclasses. Therefore, each class constructor can call the direct superclass constructor:

```
 classdef ClassB < ClassA
    ...
    function obj = ClassB
        obj@ClassA

 classdef ClassC < ClassB
    ...
    function obj = ClassC
        obj@ClassB
```

In cases of multiple inheritance, the subclass constructor can call each superclass constructor. To ensure that a specific superclass constructor calling sequence is followed, call all direct superclass constructors explicitly from the most specific subclass constructor:
See Also

Related Examples

- “Call Only Direct Superclass from Constructor” on page 12-12
- “Class Constructor Methods” on page 9-21
Modify Inherited Methods

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“When to Modify Superclass Methods” on page 12-16</td>
</tr>
<tr>
<td>“Extend Superclass Methods” on page 12-16</td>
</tr>
<tr>
<td>“Redefine Superclass Methods in Subclass” on page 12-17</td>
</tr>
<tr>
<td>“Override Superclass Methods” on page 12-18</td>
</tr>
</tbody>
</table>

When to Modify Superclass Methods

Class design enable you to pass a subclass object to a superclass method and have the method execute properly. At the same time, subclass methods can apply special processing to the unique aspects of the subclass. Some useful techniques include:

- Extend — Calling a superclass method from within a subclass method
- Redefine — Reimplementing in the subclass, the protected methods that are called from within a public superclass method
- Override — Defining the same named methods in both super and subclass, but using different implementations

Extend Superclass Methods

Subclass methods can call superclass methods of the same name. This technique enables you to extend a superclass method in a subclass without completely redefining the superclass method.

For example, suppose that both superclass and subclass define a method called foo. The method names are the same so the subclass method can call the superclass method. However, the subclass method can also perform other steps before and after the call to the superclass method. It can operate on the specialized parts to the subclass that are not part of the superclass.

For example, this subclass defines a foo method that calls the superclass foo method

```classdef Sub < Super
  methods
    function foo(obj)
      % preprocessing steps```
Redefine Superclass Methods in Subclass

A superclass method can define a process that executes in a series of steps using a protected method for each step (Access attribute set to protected). Then, subclasses can create their own versions of the protected methods that implement the individual steps in the process.

Implement this technique as shown here:

```matlab
classdef Super
    methods
        function foo(obj)
            step1(obj) % Call step1
            step2(obj) % Call step2
            step3(obj) % Call step3
        end
    end
    methods (Access = protected)
        function step1(obj)
            % Superclass version
        end
        function step2(obj)
            % Superclass version
        end
        function step3(obj)
            % Superclass version
        end
    end
end
```

The subclass does not reimplement the `foo` method, it reimplements only the methods that carry out the series of steps (`step1(obj)`, `step2(obj)`, `step3(obj)`). That is, the subclass can specialize the actions taken by each step, but does not control the order of the steps in the process. When you pass a subclass object to the superclass `foo` method, MATLAB calls the subclass step methods because of the dispatching rules.
classdef Sub < Super
...
methods (Access = protected)
    function step1(obj)
        % Subclass version
    end
    function step2(obj)
        % Subclass version
    end
    function step3(obj)
        % Subclass version
    end
...
end
end

**Override Superclass Methods**

You can completely redefine a superclass method in a subclass. In this case, both the superclass and the subclass would define a method with the same name. However, the implementation would be different and the subclass method would not call the superclass method.

```matlab
classdef Super
    methods
        function foo(obj)
            % Superclass implementation
        end
    end
end
classdef Sub < Super
    methods
        function foo(obj)
            % Subclass implementation
        end
    end
end```

12-18 How to Build on Other Classes
See Also

Related Examples
• “Invoking Superclass Methods in Subclass Methods” on page 9-19
Modify Inherited Properties

In this section...

“Superclass Property Modification” on page 12-20
“Private Local Property Takes Precedence in Method” on page 12-20

Superclass Property Modification

There are two separate conditions under which you can redefine superclass properties:

- The value of the superclass property Abstract attribute is true
- The values of the superclass property SetAccess and GetAccess attributes are private

If a superclass defines a property as abstract, the subclass must implement a concrete version of this property or the subclass is also abstract. Superclasses define abstract properties to create a consistent interface among subclasses.

If a superclass defines a property with private access, then only the superclass can access this property. The subclass can implement a different property with the same name.

Private Local Property Takes Precedence in Method

When superclass and subclass define a property with the same name, methods that refer to this property access the property of the class defining the method.

For example, if a subclass property has the same name as a superclass private property, and a method of the superclass references the property name, MATLAB accesses the property defined by the superclass.

Consider the following classes, Super and Sub:

```matlab
classdef Super
    properties (Access = private)
        Prop = 2
    end
methods
    function p = superMethod(obj)
        p = obj.Prop;
end
```
If you create an instance of the `Sub` class and use it to call the superclass method, MATLAB accesses the private property of the superclass:

```matlab
subObj = Sub
subObj = Sub with properties:
    Prop: 1
subObj.superMethod
ans = 2
```

**See Also**

**More About**

- “Property Attributes” on page 8-9
Subclassing Multiple Classes

In this section...

“Specify Multiple Superclasses” on page 12-22
“Class Member Compatibility” on page 12-22
“Multiple Inheritance” on page 12-23

Specify Multiple Superclasses

When inheriting from multiple classes, use the & character to indicate the combination of the superclasses:

classdef ClassName < SuperClass1 & SuperClass2

For more information on class syntax, see “Subclass Syntax” on page 12-7.

Class Member Compatibility

When you create a subclass derived from multiple superclasses, the subclass inherits the properties, methods, and events defined by all specified superclasses. If more than one superclass defines a property, method, or event having the same name, there must be an unambiguous resolution to the multiple definitions. You cannot derive a subclass from any two or more classes that define incompatible class members.

Here are various situations where you can resolve name and definition conflicts.

Property Conflicts

If two or more superclasses define a property with the same name, then at least one of the following must be true:

• All, or all but one of the properties must have their SetAccess and GetAccess attributes set to private
• The properties have the same definition in all superclasses (for example, when all superclasses inherited the property from a common base class)
Method Conflicts

If two or more superclasses define methods with the same name, then at least one of the following must be true:

• The method Access attribute is private so only the defining superclass can access the method.
• The method has the same definition in all subclasses. This situation can occur when all superclasses inherit the method from a common base class and none of the superclasses override the inherited definition.
• The subclass redefines the method to disambiguate the multiple definitions across all superclasses. Therefore, the superclass methods must not have their Sealed attribute set to true.
• Only one superclass defines the method as Sealed, in which case, the subclass adopts the sealed method definition.
• The superclasses define the methods as Abstract and rely on the subclass to define the method.

Event Conflicts

If two or more superclasses define events with the same name, then at least one of the following must be true:

• The event ListenAccess and NotifyAccess attributes must be private.
• The event has the same definition in all superclasses (for example, when all superclasses inherited the event from a common base class)

Multiple Inheritance

Resolving the potential conflicts involved when defining a subclass from multiple classes often reduces the value of this approach. For example, problems can arise when you enhance superclasses in future versions and introduce new conflicts.

Reduce potential problems by implementing only one unrestricted superclass. In all other superclasses, all methods are

• Abstract
• Defined by a subclass
• Inherited from the unrestricted superclass

When using multiple inheritance, ensure that all superclasses remain free of conflicts in definition.

See Also

Related Examples
• “Design Subclass Constructors” on page 12-9
• “Handle Compatible Classes” on page 12-39
Specify Allowed Subclasses

In this section...

“Basic Knowledge” on page 12-25
“Why Control Allowed Subclasses” on page 12-25
“Specify Allowed Subclasses” on page 12-25
“Define Sealed Hierarchy of Classes” on page 12-27

Basic Knowledge

The material presented in this section builds on an understanding of the following information:

• “Class Metadata” on page 16-2
• “Attribute Specification” on page 5-22

Why Control Allowed Subclasses

A class definition can specify a list of classes that it allows as subclasses. Classes not in the list cannot be defined as subclass of the class. To specify the allowed subclasses, use the AllowedSubclasses class attribute.

The AllowedSubclasses attribute provides a design point between Sealed classes, which do not allow subclassing, and the default behavior, which does not restrict subclassing.

By controlling the allowed subclasses, you can create a sealed hierarchy of classes. That is, a system of classes that enables a specific set of classes to derive from specific base classes, but that does not allow unrestricted subclassing.

See “Define Sealed Hierarchy of Classes” on page 12-27 for more about this technique.

Specify Allowed Subclasses

Specify a list of one or more allowed subclasses in the classdef statement by assigning meta.class objects to the AllowedSubclasses attribute. Create the meta.class object referencing a specific class using the ? operator and the class name:
classdef (AllowedSubclasses = ?ClassName) MySuperClass
  ...
end

Use a cell array of meta.class objects to define more than one allowed subclass:

```matlab
classdef (AllowedSubclasses = {?ClassName1,?ClassName2,...?ClassNameN}) MySuperClass
  ...
end
```

Always use the fully qualified class name when referencing the class name:

```matlab
classdef (AllowedSubclasses = {?Package.SubPackage.ClassName1}) MySuperClass
  ...
end
```

Assigning an empty cell array to the AllowedSubclasses attribute is effectively the same as defining a Sealed class.

```matlab
classdef (AllowedSubclasses = {[]}) MySuperClass
  ...
end
```

**Note** Use only the ? operator and the class name to generate meta.class objects. Values assigned to the AllowedSubclasses attribute cannot contain any other MATLAB expressions, including functions that return either meta.class objects or cell arrays of meta.class objects.

**Result of Declaring Allowed Subclasses**

Including a class in the list of AllowedSubclasses does not define that class as a subclass or require you to define the class as a subclass. It just allows the referenced class to be defined as a subclass.

Declaring a class as an allowed subclass does not affect whether this class can itself be subclassed.

A class definition can contain assignments to the AllowedSubclasses attribute that reference classes that are not currently defined or available on the MATLAB path. However, any referenced subclass that MATLAB cannot find when loading the class is effectively removed from the list without causing an error or warning.

**Note** If MATLAB does not find any of the classes in the allowed classes list, the class is effectively Sealed. A sealed class is equivalent to AllowedSubclasses = {[]}.
Use the `meta.class` property `RestrictsSubclassing` to determine if a class is **Sealed** or specifies `AllowedSubclasses`.

**Define Sealed Hierarchy of Classes**

The `AllowedSubclasses` attribute enables you to define a sealed class hierarchy by sealing the allowed subclasses:

```matlab
classdef (AllowedSubclasses = {?SubClass1,?SubClass2}) SuperClass
...
end
```

Define the allowed subclasses as **Sealed**:

```matlab
classdef (Sealed) SubClass1
...
end
```

```matlab
classdef (Sealed) SubClass2
...
end
```

Sealed class hierarchies enable you to use the level of abstraction that your design requires while maintaining a closed system of classes.

**See Also**

**Related Examples**

- “Handle Compatible Classes” on page 12-39
Class Members Access

In this section...

“Basic Knowledge” on page 12-28
“Applications for Access Control Lists” on page 12-29
“Specify Access to Class Members” on page 12-29
“Properties with Access Lists” on page 12-31
“Methods with Access Lists” on page 12-31
“Abstract Methods with Access Lists” on page 12-35

Basic Knowledge

The material presented in this section builds on an understanding of the following information:

Related Topics

• “Class Metadata” on page 16-2
• “Attribute Specification” on page 5-22

Terminology and Concepts

• Class members — Properties, methods, and events defined by a class
• Defining class — The class defining the class member for which access is being specified
• Get access — Permission to read the value of a property, controlled by the property GetAccess attribute
• Set access — Permission to assign a value to a property; controlled by the property SetAccess attribute
• Method access - Determines what other methods and functions can call the class method; controlled by the method Access attribute
• Listen access — Permission to define listeners; controlled by the event ListenAccess attribute
• Notify access — Permission to trigger events, controlled by the event NotifyAccess attribute
Possible Values for Access to Class Members

The following class member attributes can contain a list of classes:

- **Properties** — Access, GetAccess, and SetAccess. For a list of all property attributes, see “Property Attributes” on page 8-9.
- **Methods** — Access. For a list of all method attributes, see “Method Attributes” on page 9-5.
- **Events** — ListenAccess and NotifyAccess. For a list of all event attributes, see “Event Attributes” on page 11-19.

These attributes accept the following possible values:

- **public** — Unrestricted access
- **protected** — Access by defining class and its subclasses
- **private** — Access by defining class only
- **Access list** — A list of one or more classes. Only the defining class and the classes in the list have access to the class members to which the attribute applies. If you specify a list of classes, MATLAB does not allow access by any other class (that is, access is private, except for the listed classes).

Applications for Access Control Lists

Access control lists enable you to control access to specific class properties, methods, and events. Access control lists specify a list of classes to which you grant access to these class members.

This technique provides greater flexibility and control in the design of a system of classes. For example, use access control lists to define separate classes, but not allow access to class members from outside the class system.

Specify Access to Class Members

Specify the classes that are allowed to access a particular class member in the member access attribute statement. For example:

```matlab
methods (Access = {?ClassName1,?ClassName2,...})
```
Use the class `meta.class` object to refer to classes in the access list. To specify more than one class, use a cell array of `meta.class` objects. Use the package name when referring to classes that are in packages.

**Note** Specify the `meta.class` objects explicitly (created with the `?` operator), not as values returned by functions or other MATLAB expressions.

### How MATLAB Interprets Attribute Values

- Granting access to a list of classes restricts access to only:
  - The defining class
  - The classes in the list
  - Subclasses of the classes in the list
- Including the defining class in the access list gives all subclasses of the defining class access.
- MATLAB resolves references to classes in the access list only when the class is loaded. If MATLAB cannot find a class that is included in the access list, that class is effectively removed from access.
- MATLAB replaces unresolved `meta.class` entries in the list with empty `meta.class` objects.
- An empty access list (that is, an empty cell array) is equivalent to `private` access.

### Specify Metaclass Objects

Generate the `meta.class` objects using only the `?` operator and the class name. Values assigned to the attributes cannot contain any other MATLAB expressions, including functions that return allowed attribute values:

- `meta.class` objects
- Cell arrays of `meta.class` objects
- The values `public`, `protected`, or `private`

Specify these values explicitly, as shown in the example code in this section.
Properties with Access Lists

These sample classes show the behavior of a property that grants read access (GetAccess) to a class. The GrantAccess class gives GetAccess to the NeedAccess class for the Prop1 property:

```matlab
classdef GrantAccess
    properties (GetAccess = ?NeedAccess)
        Prop1 = 7
    end
end
```

The NeedAccess class defines a method that uses the value of the GrantAccess Prop1 value. The dispObj method is defined as a Static method, however, it could be an ordinary method.

```matlab
classdef NeedAccess
    methods (Static)
        function dispObj(GrantAccessObj)
            disp(['Prop1 is: ',num2str(GrantAccessObj.Prop1)])
        end
    end
end
```

Get access to Prop1 is private so MATLAB returns an error if you attempt to access the property from outside the class definition. For example, from the command line:

```matlab
a = GrantAccess;
a.Prop1
```

Getting the 'Prop1' property of the 'GrantAccess' class is not allowed.

However, MATLAB allows access to Prop1 by the NeedAccess class:

```matlab
NeedAccess.dispObj(a)
```

Prop1 is: 7

Methods with Access Lists

Classes granted access to a method can:

- Call the method using an instance of the defining class.
• Define their own method with the same name (if not a subclass).
• Override the method in a subclass only if the superclass defining the method includes itself or the subclass in the access list.

These sample classes show the behavior of methods called from methods of other classes that are in the access list. The class AcListSuper gives the AcListNonSub class access to its m1 method:

```matlab
classdef AcListSuper
    methods (Access = {?AcListNonSub})
        function obj = m1(obj)
            disp('Method m1 called')
        end
    end
end
```

Because AcListNonSub is in the access list of m1, its methods can call m1 using an instance of AcListSuper:

```matlab
classdef AcListNonSub
    methods
        function obj = nonSub1(obj,AcListSuper_Obj)
            % Call m1 on AcListSuper class
            AcListSuper_Obj.m1;
        end
        function obj = m1(obj)
            % Define a method named m1
            disp(['Method m1 defined by ',class(obj)])
        end
    end
end
```

Create objects of both classes:

```matlab
a = AcListSuper;
b = AcListNonSub;
```

Call the AcListSuper m1 method using an AcListNonSub method:

```matlab
b.nonSub1(a);
```

Method m1 called

Call the AcListNonSub m1 method:
b.m1;

Method m1 defined by AcListNonSub

**Subclasses Without Access**

Including the defining class in the access list for a method grants access to all subclasses derived from that class. When you derive from a class that has a method with an access list and that list does not include the defining class:

- Subclass methods cannot call the superclass method.
- Subclass methods can call the superclass method indirectly using an instance of a class that is in the access list.
- Subclasses cannot override the superclass method.
- Methods of classes that are in the superclass method access list, but that are not subclasses, can call the superclass method.

For example, AcListSub is a subclass of AcListSuper. The AcListSuper class defines an access list for method m1. However, this list does not include AcListSuper, so subclasses of AcListSuper do not have access to method m1:

```matlab
classdef AcListSub < AcListSuper
    methods
        function obj = sub1(obj,AcListSuper_Obj)
            % Access m1 via superclass object (**NOT ALLOWED**)
            AcListSuper_Obj.m1;
        end
        function obj = sub2(obj,AcListNonSub_Obj,AcListSuper_obj)
            % Access m1 via object that is in access list (is allowed)
            AcListNonSub_Obj.nonSub1(AcListSuper_Obj);
        end
    end
end
```

**No Direct Call to Superclass Method**

Attempting to call the superclass m1 method from the sub1 method results in an error because subclasses are not in the access list for m1:

```matlab
a = AcListSuper;
c = AcListSub;
c.sub1(a);
```

Cannot access method 'm1' in class 'AcListSuper'.

12-33
Error in AcListSub/sub1 (line 4)
    AcListSuper_Obj.m1;

**Indirect Call to Superclass Method**

You can call a superclass method from a subclass that does not have access to that method using an object of a class that is in the superclass method access list.

The AcListSub `sub2` method calls a method of a class (AcListNonSub) that is on the access list for `m1`. This method, `nonSub1`, does have access to the superclass `m1` method:

```matlab
a = AcListSuper;
b = AcListNonSub;
c = AcListSub;
c.sub2(b,a);
```

Method `m1` called

**No Redefining Superclass Method**

When subclasses are not included in the access list for a method, those subclasses cannot define a method with the same name. This behavior is not the same as cases in which the method `Access` is explicitly declared as `private`.

For example, adding the following method to the AcListSub class definition produces an error when you attempt to instantiate the class:

```matlab
methods (Access = {?AcListNonSub})
    function obj = m1(obj)
        disp('AcListSub m1 method')
    end
end

Class 'AcListSub' is not allowed to override the method 'm1' because neither it nor its superclasses have been granted access to the method by class 'AcListSuper'.

**Call Superclass from Listed Class Via Subclass**

The AcListNonSub class is in the `m1` method access list. This class can define a method that calls the `m1` method using an object of the AcListSub class. While AcListSub is not in the access list for method `m1`, it is a subclass of AcListSuper.

For example, add the following method to the AcListNonSub class:
methods
    function obj = nonSub2(obj,AcListSub_Obj)
        disp('Call m1 via subclass object:')
        AcListSub_Obj.m1;
    end
end

Calling the nonSub2 method results in execution of the superclass m1 method:

b = AcListNonSub;
c = AcListSub;
b.nonSub2(c);

Call m1 via subclass object:
Method m1 called

This behavior is consistent with the behavior of any subclass object, which can substitute for an object of its superclass.

Abstract Methods with Access Lists

A class containing a method declared as Abstract is an abstract class. It is the responsibility of subclasses to implement the abstract method using the function signature declared in the class definition.

When an abstract method has an access list, only the classes in the access list can implement the method. A subclass that is not in the access list cannot implement the abstract method so that subclass is itself abstract.

See Also

Related Examples

- “Property Access List” on page 12-36
- “Method Access List” on page 12-37
- “Event Access List” on page 12-38
Property Access List

This class declares access lists for the property GetAccess and Access attributes:

```matlab
classdef PropertyAccess
    properties (GetAccess = {?ClassA, ?ClassB}, SetAccess = private)
        Prop1
    end
    properties (Access = ?ClassC)
        Prop2
    end
end
```

The class PropertyAccess specifies the following property access:

- Gives the classes ClassA and ClassB get access to the Prop1 property.
- Gives all subclasses of ClassA and ClassB get access to the Prop1 property.
- Does not provide get access to Prop1 from subclasses of PropertyAccess.
- Defines private set access for the Prop1 property.
- Gives set and get access to Prop2 for ClassC and its subclasses.

See Also

Related Examples
- “Properties with Access Lists” on page 12-31
Method Access List

This class declares an access list for the method Access attribute:

```matlab
classdef MethodAccess
        function listMethod(obj)
            ...
            end
        end
    end
end
```

The `MethodAccess` class specifies the following method access:

- Access to `listMethod` from an instance of `MethodAccess` by methods of the classes `ClassA` and `ClassB`.
- Access to `listMethod` from an instance of `MethodAccess` by methods of subclasses of `MethodAccess`, because of the inclusion of `MethodAccess` in the access list.
- Subclasses of `ClassA` and `ClassB` are allowed to define a method named `listMethod`, and `MethodAccess` is allowed to redefine `listMethod`. However, if `MethodAccess` was not in the access list, its subclasses could not redefine `listMethod`.

See Also

Related Examples

- “Methods with Access Lists” on page 12-31
Event Access List

This class declares an access list for the event ListenAccess attribute:

```matlab
classdef EventAccess
    events (NotifyAccess = private, ListenAccess = {?ClassA, ?ClassB})
    Event1
    Event2
    end
end
```

The class EventAccess specifies the following event access:

- Limits notify access for Event1 and Event2 to EventAccess; only methods of the EventAccess can trigger these events.
- Gives listen access for Event1 and Event2 to methods of ClassA and ClassB. Methods of EventAccess, ClassA, and ClassB can define listeners for these events. Subclasses of MyClass cannot define listeners for these events.

See Also

Related Examples

- “Events and Listeners Syntax” on page 11-22
Handle Compatible Classes

In this section...

“Basic Knowledge” on page 12-39
“When to Use Handle Compatible Classes” on page 12-39
“Handle Compatibility Rules” on page 12-40
“Identify Handle Objects” on page 12-40

Basic Knowledge

The material presented in this section builds on knowledge of the following information.

• “Design Subclass Constructors” on page 12-9
• “Subclassing Multiple Classes” on page 12-22
• “Comparison of Handle and Value Classes” on page 7-2

Key Concepts

Handle-compatible class — a class that you can include with handle classes in a class hierarchy, even if the class is not a handle class.

• All handle classes are handle-compatible.
• All superclasses of handle-compatible classes must also be handle compatible.

HandleCompatible — the class attribute that defines nonhandle classes as handle compatible.

When to Use Handle Compatible Classes

Typically, when deriving a MATLAB class from other classes, all the superclasses are handle classes, or else none of them are handle classes. However, there are situations in which a class provides some utility that is used by both handle and non-handle subclasses. Because it is not legal to combine handle and non-handle classes, the author of the utility class must implement two distinct versions of the utility.

The solution is to use handle-compatible classes. Handle compatible classes are a type of class that you can use with handle classes when forming sets of superclasses. Designate a
nonhandle compatible class as handle-compatible by using the HandleCompatible class attribute.

classdef (HandleCompatible) MyClass
    ...
end

Handle Compatibility Rules

Handle-compatible classes (that is, classes whose HandleCompatible attribute is set to true) follow these rules:

- All superclasses of a handle-compatible class must also be handle compatible
- If a class explicitly sets its HandleCompatibility attribute to false, then none of the class superclasses can be handle classes.
- If a class does not explicitly set its HandleCompatible attribute and, if any superclass is a handle, then all superclasses must be handle compatible.
- The HandleCompatible attribute is not inherited.

A class that does not explicitly set its HandleCompatible attribute to true is:

- A handle class if any of its superclasses are handle classes
- A value class if none of the superclasses are handle classes

Identify Handle Objects

To determine if an object is a handle object, use the isa function:

isa(obj,'handle')

See Also

Related Examples

- “How to Define Handle-Compatible Classes” on page 12-41
How to Define Handle-Compatible Classes

In this section...

“What Is Handle Compatibility?” on page 12-41
“Subclassing Handle-Compatible Classes” on page 12-43

What Is Handle Compatibility?

A class is handle compatible if:

- It is a handle class
- Its HandleCompatible attribute is set to true

The HandleCompatible class attribute identifies classes that you can combine with handle classes when specifying a set of superclasses.

Handle compatibility provides greater flexibility when defining abstract superclasses. For example, when using superclasses that support both handle and value subclasses, handle compatibility removes the need to define both a handle version and a nonhandle version of a class.

A Handle Compatible Class

The Utility class is useful to both handle and value subclasses. In this example, the Utility class defines a method to reset property values to the default values defined in the respective class definition:

classdef (HandleCompatible) Utility
    methods
        function obj = resetDefaults(obj)
            mc = metaclass(obj);
            mp = mc.PropertyList;
            for k=1:length(mp)
                if mp(k).HasDefault && ~strcmp(mp(k).SetAccess,'private')
                    obj.(mp(k).Name) = mp(k).DefaultValue;
                end
            end
        end
    end
end
end
The Utility class is handle compatible. Therefore, you can use it in the derivation of classes that are either handle classes or value classes. See “Class Introspection and Metadata” for information on using meta-data classes.

**Return Modified Objects**

The resetDefaults method defined by the Utility class returns the object it modifies. When you call resetDefaults with a value object, the method must return the modified object. It is important to implement methods that work with both handle and value objects in a handle compatible superclass. See “Object Modification” on page 5-62 for more information on modifying handle and value objects.

Consider the behavior of a value class that subclasses the Utility class. The PropertyDefaults class defines three properties, all of which have default values:

```matlab
classdef PropertyDefaults < Utility
    properties
        p1 = datestr(rem(now,1)) % Current time
        p2 = 'red' % Character vector
        p3 = p1/2 % Result of division operation
    end
end
```

Create a PropertyDefaults object. MATLAB evaluates the expressions assigned as default property values when the class is first loaded. MATLAB uses these same default values whenever you create an instance of this class in the current MATLAB session.

```matlab
pd = PropertyDefaults
```

```
PropertyDefaults with properties:
  p1:  ' 4:42 PM'
  p2: 'red'
  p3: 1.5708
```

Assign new values that are different from the default values:

```matlab
pd.p1 = datestr(rem(now,1));
pd.p2 = 'green';
pd.p3 = p1/4;
```

All pd object property values now contain values that are different from the default values originally defined by the class:
pd

pd =

PropertyDefaults with properties:
:
  p1: ' 4:45 PM'
  p2: 'green'
  p3: 0.7854

Call the resetDefaults method, which is inherited from the Utility class. Because the PropertyDefaults class is not a handle class, return the modified object.

pd = pd.resetDefaults

pd =

PropertyDefaults with properties:
:
  p1: ' 4:54 PM'
  p2: 'red'
  p3: 1.5708

If the PropertyDefaults class was a handle class, then you would not need to save the object returned by the resetDefaults method. To design a handle compatible class like Utility, ensure that all methods work with both kinds of classes.

**Subclassing Handle-Compatible Classes**

According to the rules described in “Handle Compatibility Rules” on page 12-40, when you combine a handle superclass with a handle-compatible superclass, the result is a handle subclass, which is handle compatible.

However, subclassing a handle-compatible class does not necessarily result in the subclass being handle compatible. Consider the following two cases, which demonstrate two possible results.

**Combine Nonhandle Utility Class with Handle Classes**

Suppose that you define a class that subclasses a handle class, and the handle compatible Utility class discussed in “A Handle Compatible Class” on page 12-41. The HPropertyDefaults class has these characteristics:
• It is a handle class (it derives from handle).
• All its superclasses are handle compatible (handle classes are handle compatible by
definition).

classdef HPropertyDefaults < handle & Utility
    properties
        GraphPrim = line
        Width = 1.5
        Color = 'black'
    end
end

The HPropertyDefaults class is handle compatible:

hpd = HPropertyDefaults;
mc = metaclass(hpd);
mc.HandleCompatible

ans =

   1

Nonhandle Subclasses of a Handle-Compatible Class

If you subclass both a value class that is not handle compatible and a handle compatible
class, the subclass is a nonhandle compatible value class. The ValueSub class:

• Is a value class (it does not derive from handle.)
• One of its superclasses is handle compatible (the Utility class).

classdef ValueSub < MException & Utility
    methods
        function obj = ValueSub(str1,str2)
            obj = obj@MException(str1,str2);
        end
    end
end

The ValueSub class is a nonhandle-compatible value class because the MException
class does not define the HandleCompatible attribute as true:

hv = ValueSub('MATLAB:narginchk:notEnoughInputs','Not enough input arguments.');
mc = metaclass(hv);
mc.HandleCompatible
ans = 0

See Also

Related Examples
• “Methods for Handle Compatible Classes” on page 12-46
Methods for Handle Compatible Classes

In this section...

“Methods for Handle and Value Objects” on page 12-46
“Modify Value Objects in Methods” on page 12-46

Methods for Handle and Value Objects

Objects passed to methods of handle compatible classes can be either handle or value objects. There are two different behaviors to consider when implementing methods for a class that operate on both handles and values:

• If an input object is a handle object and the method alters the handle object, these changes are visible to all workspaces that contain the same handle.
• If an input object is a value object, then changes to the object made inside the method affect only the value inside the method workspace.

Handle compatible methods generally do not alter input objects because the effects of such changes are not the same for handle and nonhandle objects.

See “Object Modification” on page 5-62 for information about modifying handle and value objects.

Modify Value Objects in Methods

If a method operates on both handle and value objects, the method must return the modified object. For example, the `setTime` method returns the object it modifies:

```matlab
classdef (HandleCompatible) Util
    % Utility class that adds a time stamp
    properties
        TimeStamp
    end
    methods
        function obj = setTime(obj)
            obj.TimeStamp = now;
        end
    end
end
```
See Also

Related Examples
• “Handle-Compatible Classes and Heterogeneous Arrays” on page 12-48
Handle-Compatible Classes and Heterogeneous Arrays

Heterogeneous Arrays

A heterogeneous array contains objects of different classes. Members of a heterogeneous array have a common superclass, but can belong to different subclasses. See the `matlab.mixin.Heterogeneous` class for more information on heterogeneous arrays. The `matlab.mixin.Heterogeneous` class is a handle-compatible class.

Methods Must Be Sealed

You can invoke only those methods that are sealed by the common superclass on heterogeneous arrays (Sealed attribute set to true). Sealed methods prevent subclasses from overriding those methods and guarantee that methods called on heterogeneous arrays have the same definition for the entire array.

Subclasses cannot override sealed methods. In situations requiring subclasses to specialize methods defined by a utility class, you can employ the design pattern referred to as the template method.

Template Technique

Suppose that you implement a handle compatible class that works with heterogeneous arrays. This approach enables you to seal public methods, while providing a way for each subclass to specialize how the method works on each subclass instance. In the handle compatible class:

- Define a sealed method that accepts a heterogeneous array as input.
- Define a protected, abstract method that each subclass must implement.
- Within the sealed method, call the overridden method for each array element.
Each subclass in the heterogeneous hierarchy implements a concrete version of the abstract method. The concrete method provides specialized behavior required by the particular subclass.

The `Printable` class shows how to implement a template method approach:

```matlab
classdef (HandleCompatible) Printable
    methods(Sealed)
    function print(aryIn)
        n = numel(aryIn);
        for k=1:n
            printElement(aryIn(k));
        end
    end
end
methods(Access=protected, Abstract)
end
end
```

**See Also**

**Related Examples**

- "Handle Compatible Classes" on page 12-39
Subclasses of MATLAB Built-In Types

**In this section...**

- “MATLAB Built-In Types” on page 12-50
- “Built-In Types You Can Subclass” on page 12-50
- “Why Subclass Built-In Types” on page 12-51
- “Which Functions Work with Subclasses of Built-In Types” on page 12-51
- “Behavior of Built-In Functions with Subclass Objects” on page 12-51
- “Built-In Subclasses That Define Properties” on page 12-52

**MATLAB Built-In Types**

Built-in types represent fundamental kinds of data such as numeric arrays, logical arrays, and character arrays. Other built-in types like cell arrays and structures contain data belonging to any class.

Built-in types define methods that perform operations on objects of these classes. For example, you can perform operations on numeric arrays such as, sorting, arithmetic, and logical operations.

See “Fundamental MATLAB Classes” for more information on MATLAB built-in classes.

**Note** It is an error to define a class that has the same name as a built-in class.

**Built-In Types You Can Subclass**

You can subclass MATLAB numeric classes and the logical class. For a list of numeric types, see “Numeric Types”.

You cannot subclass any class that has its Sealed attribute set to true. To determine if the class is Sealed, query the class metadata:

```matlab
c = ?ClassName;
c.Sealed
```

A value of 0 indicates that the class is not Sealed and can be subclasses.
**Why Subclass Built-In Types**

Subclass a built-in class to extend the operations that you can perform on a particular class of data. For example, when you want to:

- To perform unique operations on class data.
- Be able to use methods of the built-in class and other built-in functions directly with objects of the subclass. For example, you do not need to reimplement all the mathematical operators if you derived from a class such as `double` that defines these operators.

**Which Functions Work with Subclasses of Built-In Types**

Consider a class that defines enumerations. It can derive from an integer class and inherit methods that enable you to compare and sort values. For example, integer classes like `int32` support all the relational methods (`eq`, `ge`, `gt`, `le`, `lt`, `ne`).

To see a list of functions that the subclass has inherited as methods, use the `methods` function:

```
methods('SubclassName')
```

Generally, you can use an object of the subclass with any:

- Inherited methods
- Functions that normally accept input arguments of the same class as the superclass.

**Behavior of Built-In Functions with Subclass Objects**

When you define a subclass of a built-in class, the subclass inherits all the methods defined by that built-in class. MATLAB also provides additional methods to subclasses of built-in classes that override several built-in functions.

Built-in functions and methods that work on built-in classes can behave differently when called with subclasses of built-in classes. Their behavior depends on which function you are using and whether your subclass defines properties.
Behavior Categories

When you call an inherited method on a subclass of a built-in class, the result depends on the nature of the operation performed by the method. The behaviors of these methods fit into several categories.

• Operations on data values return objects of the superclass. For example, if you subclass `double` and perform addition on two subclass objects, MATLAB adds the numeric values and returns a value of class `double`.

• Operations on the orientation or structure of the data return objects of the subclass. Methods that perform these kinds of operations include, `reshape`, `permute`, `transpose`, and so on.

• Converting a subclass object to a built-in class returns an object of the specified class. Functions such as `uint32`, `double`, `char` work with subclass objects the same as they work with built-in objects.

• Comparing objects or testing for inclusion in a specific set returns logical or built-in objects, depending on the function. Functions such as `isequal`, `ischar`, `isobject` work with subclass objects the same as they work with superclass objects.

• Indexing expressions return objects of the subclass. If the subclass defines properties, then default indexing no longer works. The subclass must define its own indexing methods.

• Concatenation returns an object of the subclass. If the subclass defines properties, then default concatenation no longer works and the subclass must define its own concatenation methods.

To list the built-in functions that work with a subclass of a built-in class, use the `methods` function.

Built-In Subclasses That Define Properties

When a subclass of a built-in class defines properties, MATLAB no longer supports indexing and concatenation operations. MATLAB cannot use the built-in functions normally called for these operations because subclass properties can contain any data.

The subclass must define what indexing and concatenation mean for a class with properties. If your subclass needs indexing and concatenation functionality, then the subclass must implement the appropriate methods.
Methods for Indexing

To support indexing operations, the subclass must implement these methods:

- `subsasgn` — Implement dot notation and indexed assignments
- `subsref` — Implement dot notation and indexed references
- `subsindex` — Implement object as index value

Methods for Concatenation

To support concatenation, the subclass must implement the following methods:

- `horzcat` — Implement horizontal concatenation of objects
- `vertcat` — Implement vertical concatenation of objects
- `cat` — Implement concatenation of object arrays along specified dimension

See Also

Related Examples

- “Class to Represent Hardware” on page 12-83
- “Subclasses of Built-In Types with Properties” on page 12-68
- “Subclasses of Built-In Types Without Properties” on page 12-60
Behavior of Inherited Built-In Methods

In this section...

“Subclass double” on page 12-54
“Built-In Data Value Methods” on page 12-55
“Built-In Data Organization Methods” on page 12-56
“Built-In Indexing Methods” on page 12-57
“Built-In Concatenation Methods” on page 12-57

Subclass double

Most built-in functions used with built-in classes are actually methods of the built-in class. For example, the `double` and `single` classes define several methods to perform arithmetic operations, indexing, matrix operation, and so on. All these built-in class methods work with subclasses of the built-in class.

Subclassing `double` enables your class to use features without implementing the methods that a MATLAB built-in class defines.

The `DocSimpleDouble` class subclasses the built-in `double` class.

```matlab
classdef DocSimpleDouble < double
    methods
        function obj = DocSimpleDouble(data)
            if nargin == 0
                data = 0;
            end
            obj = obj@double(data);
        end
    end
end
```

Create an instance of the class `DocSimpleDouble`.

```matlab
sc = DocSimpleDouble(1:10)
```

```
sc =
    1x10 DocSimpleDouble:
double data:
   1    2    3    4    5    6    7    8    9   10
```
Call a method inherited from class `double` that operates on the data, such as `sum`. `sum` returns a `double` and, therefore, uses the `display` method of class `double`:

```matlab
sum(sc)
```

```matlab
ans =
55
```

Index `sc` like an array of doubles. The returned value is the class of the subclass:

```matlab
a = sc(2:4)
```

```matlab
a =
    1x3 DocSimpleDouble:
    double data:
        2 3 4
```

Indexed assignment works the same as the built-in class. The returned value is the class of the subclass:

```matlab
sc(1:5) = 5:-1:1
```

```matlab
sc =
    1x10 DocSimpleDouble:
    double data:
        5 4 3 2 1 6 7 8 9 10
```

Calling a method that modifies the order of the data elements operates on the data, but returns an object of the subclass:

```matlab
sc = DocSimpleDouble(1:10);
sc(1:5) = 5:-1:1;
a = sort(sc)
```

```matlab
a =
    1x10 DocSimpleDouble:
    double data:
        1 2 3 4 5 6 7 8 9 10
```

**Built-In Data Value Methods**

When you call a built-in data value method on a subclass object, MATLAB uses the superclass part of the subclass object as inputs to the method. The value returned is same class as the built-in class. For example:
sc = DocSimpleDouble(1:10);
a = sin(sc);
class(a)

ans =
double

Built-In Data Organization Methods

This group of built-in methods reorders or reshapes the input argument array. These methods operate on the superclass part of the subclass object, but return an object of the same type as the subclass.

sc = DocSimpleDouble(randi(9,1,10))

sc = DocSimpleDouble(randi(9,1,10))

sc =

1x10 DocSimpleDouble:

double data:
6 1 8 9 7 7 7 4 6 2

b = sort(sc)

b =

1x10 DocSimpleDouble:

double data:
1 2 4 6 6 7 7 7 8 9

Methods in this group include:

• reshape
• permute
• sort
• sort
• transpose
• ctranspose
Built-In Indexing Methods

Built-in classes use specially implemented versions of the `subsref`, `subsasgn`, and `subsindex` methods to implement indexing. When you index a subclass object, only the built-in data is referenced (not the properties defined by your subclass).

For example, indexing element 2 in the `DocSimpleDouble` subclass object returns the second element in the vector:

```matlab
sc = DocSimpleDouble(1:10);
a = sc(2)
```

```matlab
a =
    DocSimpleDouble
      double data:
        2
```

The value returned from an indexing operation is an object of the subclass. You cannot make indexed references if your subclass defines properties, unless your subclass overrides the default `subsref` method.

Assigning a new value to the second element in the `DocSimpleDouble` object operates only on the superclass data:

```matlab
sc(2) = 12
```

```matlab
sc =
    1x10 DocSimpleDouble:
      double data:
        1    12    3    4    5    6    7    8    9   10
```

The `subsref` method also implements dot notation for methods.

Built-In Concatenation Methods

Built-in classes use the functions `horzcat`, `vertcat`, and `cat` to implement concatenation. When you use these functions with subclass objects of the same type, MATLAB concatenates the superclass data to form a new object. For example, you can concatenate objects of the `DocSimpleDouble` class:

```matlab
sc1 = DocSimpleDouble(1:10);
sc2 = DocSimpleDouble(11:20);
[sc1,sc2]
```
Concatenate two objects along a third dimension:

c = cat(3,sc1,sc2)

c =

1x10x2 DocSimpleDouble:

double data:

(:, :, 1) =

1 2 3 4 5 6 7 8 9 10

(:, :, 2) =

11 12 13 14 15 16 17 18 19 20

If the subclass of a built-in class defines properties, you cannot concatenate objects of the subclass. There is no way to determine how to combine properties of different objects. However, your subclass can define custom horzcat and vertcat methods to support concatenation in whatever way makes sense for your subclass.

See Also

Related Examples
• “Subclasses of Built-In Types Without Properties” on page 12-60
See Also

- “Subclasses of Built-In Types with Properties” on page 12-68
Subclasses of Built-In Types Without Properties

In this section...

“Specialized Numeric Types” on page 12-60
“A Class to Manage uint8 Data” on page 12-60
“Using the DocUint8 Class” on page 12-61

Specialized Numeric Types

Subclass built-in numeric types to create customized data types that inherit the functionality of the built-in type. Add functionality to that provided by the superclass by implementing class methods. Subclasses without properties store numeric data as the superclass type. If your subclass design does not require properties to store other data, the implementation is simpler because you do not need to define indexing and concatenation methods.

For more information, see “Subclasses of MATLAB Built-In Types” on page 12-50.

A Class to Manage uint8 Data

This example shows a class derived from the built-in uint8 class. This class simplifies the process of maintaining a collection of intensity image data defined by uint8 values. The basic operations of the class include:

- Capability to convert various classes of image data to uint8 to reduce object data storage.
- A method to display the intensity images contained in the subclass objects.
- Ability to use all the methods supported by uint8 data (for example, size, indexing, reshape, bitshift, cat, fft, arithmetic operators, and so on).

The class data are matrices of intensity image data stored in the superclass part of the subclass object. This approach requires no properties.

The DocUint8 class stores the image data, which converts the data, if necessary:

```matlab
classdef DocUint8 < uint8
    methods
        function obj = DocUint8(data)
```
if nargin == 0
    data = uint8(0);
end
obj = obj@uint8(data); % Store data on superclass
end
end
function h = showImage(obj)
    data = uint8(obj);
    figure; colormap(gray(256))
    h = imagesc(data,[0 255]);
    axis image
    brighten(.2)
end
end
end

Using the DocUint8 Class

Create DocUint8 Objects

The DocUint8 class provides a method to display all images stored as DocUint8 objects in a consistent way. For example:

cir = imread('circuit.tif');
img1 = DocUint8(cir);
img1.showImage;
Because DocUint8 subclasses uint8, you can use any uint8 methods. For example,

```
size(img1)
```

```
ans =
   280   272
```

returns the size of the image data.

**Indexing Operations**

Inherited methods perform indexing operations, but return objects of the same class as the subclass.

Therefore, you can index into the image data and call a subclass method:
showImage(img1(100:200,1:160));

Subscripted reference operations (controlled by the inherited `subsref` method) return a `DocUint8` object.

You can assign values to indexed elements:

```plaintext
img1(100:120,140:160) = 255;
img1.showImage;
```

Subscripted assignment operations (controlled by the inherited `subsasgn` method) return a `DocUint8` object.
Concatenation Operations

Concatenation operations work on DocUint8 objects because this class inherits the uint8 horzcat and vertcat methods, which return a DocUint8 object:

showImage([img1 img1]);
Data Operations

Methods that operate on data values, such as arithmetic operators, always return an object of the built-in type (not of the subclass type). For example, multiplying DocUint8 objects returns a uint8 object:

```plaintext
a = img1.*1.8;
showImage(a);
```

Undefined function 'showImage' for input arguments of type 'uint8'.

To perform operations of this type, implement a subclass method to override the inherited method. The times method implements array (element-by-element) multiplication.

Add this method to the DocUint8 class:
function o = times(obj,val)
    u8 = uint8(obj).*val;
    o = DocUint8(u8);
end

When you override a uint8 method, MATLAB calls the subclass method, not the base class method. The subclass method must:

- Call the uint8 times method on the DocUint8 object data.
- Construct a new DocUint8 object using the uint8 data.

After adding the times method to the DocUint8 class, the output of multiplication expressions is an object of the DocUint8 class:

showImage(img1.*1.8);
See Also

Related Examples

- “Operator Overloading” on page 17-47
- “Subclasses of Built-In Types with Properties” on page 12-68
Subclasses of Built-In Types with Properties

In this section...

| “Specialized Numeric Types with Additional Data Storage” on page 12-68 |
| “Subclasses with Properties” on page 12-68 |
| “Property Added” on page 12-69 |
| “Methods Implemented” on page 12-69 |
| “Class Definition Code” on page 12-69 |
| “Using ExtendDouble” on page 12-72 |
| “Concatenation of ExtendDouble Objects” on page 12-75 |

Specialized Numeric Types with Additional Data Storage

Subclass built-in numeric types to create customized data types that inherit the functionality of the built-in type. Add or modify functionality to that provided by the superclass by implementing class methods.

Providing additional data storage in the subclass by defining properties can be a useful extension to the built-in data class. However, the addition of properties to the subclass requires the subclass to define methods to implement standard array behaviors.

For more information, see “Subclasses of MATLAB Built-In Types” on page 12-50.

Subclasses with Properties

When a subclass of a built-in class defines properties, default indexing and concatenation do not work. The default subsref, subsasgn, horzcat, and vertcat functions cannot work with unknown property types and values. Therefore, the subclass must define these behaviors by implementing these methods.

This sample implementation of the ExtendDouble class derives from the double class and defines a single property. The ExtendDouble class definition demonstrates how to implement indexing and concatenation for subclasses of built-in classes.
Property Added

The ExtendDouble class defines the DataString property to contain text that describes the data. The superclass part of the class contains the numeric data.

Methods Implemented

The following methods modify the behavior of the ExtendDouble class:

- **ExtendDouble** — The constructor supports a no argument syntax that initializes properties to empty values.
- **subsref** — Enables subscripted reference to the superclass part of the subclass, dot notation reference to the DataString property, and dot notation reference the built-in data via the name Data.
- **subsasgn** — Enables subscripted assignment to the superclass part of the subclass, dot notation reference to the DataString property, and dot notation reference the built-in data via the name Data.
- **horzcat** — Defines horizontal concatenation of ExtendDouble objects. Concatenates the superclass part using the double class horzcat method and forms a cell array of the DataString properties.
- **vertcat** — The vertical concatenation equivalent of horzcat (both are required).
- **char** — A ExtendDouble to char converter used by horzcat and vertcat.
- **disp** — ExtendDouble implements a disp method to provide a custom display for the object.

Class Definition Code

The ExtendDouble class extends double and implements methods to support subscripted indexing and concatenation.

```matlab
classdef ExtendDouble < double
    properties
        DataString
    end

    methods
        function obj = ExtendDouble(data,str)
            if nargin == 0
```
data = 0;
str = ''; 
elseif nargin == 1
str = ''; 
end
obj = obj@double(data);
obj.DataString = str;
end

function sref = subsref(obj,s)
switch s(1).type
  case '.
    switch s(1).subs
      case 'DataString'
        sref = obj.DataString;
      case 'Data'
        d = double(obj);
        if length(s)<2
          sref = d;
        elseif length(s)>1 && strcmp(s(2).type,'()')
          sref = subsref(d,s(2:end));
        end
      otherwise
        error('Not a supported indexing expression')
    end
  case '{}'
    error('Not a supported indexing expression')
  end
end

function obj = subsasgn(obj,s,b)
switch s(1).type
  case '.
    switch s(1).subs
      case 'DataString'
        obj.DataString = b;
      case 'Data'
        if length(s)<2
          obj = ExtendDouble(b,obj.DataString);
        elseif length(s)>1 && strcmp(s(2).type,'()')
          obj = ExtendDouble(b,obj.DataString);
        end
      otherwise
        error('Not a supported indexing expression')
    end
end
d = double(obj);
newd = subsasgn(d,s(2:end),b);
obj = ExtendDouble(newd,obj.DataString);
end
otherwise
error('Not a supported indexing expression')
end
case '()'
d = double(obj);
newd = subsasgn(d,s(1),b);
obj = ExtendDouble(newd,obj.DataString);
case '{}'
error('Not a supported indexing expression')
end
end

function newobj = horzcat(varargin)
d1 = cellfun(@double,varargin,'UniformOutput',false);
data = horzcat(d1{:});
str = horzcat(cellfun(@char,varargin,'UniformOutput',false));
newobj = ExtendDouble(data,str);
end

function newobj = vertcat(varargin)
d1 = cellfun(@double,varargin,'UniformOutput',false);
data = vertcat(d1{:});
str = vertcat(cellfun(@char,varargin,'UniformOutput',false));
newobj = ExtendDouble(data,str);
end

function str = char(obj)
str = obj.DataString;
end

function disp(obj)
disp(obj.DataString)
disp(double(obj))
end
Using ExtendDouble

Create an instance of ExtendDouble and notice that the display is different from the default:

```matlab
ed = ExtendDouble(1:10,'One to ten')
ed =
One to ten
   1    2    3    4    5    6    7    8    9   10
```

Inherited Methods

The ExtendDouble class inherits methods from the class double. To see a list of all public methods defined by the double class, use the `methods` function:

```matlab
methods(double.empty)
```

The `sum` function continues to operate on the superclass part of the object:

```matlab
sum(ed)
ans =
   55
```

The `sort` function works on the superclass part of the object:

```matlab
sort(ed(10:-1:1))
ans =
   1    2    3    4    5    6    7    8    9   10
```

Arithmetic operators work on the superclass part of the object:

```matlab
ed.^2
ans =
   1     4     9    16    25    36    49    64    81   100
```

Subscripted Indexing

Because the ExtendDouble class defines a property, the class must implement its own `subsref` and `subsasgn` methods.
This class implements the following subscripted indexing expressions for reference and assignment.

- **obj.DataString** — access the DataString property.
- **obj.Data, obj.Data(ind)** — access the data using a property-style reference. Reference returns values of type double.
- **obj(ind)** — access the numeric data (same as obj.Data(ind)). Reference returns values of type ExtendDouble.

The class `subsref` method enables you to use `ExtendDouble` objects like numeric arrays to reference the numeric data:

```matlab
ed = ExtendDouble(1:10,'One to ten');
ed(10:-1:1)
```

```matlab
ans =
One to ten
     10   9   8   7   6   5   4   3   2   1
```

Access the numeric data of the `ExtendDouble` using property-style indexing with the arbitrarily chosen name `Data`:

```matlab
ed.Data(10:-1:1)
```

```matlab
ans =
One to ten
     10   9   8   7   6   5   4   3   2   1
```

Access the `DataString` property:

```matlab
ed.DataString
```

```matlab
ans =
One to ten
```

Subscripted assignment implements similar syntax in the class `subsasgn` method.

```matlab
ed = ExtendDouble(1:10,'One to ten');
ed(11:13) = [11,12,13];
ed.DataString = 'one to thirteen';
ed
```
ed =
"One to thirteen"

1  2  3  4  5  6  7  8  9 10 11 12 13

The ExtendDouble inherits converter methods from the double class. For example, MATLAB calls the char method to perform this assignment statement.

```matlab
ed(11:13) = ['a','b','c']
ed =
"one to thirteen"

1  2  3  4  5  6  7  8  9 10 97 98 99
```

**Class of Value Returned by Indexing Expression**

The ExtendDouble implements two forms of indexed reference in the `subsref` method:

- `obj.Data` and `obj.Data(ind)` — Return values of class `double`
- `obj(ind)` — Return values of class `ExtendDouble`

For example, compare the values returned by these expressions.

```matlab
ed = ExtendDouble(1:10,'One to ten');
a = ed(1)
a =
"One to ten"

1
b = ed.Data(1)
b =

1
```

```
whos
Name      Size            Bytes  Class           Attributes
---------  --------  ----------  --------  -----------------------------------
a         1x1               132  ExtendDouble
b         1x1                 8  double
ed        1x10              204  ExtendDouble
```

12-74
The increased flexibility of the implementation of indexed reference in the ExtendDouble class.

**Concatenation of ExtendDouble Objects**

Create these two objects:

```plaintext
ed1 = ExtendDouble([1:10], 'One to ten');
ed2 = ExtendDouble([10:-1:1], 'Ten to one');
```

Concatenate these objects along the horizontal dimension:

```plaintext
hcat = [ed1, ed2]
```

```plaintext
hcat =
    'One to ten'    'Ten to one'
Columns 1 through 13
   1     2     3     4     5     6     7     8     9    10    10     9     8
Columns 14 through 20
    7     6     5     4     3     2     1
```

```plaintext
whos
Name      Size            Bytes  Class           Attributes
---------- ---------           ------- ---------    --------
ed1       1x10              204  ExtendDouble
ed2       1x10              204  ExtendDouble
hcat      1x20              528  ExtendDouble
```

Vertical concatenation works in a similar way:

```plaintext
vcat = [ed1; ed2]
```

```plaintext
vcat =
    'One to ten'    'Ten to one'
     1     2     3     4     5     6     7     8     9    10
    10     9     8     7     6     5     4     3     2     1
```

Both horzcat and vertcat return a new object of the same class as the subclass.
See Also

Related Examples

• “Subclasses of Built-In Types Without Properties” on page 12-60
Use of size and numel with Classes

In this section...

“size and numel” on page 12-77
“Built-In Class Behavior” on page 12-77
“Subclasses Inherit Behavior” on page 12-78
“Classes Not Derived from Built-In Classes” on page 12-79
“Change the Behavior of size or numel” on page 12-81
“Overload numArgumentsFromSubscript Instead of numel” on page 12-82

size and numel

The size function returns the dimensions of an array. The numel function returns the number of elements in an array, which is equivalent to `prod(size(objArray))`. That is, the product of the array dimensions.

The size and numel functions work consistently with arrays of user-defined objects. There is generally no need to overload size or numel in user-defined classes.

Several MATLAB functions use size and numel to perform their operations. Therefore, if you do overload either of these functions in your class, be sure that objects of your class work as designed with other MATLAB functions.

If your class modifies array indexing, see “Overload numArgumentsFromSubscript Instead of numel” on page 12-82

Built-In Class Behavior

When you use the size and numel functions in classes derived from built-in classes, these functions behave the same as they behave in the superclass.

Consider the built-in class double:

```matlab
d = 1:10;
size(d)
```

```matlab
ans =
    1    10
```
numel(d)
ans =

   10
dsub = d(7:end);
size(dsub)
ans =

   1   4

The `double` class defines these behaviors, including parentheses indexing.

**Subclasses Inherit Behavior**

Unless the subclass explicitly overrides superclass behavior, subclasses behave like their superclasses. For example, `SimpleDouble` subclasses `double` and defines no properties:

```matlab
classdef SimpleDouble < double
    methods
        function obj = SimpleDouble(data)
            if nargin == 0
                data = 0;
            end
            obj = obj@double(data);
        end
    end
end
```

Create an object and assign the values 1:10:

```matlab
sd = SimpleDouble(1:10);
```

The `size` function returns the size of the superclass part:

```matlab
size(sd)
ans =

   1   10
```

The `numel` function returns the number of elements in the superclass part:
numel(sd)
ans =
    10

Object arrays return the size of the superclass arrays:
size([sd;sd])
ans =
    2 10
numel([sd;sd])
ans =
    20

The SimpleDouble class inherits the indexing behavior of the double class:

sdsub = sd(7:end);
size(sdsub)
ans =
    1 4

Classes Not Derived from Built-In Classes

Consider a simple value class. This class does not inherit the array-like behaviors of the double class. For example:

classdef VerySimpleClass
    properties
        Value
    end
end

Create an object and assign a 10-element array to the Value property:

vs = VerySimpleClass;
vs.Value = 1:10;
size(vs)
ans =
    1  1
numel(vs)
an =
    1
size([vs;vs])
an =
    2  1
numel([vs;vs])
an =
    2

vs is a scalar object. The Value property is an array of doubles:
size(vs.Value)
an =
    1  10

Apply indexing expressions to the object property:
        vssub = vs.Value(7:end);
        size(vssub)
an =
    1  4

The vs.Value property is an array of class double:
class(vs.Value)
an =
double
Create an array of VerySimpleClass objects:

```matlab
vsArray(1:10) = VerySimpleClass;
```

The Value property for array elements 2 through 10 is empty:

```matlab
isempty([vsArray(2:10).Value])
```

```matlab
ans =

1
```

MATLAB does not apply scalar expansion to object array property value assignment. Use the `deal` function for this purpose:

```matlab
[vsArray.Value] = deal(1:10);
isempty([vsArray.Value])
```

```matlab
ans =

0
```

The `deal` function assigns values to each Value property in the `vsArray` object array.

Indexing rules for object arrays are equivalent to the rules for arrays of `struct`:

```matlab
vsArray(1).Value
```

```matlab
ans =

1     2     3     4     5     6     7     8     9    10
```

```matlab
vsArray(1).Value(6)
```

```matlab
ans =

6
```

**Change the Behavior of size or numel**

Subclasses of built-in numeric classes inherit a `size` method, which operates on the superclass part of the subclass object (these methods are hidden). If you want `size` or `numel` to behave differently, override them by defining a `size` or `numel` method in your subclass.
Other MATLAB functions use the values returned by these functions. If you change the way that `size` and `numel` behave, ensure that the values returned make sense for the intended use of your class.

**Overload numArgumentsFromSubscript Instead of numel**

If classes implement a `numArgumentsFromSubscript` method, MATLAB calls it instead of `numel` to determine the number of elements returned by indexed expressions that return comma-separated lists. For example, expressions such as:

```matlab
A(1:2).Prop
```

Both `subsref` and `subsasgn` use `numArgumentsFromSubscript`:

- `subsref` — `numArgumentsFromSubscript` computes the number of expected outputs (`nargout`) returned `subsref`.
- `subsasgn` — `numArgumentsFromSubscript` computes the number of expected inputs (`nargin`) that MATLAB assigns as a result of a call to `subsasgn`.

Subclasses of built-in classes always return scalar objects as a result of subscripted reference and always use scalar objects for subscripted assignment.

If you define a class in which `nargout` for `subsref` or `nargin` for `subsasgn` must be a specific value, then overload `numArgumentsFromSubscript` to return that value.

**See Also**

`numArgumentsFromSubscript`

**Related Examples**

- “Modify `nargout` and `nargin` for Indexing Methods” on page 17-9
Class to Represent Hardware

In this section...

“Objective” on page 12-83
“Why Derive from int32” on page 12-83
“Implementation” on page 12-83
“Construct MuxCard Object” on page 12-84
“Call Methods of int32” on page 12-85

Objective

This example implements a class to represent an optical multiplex card. These cards typically have several input ports and an output port. The MuxCard class represents the ports by the port names and port data rates. The output rate of a multiplex card is the sum of the input port data rates.

Why Derive from int32

The MuxCard class derives from the int32 class because 32-bit integers represent the input port data rates. The MuxCard class inherits the methods of the int32 class, which simplifies the implementation of this subclass. For example, numeric array indexing and arithmetic operations work on MuxCard objects because the class inherits these operations from the int32 class.

Implementation

Here is the definition of the MuxCard class. Notice that the input port rates initialize the int32 portion of class.

```matlab
classdef MuxCard < int32
    properties
        InPutNames
        OutPutName
    end

    properties (Dependent = true)
        OutPutRate
end
```
Construct MuxCard Object

The constructor takes three arguments:

- `inptnames` — Cell array of input port names
- `inptrates` — Vector of input port rates
- `outpname` — Name for the output port

```matlab
omx = MuxCard({'inp1','inp2','inp3','inp4'},[3 12 12 48],'outp')
```

```text
1x4 MuxCard array with properties:

   InPutNames: {'inp1' 'inp2' 'inp3' 'inp4'}
```

12-84
OutPutName: 'outp'
OutPutRate: 75

int32 data:
   3   12   12   48

Call Methods of int32

Use a MuxCard object like an array of int32 values. For example, this indexing statement accesses the data in the object to determine the names of the input ports that have a rate of 12:

omx.InPutNames(omx==12)

ans =
   inp2   inp3

The indexing statement generates a logical array index:

omx == 12

ans =
   0   1   1   0

Indexing the MuxCard object accesses the int32 vector of input port rates:

omx(1:2)

ans =
   3   12

The OutPutRate property get access method uses sum to sum the output port rates:

omx.OutPutRate

ans =
   75
See Also

Related Examples
• “Subclasses of Built-In Types with Properties” on page 12-68
Determine Array Class

In this section...

“Query the Class Name” on page 12-87
“Test for Array Class” on page 12-87
“Test for Specific Types” on page 12-88
“Test for Most Derived Class” on page 12-89

Query the Class Name

To determine the class of an array, use the class function:

\[
\begin{align*}
a &= [2,5,7,11]; \\
\text{class}(a) \\
\text{ans} &= \text{double} \\
\text{str} &= \text{'Character array'}; \\
\text{class}(\text{str}) \\
\text{ans} &= \text{char}
\end{align*}
\]

Test for Array Class

The isa function enables you to test for a specific class or a category of numeric class (numeric, float, integer):

\[
\begin{align*}
a &= [2,5,7,11]; \\
\text{isa}(a,\text{'double'}) \\
\text{ans} &= 1 \\
\text{isa}(a,\text{'float'}) \\
\text{ans} &= 1
\end{align*}
\]
Numeric values (floating-point and integer values):

```matlab
isa(a,'numeric')
ans =
    1
```

**isa Returns True for Subclasses**

`isa` returns true for classes derived from the specified class. For example, the `SubInt` class derives from the built-in type `int16`:

```matlab
classdef SubInt < int16
    methods
        function obj = SubInt(data)
            if nargin == 0
                data = 0;
            end
            obj = obj@int16(data);
        end
    end
end
```

By definition, an instance of the `SubInt` class is also an instance of the `int16` class:

```matlab
aInt = SubInt;
isa(aInt,'int16')
ans =
    1
```

Using the `integer` category also returns true:

```matlab
isa(aInt,'integer')
ans =
    1
```

**Test for Specific Types**

The `class` function returns the name of the *most derived* class of an object:

```matlab
class(aInt)
```
Use the `strcmp` function with the `class` function to check for a specific class of an object:

```matlab
a = int16(7);
strcmp(class(a), 'int16')
```

```matlab
ans =
   1
```

Because the `class` function returns the class name as a `char` vector, the inheritance does not affect the result of the comparison performed by `strcmp`:

```matlab
aInt = SubInt;
strcmp(class(aInt), 'int16')
```

```matlab
ans =
   0
```

**Test for Most Derived Class**

If you define functions that require inputs that are:

- MATLAB built-in types
- Not subclasses of MATLAB built-in types

Use the following techniques to exclude subclasses of built-in types from the input arguments.

- Define a cell array that contains the names of built-in types accepted by your function.
- Call `class` and `strcmp` to test for specific types in a MATLAB control statement.

Test an input argument:

```matlab
if strcmp(class(inputArg), 'single')
   % Call function
else
   inputArg = single(inputArg);
end
```
Test for Category of Types

Suppose that you create a MEX-function, myMexFcn, that requires two numeric inputs that must be of type double or single:

```matlab
outArray = myMexFcn(a,b)
```

Define a cell array that contains the character arrays double and single:

```matlab
floatTypes = {'double','single'};
```

% Test for proper types
```matlab
if any(strcmp(class(a),floatTypes)) && ...
    any(strcmp(class(b),floatTypes))
    outArray = myMexFcn(a,b);
else
    % Try to convert inputs to avoid error
    ...
end
```

Another Test for Built-In Types

Use isobject to separate built-in types from subclasses of built-in types. The isobject function returns false for instances of built-in types:

```matlab
% Create a int16 array
a = int16([2,5,7,11]);
isobject(a)
```

ans =
0

Determine if an array is one of the built-in integer types:

```matlab
if isa(a,'integer') && ~isobject(a)
    % a is a built-in integer type
    ...
end
```
Abstract Classes

Abstract Classes

Abstract classes are useful for describing functionality that is common to a group of classes, but requires unique implementations within each class.

Abstract Class Terminology

abstract class — A class that cannot be instantiated, but that defines class components used by subclasses.

abstract members — Properties or methods declared in an abstract class, but implemented in subclasses.

concrete members — Properties or methods that are fully implemented by a class.

concrete class — A class that can be instantiated. Concrete classes contain no abstract members.

interface — An abstract class describing functionality that is common to a group of classes, but that requires unique implementations within each class. The abstract class defines the interface of each subclass without specifying the actual implementation.

An abstract class serves as a basis (that is, a superclass) for a group of related subclasses. An abstract class can define abstract properties and methods that subclasses implement. Each subclass can implement the concrete properties and methods in a way that supports their specific requirements.

Implementing a Concrete Subclass

A subclass must implement all inherited abstract properties and methods to become a concrete class. Otherwise, the subclass is itself an abstract class.
MATLAB does not force subclasses to implement concrete methods with the same signature or attributes.

Abstract classes:
• Can define properties and methods that are not abstract
• Pass on their concrete members through inheritance
• Do not need to define any abstract members

**Declare Classes as Abstract**

A class is abstract when it declares:
• The Abstract class attribute
• An abstract method
• An abstract property

If a subclass of an abstract class does not define concrete implementations for all inherited abstract methods or properties, it is also abstract.

**Abstract Class**

Declare a class as abstract in the classdef statement:

```matlab
classdef (Abstract) AbsClass
    ...
end
```

For classes that declare the Abstract class attribute:
• Concrete subclasses must redefine any properties or methods that are declared as abstract.
• The abstract class does not need to define any abstract methods or properties.

When you define any abstract methods or properties, MATLAB automatically sets the class Abstract attribute to true.

**Abstract Methods**

Define an abstract method:
methods (Abstract)
    abstMethod(obj)
end

For methods that declare the Abstract method attribute:

- Do not use a function...end block to define an abstract method, use only the method signature.
- Abstract methods have no implementation in the abstract class.
- Concrete subclasses are not required to support the same number of input and output arguments and do not need to use the same argument names. However, subclasses generally use the same signature when implementing their version of the method.

**Abstract Properties**

Define an abstract property:

properties (Abstract)
    AbsProp
end

For properties that declare the Abstract property attribute:

- Concrete subclasses must redefine abstract properties without the Abstract attribute.
- Concrete subclasses must use the same values for the SetAccess and GetAccess attributes as those attributes used in the abstract superclass.
- Abstract properties cannot define access methods and cannot specify initial values. The subclass that defines the concrete property can create access methods and specify initial values.

For more information on access methods, see “Property Access Methods” on page 8-49.

**Determine If a Class Is Abstract**

Determine if a class is abstract by querying the Abstract property of its meta.class object. For example, the AbsClass defines two abstract methods:

```
classdef AbsClass
    methods (Abstract)
        result = absMethodOne(obj)
```
output = absMethodTwo(obj)
end
end

Use the logical value of the meta.class Abstract property to determine if the class is abstract:

mc = ?AbsClass;
if ~mc.Abstract
% not an abstract class
end

**Display Abstract Member Names**

Use the meta.abstractDetails function to display the names of abstract properties or methods and the names of the defining classes:

meta.abstractDetails('AbsClass');

Abstract methods for class AbsClass:
  absMethodTwo    % defined in AbsClass
  absMethodOne    % defined in AbsClass

**Find Inherited Abstract Properties and Methods**

The meta.abstractDetails function returns the names and defining class of any inherited abstract properties or methods that you have not implemented in your subclass. Use this function if you want the subclass to be concrete and must determine what abstract members the subclass inherits.

For example, suppose that you create a subclass of the AbsClass class that is defined in the previous section. In this case, the subclass implements only one of the abstract methods defined by AbsClass.

classdef SubAbsClass < AbsClass
  % Does not implement absMethodOne
  % defined as abstract in AbsClass
  methods
    function out = absMethodTwo(obj)
      ...
    end
  end
end
Determine if you implemented all inherited class members using `meta.abstractDetails`:

```
meta.abstractDetails(?SubAbsClass)
```

Abstract methods for class SubAbsClass:
```
absMethodOne % defined in AbsClass
```

The `SubAbsClass` class is abstract because it has not implemented the `absMethodOne` method defined in `AbsClass`.

```
msub = ?SubAbsClass;
msub.Abstract

ans = 1
```

If you implement both methods defined in `AbsClass`, the subclass becomes concrete.

**See Also**

**Related Examples**
- “Define an Interface Superclass” on page 12-96
Define an Interface Superclass

In this section...

| “Interfaces” on page 12-96 |
| “Interface Class Implementing Graphs” on page 12-96 |

Interfaces

The properties and methods defined by a class form the interface that determines how class users interact with objects of the class. When creating a group of related classes, interfaces define a common interface to all these classes. The actual implementations of the interface can differ from one class to another.

Consider a set of classes designed to represent various types of graphs. All classes must implement a Data property to contain the data used to generate the graph. However, the form of the data can differ considerably from one type of graph to another. Each class can implement the Data property differently.

The same differences apply to methods. All classes can have a draw method that creates the graph, but the implementation of this method changes with the type of graph.

The basic idea of an interface class is to specify the properties and methods that each subclass must implement without defining the actual implementation. This approach enables you to enforce a consistent interface to a group of related objects. As you add more classes in the future, the interface remains the same.

Interface Class Implementing Graphs

This example creates an interface for classes used to represent specialized graphs. The interface is an abstract class that defines properties and methods that the subclasses must implement, but does not specify how to implement these components.

This approach enforces the use of a consistent interface while providing the necessary flexibility to implement the internal workings of each specialized subclass differently.

In this example, a package folder contains the interface, derived subclasses, and a utility function:

+ graphics/GraphInterface.m % abstract interface class
+ graphics/LineGraph.m % concrete subclass
Interface Properties and Methods

The graph class specifies the following properties, which the subclasses must define:

- **Primitive** — Handle of the graphics object used to implement the specialized graph. The class user has no need to access these objects directly so this property has protected SetAccess and GetAccess.
- **AxesHandle** — Handle of the axes used for the graph. The specialized graph objects can set axes object properties. This property has protected SetAccess and GetAccess.
- **Data** — All subclasses of the GraphInterface class must store data. The type of data varies and each subclass defines the storage mechanism. Subclass users can change the data values so this property has public access rights.

The GraphInterface class names three abstract methods that subclasses must implement. The GraphInterface class also suggests in comments that each subclass constructor must accept the plot data and property name/property value pairs for all class properties.

- **Subclass constructor** — Accept data and P/V pairs and return an object.
- **draw** — Used to create a drawing primitive and render a graph of the data according to the type of graph implemented by the subclass.
- **zoom** — Implementation of a zoom method by changing the axes CameraViewAngle property. The interface suggests the use of the camzoom function for consistency among subclasses. The zoom buttons created by the addButton static method use this method as a callback.
- **updateGraph** — Method called by the set.Data method to update the plotted data whenever the Data property changes.

Interface Guides Class Design

The package of classes that derive from the GraphInterface abstract class implement the following behaviors:

- Creating an instance of a specialized GraphInterface object (subclass object) without rendering the plot
- Specifying any or none of the object properties when you create a specialized GraphInterface object
- Changing any object property automatically updates the currently displayed plot
• Allowing each specialized `GraphInterface` object to implement whatever additional properties it requires to give class users control over those characteristics.

**Define the Interface**

The `GraphInterface` class is an abstract class that defines the methods and properties used by the subclasses. Comments in the abstract class describe the intended implementation:

```matlab
classdef GraphInterface < handle
    % Abstract class for creating data graphs
    % Subclass constructor should accept
    % the data that is to be plotted and
    % property name/property value pairs
    properties (SetAccess = protected, GetAccess = protected)
        Primitive
        AxesHandle
    end
    properties
        Data
    end
    methods (Abstract)
        draw(obj)
            % Use a line, surface,
            % or patch graphics primitive
        zoom(obj,factor)
            % Change the CameraViewAngle
            % for 2D and 3D views
            % use camzoom for consistency
        updateGraph(obj)
            % Update the Data property and
            % update the drawing primitive
    end

    methods
        function set.Data(obj,newdata)
            obj.Data = newdata;
            updateGraph(obj)
        end
        function addButtons(gobj)
            hfig = get(gobj.AxesHandle,'Parent');
            uicontrol(hfig,'Style','pushbutton','String','Zoom Out',...  
                'Callback',@(src,evnt)zoom(gobj,.5));
            uicontrol(hfig,'Style','pushbutton','String','Zoom In',...
```
The GraphInterface class implements the property set method (set.Data) to monitor changes to the Data property. An alternative is to define the Data property as Abstract and enable the subclasses to determine whether to implement a set access method for this property. The GraphInterface class defines a set access method that calls an abstract method (updateGraph, which each subclass must implement). The GraphInterface interface imposes a specific design on the whole package of classes, without limiting flexibility.

**Method to Work with All Subclasses**

The addButtons method adds push buttons for the zoom methods, which each subclass must implement. Using a method instead of an ordinary function enables addButtons to access the protected class data (the axes handle). Use the object zoom method as the push-button callback.

```matlab
function addButtons(gobj)
    hfig = get(gobj.AxesHandle,'Parent');
    uicontrol(hfig,'Style','pushbutton',...
        'String','Zoom Out',...
        'Callback',@(src,evnt)zoom(gobj,.5));
    uicontrol(hfig,'Style','pushbutton',...
        'String','Zoom In',...
        'Callback',@(src,evnt)zoom(gobj,2),...
        'Position',[100 20 60 20]);
end
```

**Derive a Concrete Class — LineGraph**

This example defines only a single subclass used to represent a simple line graph. It derives from GraphInterface, but provides implementations for the abstract methods draw, zoom, updateGraph, and its own constructor. The base class GraphInterface and subclass are all contained in a package (graphics), which you must use to reference the class name:

```matlab
classdef LineGraph < graphics.GraphInterface
```
Add Properties

The LineGraph class implements the interface defined in the GraphInterface class and adds two additional properties—LineColor and LineType. This class defines initial values for each property, so specifying property values in the constructor is optional. You can create a LineGraph object with no data, but you cannot produce a graph from that object.

properties
    LineColor = [0 0 0];
    LineType = '-';
end

The LineGraph Constructor

The constructor accepts a struct with x and y coordinate data, and property name/property value pairs:

```matlab
function gobj = LineGraph(data,varargin)
    if nargin > 0
        gobj.Data = data;
        if nargin > 2
            for k=1:2:length(varargin)
                gobj.(varargin{k}) = varargin{k+1};
            end
        end
    end
end
```

Implement the draw Method

The LineGraph draw method uses property values to create a line object. The LineGraph class stores the line handle as protected class data. To support the use of no input arguments for the class constructor, draw checks the Data property to determine if it is empty before proceeding:

```matlab
function gobj = draw(gobj)
    if isempty(gobj.Data)
        error('The LineGraph object contains no data')
    end
    h = line(gobj.Data.x,gobj.Data.y,...
             'Color',gobj.LineColor,...
             'LineStyle',gobj.LineType);
    gobj.Primitive = h;
```
Implement the zoom Method

The LineGraph zoom method follows the comments in the GraphInterface class which suggest using the camzoom function. camzoom provides a convenient interface to zooming and operates correctly with the push buttons created by the addButtons method.

Define the Property Set Methods

Property set methods provide a convenient way to execute code automatically when the value of a property changes for the first time in a constructor. (See “Property Set Methods” on page 8-55.) The linegraph class uses set methods to update the line primitive data (which causes a redraw of the plot) whenever a property value changes. The use of property set methods provides a way to update the data plot quickly without requiring a call to the draw method. The draw method updates the plot by resetting all values to match the current property values.

Three properties use set methods: LineColor, LineType, and Data. LineColor and LineType are properties added by the LineGraph class and are specific to the line primitive used by this class. Other subclasses can define different properties unique to their specialization (for example, FaceColor).

The GraphInterface class implements the Data property set method. However, the GraphInterface class requires each subclass to define a method called updateGraph, which handles the update of plot data for the specific drawing primitive used.

The LineGraph Class

Here is the LineGraph class definition.

```matlab
classdef LineGraph < graphics.GraphInterface
    properties
        LineColor = [0 0 0]
        LineType = '-'
    end

    methods
        function gobj = LineGraph(data,varargin)
            if nargin > 0
                gobj.Data = data;
```
if nargin > 1
    for k=1:2:length(varargin)
        gobj.(varargin{k}) = varargin{k+1};
    end
end
end

function gobj = draw(gobj)
    if isempty(gobj.Data)
        error('The LineGraph object contains no data')
    end
    h = line(gobj.Data.x,gobj.Data.y,...
    'Color',gobj.LineColor,...
    'LineStyle',gobj.LineType);
    gobj.Primitive = h;
    gobj.AxesHandle = h.Parent;
end

function zoom(gobj,factor)
    camzoom(gobj.AxesHandle,factor)
end

function updateGraph(gobj)
    set(gobj.Primitive,...
    'XData',gobj.Data.x,...
    'YData',gobj.Data.y)
end

function set.LineColor(gobj,color)
    gobj.LineColor = color;
    set(gobj.Primitive,'Color',color)
end

function set.LineType(gobj,ls)
    gobj.LineType = ls;
    set(gobj.Primitive,'LineStyle',ls)
end
Use the LineGraph Class

The LineGraph class defines the simple API specified by the graph base class and implements its specialized type of graph:

```matlab
d.x = 1:10;
d.y = rand(10,1);
lg = graphics.LineGraph(d,'LineColor','b','LineType',':');
lg.draw;
lg.addButtons;
```

Clicking the **Zoom In** button shows the zoom method providing the callback for the button.
Changing properties updates the graph:

```matlab
d.y = rand(10,1);
lg.Data = d;
lg.LineColor = [0.9,0.1,0.6];
```

Now click **Zoom Out** and see the new results:
See Also

Related Examples
•  “Abstract Classes” on page 12-91
Saving and Loading Objects

- “Save and Load Process for Objects” on page 13-2
- “Reduce MAT-File Size for Saved Objects” on page 13-5
- “Save Object Data to Recreate Graphics Objects” on page 13-7
- “Improve Version Compatibility with Default Values” on page 13-10
- “Avoid Property Initialization Order Dependency” on page 13-12
- “Modify the Save and Load Process” on page 13-16
- “Basic saveobj and loadobj Pattern” on page 13-19
- “Maintain Class Compatibility” on page 13-23
- “Initialize Objects When Loading” on page 13-30
- “Save and Load Objects from Class Hierarchies” on page 13-33
- “Restore Listeners” on page 13-36
- “Save and Load Dynamic Properties” on page 13-39
Save and Load Process for Objects

In this section...

“Save and Load Objects” on page 13-2
“What Information Is Saved?” on page 13-2
“How Is the Property Data Loaded?” on page 13-3
“Errors During Load” on page 13-3

Save and Load Objects

Use save and load to store and reload objects:

save filename object
load filename object

What Information Is Saved?

Saving objects in MAT-files saves:

• The full name of the object class, including any package qualifiers
• Values of dynamic properties
• All property default values defined by the class at the time the first object of the class is saved to the MAT-file.
• The names and values of all properties, with the following exceptions:

  • Properties are not saved if their current values are the same as the default values specified in the class definition.
  • Properties are not saved if their Transient, Constant, or Dependent attributes set to true.

For a description of property attributes, see “Specify Property Attributes” on page 8-7

To save graphics objects, see savefig.

Note  Do not use the pack command with objects that define events and listeners. The pack command causes the destruction of any listeners defined for the objects in the
workspace. For information on restoring listeners when saving objects, see “Restore Listeners” on page 13-36.

**How Is the Property Data Loaded?**

When loading objects from MAT-files, the `load` function restores the object.

- `load` creates a new object.
- If the class `ConstructOnLoad` attribute is set to `true`, `load` calls the class constructor with no arguments. Otherwise, `load` does not call the class constructor.
- `load` assigns the saved property values to the object properties. These assignments result in calls to property set methods defined by the class (except in the case of `Dependent`, `Constant`, or `Transient` properties, which are not saved or loaded).
- `load` assigns the default values saved in the MAT-file to properties whose values were not saved because the properties were set to the default values when saved. These assignments result in calls to property set methods defined by the class.

MATLAB calls property set methods to ensure that property values are still valid in cases where the class definition has changed.

For information on property set methods, see “Property Set Methods” on page 8-55.

**Errors During Load**

If a new version of a class removes, renames, or changes the validation for a property, `load` can generate an error when attempting to set the altered or deleted property.

When an error occurs while an object is being loaded from a file, MATLAB does one of the following:

- If the class defines a `loadobj` method, MATLAB returns the saved values to the `loadobj` method in a `struct`.
- If the class does not define a `loadobj` method, MATLAB silently ignores the errors. The `load` function reconstitutes the object with property values that do not produce an error.

In the `struct` passed to the `loadobj` method, the field names correspond to the property names. The field values are the saved values for the corresponding properties.
If the saved object derives from multiple superclasses that have private properties with the same name, the `struct` contains only the property value of the most direct superclass.

For information on how to implement `saveobj` and `loadobj` methods, see “Modify the Save and Load Process” on page 13-16.

**Changes to Property Validation**

If a class definition changes property validation such that loaded property values are no longer valid, MATLAB substitutes the currently defined default value for that property. The class can define a `loadobj` method or converter methods to provide compatibility among class versions.

For information on property validation, see “Validate Property Values” on page 8-24

**See Also**

**Related Examples**

- “Object Save and Load”
Reduce MAT-File Size for Saved Objects

Default Values

If a property often has the same value, define a default value for that property. When the user saves the object to a MAT-file, MATLAB does not save the value of a property if the current value equals the default value. MATLAB saves the default value on a per class basis to avoid saving the value for every object.

For more information on how MATLAB evaluates default value expressions, see “Property Default Values” on page 8-18.

Dependent Properties

Use a dependent property when the property value must be calculated at run time. A dependent property is not saved in the MAT-file when you save an object. Instances of the class do not allocate memory to hold a value for a dependent property.

Dependent is a property attribute (see “Property Attributes” on page 8-9 for a complete list.)

Transient Properties

MATLAB does not store the values of transient properties. Transient properties can store data in the object temporarily as an intermediate computation step or for faster retrieval. Use transient properties when you easily can reproduce the data at run time or when the data represents intermediate state that can be discarded.
Avoid Saving Unwanted Variables

Do not save variables that you do not want to load. Be sure that an object is still valid before you save it. For example, if you save a deleted handle object, MATLAB loads it as a deleted handle.

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-16
- “Object Save and Load”
Save Object Data to Recreate Graphics Objects

In this section...

“What to Save” on page 13-7
“Regenerate When Loading” on page 13-7
“Change to a Stairstep Chart” on page 13-8

What to Save

Use transient properties to avoid saving what you can recreate when loading the object. For example, an object can contain component parts that you can regenerate from data that is saved. Regenerating these components also enables newer versions of the class to create the components in a different way.

Regenerate When Loading

The YearlyRainfall class illustrates how to regenerate a graph when loading objects of that class. YearlyRainfall objects contain a bar chart of the monthly rainfall for a given location and year. The Location and Year properties are ordinary properties whose values are saved when you save the object.

The Chart property contains the handle to the bar chart. When you save a bar chart, MATLAB also saves the figure, axes, and Bar object and the data required to create these graphics objects. The YearlyRainfall class design eliminates the need to save objects that it can regenerate:

- The Chart property is Transient so the graphics objects are not saved.
- ChartData is a private property that provides storage for the Bar object data (YData).
- The load function calls the set.ChartData method, passing it the saved bar chart data.
- The setup method regenerates the bar chart and assigns the handle to the Chart property. Both the class constructor and the set.ChartData method call setup.

```matlab
classdef YearlyRainfall < handle
    properties
        Location
```
Year
end
properties(Transient)
  Chart
end
properties(Access = private)
  ChartData
end
methods
  function rf = YearlyRainfall(data)
    setup(rf,data);
  end
  function set.ChartData(obj,V)
    setup(obj,V);
  end
  function V = get.ChartData(obj)
    V = obj.Chart.YData;
  end
end
methods(Access = private)
  function setup(rf,data)
    rf.Chart = bar(data);
  end
end

Change to a Stairstep Chart

An advantage of the YearlyRainfall class design is the flexibility to modify the type of graph used without making previously saved objects incompatible. Loading the object recreates the graph based only on the data that is saved to the MAT-file.

For example, change the type of graph from a bar chart to a stair-step graph by modifying the setup method:

methods(Access = private)
  function setup(rf,data)
    rf.Chart = stairs(data);
  end
end
See Also

Related Examples

• “Modify the Save and Load Process“ on page 13-16
• “Object Save and Load“
Improve Version Compatibility with Default Values

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Version Compatibility” on page 13-10</td>
</tr>
<tr>
<td>“Using a Default Property Value“ on page 13-10</td>
</tr>
</tbody>
</table>

**Version Compatibility**

Default property values can help you implement version compatibility for saved objects. For example, suppose that you add a property to version 2 of your class. Having a default value enables MATLAB to assign a value to the new property when loading a version 1 object.

Similarly, suppose version 2 of your class removes a property. If a version 2 object is saved and loaded into version 1, your `loadobj` method can use the default value from version 1.

**Using a Default Property Value**

The `EmployeeInfo` class shows how to use property default values as a way to enhance compatibility among versions. Version 1 of the `EmployeeInfo` class defines three properties — Name, JobTitle, and Department.

```matlab
classdef EmployeeInfo
    properties
        Name
        JobTitle
        Department
    end
end
```

Version 2 of the `EmployeeInfo` class adds a property, Country, for the country name of the employee location. The Country property has a default value of 'USA'.

```matlab
classdef EmployeeInfo
    properties
        Name
        JobTitle
        Department
        Country = 'USA'
    end
end
```
The character array, 'USA', is a good default value because:

- MATLAB assigns an empty double [] to properties that do not have default values defined by the class. Empty double is not a valid value for the Country property.
- In version 1, all employees were in the USA. Therefore, any version 1 object loaded into version 2 receives a valid value for the Country property.

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-16
- “Object Save and Load”
Avoid Property Initialization Order Dependency

In this section...

“Control Property Loading” on page 13-12
“Dependent Property with Private Storage” on page 13-13
“Property Value Computed from Other Properties” on page 13-14

Control Property Loading

Problems can occur if property values depend on the order in which load sets the property values.

Suppose that your class design is such that both of the following are true:

• A property set method changes another property value.
• A property value is computed from other property values.

Then the final state of an object after changing a series of property values can depend on the order in which you set the properties. This order dependency can affect the result of loading an object.

The load function sets property values in a particular order. This order can be different from the order in which you set the properties in the saved object. As a result, the loaded object can have different property values than the object had when it was saved.

Restore Nondependent Properties

If a property set function changes the values of other properties, then define the Dependent attribute of that property as true. MATLAB does not save or restore dependent property values.

Use nondependent properties for storing the values set by the dependent property. Then the load function restores the nondependent properties with the same values that were saved. The load function does not call the dependent property set method because there is no value in the saved file for that property.
Dependent Property with Private Storage

The Odometer class avoids order dependences when loading objects by controlling which properties are restored when loading:

- The Units property is dependent. Its property set method sets the TotalDistance property. Therefore load does not call the Units property set method.
- The load function restores TotalDistance to whatever value it had when you saved the object.

classdef Odometer
    properties(Constant)
        ConversionFactor = 1.6
    end
    properties
        TotalDistance = 0
    end
    properties(Dependent)
        Units
    end
    properties(Access=private)
        PrivateUnits = 'mi'
    end
methods
    function unit = get.Units(obj)
        unit = obj.PrivateUnits;
    end
    function obj = set.Units(obj,newUnits)
        % validate newUnits to be a char vector
        switch(newUnits)
            case 'mi'
                if strcmp(obj.PrivateUnits,'km')
                    obj.TotalDistance = obj.TotalDistance / ...
                        obj.ConversionFactor;
                    obj.PrivateUnits = newUnits;
                end
            case 'km'
                if strcmp(obj.PrivateUnits,'mi')
                    obj.TotalDistance = obj.TotalDistance * ...
                        obj.ConversionFactor;
                    obj.PrivateUnits = newUnits;
                end
            otherwise
                error('Odometer:InvalidUnits', ...
Suppose that you create an instance of Odometer and set the following property values:

```matlab
odObj = Odometer;
odObj.Units = 'km';
odObj.TotalDistance = 16;
```

When you save the object:

- `ConversionFactor` is not saved because it is a `Constant` property.
- `TotalDistance` is saved.
- `Units` is not saved because it is a `Dependent` property.
- `PrivateUnits` is saved and provides the storage for the current value of `Units`.

When you load the object:

- `ConversionFactor` is obtained from the class definition.
- `TotalDistance` is loaded.
- `Units` is not loaded, so its set method is not called.
- `PrivateUnits` is loaded from the saved object.

If the `Units` property was not `Dependent`, loading it calls its set method and causes the `TotalDistance` property to be set again.

**Property Value Computed from Other Properties**

The `Odometer2` class `TripDistance` property depends only on the values of two other properties, `TotalDistance` and `TripMarker`.

The class avoids order dependence when initializing property values during the load process by making the `TripDistance` property dependent. MATLAB does not save or load a value for the `TripDistance` property, but does save and load values for the two properties used to calculate `TripDistance` in its property get method.

```matlab
classdef Odometer2
    properties
```
end

properties Dependent
    TripDistance
end

methods
    function distance = get.TripDistance(obj)
        distance = obj.TotalDistance - obj.TripMarker;
    end
end end

See Also

Related Examples

• “Modify the Save and Load Process” on page 13-16
• “Object Save and Load”
Modify the Save and Load Process

When to Modify the Save and Load Process

The primary reason for modifying the save and load process is to support backward and forward compatibility of classes. Consider modifying the save and load process when you:

- Rename a class
- Remove properties
- Define a circular reference of handle objects where initialization order is important
- Must call the constructor with arguments and, therefore, cannot use `ConstructOnLoad`

How to Modify the Save and Load Process

The most versatile technique for modifying the save and load process is to implement `loadobj`, and if necessary, `saveobj` methods for your class. MATLAB executes these methods when you call `save` or `load` on an object of the class.

The `save` function calls your class `saveobj` method before performing the save operation. The `save` function then saves the value returned by the `saveobj` method. You can use `saveobj` to return a modified object or a struct that contains property values.

`load` calls your class `loadobj` method after loading the object. The `load` function loads the value returned by the `loadobj` method into the workspace. A `loadobj` method can modify the object being loaded or can reconstruct an object from the data saved by the class `saveobj` method.
Implementing saveobj and loadobj Methods

Implement a saveobj method that modifies the object being saved, then implement a loadobj method to return the object to the correct state when loading it.

Implement the loadobj method as a Static method because MATLAB can call the loadobj method with a struct instead of an object of the class.

Implement the saveobj method as an ordinary method (that is, calling it requires an instance of the class).

MATLAB saves the object class name so that load can determine which loadobj method to call in cases where your saveobj method saves only the object data in a structure. Therefore, the class must be accessible to MATLAB when you load the object.

Use a loadobj method when:

- The class definition has changed since the object was saved, requiring you to modify the object before loading.
- A saveobj method modified the object during the save operation, possibly saving data in a struct. Implement the loadobj method to reconstruct the object from the output of saveobj.

Additional Considerations

When you decide to modify the default save and load process, keep the following points in mind:

- If loading any property value from the MAT-file produces an error, load passes a struct to loadobj. The struct field names correspond to the property names extracted from the file.
- loadobj must always be able to accept a struct as input and return an object, even if there is no saveobj or saveobj does not return a struct.
- If saveobj returns a struct, then load always passes that struct to loadobj.
- Subclass objects inherit superclass loadobj and saveobj methods. Therefore, if you do not implement a loadobj or saveobj method in the subclass, MATLAB calls only the inherited methods.

If a superclass implements a loadobj or saveobj method, then a subclass can also implement a loadobj or saveobj method that calls the superclass methods. For
more information, see “Save and Load Objects from Class Hierarchies” on page 13-33.

- The load function does not call the constructor by default. For more information, see “Initialize Objects When Loading” on page 13-30.

**See Also**

**Related Examples**

- “Basic saveobj and loadobj Pattern” on page 13-19
- “Object Save and Load”
Basic saveobj and loadobj Pattern

In this section...

“Using saveobj and loadobj” on page 13-19
“Handle Load Problems” on page 13-20

Using saveobj and loadobj

Depending on the requirements of your class, there are various ways you can use saveobj and loadobj methods. This pattern is a flexible way to solve problems that you cannot address by simpler means.

The basic process is:

- Use saveobj to save all essential data in a struct and do not save the object.
- Use loadobj to reconstruct the object from the saved data.

This approach is not useful in cases where you cannot save property values in a struct field. Data that you cannot save, such as a file identifier, you can possibly regenerate in the loadobj method.

saveobj

For this pattern, define saveobj as an ordinary method that accepts an object of the class and returns a struct.

- Copy each property value to a structure field of the same name.
- You can save only the data that is necessary to rebuild the object. Avoid saving whole objects hierarchies, such as those created by graphs.

methods
function s = saveobj(obj)
    s.Prop1 = obj.Prop1;
    s.Prop2 = obj.Prop2;
    s.Data = obj.GraphHandle.YData;
end
end
**loadobj**

Define `loadobj` as a static method. Create an object by calling the class constructor. Then assign values to properties from the `struct` passed to `loadobj`. Use the data to regenerate properties that were not saved.

```matlab
methods(Static)
    function obj = loadobj(s)
        if isstruct(s)
            newObj = ClassConstructor;
            newObj.Prop1 = s.Prop1;
            newObj.Prop2 = s.Prop2
            newObj.GraphHandle = plot(s.Data);
            obj = newObj;
        else
            obj = s;
        end
    end
end
```

If the `load` function encounters an error, `load` passes `loadobj` a `struct` instead of an object. Your `loadobj` method must always be able to handle a `struct` as the input argument. The input to `loadobj` is always a scalar.

**Handle Load Problems**

`loadobj` can handle a `struct` input even if you are not using a `saveobj` method.

The `GraphExpression` class creates a graph of a MATLAB expression over a specified range of data. `GraphExpression` uses its `loadobj` method to regenerate the graph, which is not saved with the object.

```matlab
classdef GraphExpression
    properties
        FuncHandle
        Range
    end
    methods
        function obj = GraphExpression(fh,rg)
            obj.FuncHandle = fh;
            obj.Range = rg;
            makeGraph(obj)
        end
end
```
function makeGraph(obj)
    rg = obj.Range;
    x = min(rg):max(rg);
    data = obj.FuncHandle(x);
    plot(data)
end
end
methods (Static)
    function obj = loadobj(s)
        if isstruct(s)
            fh = s.FuncHandle;
            rg = s.Range;
            obj = GraphExpression(fh,rg);
        else
            makeGraph(s);
            obj = s;
        end
    end
end
end

**Save and Load Object**

Create an object with an anonymous function and a range of data as inputs:

```
    h = GraphExpression(@(x)x.^4,[1:25])
```

```
    h =

    GraphExpression with properties:

        FuncHandle: @(x)x.^4
            Range: [1x25 double]
```

Save the `GraphExpression` object and close the graph:

```
    save myFile h
    close
```

Load the object. MATLAB recreates the graph:

```
    load myFile h
```

If the `load` function cannot create the object and passes a `struct` to `loadobj`, `loadobj` attempts to create an object with the data supplied.
See Also

Related Examples

- “Modify the Save and Load Process” on page 13-16
- “Object Save and Load”
Maintain Class Compatibility

**In this section...**

```
“Rename Property” on page 13-23
“Update Property When Loading” on page 13-25
“Maintaining Compatible Versions of a Class” on page 13-26
“Version 2 of the PhoneBookEntry Class” on page 13-27
```

**Rename Property**

Suppose that you want to rename a property, but do not want to cause errors in existing code that refer to the original property. For example, rename a public property called `OfficeNumber` to `Location`. Here is the original class definition:

```matlab
classdef EmployeeList
    properties
        Name
        Email
        OfficeNumber % Rename as Location
    end
end
```

Use of a hidden dependent property can achieve the desired results.

- In the class definition, set the `OfficeNumber` property attributes to `Dependent` and `Hidden`.
- Create a property set method for `OfficeNumber` that sets the value of the `Location` property.
- Create a property get method for `OfficeNumber` that returns the value of the `Location` location property.

While the `OfficeNumber` property is hidden, existing code can continue to access this property. The `Hidden` attribute does not affect access.

Because `OfficeNumber` is dependent, there is no redundancy in storage required by adding the new property. MATLAB does not store or save dependent properties.

Here is the updated class definition.
classdef EmployeeList
    properties
        Name
        Email
        Location
    end
    properties (Dependent, Hidden)
        OfficeNumber
    end
    methods
        function obj = set.OfficeNumber(obj,val)
            obj.Location = val;
        end
        function val = get.OfficeNumber(obj)
            val = obj.Location;
        end
    end
end

Saving and Loading EmployeeList Objects

You can load old instances of the EmployeeList class in the presence of the new class version. Code that refers to the OfficeNumber property continues to work.

Forward and Backward Compatibility

Suppose that you want to be able to load new EmployeeList objects into systems that still have the old version of the EmployeeList class. To achieve compatibility with old and new versions:

- Define the OfficeNumber property as Hidden, but not Dependent.
- Define the Location property as Dependent.

In this version of the EmployeeList class, the OfficeNumber property saves the value used by the Location property. Loading an object assigns values of the three original properties (Name, Email, and OfficeNumber), but does not assign a value to the new Location property. The lack of the Location property in the old class definition is not a problem.

classdef EmployeeList
    properties
        Name
        Email
properties (Dependent)
  Location
end

properties (Hidden)
  OfficeNumber
end

methods
  function obj = set.Location(obj,val)
    obj.OfficeNumber = val;
  end
  function val = get.Location(obj)
    val = obj.OfficeNumber;
  end
end

Update Property When Loading

Suppose that you modify a class so that a property value changes in its form or type. Previously saved objects of the class must be updated when loaded to have a conforming property value.

Consider a class that has an AccountID property. Suppose that all account numbers must migrate from eight-digit numeric values to 12-element character arrays.

You can accommodate this change by implementing a loadobj method.

The loadobj method:

- Tests to determine if the load function passed a struct or object. All loadobj methods must handle both struct and object when there is an error in load.
- Tests to determine if the AccountID number contains eight digits. If so, change it to a 12-element character array by calling the paddAccID method.

After updating the AccountID property, loadobj returns a MyAccount object that MATLAB loads into the workspace.

classdef MyAccount
  properties
    AccountID
  end
  methods
function obj = padAccID(obj)
    ac = obj.AccountID;
    acstr = num2str(ac);
    if length(acstr) < 12
        obj.AccountID = [acstr,repmat('0',1,12-length(acstr))];
    end
end

methods (Static)
function obj = loadobj(a)
    if isstruct(a)
        obj = MyAccount;
        obj.AccountID = a.AccountID;
        obj = padAccID(obj);
    elseif isa(a,'MyAccount')
        obj = padAccID(a);
    end
end
end

You do not need to implement a saveobj method. You are using loadobj only to ensure that older saved objects are brought up to date while loading.

**Maintaining Compatible Versions of a Class**

The \texttt{PhoneBookEntry} class uses a combination of techniques to maintain compatibility with new versions of the class.

Suppose that you define a class to represent an entry in a phone book. The \texttt{PhoneBookEntry} class defines three properties: \texttt{Name}, \texttt{Address}, and \texttt{PhoneNumber}.

```matlab
classdef PhoneBookEntry
    properties
        Name
        Address
        PhoneNumber
    end
end
```

However, in future releases, the class adds more properties. To provide flexibility, \texttt{PhoneBookEntry} saves property data in a \texttt{struct} using its \texttt{saveobj} method.
methods
  function s = saveobj(obj)
    s.Name = obj.Name;
    s.Address = obj.Address;
    s.PhoneNumber = obj.PhoneNumber;
  end
end

The `loadobj` method creates the `PhoneBookEntry` object, which is then loaded into the workspace.

methods (Static)
  function obj = loadobj(s)
    if isstruct(s)
      newObj = PhoneBookEntry;
      newObj.Name = s.Name;
      newObj.Address = s.Address;
      newObj.PhoneNumber = s.PhoneNumber;
      obj = newObj;
    else
      obj = s;
    end
  end
end

**Version 2 of the PhoneBookEntry Class**

In version 2 of the `PhoneBookEntry` class, you split the `Address` property into `StreetAddress`, `City`, `State`, and `ZipCode` properties.

With these changes, you could not load a version 2 object in a previous release. However, version 2 employs several techniques to enable compatibility:

- Preserve the `Address` property (which is used in version 1) as a `Dependent` property with private `SetAccess`.
- Define an `Address` property get method (`get.Address`) to build a `char` vector that is compatible with the version 2 `Address` property.
- The `saveobj` method invokes the `get.Address` method to assign the object data to a `struct` that is compatible with previous versions. The `struct` continues to have only an `Address` field built from the data in the new `StreetAddress`, `City`, `State`, and `ZipCode` properties.
• When the `loadobj` method sets the `Address` property, it invokes the property set method (`set.Address`), which extracts the substrings required by the `StreetAddress`, `City`, `State`, and `ZipCode` properties.

• The `Transient` (not saved) property `SaveInOldFormat` enables you to specify whether to save the version 2 object as a `struct` or an object.

classdef PhoneBookEntry
    properties
        Name
        StreetAddress
        City
        State
        ZipCode
        PhoneNumber
    end
    properties (Constant)
        Sep = ',', '
    end
    properties (Dependent, SetAccess=private)
        Address
    end
    properties (Transient)
        SaveInOldFormat = false;
    end
    methods (Static)
        function obj = loadobj(s)
            if isstruct(s)
                obj = PhoneBookEntry;
                obj.Name = s.Name;
                obj.Address = s.Address;
                obj.PhoneNumber = s.PhoneNumber;
            else
                obj = s;
            end
        end
    end
    methods
        function address = get.Address(obj)
            address = [obj.StreetAddress, obj.Sep, obj.City, obj.Sep, ...
                        obj.State, obj.Sep, obj.ZipCode];
        end
        function obj = set.Address(obj,address)
            addressItems = regexp(address, obj.Sep, 'split');
            if length(addressItems) == 4
            end
        end
end
obj.StreetAddress = addressItems{1};
obj.City = addressItems{2};
obj.State = addressItems{3};
obj.ZipCode = addressItems{4};
else
    error('PhoneBookEntry:InvalidAddressFormat', ...'
        'Invalid address format.');
end
end
end
function s = saveobj(obj)
    if obj.SaveInOldFormat
        s.Name = obj.Name;
        s.Address = obj.Address;
        s.PhoneNumber = obj.PhoneNumber;
    end
end
end

See Also

Related Examples

• "Modify the Save and Load Process" on page 13-16
• "Object Save and Load"
Initialize Objects When Loading

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Calling Constructor When Loading Objects” on page 13-30</td>
</tr>
<tr>
<td>“Initializing Objects in the loadobj Method” on page 13-30</td>
</tr>
</tbody>
</table>

Calling Constructor When Loading Objects

MATLAB does not call the class constructor when loading an object from a MAT-file. However, if you set the ConstructOnLoad class attribute to true, load does call the constructor with no arguments.

Enable ConstructOnLoad when you do not want to implement a loadobj method, but must perform some actions at construction time. For example, enable ConstructOnLoad when you are registering listeners for another object. Ensure that MATLAB can call the class constructor with no arguments without generating an error.

Attributes set on superclasses are not inherited by subclasses. Therefore, MATLAB does use the value of the superclass ConstructOnLoad attribute when loading objects. If you want MATLAB to call the class constructor, set the ConstructOnLoad attribute in your specific subclass.

If the constructor requires input arguments, use a loadobj method.

Initializing Objects in the loadobj Method

Use a loadobj method when the class constructor requires input arguments to perform object initialization.

The LabResults class shares the constructor object initialization steps with the loadobj method by performing these steps in the assignStatus method.

Objects of the LabResults class:

- Hold values for the results of tests.
- Assign a status for each value based on a set of criteria.

```matlab
classdef LabResult
    properties
```
The `LabResults` class uses `loadobj` to determine the status of a given test value. This approach provides a way to:

- Modify the criteria for determining status
- Ensure that objects always use the current criteria

You do not need to implement a `saveobj` method.
See Also

Related Examples
- “Modify the Save and Load Process” on page 13-16
- “Object Save and Load”
Save and Load Objects from Class Hierarchies

In this section...

“Saving and Loading Subclass Objects” on page 13-33
“Reconstruct the Subclass Object from a Saved struct” on page 13-33

Saving and Loading Subclass Objects

If the most specific class of an object does not define a loadobj or saveobj method, this class can inherit loadobj or saveobj methods from a superclass.

If any class in the hierarchy defines saveobj or loadobj methods:

• Define saveobj for all classes in the hierarchy.
• Call superclass saveobj methods from the subclass saveobj method because the save function calls only the most specific saveobj method.
• The subclass loadobj method can call the superclass loadobj, or other methods as required, to assign values to their properties.

Reconstruct the Subclass Object from a Saved struct

Suppose that you want to save a subclass object by first converting its property data to a struct in the class saveobj method. Then you reconstruct the object when loaded using its loadobj method. This action requires that:

• Superclasses implement saveobj methods to save their property data in the struct.
• The subclass saveobj method calls each superclass saveobj method and returns the completed struct to the save function. Then the save function writes the struct to the MAT-file.
• The subclass loadobj method creates a subclass object and calls superclass methods to assign their property values in the subclass object.
• The subclass loadobj method returns the reconstructed object to the load function, which loads the object into the workspace.

The following superclass (MySuper) and subclass (MySub) definitions show how to code these methods.
• The `MySuper` class defines a `loadobj` method to enable an object of this class to be loaded directly.

• The subclass `loadobj` method calls a method named `reload` after it constructs the subclass object.

• `reload` first calls the superclass `reload` method to assign superclass property values and then assigns the subclass property value.

```matlab
classdef MySuper
    properties
        X
        Y
    end
    methods
        function S = saveobj(obj)
            S.PointX = obj.X;
            S.PointY = obj.Y;
        end
        function obj = reload(obj,S)
            obj.X = S.PointX;
            obj.Y = S.PointY;
        end
    end
    methods (Static)
        function obj = loadobj(S)
            if isstruct(s)
                obj = MySuper;
                obj = reload(obj,S);
            end
        end
    end
end
```

Call the superclass `saveobj` and `loadobj` methods from the subclass `saveobj` and `loadobj` methods.

```matlab
classdef MySub < MySuper
    properties
        Z
    end
    methods
        function S = saveobj(obj)
            S = saveobj@MySuper(obj);
            S.PointZ = obj.Z;
    end
end
```

13-34
function obj = reload(obj,S)
    obj = reload@MySuper(obj,S);
    obj.Z = S.PointZ;
end

methods (Static)
    function obj = loadobj(S)
        if isstruct(s)
            obj = MySub;
            obj = reload(obj,S);
        end
    end
end
end

See Also

Related Examples

• “Modify the Save and Load Process” on page 13-16
• “Object Save and Load”
Restore Listeners

Create Listener with loadobj

Suppose that you create a property listener and want to be able to save and restore the event source and the listener. One approach is to create a listener from the `loadobj` method.

Use Transient Property to Load Listener

The `BankAccount` class stores the account balance and an account status. A `PostSet` listener attached to the `AccountBalance` property controls the account status.

When the `AccountBalance` property value changes, the listener callback determines the account status. Important points include:

- The `BankAccount` class defines the `AccountManagerListener` property to contain the listener handle. This property enables the `loadobj` method to create a listener and return a reference to it in the object that is loaded into the workspace.
- The `AccountManagerListener` property is `Transient` because there is no need to store the listener handle with a `BankAccount` object. Create a listener that is attached to the new `BankAccount` object created during the load process.
- The `AccountBalance` listener sets the `AccountStatus`.
- Only the `AccountManager` class can access `AccountStatus` property.

```matlab
classdef BankAccount < handle
    properties (SetObservable, AbortSet)
        AccountBalance
    end
    properties (Transient)
        AccountManagerListener
    end
    properties (Access = ?AccountManager)
```
AccountStatus
end
methods
function obj = BankAccount(initialBalance)
    obj.AccountBalance = initialBalance;
    obj.AccountStatus = 'New Account';
    obj.AccountManagerListener = AccountManager.addAccount(obj);
end
end
methods (Static)
function obj = loadobj(obj)
    if isstruct(obj) % Handle error
        initialBalance = obj.AccountBalance;
        obj = BankAccount(initialBalance);
    else
        obj.AccountManagerListener = AccountManager.addAccount(obj);
    end
end
end
end

Assume the AccountManager class provides services for various types of accounts. For the BankAccount class, the AccountManager class defines two Static methods:

- assignStatus — Callback for the AccountBalance property PostSet listener. This method determines the value of the BankAccount AccountStatus property.
- addAccount — Creates the AccountBalance property PostSet listener. The BankAccount constructor and loadobj methods call this method.

classdef AccountManager
    methods (Static)
        function assignStatus(BA,~)
            if BA.AccountBalance < 0 && BA.AccountBalance >= -100
                BA.AccountStatus = 'overdrawn';
            elseif BA.AccountBalance < -100
                BA.AccountStatus = 'frozen';
            else
                BA.AccountStatus = 'open';
            end
        end
        function lh = addAccount(BA)
            lh = addlistener(BA,'AccountBalance','PostSet', ... @sourcesamples_EVT_AccountManager.assignStatus(BA));
        end
    end
end
Using the BankAccount and AccountManager Classes

Create an instance of the BankAccount class.

```matlab
ba = BankAccount(100)
ba =

      BankAccount with properties:
                  AccountBalance: 100
          AccountManagerListener: [1x1 event.proplistener]
                  AccountStatus: 'New Account'
```

Now set an account value to confirm that the AccountManager sets AccountStatus appropriately:

```matlab
ba.AccountBalance = -10;
ba.AccountStatus

ans =
      overdrawn
```

See Also

Related Examples

- “Modify the Save and Load Process” on page 13-16
- “Property Attributes” on page 8-9
- “Listen for Changes to Property Values” on page 11-40
- “Object Save and Load”
Save and Load Dynamic Properties

In this section...

“Saving Dynamic Properties” on page 13-39
“When You Need saveobj and loadobj Methods” on page 13-39
“Implementing saveobj and loadobj Methods” on page 13-39

Saving Dynamic Properties

Use the addprop method to add dynamic properties to a class that is derived from the dynamicprops class. The save function saves dynamic properties with the object to which they are attached. For more information on dynamic properties, see “Dynamic Properties — Adding Properties to an Instance” on page 8-68.

When You Need saveobj and loadobj Methods

The save function saves dynamic properties and their values. However, save does not save dynamic property attributes because these attributes are not specified in the class definition. If you save an object that has dynamic properties with nondefault attributes values, use saveobj and loadobj to manage the saving and loading of attribute values.

If the dynamic property has nondefault attribute values, convert the object to a struct in the saveobj method. Save the dynamic property attribute values in the struct so that the loadobj method can restore these values.

Implementing saveobj and loadobj Methods

Your saveobj method can obtain the nondefault attribute values from the meta.DynamicProperty object associated with the dynamic property. Suppose the object that you are saving has a dynamic property called DynoProp. Create a struct in the saveobj method to save the data that the loadobj method uses to reconstruct the object.

Here is how the saveobj method works:

• Obtain the meta.DynamicProperty object for the dynamic property.
• Store the name and value of the dynamic property in structs.
• Store the nondefault dynamic property attributes values for SetAccess and GetAccess in the struct. The `loadobj` function restores these values.

methods

```matlab
function s = saveobj(obj)
    metaDynoProp = findprop(obj,'DynoProp');
    s.dynamicprops(1).name = metaDynoProp.Name;
    s.dynamicprops(1).value = obj.DynoProp;
    s.dynamicprops(1).setAccess = metaDynoProp.SetAccess;
    s.dynamicprops(1).getAccess = metaDynoProp.GetAccess;
    ...
end
end
```

Your `loadobj` method can add the dynamic property and set the attribute values:

• Create an instance of the class.
• Use `addprop` to add a new dynamic property to the object.
• Restore the attributes of the dynamic property.

methods (Static)

```matlab
function obj = loadobj(s)
    if isstruct(s)
        obj = ClassConstructor;
        ...
        metaDynoProp = addprop(obj,s.dynamicprops(1).name);
        obj.(s.dynamicprops(1).name) = s.dynamicprops(1).value;
        metaDynoProp.SetAccess = s.dynamicprops(1).setAccess;
        metaDynoProp.GetAccess = s.dynamicprops(1).getAccess;
    end
end
end
```

See Also

Related Examples

• “Modify the Save and Load Process” on page 13-16
• “Object Save and Load”
Enumerations

• “Named Values” on page 14-2
• “Define Enumeration Classes” on page 14-5
• “Refer to Enumerations” on page 14-11
• “Enumerations for Property Values” on page 14-17
• “Operations on Enumerations” on page 14-19
• “Enumeration Class Restrictions” on page 14-27
• “Enumerations Derived from Built-In Types” on page 14-28
• “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-34
• “Enumerations That Encapsulate Data” on page 14-41
• “Save and Load Enumerations” on page 14-45
Named Values

In this section...

“Kinds of Predefined Names” on page 14-2
“Techniques for Defining Enumerations” on page 14-3

Kinds of Predefined Names

MATLAB supports two kinds of predefined names:

• Constant properties
• Enumerations

Constant Properties

Use constant properties when you want a collection of related constant values whose values can belong to different types (numeric values, character strings, and so on). Define properties with constant values by setting the property Constant attribute. Reference constant properties by name whenever you need access to that particular value.

See “Define Class Properties with Constant Values” on page 15-2 for more information.

Enumerations

Use enumerations when you want to create a fixed set of names representing a single type of value. Use this new type in multiple places without redefining it for each class.

You can derive enumeration classes from other classes to inherit the operations of the superclass. For example, if you define an enumeration class that subclasses a MATLAB numeric class like double or int32, the enumeration class inherits all the mathematical and relational operations that MATLAB defines for those classes.

Using enumerations instead of character strings to represent a value, such as colors ('red'), can result in more readable code because:

• You can compare enumeration members with == instead of using strcmp
• Enumerations maintain type information, char vectors do not. For example, passing a char vector 'red' to functions means that every function must interpret what 'red' means. If you define red as an enumeration, the actual value of 'red' can change
Define enumerations by creating an enumeration block in the class definition.

See “Define Enumeration Classes” on page 14-5 for more information.

**Techniques for Defining Enumerations**

Enumerations enable you to define names that represent entities useful to your application, without using numeric values or character strings. All enumerations support equality and inequality operations. Therefore, `switch`, `if`, and several comparison functions like `isequal` and `ismember` work with enumeration members.

You can define enumeration classes in ways that are most useful to your application, as described in the following sections.

**Simple Enumerated Names**

Simple enumeration classes have no superclasses and no properties. These classes define a set of related names that have no underlying values associated with them. Use this kind of enumeration when you want descriptive names, but your application does not require specific information associated with the name.

See the `WeekDays` class in the “Enumeration Class” on page 14-5 and the “Define Methods in Enumeration Classes” on page 14-7 sections.

**Enumerations with Built-In Class Behaviors**

Enumeration classes that subclass MATLAB built-in classes inherit most of the behaviors of those classes. For example, an enumeration class derived from the `double` class inherits the mathematical, relational, and set operations that work with variables of the class.

Enumerations do not support the colon (:) operator, even if the superclass does.

**Enumerations with Properties for Member Data**

Enumeration classes that do not subclass MATLAB built-in numeric and logical classes can define properties. These classes can define constructors that set each member's unique property values.
The constructor can save input arguments in property values. For example, a Color class can specify a Red enumeration member color with three (Red, Green, Blue) values:

```
enumeration
   Red (1,0,0)
end
```

See Also

Related Examples

- “Enumeration Class Restrictions” on page 14-27
- “Enumerations Derived from Built-In Types” on page 14-28
- “Enumerations That Encapsulate Data” on page 14-41
Define Enumeration Classes

**Enumeration Class**

Create an enumeration class by adding an enumeration block to a class definition. For example, the `WeekDays` class enumerates a set of days of the week.

```matlab
classdef WeekDays
    enumeration
        Monday, Tuesday, Wednesday, Thursday, Friday
    end
end
```

To execute the MATLAB code in the following sections, place the `WeekDays` class definition in a `.m` file on your path.

**Construct an Enumeration Member**

Refer to an enumeration member using the class name and the member name:

```matlab
ClassName.MemberName
```

For example, assign the enumeration member `WeekDays.Tuesday` to the variable `today`:

```matlab
today = WeekDays.Tuesday;
today
```

today is a variable of class `WeekDays`:

```matlab
whos
```
Convert to Superclass Value

If an enumeration class specifies a superclass, convert an enumeration object to the superclass by passing the object to the superclass constructor. However, the superclass constructor must be able to accept its own class as input and return an instance of the superclass. MATLAB built-in numeric classes, such as `uint32`, allow this conversion.

For example, the `Bearing` class derives from the `uint32` built-in class:

```matlab
classdef Bearing < uint32
    enumeration
        North (0)
        East  (90)
        South (180)
        West  (270)
    end
end
```

Assign the `Bearing.East` member to the variable `a`:

```matlab
a = Bearing.East;
```

Pass `a` to the superclass constructor and return a `uint32` value:

```matlab
b = uint32(a);
```

The `uint32` constructor accepts an object of the subclass `Bearing` and returns and object of class `uint32`. 
Define Methods in Enumeration Classes

Define methods in an enumeration class like any MATLAB class. For example, here is the WeekDays class with a method called isMeetingDay added:

```matlab
classdef WeekDays
enumeration
    Monday, Tuesday, Wednesday, Thursday, Friday
end
methods
    function tf = isMeetingDay(obj)
        tf = ~(WeekDays.Tuesday == obj);
    end
end
end
```

Call isMeetingDay with an instance of the WeekDays class:

```matlab
today = WeekDays.Tuesday;
today.isMeetingDay
```

```matlab
ans =
    0
```

Use the enumeration member directly as input to the method:

```matlab
isMeetingDay(WeekDays.Wednesday)
```

```matlab
ans =
    1
```

Define Properties in Enumeration Classes

Add properties to an enumeration class when you must store data related to the enumeration members. Set the property values in the class constructor. For example, the SyntaxColors class defines three properties whose values the constructor assigns to the values of the input arguments when you reference a class member.

```matlab
classdef SyntaxColors
    properties
        R
        G
end
```
When you refer to an enumeration member, the constructor initializes the property values:

e = SyntaxColors.Error;
e.R

ans =

1

Because SyntaxColors is a value class (it does not derive from handle), only the class constructor can set property values:

e.R = 0

You cannot set the read-only property 'R' of SyntaxColors.

For more information on enumeration classes that define properties, see “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-34.

**Enumeration Class Constructor Calling Sequence**

Each statement in an enumeration block is the name of an enumeration member, optionally followed by an argument list. If the enumeration class defines a constructor, MATLAB calls the constructor to create the enumerated instances.

MATLAB provides a default constructor for all enumeration classes that do not explicitly define a constructor. The default constructor creates an instance of the enumeration class:
• Using no input arguments, if the enumeration member defines no input arguments
• Using the input arguments defined in the enumeration class for that member

For example, the input arguments for the `Bool` class are 0 for `Bool.No` and 1 for `Bool.Yes`.

```matlab
classdef Bool < logical
    enumeration
        No (0)
        Yes (1)
    end
end
```

The values of 0 and 1 are of class `logical` because the default constructor passes the argument to the first superclass. That is, this statement:

```matlab
n = Bool.No;
```

Results in a call to `logical` that is equivalent to the following statement in a constructor:

```matlab
function obj = Bool(val)
    obj@logical(val)
end
```

MATLAB passes the member argument only to the first superclass. For example, suppose `Bool` derived from another class:

```matlab
classdef Bool < logical & MyBool
    enumeration
        No (0)
        Yes (1)
    end
end
```

The `MyBool` class can add some specialized behavior:

```matlab
classdef MyBool
    methods
        function boolValues = testBools(obj)
            ...
        end
    end
end
```

The default `Bool` constructor behaves as if defined like this function:
• Argument passed to first superclass constructor
• No arguments passed to subsequent constructors

```matlab
function obj = Bool(val)
    obj@logical(val)
    obj@MyBool
end
```

See Also

Related Examples
• “Refer to Enumerations” on page 14-11
• “Operations on Enumerations” on page 14-19
Refer to Enumerations

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Instances of Enumeration Classes” on page 14-11</td>
</tr>
<tr>
<td>“Conversion of Characters to Enumerations” on page 14-13</td>
</tr>
<tr>
<td>“Enumeration Arrays” on page 14-15</td>
</tr>
</tbody>
</table>

Instances of Enumeration Classes

Enumeration members are instances of the enumeration class. You can assign enumeration members to variables and form arrays of enumeration members. If an enumeration class derives from a superclass, you can substitute an enumeration member for an instance of the superclass.

The WeekDays class defines enumeration members for five days of the week.

classdef WeekDays
  enumeration
    Monday, Tuesday, Wednesday, Thursday, Friday
  end
end

Create objects of the WeekDays class representing specific days.

today = WeekDays.Monday;
tomorrow = WeekDays.Tuesday;

The variables today and tomorrow are objects of the WeekDays class.

The PPM class defines three enumeration members. Each member has an associated numeric value derived from the class superclass.

classdef PPM < double
  enumeration
    High (1000)
    Medium (100)
    Low (10)
  end
end

Assign an enumeration member to a variable.
level = PPM.High;

When you substitute enumeration members for instances of the superclass, MATLAB coerces the enumeration member to the superclass. For example, add an enumeration member of the PPM class with a numeric values

levelNew = level + 100

levelNew =

    1100

The result is of class double.

whos

Name          Size            Bytes  Class    Attributes
level         1x1               108  PPM
levelNew      1x1                 8  double

You can substitute superclass values for enumeration members when the values correspond. For example, pass one of the numeric values defined in the enumeration class to the PPMSwitch function.

function PPMSwitch(ppm)
    switch ppm
        case PPM.Low
            disp Low
        case PPM.Medium
            disp Medium
        case PPM.High
            disp High
    end
end

PPMSwitch(100)

Medium

You can also use an enumeration member directly:

PPMSwitch(PPM.Medium)

Medium
For information on operations you can perform on enumeration class instances, see “Operations on Enumerations” on page 14-19.

**Conversion of Characters to Enumerations**

Enumeration classes can convert char vectors to enumeration members when the char vector represents an enumeration member defined by the class. This conversion enables you to pass a valid char vector or a cell array of char vectors when enumerations are expected.

Use a char vector instead of a direct reference to an enumeration member when you want to use a simple character strings to specify an enumeration member. However, specifying an enumeration member directly eliminates the conversion from char to enumeration.

Enumeration classes provide a converter function using the constructor syntax.

```matlab
today = WeekDays('Tuesday');
```

Because the char vector 'Tuesday' matches the enumeration member WeekDays.Tuesday, the Weekdays char method can perform the conversion.

```matlab
class(today)
anst =
WeekDays
```

Create an enumeration array using the WeekDay class constructor and a cell array of char vectors.

```matlab
wd = WeekDays({'Monday','Wednesday','Friday'})
```

```matlab
wd =
    Monday       Wednesday    Friday
```

```matlab
class(wd)
anst =
WeekDays
```
All char vectors in the cell array must correspond to an enumeration member defined by the class.

**Coercion of char to Enumerations**

MATLAB coerces char vectors into enumerations members when the dominant argument is an enumeration. Because user-defined classes are dominant over the char class, MATLAB attempts to convert the char vector to a member of the enumeration class.

Create an enumeration array. Then insert a char vector that represents an enumeration member into the array.

```matlab
a = [WeekDays.Monday,WeekDays.Wednesday,WeekDays.Friday]
a =
    Monday       Wednesday    Friday
```

Add a char vector to the WeekDays array.

```matlab
a(end+1) = 'Tuesday'
a =
    Monday       Wednesday    Friday       Tuesday
```

MATLAB coerces the char vector to a WeekDays enumeration member.

```matlab
class(a)
ans =
    WeekDays
```

**Substitute Enumeration Members for char Vectors**

You can use enumeration members in place of char vectors in cases where functions require char vectors. For example, this call to sprintf expects a char vector, designated by the %s format specifier.

```matlab
sprintf('Today is %s',WeekDays.Friday)
ans =
    Today is Friday
```
The automatic conversion of enumeration classes to `char` enable you to use enumeration members in this case.

**Enumeration Arrays**

Create enumeration arrays by:

- Concatenating enumeration members using `[]`
- Assigning enumeration members to an array using indexed assignment

Create an enumeration array of class `WeekDays` by concatenating enumeration members:

```matlab
wd = [WeekDays.Tuesday,WeekDays.Wednesday,WeekDays.Friday];
```

Create an enumeration array of class `WeekDays` by indexed assignment:

```matlab
a(1) = WeekDays.Tuesday;
a(2) = WeekDays.Wednesday;
a(3) = WeekDays.Friday;
```

**Mixed Enumeration Members and char Vectors**

You can concatenate enumeration members and `char` vectors as long as the `char` vector represents an enumeration member.

```matlab
clear a
a = [WeekDays.Wednesday,'Friday'];
class(a)
```

```matlab
ans =
WeekDays
```

You can also assign a `char` vector to an enumeration array:

```matlab
clear a
a(1) = WeekDays.Wednesday;
a(2) = 'Friday';
class(a)
```

```matlab
ans =
WeekDays
```
**Default Enumeration Member**

The default member an enumeration class is the first enumeration member defined in the enumeration block. For the `WeekDays` class, the default enumeration member is `WeekDays.Monday`.

```matlab
classdef WeekDays
    enumeration
        Monday, Tuesday, Wednesday, Thursday, Friday
    end
end
```

MATLAB allows assignment to any element of an array, even if the array variable does not previously exist. To fill in unassigned array elements, MATLAB uses the default enumeration member.

For example, assign a value to element 5 of an array, `a`:

```matlab
clear a
a(5) = WeekDays.Tuesday;
```

MATLAB must initialize the values of array elements `a(1:4)` with the default enumeration member. The result of the assignment to the fifth element of the array `a` is:

```matlab
a
```

```
Monday    Monday    Monday    Monday    Tuesday
```

**See Also**

**Related Examples**

- “Operations on Enumerations” on page 14-19
Enumerations for Property Values

Syntax for Property/Enumeration Definition

You can restrict the values that are allowed for a property to members of an enumeration class. Define the property as restricted to a specific enumeration class in the class definition using this syntax:

```matlab
properties
  PropName EnumerationClass
end
```

This syntax restricts values of `PropName` to members of the enumeration class `EnumerationClass`.

Example of Restricted Property

For example, the `Days` class defines a property named `Today`. The allowed values for the `Today` property are enumeration members of the `WeekDays` class.

The `WeekDays` class defines the enumerations:

```matlab
classdef WeekDays
    enumeration
      Monday, Tuesday, Wednesday, Thursday, Friday
    end
end
```

Use the `WeekDays` enumerations to restrict the allowed values of the `Today` property:

```matlab
classdef Days
    properties
      Today WeekDays
    end
end
```

Create an object of the `Days` class.
d = Days;
d.Today = WeekDays.Tuesday;

d =

Days with properties:

Today: Tuesday

**Representing Enumeration Members with char Vectors**

The automatic conversion feature enables users of the Days class to assign values to the Today property as either enumeration members, char vectors, or string scalars. The Today property is restricted to members of the WeekDays enumeration class. Therefore, you can assign a char vector that represents a member of the WeekDays class.

\[
d = Days;
d.Today = 'Tuesday';
\]

Also, you can use a string scalar:

\[
d = Days;
d.Today = "Tuesday";
\]

For more information on restricting property values, see “Validate Property Values” on page 8-24 and “Property Class and Size Validation” on page 8-31.
Operations on Enumerations

In this section...

“Operations Supported by Enumerations” on page 14-19
“Enumeration Class” on page 14-19
“Default Methods” on page 14-20
“Convert Enumeration Member to Characters” on page 14-20
“Convert Enumeration Array to Cell Array of char Vectors” on page 14-20
“Enumerations and char Vectors in Relational Operations” on page 14-21
“Enumerations in switch Statements” on page 14-22
“Enumeration Set Membership” on page 14-23
“Enumeration Text Comparison Methods” on page 14-24
“How to Get Information About Enumerations” on page 14-25
“Testing for an Enumeration” on page 14-25

Operations Supported by Enumerations

You can use logical, set membership, and string comparison operations on enumerations. These operations also allow the use of enumeration in conditional statements, such as switch and if statements. Converters enable you to use char vectors and cell arrays of strings as enumerations.

Enumeration Class

The WeekDays class defines members that enumerate days of the week. This topic uses the WeekDays class to illustrate how to perform operations on enumerations.

classdef WeekDays
    enumeration
        Monday, Tuesday, Wednesday, Thursday, Friday
    end
end

For information on defining enumerations, see “Define Enumeration Classes” on page 14-5.
**Default Methods**

Enumeration classes have the following default methods:

```matlab
methods('WeekDays')
```

Methods for class WeekDays:

<table>
<thead>
<tr>
<th>WeekDays</th>
<th>char</th>
<th>intersect</th>
<th>ne</th>
<th>setxor</th>
<th>strcmpi</th>
<th>strncmp</th>
<th>union</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellstr</td>
<td>eq</td>
<td>ismember</td>
<td>setdiff</td>
<td>strcmp</td>
<td>string</td>
<td>strncmpi</td>
<td></td>
</tr>
</tbody>
</table>

The `WeekDays` method converts char vectors, a cell array of char vectors, or string array elements to enumerations.

Other methods behave similarly to the equivalent function when used with enumerations. For information on a specific method, see the documentation for that function.

**Convert Enumeration Member to Characters**

Conversion to char is useful because you can define enumeration members with descriptive names. For example:

```matlab
today = WeekDays.Friday;
['Today is ',char(today)]
```

ans =

Today is Friday

**Convert Enumeration Array to Cell Array of char Vectors**

Use `cellstr` to convert an enumeration array to a cell array of char vectors.

```matlab
ca = cellstr({'WeekDays.Tuesday','WeekDays.Thursday'});
class(ca)
```

ans =

cell

Both cells in the cell array contain char vectors:

```matlab
class([ca{1:2}])
```
Expressions involving the relational operators, `eq` and `ne`, in which one operand is an enumeration allow the other operand to be of type `char`. Before performing the operation, MATLAB converts `char` vectors to scalar enumerations or cell arrays of `char` vectors to enumeration arrays.

**Note**  
Enumeration classes that derive from MATLAB built-in classes cannot substitute `char` vectors for enumeration members.

```matlab
today = WeekDays.Friday;
today == 'Friday'
ans =
    1

Compare enumeration array to `char` vector:
```
wd = [WeekDays.Monday,WeekDays.Wednesday,WeekDays.Friday];
wd == 'Friday'
ans =
    0     0     1
```

Compare enumeration array to cell array of `char` vectors:
```
cv = {'Monday','Wednesday','Friday'};
md = [WeekDays.Tuesday,WeekDays.Thursday,WeekDays.Friday];
md ~= cv
ans =
    1     1     0
```

Test equality to implement if statements. The `char` vector `Wednesday` is equal to `==` the enumeration member `WeekDays.Wednesday`:
today = 'Wednesday';
...
if today == WeekDays.Wednesday
    disp('Team meeting at 2:00')
end

Enumerations in switch Statements

Equality (eq) and inequality (ne) methods enable you to use enumeration members in switch statements. For example, using the WeekDays class defined previously, construct a switch statement:

```matlab
function c = Reminder(day)
    % Add error checking here
    switch(day)
    case WeekDays.Monday
        c = 'Department meeting at 10:00';
    case WeekDays.Tuesday
        c = 'Meeting Free Day!';
    case {WeekDays.Wednesday WeekDays.Friday}
        c = 'Team meeting at 2:00';
    case WeekDays.Thursday
        c = 'Volleyball night';
    end
end
```

Pass a member of the WeekDays enumeration class to the Reminder function:

today = WeekDays.Wednesday;
Reminder(today)

ans =

Team meeting at 2:00

For more information, see “Objects In Conditional Statements” on page 5-37.

Substitute char Vectors

**Note** Enumeration classes that derive from MATLAB built-in classes cannot substitute char vectors for enumeration members.
You can use `char` vectors to represent specific enumeration members:

```matlab
function c = Reminder2(day)
    switch(day)
    case 'Monday'
        c = 'Department meeting at 10:00';
    case 'Tuesday'
        c = 'Meeting Free Day!';
    case {'Wednesday' 'Friday'}
        c = 'Team meeting at 2:00';
    case 'Thursday'
        c = 'Volleyball night';
    end
end
```

Although you can use `char` vectors instead of specifying enumerations explicitly, MATLAB must convert the `char` to an enumeration. Eliminate the need for this conversion if it is not necessary.

### Enumeration Set Membership

Enumeration classes provide methods to determine set membership.

- `ismember` — True for elements of an enumeration array if in a set
- `setdiff` — Set difference for enumeration arrays
- `intersect` — Set intersection for enumeration arrays
- `setxor` — Set exclusive-or for enumeration arrays
- `union` — Set union for enumeration arrays

Determine if today is a meeting day for your team. Create a set of enumeration members corresponding to the days on which the team has meetings.

```matlab
today = WeekDays.Tuesday;
teamMeetings = [WeekDays.Wednesday WeekDays.Friday];
```

Use `ismember` to determine if `today` is part of the `teamMeetings` set:

```matlab
ismember(today,teamMeetings)
```

```matlab
ans =
0
```
**Mixed Sets of Enumeration and char**

If you pass both enumeration and char arguments to an enumeration class method, the class attempts to convert the char to the class of the enumeration.

Determine if char vector is a member of the enumeration array.

```matlab
teamMeetings = [WeekDays.Wednesday WeekDays.Friday];
ismember('Friday',teamMeetings)
```

```matlab
ans =
1
```

Determine if the enumeration member is a member of the cell array of char vectors.

```matlab
ismember(WeekDays.Friday,{'Wednesday','Friday'})
```

```matlab
ans =
1
```

**Enumeration Text Comparison Methods**

Enumeration classes provide methods to compare enumeration members with char vectors. One of the arguments to the string comparison method must be a char vector. Comparing two enumeration members returns false.

- `strcmp` — Compare enumeration members
- `strncmp` — Compare first n characters of enumeration members
- `strcmpi` — Case insensitive comparison of enumeration members
- `strncmpi` — Case insensitive first n character comparison of enumeration members

**Comparing Enumeration Member with char Vector**

The string comparison methods can compare enumeration members and char vectors.

```matlab
today = WeekDays.Tuesday;
strcmp(today,'Friday')
```

```matlab
ans =
0
```
strcmp(today,'Tuesday')
ans =
   1

**How to Get Information About Enumerations**

Obtain information about enumeration classes using the `enumeration` function. For example:

```matlab
enumeration WeekDays

Enumeration members for class 'WeekDays':
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
```

See also “Metaclass EnumeratedValues Property” on page 16-8

**Testing for an Enumeration**

To determine if a variable is an enumeration, use the `isenum` function. For example:

```matlab
today = WeekDays.Wednesday;
isenum(today)
ans =
   1

isenum returns true for empty enumeration objects:

```matlab
noday = WeekDays.empty;
isenum(noday)
ans =
   1
```
To determine if the class of a variable class is an enumeration class, use the `meta.class` object.

today = WeekDays.Wednesday;
mc = metaclass(today);
mc.Enumeration

ans =
  1

**See Also**

**Related Examples**

- “Enumeration Class Restrictions” on page 14-27
Enumeration Class Restrictions

Enumeration classes restrict certain aspects of their use and definition:

- Enumeration classes are implicitly **Sealed**. You cannot define a subclass of an enumeration class because doing so would expand the set.

- The properties of value-based enumeration classes are immutable. Only the constructor can assign property values. MATLAB implicitly defines the `SetAccess` attributes of all properties defined by value-based enumeration classes as immutable. You cannot set the `SetAccess` attribute to any other value.

- All properties inherited by a value-based enumeration class that are not defined as `Constant` must have immutable `SetAccess`.

- The properties of handle-based enumeration classes are mutable. You can set property values on instances of the enumeration class. See “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-34.

- An enumeration member cannot have the same name as a property, method, or event defined by the same class.

- Enumerations do not support colon (a:b) operations. For example, `FlowRate.Low:FlowRate.High` causes an error even if the `FlowRate` class derives from a numeric superclass.

- Classes that define enumerations cannot restrict properties of the same class to an enumeration type. You must create a separate enumeration class to restrict property values to an enumeration. For information on restricting property values, see “Example of Restricted Property” on page 14-17.

See Also

Related Examples

- “Enumerations Derived from Built-In Types” on page 14-28
Enumerations Derived from Built-In Types

In this section...

- “Subclassing Built-In Types” on page 14-28
- “Derive Enumeration Class from Numeric Type” on page 14-28
- “How to Alias Enumeration Names” on page 14-30
- “Superclass Constructor Returns Underlying Value” on page 14-31
- “Default Converter” on page 14-32

Subclassing Built-In Types

Enumeration classes can subclass MATLAB built-in classes. Deriving an enumeration class from a built-in classes is useful to extend the usefulness of the enumeration members.

- Enumeration inherit functionality from the built-in class.
- You can associate a numeric or logical value with enumeration members.

For a more basic discussion of enumeration classes, see “Define Enumeration Classes” on page 14-5.

Derive Enumeration Class from Numeric Type

Note: Enumeration classes derived from built-in numeric and logical classes cannot define properties.

If an enumeration class subclasses a built-in numeric class, the subclass inherits ordering and arithmetic operations that you can apply to the enumerated names.

For example, the Results class subclasses the int32 built-in class. This class associates an integer value with each of the four enumeration members — First, Second, Third, and NoPoints.

```matlab
classdef Results < int32	enumeration
    First   (100)
```
Second   (50)
Third    (10)
NoPlace (0)

The enumeration member inherits the methods of the int32 class (except the colon operator). Use these enumerations like numeric values (summed, sorted, averaged).

isa(Results.Second,'int32')
ans =
    1

For example, use enumeration names instead of numbers to rank two teams:

Team1 = [Results.First, Results.NoPlace, Results.Third, Results.Second];
Team2 = [Results.Second, Results.Third, Results.First, Results.First];

Perform int32 operations on these Results enumerations:

sum(Team1)
ans =
    160

mean(Team1)
ans =
    40

sort(Team2,'descend')
ans =
    First    First    Second    Third

Team1 > Team2
ans =
    1     0     0     0

sum(Team1) < sum(Team2)
How to Create Enumeration Instances

When you first refer to an enumeration class that derives from a built-in class such as, `int32`, MATLAB passes the input arguments associated with the enumeration members to the superclass constructor. For example, referencing the `Second Results` member, defined as:

`Second (50)`

means that MATLAB calls:

`int32(50)`

to initialize the `int32` aspect of this `Results` object.

How to Alias Enumeration Names

Enumeration classes that derive from MATLAB built-in numeric and logical classes can define more than one name for an underlying value. The first name in the enumeration block with a given underlying value is the actual name for that underlying value and subsequent names are aliases.

Specify aliased names with the same superclass constructor argument as the actual name:

```matlab
classdef Bool < logical
    enumeration
        No  (0)
        Yes (1)
        off (0)
        on  (1)
    end
end
```

For example, the actual name of an instance of the `Bool.off` enumeration member is `No`:

```matlab
a = Bool.No
```
Superclass Constructor Returns Underlying Value

The actual underlying value associated with an enumeration member is the value returned by the built-in superclass. For example, consider the `Bool` class defined with constructor arguments that are of class `double`:

```matlab
classdef Bool < logical
    enumeration
        No (0)
        Yes (100)
    end
end
```

This class derives from the built-in `logical` class. Therefore, underlying values for an enumeration member depend only on what value `logical` returns when passed that value:

```matlab
a = Bool.Yes
a =
    Yes
logical(a)
an =
    1
```

How to Subclass Numeric Built-In Classes

The `FlowRate` enumeration class defines three members, `Low`, `Medium`, and `High`.

```matlab
classdef FlowRate < int32
    enumeration
```
Reference an instance of an enumeration member:

```matlab
setFlow = FlowRate.Medium;
```

This statement causes MATLAB to call the default constructor with the argument value of 50. MATLAB passes this argument to the first superclass constructor (int32(50) in this case). The result is an underlying value of 50 as a 32-bit integer for the FlowRate.Medium member.

Because FlowRate subclasses a built-in numeric class (int32), this class cannot define properties. However FlowRate inherits int32 methods including a converter method. Programs can use the converter to obtain the underlying value:

```matlab
setFlow = FlowRate.Medium;
int32(setFlow)

ans =

  50
```

**Default Converter**

If an enumeration is a subclass of a built-in numeric class, you can convert from built-in numeric data to the enumeration using the name of the enumeration class. For example:

```matlab
a = Bool(1)
a =

Yes
```

An enumerated class also accepts enumeration members of its own class as input arguments:

```matlab
Bool(a)
```
ans =
   Yes

The converter returns an object of the same size as in input:

`Bool([0,1])`

ans =
   No     Yes

Create an empty enumeration array using the `empty` static method:

`Bool.empty`

ans =
   0x0 empty Boolean enumeration.

**See Also**

**Related Examples**
- “Mutable Handle vs. Immutable Value Enumeration Members” on page 14-34
- “Fundamental MATLAB Classes”
Mutable Handle vs. Immutable Value Enumeration Members

In this section...

"Select Handle- or Value-Based Enumerations" on page 14-34
"Value-Based Enumeration Classes" on page 14-34
"Handle-Based Enumeration Classes" on page 14-36
"Represent State with Enumerations" on page 14-39

Select Handle- or Value-Based Enumerations

Use a handle enumeration to enumerate a set of objects whose state can change over time. Use a value enumeration to enumerate a set of abstract (and immutable) values. For information about handle and value classes, see “Comparison of Handle and Value Classes” on page 7-2.

Value-Based Enumeration Classes

A value-based enumeration class has a fixed set of specific values. Modify these values by changing the values of properties. Doing so expands or changes the fixed set of values for this enumeration class.

Inherited Property SetAccess Must Be Immutable

Value-based enumeration classes implicitly define the SetAccess attributes of all properties as immutable. You cannot set the SetAccess attribute to any other value.

However, all superclass properties must explicitly define property SetAccess as immutable.

Enumeration Members Remain Constant

An instance of a value-based enumeration class is unique until the class is cleared and reloaded. For example, given this class:

```matlab
classdef WeekDays
    enumeration
        Monday, Tuesday, Wednesday, Thursday, Friday
```
MATLAB considers \( a \) and \( b \) as equivalent:

\[
a = \text{WeekDays.Monday}; \\
b = \text{WeekDays.Monday}; \\
\text{isequal}(a,b)
\]

\[
\text{ans} = \\
1
\]

\[
a == b \\
\text{ans} = \\
1
\]

**Enumeration Member Properties Remain Constant**

Value-based enumeration classes that define properties are immutable. For example, the `Colors` enumeration class associates RGB values with color names.

```matlab
classdef Colors 
    properties 
        R = 0 
        G = 0 
        B = 0 
    end 
    methods 
        function c = Colors(r,g,b) 
            c.R = r; c.G = g; c.B = b; 
        end 
    end 
    enumeration 
        Red   (1, 0, 0) 
        Green (0, 1, 0) 
        Blue  (0, 0, 1) 
    end 
end
```

The constructor assigns the input arguments to \( R \), \( G \), and \( B \) properties:

```matlab
red = Colors.Red; 
```
You cannot change a property value:

red.G = 1;

You cannot set the read-only property 'G' of Colors.

Handle-Based Enumeration Classes

Handle-based enumeration classes that define properties are mutable. Derive enumeration classes from the handle class when you must be able to change property values on instances of that class.

Note You cannot derive an enumeration class from matlab.mixin.Copyable because the number of instances you can create are limited to the ones defined inside the enumeration block.

An Enumeration Member Remains Constant

Given a handle-based enumeration class with properties, changing the property value of an instance causes all references to that instance to reflect the changed value.

For example, the HandleColors enumeration class associates RGB values with color names, the same as the Colors class in the previous example. However, HandleColors derives from handle:

```matlab
classdef HandleColors < handle
    properties
        R = 0
        G = 0
        B = 0
    end
    methods
        function c = HandleColors(r, g, b)
            c.R = r; c.G = g; c.B = b;
        end
    end
end
```

```plaintext
ans =
    1     0     0
```
Create an instance of HandleColors.Red and return the value of the R property:

```matlab
a = HandleColors.Red;
a.R
ans =
    1
```

MATLAB constructs the `HandleColors.Red` enumeration member, which sets the R property to 1, the G property to 0, and the B property to 0.

Change the value of the R property to 0.8:

```matlab
a.R = 0.8;
```

After setting the value of the R property to 0.8, create another instance, b, of `HandleColors.Red`:

```matlab
b = HandleColors.Red;
b.R
ans =
    0.8000
```

The value of the R property of the newly created instance is also 0.8. A MATLAB session has only one value for any enumeration member at any given time.

Clearing the workspace variables does not change the current definition of the enumeration member `HandleColors.Red`:

```matlab
clear
a = HandleColors.Red;
a.R
ans =
    0.8000
```
Clear the class to reload the definition of the HandleColors class:

```matlab
clear classes
a = HandleColors.Red;
a.R

ans =
    1
```

To prevent reassignment of a given property value, set that property's SetAccess attribute to immutable.

**Equality of Handle-Based Enumerations**

Assign two variables to a particular enumeration member:

```matlab
a = HandleColors.Red;
b = HandleColors.Red;
```

Compare `a` and `b` using `isequal`:

```matlab
isequal(a,b)

ans =
    1
```

The property values of `a` and `b` are the same, so `isequal` returns `true`. However, unlike handle classes that are not enumeration classes, `a` and `b` are the same handle because there is only one enumeration member. Determine handle equality using `==` (the handle `eq` method).

```matlab
a == b

ans =
    1
```

See the handle `eq` method for information on how `isequal` and `==` differ when used with handles.
Represent State with Enumerations

The **MachineState** class defines two enumeration members to represent the state of a machine, either running or not running.

```matlab
classdef MachineState
    enumeration
        Running
        NotRunning
    end
end
```

The **Machine** class represents a machine with start and stop operations. The **MachineState** enumerations are easy to work with because of their `eq` and `char` methods, and they result in code that is easy to read.

```matlab
classdef Machine < handle
    properties (SetAccess = private)
        State = MachineState.NotRunning
    end

    methods
        function start(machine)
            if machine.State == MachineState.NotRunning
                machine.State = MachineState.Running;
            end
            disp (machine.State.char)
        end
        function stop(machine)
            if machine.State == MachineState.Running
                machine.State = MachineState.NotRunning;
            end
            disp (machine.State.char)
        end
    end
end
```

Create a **Machine** object and call `start` and `stop` methods

```matlab
m = Machine;
m.start
Running
m.stop
```
See Also

Related Examples
• “Enumerations That Encapsulate Data” on page 14-41
Enumerations That Encapsulate Data

In this section...

Enumeration Classes with Properties

“Enumeration Classes with Properties” on page 14-41
“Store Data in Properties” on page 14-41

Enumeration Classes with Properties

Enumeration classes can define properties to store data values. The enumeration members represent specific values for these properties, which MATLAB assigns in the class constructor. For information on defining enumeration classes, see “Define Enumeration Classes” on page 14-5.

Store Data in Properties

Note

Enumeration classes that subclass built-in numeric or logical classes cannot define or inherit properties. For more information on this kind of enumeration class, see “Enumerations Derived from Built-In Types” on page 14-28.

Define properties in an enumeration class if you want to associate specific data with enumeration members, but do not need to inherit arithmetic, ordering, or other operations that MATLAB defines for specific built-in classes.

Representing Colors

Define an enumeration class to represent the RGB values of the colors in a color set. The Colors class defines names for the colors, each of which uses the RGB values as arguments to the class constructor:

classdef Colors
    properties
        R = 0
        G = 0
        B = 0
    end
    methods
        function c = Colors(r, g, b)
            c.R = r; c.G = g; c.B = b;
    end
end

14-41
You can access the property values via the enumeration member:

```matlab
Colors.Reddish.R
```

ans =

```matlab
0.9294
```

Suppose that you want to create a plot with the new shade of red named Reddish:

```matlab
a = Colors.Reddish;
```

ans =

```matlab
0.9294   0.1412   0.1490
```

Use these values by accessing the enumeration member properties. For example, the `myPlot` function accepts a `Colors` enumeration member as an input argument. The function accesses the RGB values defining the color from the property values.

```matlab
function h = myPlot(x,y,LineColor)
    h = line('XData',x,'YData',y);
    r = LineColor.R;
    g = LineColor.G;
    b = LineColor.B;
    h.Color = [r g b];
end
```

Create a plot using a reddish color line:

```matlab
h = myPlot(1:10,1:10,Colors.Reddish);
```
The `Colors` class encapsulates the definitions of a standard set of colors. You can change the enumeration class definition of the colors and not affect functions that use the enumerations.

**Enumerations Defining Categories**

The `Cars` class defines categories used to inventory automobiles. The `Cars` class derives from the `CarPainter` class, which derives from `handle`. The abstract `CarPainter` class defines a `paint` method, which modifies the `Color` property when a car is painted another color.

The `Cars` class uses the `Colors` enumeration members to specify a finite set of available colors. The exact definition of any given color can change independently of the `Cars` class.

```matlab
classdef Cars < CarPainter
    enumeration
        Hybrid (2,'Manual',55,Colors.Reddish)
        Compact(4,'Manual',32,Colors.Greenish)
        MiniVan(6,'Automatic',24,Colors.Blueish)
        SUV (8,'Automatic',12,Colors.Yellowish)
    end
    properties  (SetAccess = private)
        Cylinders
        Transmission
        MPG
        Color
    end
    methods
        function obj = Cars(cyl,trans,mpg,colr)
            obj.Cylinders = cyl;
            obj.Transmission = trans;
            obj.MPG = mpg;
            obj.Color = colr;
        end
        function paint(obj,colorobj)
            if isa(colorobj,'Colors')
                obj.Color = colorobj;
            else
                [~,cls] = enumeration('Colors');
                disp('Not an available color')
                disp(cls)
            end
        end
    end
```
The `CarPainter` class requires its subclasses to define a method called `paint`:

```matlab
classdef CarPainter < handle
    methods (Abstract)
        paint(carobj,colorobj)
    end
end
```

Define an instance of the `Cars` class:

```matlab
c1 = Cars.Compact;
```

The color of this car is `Greenish`, as defined by the `Colors.Greenish` enumeration:

```matlab
c1.Color
ans =
    Greenish
```

Use the `paint` method to change the car color:

```matlab
c1.paint(Colors.Reddish)
c1.Color
ans =
    Reddish
```

### See Also

#### Related Examples

- “Save and Load Enumerations” on page 14-45
- “Enumerations for Property Values” on page 14-17
Save and Load Enumerations

In this section...

“Basic Knowledge” on page 14-45
“Built-In and Value-Based Enumeration Classes” on page 14-45
“Simple and Handle-Based Enumeration Classes” on page 14-45
“Causes: Load as struct Instead of Object” on page 14-46

Basic Knowledge

See the `save` and `load` functions and “Save and Load Process for Objects” on page 13-2 for general information on saving and loading objects.

To see a list of enumeration names defined by a class, use the `enumeration` function.

Built-In and Value-Based Enumeration Classes

When you save enumerations that derive from built-in classes or that are value-based classes with properties, MATLAB saves the names of the enumeration members and the definition of each member.

When loading these enumerations, MATLAB preserves names over underlying values. If the saved named value is different from the current class definition, MATLAB uses the value defined in the current class, and then issues a warning.

Simple and Handle-Based Enumeration Classes

When you save simple enumerations that have no properties, superclasses, or values associated with the member names or enumerations derived from the `handle` class, MATLAB saves the names and any underlying values.

When loading these types of enumerations, MATLAB does not check the values associated with the names in the current class definition. This behavior results from the fact that simple enumerations have no underlying values and handle-based enumerations can legally have values that are different than those values defined by the class.
Causes: Load as struct Instead of Object

If you add a new named value or a new property to a class after saving an enumeration, MATLAB does not warn during load.

If the changes to the enumeration class definition do not prevent MATLAB from loading the object (that is, all the named values in the MAT-File are present in the modified class definition), then MATLAB issues a warning that the class has changed and loads the enumeration.

In the following cases, MATLAB issues a warning and loads as much of the saved data as possible as a struct:

- MATLAB cannot find the class definition
- The class is no longer an enumeration class
- MATLAB cannot initialize the class
- There is one or more enumeration members in the loaded enumeration that is not in the class definition
- If the class is a value-based enumeration with properties and a property that exists in the file, is not present in the class definition

**struct Fields**

The returned struct has these fields:

- **ValueNames** — A cell array of strings, one per unique value in the enumeration array.
- **Values** — An array of the same dimension as ValueNames containing the corresponding values of the enumeration members named in ValueNames. Depending on the kind of enumeration class, Values can be one of the following:
  - If the enumeration class derives from a built-in class, the array class is the same as the built-in class. The values in the array are the underlying values of each enumeration member.
  - Otherwise, a struct array representing the property name — property values pairs of each enumeration member. For simple and handle-based enumerations, the struct array has no fields.
- **ValueIndices** — a uint32 array of the same size as the original enumeration. Each element is an index into the ValueNames and Values arrays. The content of ValueIndices represents the value of each object in the original enumeration array.
See Also

More About

- “Named Values” on page 14-2
Constant Properties
Define Class Properties with Constant Values

In this section...

“Defining Named Constants” on page 15-2
“Constant Property Assigned a Handle Object” on page 15-4
“Constant Property Assigned Any Object” on page 15-4
“Constant Properties — No Support for Get Events” on page 15-6

Defining Named Constants

You can define constants that you can refer to by name by creating a MATLAB class that defines constant properties.

Use constant properties to define constant values that you can access by name. Create a class with constant properties by declaring the Constant attribute in the property blocks. Setting the Constant attribute means that, once initialized to the value specified in the property block, the value cannot be changed.

Assigning Values to Constant Properties

Assign any value to a Constant property, including a MATLAB expression. For example:

```matlab
classdef NamedConst
    properties (Constant)
        R = pi/180
        D = 1/NamedConst.R
        AccCode = '0145968740001110202NPQ'
        RN = rand(5)
    end
end
```

MATLAB evaluates the expressions when loading the class. Therefore, the values MATLAB assigns to RN are the result of a single call to the rand function and do not change with subsequent references to NamedConst.RN. Calling clear classes causes MATLAB to reload the class and reinitialize the constant properties.

Referencing Constant Properties

Refer to the constant using the class name and the property name:
For example, to use the NamedConst class defined in the previous section, reference the constant for the degree to radian conversion, R:

\[
radi = 45 * \text{NamedConst.R}
\]

\[
radi = 0.7854
\]

**Constants In Packages**

To create a library for constant values that you can access by name, first create a package folder, then define the various classes to organize the constants. For example, to implement a set of constants that are useful for making astronomical calculations, define a \texttt{AstroConstants} class in a package called constants:

\texttt{+constants/@AstroConstants/AstroConstants.m}

The class defines a set of Constant properties with values assigned:

```matlab
classdef AstroConstants
    properties (Constant)
        C = 2.99792458e8     % m/s
        G = 6.67259          % m/kgs
        Me = 5.976e24        % Earth mass (kg)
        Re = 6.378e6         % Earth radius (m)
    end
end
```

To use this set of constants, reference them with a fully qualified class name. For example, the following function uses some of the constants defined in AstroConstants:

```matlab
function E = energyToOrbit(m,r)
    E = constants.AstroConstants.G * constants.AstroConstants.Me * m * ...  
        (1/constants.AstroConstants.Re-0.5*r);
end
```

Importing the package into the function eliminates the need to repeat the package name (see import):

```matlab
function E = energyToOrbit(m,r)
    import constants.*;
    E = AstroConstants.G * AstroConstants.Me * m * ...
end
```
(1/AstroConstants.Re - 0.5 * r);
end

**Constant Property Assigned a Handle Object**

If a class defines a constant property with a value that is a handle object, you can assign values to the handle object properties. Create a local variable to access the handle object.

For example, the `ConstMapClass` class defines a constant property. The value of the constant property is a handle object (a `containers.Map` object).

```matlab
classdef ConstMapClass < handle
    properties (Constant)
        ConstMapProp = containers.Map
    end
end
```

To assign the current date to the `Date` key, return the handle from the constant property, then make the assignment using the local variable on the left side of the assignment statement:

```matlab
localMap = ConstMapClass.ConstMapProp
localMap('Date') = datestr(clock);
```

You cannot use a reference to a constant property on the left side of an assignment statement. For example, MATLAB interprets the following statement as the creation of a `struct` named `ConstantMapClass` with a field `ConstMapProp`:

```matlab
ConstMapClass.ConstMapProp('Date') = datestr(clock);
```

**Constant Property Assigned Any Object**

You can assign an instance of the defining class to a constant property. MATLAB creates the instance assigned to the constant property when loading the class. Use this technique only when the defining class is a `handle` class.

The `MyProject` is an example of such a class:

```matlab
classdef MyProject < handle
    properties (Constant)
        ProjectInfo = MyProject
    end
```
properties
    Date
    Department
    ProjectNumber
end
methods (Access = private)
    function obj = MyProject
        obj.Date = datestr(clock);
        obj.Department = 'Engineering';
        obj.ProjectNumber = 'P29.367';
    end
end
end

Reference property data via the Constant property:

MyProject.ProjectInfo.Date

ans =
18-Apr-2002 09:56:59

Because MyProject is a handle class, you can get the handle to the instance that is assigned to the constant property:

p = MyProject.ProjectInfo;

Access the data in the MyProject class using this handle:

p.Department

ans =
Engineering

Modify the nonconstant properties of the MyProject class using this handle:

p.Department = 'Quality Assurance';

p is a handle to the instance of MyProject that is assigned to the ProjectInfo constant property:

MyProject.ProjectInfo.Department
ans =

Quality Assurance

Clearing the class results in the assignment of a new instance of MyProject to the ProjectInfo property.

clear MyProject
MyProject.ProjectInfo.Department
ans =

Engineering

You can assign an instance of the defining class as the default value of a property only when the property is declared as Constant

**Constant Properties — No Support for Get Events**

Constant properties do not support property PreGet or PostGet events. MATLAB issues a warning during class initialization if you set the GetObservable attribute of a Constant property to true.

**See Also**

**Related Examples**

- “Static Data” on page 4-2

**More About**

- “Named Values” on page 14-2
Information from Class Metadata

- “Class Metadata” on page 16-2
- “Class Introspection with Metadata” on page 16-5
- “Find Objects with Specific Values” on page 16-10
- “Get Information About Properties” on page 16-14
- “Find Default Values in Property Metadata” on page 16-20
Class Metadata

In this section...

“What Is Class Metadata?” on page 16-2
“The meta Package” on page 16-3
“Metaclass Objects” on page 16-3
“Metaclass Object Lifecycle” on page 16-4

What Is Class Metadata?

Class metadata is information about class definitions that is available from various metaclasses objects. Use metaclass objects to obtain information without having to create instances of the class. Metadata enables the programmatic inspection of classes. Each metaclass has properties, methods, and events that contain information about the class or class component it describes.

All class components have an associated metaclass, which you access from the meta.class object. For example, create the meta.class object for the matlab.mixin.Copyable class:

```matlab
mc = ?matlab.mixin.Copyable
```

```plaintext
mc =

class with properties:

    Name: 'matlab.mixin.Copyable'
    Description: 'Implement copy method for handle objects in MA...'
    DetailedDescription: '
        Hidden: 0
        Sealed: 0
        Abstract: 1
        Enumeration: 0
        ConstructOnLoad: 1
        HandleCompatible: 1
        InferiorClasses: {0x1 cell}
        ContainingPackage: [1x1 meta.package]
        PropertyList: [0x1 meta.property]
        MethodList: [19x1 meta.method]
        EventList: [1x1 meta.event]
        EnumerationMemberList: [0x1 meta.EnumeratedValue]
        SuperclassList: [1x1 meta.class]
```
The meta Package

The meta package contains metaclasses that describe the definition of classes and class components. The class name indicates the component described by the metaclass. For example, each class property has a meta.property associated with it. Attributes defined for class components correspond to properties in the respective metaclass object.

• meta.package — Access from meta.class ContainingPackage property.
• meta.class — Create from class name or class object using metaclass function or ? operator.
• meta.property — Access from meta.class PropertyList property.
• meta.DynamicProperty — Obtain from the addprop method.
• meta.method — Access from meta.class MethodList property.
• meta.event — Access from meta.class EventList property.
• meta.EnumeratedValue — Access from meta.class EnumerationMemberListList property.

Metaclass Objects

You cannot instantiate metaclasses directly by calling the respective class constructor. Create metaclass objects from class instances or from the class name.

• ?ClassName — Returns a meta.class object for the named class. Use meta.class.fromName with class names stored as characters in variables.
• meta.class.fromName('ClassName') — returns the meta.class object for the named class (meta.class.fromName is a meta.class method).
• metaclass(obj) — Returns a metaclass object for the class instance (metaclass)

Create meta.class object from class name using the ? operator:
mc = ?MyClass;

Create meta.class object from class name using the fromName method:
mc = meta.class.fromName('MyClass');

Create meta.class object from class instance
obj = MyClass;
mc = metaclass(obj);
The `metaclass` function returns the `meta.class` object (that is, an object of the `meta.class` class). You can obtain other metaclass objects (`meta.property`, `meta.method`, and so on) from the `meta.class` object.

**Note** Metaclass is a term used here to refer to all the classes in the `meta` package. `meta.class` is a class in the `meta` package whose instances contain information about MATLAB classes. Metadata is information about classes contained in metaclasses.

**Metaclass Object Lifecycle**

When you change a class definition, MATLAB reloads the class definition. If instances of the class exist, MATLAB updates those objects according to the new definition.

However, MATLAB does not update existing metaclass objects to the new class definition. If you change a class definition while metaclass objects of that class exist, MATLAB deletes the metaclass objects and their handles become invalid. You must create a new metaclass object after updating the class.

For information on how to modify and reload classes, see “Automatic Updates for Modified Classes” on page 5-50.

**See Also**

**Related Examples**

- “Class Introspection with Metadata” on page 16-5
- “Find Objects with Specific Values” on page 16-10
- “Get Information About Properties” on page 16-14
- “Find Default Values in Property Metadata” on page 16-20
Class Introspection with Metadata

Using Class Metadata

Use class metadata to get information about classes and objects programmatically. For example, you can determine attribute values for class members or get a list of events defined by the class. For basic information about metadata, see “Class Metadata” on page 16-2.

Inspect the EmployeeData Class

The EmployeeData class is a handle class with two properties, one of which has private Access and defines a set access method.

classdef EmployeeData < handle
    properties
        EmployeeName
    end
    properties (Access = private)
        EmployeeNumber
    end
    methods
        function obj = EmployeeData(name,ss)
            if nargin > 0
                obj.EmployeeName = name;
                obj.EmployeeNumber = ss;
            end
        end
        function set.EmployeeName(obj,name)
            if ischar(name)
                obj.EmployeeName = name;
            else
                error('Employee name must be a char vector')
            end
        end
end
Inspect Class Definition

Using the EmployeeData class, create a meta.class object using the ? operator:

```plaintext
mc = ?EmployeeData;
```

Determine from what classes EmployeeData derives. The returned value is a meta.class object for the handle superclass:

```plaintext
a = mc.SuperclassList;
a.Name
ans =
handle
```

The EmployeeData class has only one superclass. For classes having more than one direct superclass, a contains a meta.class object for each superclass.

Use an indexed reference to refer to any particular superclass:

```plaintext
a(1).Name
or, directly from mc:
mc.SuperclassList(1).Name
ans =
handle
```

The SuperclassList property contains only direct superclasses.

Inspect Properties

Find the names of the properties defined by the EmployeeData class. First obtain an array of meta.properties objects from the meta.class PropertyList property.

```plaintext
mc = ?EmployeeData;
mpArray = mc.PropertyList;
```

The length of mpArray indicates that there are two meta.property objects, one for each property defined by the EmployeeData class:
length(mpArray)
ans =
    2

Now get a \texttt{meta.property} object from the array:

\begin{verbatim}
prop1 = mpArray(1);
prop1.Name
\end{verbatim}

ans =
EmployeeName

The \texttt{Name} property of the \texttt{meta.property} object identifies the class property represented by that \texttt{meta.property} object.

Query other \texttt{meta.property} object properties to determine the attributes of the \texttt{EmployeeName} properties.

\textbf{Find Component with Specific Attribute}

You can use indexing techniques to list class components that have specific attribute values. For example, this code lists the methods in the \texttt{EmployeeData} class that have \texttt{private} access:

\begin{verbatim}
mc = ?EmployeeData;
mc.PropertyList(ismember({mc.PropertyList(:).SetAccess},'private')).Name
\end{verbatim}

ans =
EmployeeNumber

\texttt{Access} is not a property of the \texttt{meta.property} class. Use \texttt{SetAccess} and \texttt{GetAccess}, which are properties of the \texttt{meta.property} class.

Find components with attributes that are logical values using a statement like this one:

\begin{verbatim}
mc = ?handle;
mc.MethodList(ismember([mc.MethodList(:).Hidden],true)).Name
\end{verbatim}

ans =
empty
Inspect Class Instance

Create an EmployeeData object and determine property access settings:

EdObj = EmployeeData('My Name',1234567);
mcEdObj = metaclass(EdObj);
mpArray = mcEdObj.PropertyList;
EdObj.(mpArray(1).Name) % Dynamic field names work with objects

The value of the EmployeeName property is the text My Name, which was assigned in the constructor.

ans = 
   My Name

The value of the EmployeeNumber property is not accessible because the property has private Access.

EdObj.(mpArray(2).Name)

You cannot get the 'EmployeeNumber' property of EmployeeData.

mpArray(2).GetAccess

ans =
   private

Obtain a function handle to the EmployeeName property set access function:

mpArray(1).SetMethod

ans =
   @D:\MyDir\@EmployeeData\EmployeeData.m>EmployeeData.set.EmployeeName

Metaclass EnumeratedValues Property

The meta.class EnumeratedValues property contains an array of meta.EnumeratedValue objects, one for each enumeration member. Use the meta.EnumeratedValue Name property to obtain the enumeration member names defined by an enumeration class. For example, given the WeekDays enumeration class:

```matlab
classdef WeekDays
    enumeration
        Monday, Tuesday, Wednesday, Thursday, Friday
    end
end```
Query enumeration names from the meta.class object:

```java
mc = ?WeekDays;
mc.EnumerationMemberList(2).Name

ans =

Tuesday
```

**See Also**

**Related Examples**
- “Find Objects with Specific Values” on page 16-10
Find Objects with Specific Values

In this section...

“Find Handle Objects” on page 16-10
“Find by Attribute Settings” on page 16-11

Find Handle Objects

Use the handle class `findobj` method to find objects that have properties with specific values. For example, the following class defines a `PhoneBook` object to represent a telephone book entry in a database. The `PhoneBook` class subclasses the `dynamicprops` class, which derives from `handle`.

```matlab
classdef PhoneBook < dynamicprops
    properties
        Name
        Address
        Number
    end
    methods
        function obj = PhoneBook(n,a,p)
            obj.Name = n;
            obj.Address = a;
            obj.Number = p;
        end
    end
end
```

Here are three of the `PhoneBook` entries in the database:

```matlab
PB(1) = PhoneBook('Nancy Vidal','123 Washington Street','5081234567');
PB(2) = PhoneBook('Nancy Vidal','123 Main Street','5081234568');
PB(3) = PhoneBook('Nancy Wong','123 South Street','5081234569');
```

One of these three `PhoneBook` objects has a dynamic property:

```matlab
PB(2).addprop('HighSpeedInternet');
PB(2).HighSpeedInternet = '1M';
```

Find Property/Value Pairs

Find the object representing employee Nancy Wong and display the name and number by concatenating the strings:
NW = findobj(PB,'Name','Nancy Wong');
[NW.Name,' - ',NW.Number]

ans =
Nancy Wong - 5081234569

**Find Objects with Specific Property Names**

Search for objects with specific property names using the `-property` option:

H = findobj(PB,'-property','HighSpeedInternet');
H.HighSpeedInternet

ans =
1M

The `-property` option enables you to omit the value of the property and search for objects using only the property name.

**Using Logical Expressions**

Search for specific combinations of property names and values:

H = findobj(PB,'Name','Nancy Vidal','-and','Address','123 Main Street');
H.Number

ans =
5081234568

**Find by Attribute Settings**

All metaclasses derive from the `handle` class. You can use the `handle` `findobj` method to find class members that have specific attribute settings.

For example, find the abstract methods in a class definition by searching the `meta.class MethodList` for `meta.method` objects with their `Abstract` property set to `true`:

Use the class name in character format because class is abstract. You cannot create an object of the class:

mc = meta.class.fromName('MyClass');
Search the MethodList list of meta.method objects for those methods that have their Abstract property set to true:

```matlab
absMethods = findobj(mc.MethodList,'Abstract',true);
methodNames = {absMethods.Name};
```

The cell array, methodNames, contains the names of the abstract methods in the class.

**Find Properties That Have Public Get Access**

Find the names of all properties in the containers.Map class that have public GetAccess:

- Get the meta.class object.
- Use findobj to search the array of meta.property objects.
- Use braces to create a cell array of property names.

```matlab
mc = ?containers.Map;
mpArray = findobj(mc.PropertyList,'GetAccess','public');
names = {mpArray.Name};
```

Display the names of all containers.Map properties that have public GetAccess:

```matlab
celldisp(names)
names{1} = Count
names{2} = KeyType
names{3} = ValueType
```

**Find Static Methods**

Determine if any containers.Map class methods are static:

```matlab
~isempty(findobj([mc.MethodList(:)],'Static',true))
```
findobj returns an array of meta.method objects for the static methods. In this case, the list of static methods is not empty. Therefore, there are static methods defined by this class.

Get the names of any static methods from the meta.method array:

```matlab
staticMethodInfo = findobj([mc.MethodList(:)],'Static',true);
staticMethodInfo(:).Name
```

ans =

empty

The name of the static method (there is only one in this case) is empty. Here is the information from the meta.method object for the empty method:

```matlab
staticMethodInfo
```

method with properties:

- Name: 'empty'
- Description: 'Returns an empty object array of the given size'
- DetailedDescription: ''
- Access: 'public'
- Static: 1
- Abstract: 0
- Sealed: 0
- Hidden: 1
- InputNames: {'varargin'}
- OutputNames: {'E'}
- DefiningClass: [1x1 meta.class]

See Also

eempty

Related Examples

- “Get Information About Properties” on page 16-14
Get Information About Properties

The meta.property Object

Use the meta.property class to determine the values of property attributes. The writable properties of a meta.property object correspond to the attributes of the associated property. The values of the writable meta.property properties correspond to the attribute values specified in the class definition.

Get the meta.property object for a property from the meta.class object. To get the meta.class object for a class:

- Use the metaclass function on an object of the class.
- Use the ? operator with the class name.

For example, the BasicHandle class defines three properties:

```matlab
classdef BasicHandle < handle
    properties (SetAccess = private)
        Date = date
        PassKey = randi(9,[1,7])
    end
    properties
        Category {mustBeMember(Category,{''new'',''change'})} = 'new'
    end
end
```

Create the meta.class object using the ? operator with the class name:

```matlab
mc = ?BasicHandle
```

```matlab
mc =
```

```matlab
class with properties:
```
```matlab
    Name: 'BasicHandle'
    Description: ''
```
The `meta.class` object property named `PropertyList` contains an array of `meta.property` objects, one for each property defined by the class. For example, the name of the property associated with the `meta.property` object in element 1 is:

```matlab
mc.PropertyList(1).Name
ans =
Date
```

The `meta.class` object contains a `meta.property` object for all properties, including hidden properties. The `properties` function returns only public properties.

For a handle class, use the `handle findprop` method to get the `meta.property` object for a specific property.

For example, find the `meta.property` object for the `Category` property of the `BasicHandle` class.

```matlab
mp = findprop(BasicHandle,'Category')
mp =

property with properties:

    Name: 'Category'
    Description: ''
  DetailedDescription: ''
    GetAccess: 'public'
```
The preceding `meta.property` display shows that a default `BasicHandle` object `Category` property:

- Has public `GetAccess` and `SetAccess`
- Has a default value of `new`

For a list of property attributes, see “Table of Property Attributes” on page 8-9.

**How to Index Metaclass Objects**

Access other metaclass objects directly from the `meta.class` object properties. For example, the statement:

```plaintext
mc = ?containers.Map;
```

returns a `meta.class` object:

```plaintext
class(mc)
```

```plaintext
ans =
```

```plaintext
meta.class
```

Referencing the `PropertyList` `meta.class` property returns an array with one `meta.property` object for each property of the `containers.Map` class:

```plaintext
class(mc.PropertyList)
```
ans =
meta.property

Each array element is a single meta.property object:
mc.Properties(1)
ans =
    [1x1 meta.property]

The Name property of the meta.property object contains a char vector that is the name of the property:
class(mc.PropertyList(1).Name)
ans =
    char

Apply standard MATLAB indexing to access information in metaclass objects.

For example, the meta.class PropertyList property contains an array of meta.property objects. The following expression accesses the first meta.property object in this array and returns the first and last (C and t) letters of the char vector contained in the meta.property Name property.
mc.PropertyList(1).Name([1 end])
ans =
    Ct

How to Find Properties with Specific Attributes

This example implements a function that finds properties with specific attribute values. For example, you can:

- Find objects that define constant properties (Constant attribute set to true).
- Determine what properties are read-only (GetAccess = public, SetAccess = private).
The `findAttrValue` function returns a cell array of property names that set the specified attribute.

The `findAttrValue` function accesses information from metadata using these techniques:

- If input argument, `obj`, is a char vector, use the `meta.class.fromName` static method to get the `meta.class` object.
- If input argument, `obj`, is an object, use the `metaclass` function to get the `meta.class` object.
- Every property has an associated `meta.property` object. Obtain these objects from the `meta.class.PropertyList` property.
- Use the handle class `findprop` method to determine if the requested property attribute is a valid attribute name. All property attributes are properties of the `meta.property` object. The statement, `findobj(mp,'PropertyName')` determines whether the `meta.property` object, `mp`, has a property called `PropertyName`.
- Reference `meta.property` object properties using dynamic field names. For example, if `attrName = 'Constant'`, then MATLAB converts the expression `mp.(attrName)` to `mp.Constant`.
- The optional third argument enables you to specify the value of attributes whose values are not logical `true` or `false` (such as `GetAccess` and `SetAccess`).

```matlab
function cl_out = findAttrValue(obj,attrName,varargin)
    if ischar(obj)
        mc = meta.class.fromName(obj);
    elseif isobject(obj)
        mc = metaclass(obj);
    end
    ii = 0; numb_props = length(mc.PropertyList);
    cl_array = cell(1,numb_props);
    for c = 1:numb_props
        mp = mc.PropertyList(c);
        if isempty(findprop(mp,attrName))
            error('Not a valid attribute name')
        end
        attrValue = mp.(attrName);
        if attrValue
            if islogical(attrValue) || strcmp(varargin{1},attrValue)
                ii = ii + 1;
                cl_array(ii) = {mp.Name};
            end
        end
    end
```
Find Property Attributes

Suppose that you have the following containers.Map object:

```matlab
cmap = containers.Map({'rose','bicycle'},{'flower','machine'});
```

Find properties with private SetAccess:

```matlab
findAttrValue(cmap,'SetAccess','private')
```

```matlab
ans =
    'Count'    'KeyType'    'ValueType'    'serialization'
```

Find properties with public GetAccess:

```matlab
findAttrValue(cmap,'GetAccess','public')
```

```matlab
ans =
    'Count'    'KeyType'    'ValueType'
```

See Also

Related Examples

- “Find Default Values in Property Metadata” on page 16-20
Find Default Values in Property Metadata

In this section...

“Default Values” on page 16-20
“meta.property Data” on page 16-20

Default Values

Class definitions can specify explicit default values for properties (see “Property Default Values” on page 8-18). Determine if a class defines an explicit default value for a property and what the value of the default is from the property meta.property object.

meta.property Data

The meta.class object for a class contains a meta.property object for every property defined by the class, including properties with private and protected access.

For example, get the meta.class object for the PropertyWithDefault class shown here:

```matlab
classdef PropertyWithDefault
    properties
        Date = date
        RandNumber = randi(9)
    end
end
```

Get an array of meta.property objects from the meta.class object:

```matlab
mc = ?PropertyWithDefault; % meta.class object
mp = mc.PropertyList; % meta.property array
```

The second element in the mp array is the meta.property object for the RandNumber property. Listing the meta.property object shows the information contained in its properties:

```matlab
mp(2)

    property with properties:
```
Two of the listed meta.property properties provide information on default values:

- **HasDefault** — true (displayed as 1) if the class specifies a default value for the property, false if it does not.
- **DefaultValue** — Contains the default value, when the class defines a default value for the property. If the default value is an expression, the value of DefaultValue is the result of evaluating the expression.

For more information on the evaluation of property default values defined by expressions, see “Evaluation of Expressions in Class Definitions” on page 6-10.

These properties provide a programmatic way to obtain property default values without opening class definition files. Use these meta.property object properties to obtain property default values for both built-in classes and classes defined in MATLAB code.

### Query Default Value

The procedure for querying a default value involves:

1. Getting the meta.property object for the property whose default value you want to query.
Testing the logical value of the \texttt{meta.property HasDefault} property to determine if the property defines a default value. MATLAB returns an error when you query the \texttt{DefaultValue} property if the class does not define a default value for the property.

Obtaining the default value from the \texttt{meta.property DefaultValue} property if the \texttt{HasDefault} value is true.

Use the \texttt{?} operator, the \texttt{metaclass} function, or the \texttt{meta.class.fromName} static method (works with \texttt{char} vector variable) to obtain a \texttt{meta.class} object.

The \texttt{meta.class} object \texttt{PropertyList} contains an array of \texttt{meta.property} objects. Identify which property corresponds to which \texttt{meta.property} object using the \texttt{meta.property Name} property.

For example, this class defines properties with default values:

```matlab
classdef MyDefs
    properties
        Material = 'acrylic'
        InitialValue = 1.0
    end
end
```

Follow these steps to obtain the default value defined for the \texttt{Material} property. Include any error checking that is necessary for your application.

1. Get the \texttt{meta.class} object for the class:

   ```matlab
   mc = ?MyDefs;
   ```

2. Get an array of \texttt{meta.property} objects from the \texttt{meta.class PropertyList} property:

   ```matlab
   mp = mc.PropertyList;
   ```

3. The length of the \texttt{mp} array equals the number of properties. You can use the \texttt{meta.property Name} property to find the property of interest:

   ```matlab
   for k = 1:length(mp)
       if (strcmp(mp(k).Name,'Material'))
           ...
   end
   ```

4. Before querying the default value of the \texttt{Material} property, test the \texttt{HasDefault} \texttt{meta.property} to determine if \texttt{MyClass} defines a default property for this property:
if mp(k).HasDefault
    dv = mp(k).DefaultValue;
end

The DefaultValue property is read-only. Changing the default value in the class definition changes the value of DefaultValue property. You can query the default value of a property regardless of its access settings.

Abstract and dynamic properties cannot define default values. Therefore, MATLAB returns an error if you attempt to query the default value of properties with these attributes. Always test the logical value of the meta.property HasDefault property before querying the DefaultValue property to avoid generating an error.

**Default Values Defined as Expressions**

Class definitions can define property default values as MATLAB expressions (see “Evaluation of Expressions in Class Definitions” on page 6-10 for more information). MATLAB evaluates these expressions the first time the default value is needed, such as the first time you create an instance of the class.

Querying the meta.property DefaultValue property causes MATLAB to evaluate a default value expression, if it had not yet been evaluated. Therefore, querying a property default value can return an error or warning if errors or warnings occur when MATLAB evaluates the expression. See “Property with Expression That Errors” on page 16-24 for an example.

**Property with No Explicit Default Value**

MyClass does not explicitly define a default value for the Foo property:

```matlab
classdef MyFoo
    properties
        Foo
    end
end
```

The meta.property instance for property Foo has a value of false for HasDefault. Because the class does not explicitly define a default value for Foo, attempting to access the DefaultValue property causes an error:

```matlab
mc = ?MyFoo;
mp = mc.PropertyList(1);
mp.HasDefault
```
ans =

0

dv = mp.DefaultValue;

No default value has been defined for property Foo

**Abstract Property**

MyClass defines the Foo property as Abstract:

```matlab
classdef MyAbst
    properties (Abstract)
        Foo
    end
end
```

The meta.property instance for property Foo has a value of false for its HasDefault property because you cannot define a default value for an Abstract property. Attempting to access DefaultValue causes an error:

```matlab
mc = ?MyAbst;
mp = mc.PropertyList(1);
mp.HasDefault
ans =

0

dv = mp.DefaultValue;

Property Foo is abstract and therefore cannot have a default value.

**Property with Expression That Errors**

MyPropEr defines the Foo property default value as an expression that errors when evaluated.

```matlab
classdef MyPropEr
    properties
        Foo = sin(pie/2)
    end
end
```
The `meta.property` object for property `Foo` has a value of `true` for its `HasDefault` property because `Foo` does have a default value:

\[ \sin(\pi/2) \]

However, this expression returns an error (`pie` is a function that creates a pie graph, not the value `pi`).

```matlab
mc = ?MyPropEr;
mp = mc.PropertyList(1);
mp.HasDefault
ans =
1
dv = mp.DefaultValue;
```

Error using `pie` (line 29)
Not enough input arguments.

Querying the default value causes the evaluation of the expression and returns the error.

**Property With Explicitly Defined Default Value of Empty**

`MyEmptyProp` assigns a default of `[]` (empty double) to the `Foo` property:

```matlab
classdef MyEmptyProp
    properties
        Foo = []
    end
end
```

The `meta.property` object for property `Foo` has a value of `true` for its `HasDefault` property. Accessing `DefaultValue` returns the value `[]`:

```matlab
mc = ?MyEmptyProp;
mp = mc.PropertyList(1);
mp.HasDefault
ans =
1
dv = mp.DefaultValue;
```
dv =

[]

See Also

Related Examples
- “Get Information About Properties” on page 16-14
Specialize Object Behavior

- “Methods That Modify Default Behavior” on page 17-2
- “Number of Arguments for subsref and subsasgn” on page 17-6
- “Modify nargout and nargin for Indexing Methods” on page 17-9
- “Concatenation Methods” on page 17-11
- “Object Converters” on page 17-12
- “Object Array Indexing” on page 17-15
- “Code Patterns for subsref and subsasgn Methods” on page 17-21
- “Indexed Reference” on page 17-28
- “Indexed Assignment” on page 17-31
- “end as Object Index” on page 17-35
- “Objects in Index Expressions” on page 17-37
- “Class with Modified Indexing” on page 17-39
- “Operator Overloading” on page 17-47
Methods That Modify Default Behavior

How to Customize Class Behavior

There are functions that MATLAB calls implicitly when you perform certain actions with objects. For example, a statement like \([B(1);A(3)]\) involves indexed reference and vertical concatenation.

You can change how user-defined objects behave by defining methods that control specific behaviors. To change a behavior, implement the appropriate method with the name and signature of the MATLAB function.

Which Methods Control Which Behaviors

The following table lists the methods to implement for your class and describes the behaviors that they control.

<table>
<thead>
<tr>
<th>Class Method to Implement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenating Objects</td>
<td></td>
</tr>
<tr>
<td>\texttt{cat}, \texttt{horzcat}, and \texttt{vertcat}</td>
<td>Customize behavior when concatenating objects</td>
</tr>
<tr>
<td></td>
<td>See “Subclasses of Built-In Types with Properties” on page 12-68</td>
</tr>
<tr>
<td>Creating Empty Arrays</td>
<td></td>
</tr>
<tr>
<td>\texttt{empty}</td>
<td>Create empty arrays of the specified class. See “Empty Arrays” on page 10-8</td>
</tr>
<tr>
<td>Displaying Objects</td>
<td></td>
</tr>
<tr>
<td>Class Method to Implement</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>disp</td>
<td>Called when you enter <code>disp(obj)</code> on the command line</td>
</tr>
<tr>
<td>display</td>
<td>Called by statements that are not terminated by semicolons. <code>disp</code> is often used to implement <code>display</code> methods.</td>
</tr>
<tr>
<td></td>
<td>See “Overloading the disp Function” on page 18-42</td>
</tr>
<tr>
<td></td>
<td>See “Custom Display Interface” on page 18-2</td>
</tr>
</tbody>
</table>

**Converting Objects to Other Classes**

- Converters like `double` and `char` Convert an object to a MATLAB built-in class
  - See “The Character Converter” on page 19-16 and “The Double Converter” on page 19-15

**Indexing Objects**

- `subsref` and `subsasgn` Enables you to create nonstandard indexed reference and indexed assignment
  - See “Object Array Indexing” on page 17-15

- `end` Supports `end` syntax in indexing expressions using an object; e.g., `A(1:end)`
  - See “end as Object Index” on page 17-35

- `numel` Determine the number of elements in an array
  - See “Modify nargout and nargin for Indexing Methods” on page 17-9

- `numArgumentsFromSubscript` Overload to specify the number of values to return from indexing expressions.
  - See “Number of Arguments for subsref and subsasgn” on page 17-6
<table>
<thead>
<tr>
<th>Class Method to Implement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>Determine the dimensions of an array</td>
</tr>
<tr>
<td></td>
<td>See &quot;Use of size and numel with Classes&quot; on page 12-77</td>
</tr>
<tr>
<td>subsindex</td>
<td>Support using an object in indexing expressions</td>
</tr>
<tr>
<td></td>
<td>See &quot;Objects in Index Expressions&quot; on page 17-37</td>
</tr>
</tbody>
</table>

### Saving and Loading Objects

- **loadobj** and **saveobj**
  - Customize behavior when loading and saving objects
  - See "Object Save and Load"

### Reshape and Rearrange

- **permute**
  - Rearrange dimensions of N-D array
- **transpose**
  - Transpose vector or matrix
- **ctranspose**
  - Complex conjugate transpose
- **reshape**
  - Reshape array

### Determine Size and Shape

- **isscalar**
  - Determine if the input is a scalar
- **isvector**
  - Determine if the input is a vector
- **ismatrix**
  - Determine if the input is a matrix
- **isempty**
  - Determine if the input is empty

## Overload Functions and Override Methods

Overloading and overriding are terms that describe techniques for customizing class behavior. Here is how we use these terms in MATLAB.

### Overloading

Overloading means that there is more than one function or method having the same name within the same scope. MATLAB dispatches to a particular function or method based on the dominant argument. For example, the `timeseries` class overloads the MATLAB `plot` function.
function. When you call `plot` with a `timeseries` object as an input argument, MATLAB calls the `timeseries` class method named `plot`.

To call the nonoverloaded function, use the `builtin` function.

**Overriding**

Overriding means redefining a method inherited from a superclass. MATLAB dispatches to the most specific version of the method. That is, if the dominant argument is an object of the subclass, then MATLAB calls the subclass method.

To control class dominance, use the `InferiorClasses` attribute.

**See Also**

**Related Examples**

- “Overload Functions in Class Definitions” on page 9-33
- “Object Precedence in Method Invocation” on page 9-46
- “Operator Overloading” on page 17-47
Number of Arguments for \texttt{subsref} and \texttt{subsasgn}

In this section...

| “How MATLAB Determines Number of Arguments” on page 17-6 |
| “Syntax for \texttt{subsref}, and \texttt{subsasgn} Methods” on page 17-8 |

How MATLAB Determines Number of Arguments

MATLAB calls \texttt{subsref} or \texttt{subsasgn} to determine the result of executing code that involves indexed reference or assignment. The number of elements referenced or assigned by an indexing operation determines the number of arguments MATLAB uses to call \texttt{subsref} and \texttt{subsasgn}. That is, the indexing code determines the number of arguments that MATLAB:

- Returns from the call to \texttt{subsref}
- Passes to the call to \texttt{subsasgn}

Therefore, the indexing code determines the value of \texttt{nargout} for the call to \texttt{subsref} and the value of \texttt{nargin} for the call to \texttt{subsasgn}.

For example, consider the \texttt{ValuesArray} class.

```matlab
classdef ValuesArray
    properties
        Values
    end
    methods
        function obj = ValuesArray(v)
            if nargin > 0
                obj.Values = v;
            end
        end
    end
end
```

Create an array of 10 \texttt{ValuesArray} objects.

```matlab
l = ValuesArray.empty;
for k = 1:10
    l(k) = ValuesArray(k);
end
```
This subscripted reference returns a comma-separated list of three elements. For this statement, the value of nargout in subsref is 3.

\[
l(1:3).Values
\]

\[
ans =
\]

\[
1
\]

\[
ans =
\]

\[
2
\]

\[
ans =
\]

\[
3
\]

The left side of a subscripted assignment statement affects the number of input arguments that MATLAB uses to call subsasgn. This subscripted assignment assigns three values to the three elements added to the array. For this assignment, the value of nargin within subsasgn is 5 (the object, the indexing substructure, and the three values to assign).

\[
[l(1:13).Values] = l(1:3).Values
\]

\[
l =
\]

\[
1x13 ValuesArray array with properties:
\]

\[
Values
\]

If the number of right-side arguments cannot satisfy the number of left-side arguments, MATLAB returns an error:

\[
[l(11:13).Values] = l(1).Values
\]

Insufficient number of outputs from right hand side of equal sign to satisfy assignment.
Syntax for subsref, and subsasgn Methods

If a class overloads subsref to support either '()', '. ', or both types of indexing, and the operation returns more than one value, overload subsref to return multiple values using varargout:

function varargout = subsref(A,S)
...
end

If a class overloads subsasgn to support either '()', '. ', or both types of indexing, and the operation assigns more than one value, overload subsasgn to accept multiple values using varargin:

function A = subsasgn(A,S, varargin)
...
end

See Also

More About
• “Modify nargout and nargin for Indexing Methods” on page 17-9
• “Comma-Separated Lists”
Modify nargout and nargin for Indexing Methods

When to Modify Number of Arguments

By default, the number of values referenced by an indexing operation determines how many output arguments MATLAB uses to call \texttt{subsref}. Similarly, the number of values to assign in an indexed assignment operation determines how many input arguments MATLAB uses to call \texttt{subsasgn}.

If your class design requires that indexing operations return or assign a different number of values than the number defined by the indexing operation, use \texttt{numArgumentsFromSubscript} to specify the required number. \texttt{numArgumentsFromSubscript} provides control over nargout for \texttt{subsref} and nargin for \texttt{subsasgn}.

If your class uses \texttt{numArgumentsFromSubscript}, implement \texttt{subsref} and \texttt{subsasgn} methods to define the actual values returned or assigned by indexing operations.

Before MATLAB release R2015b, MATLAB produced different results for some indexing expressions that return or assign to a comma-separated list. Use \texttt{numArgumentsFromSubscript} to support code that relies on the behavior of previous releases. Also, now you can overload \texttt{numArgumentsFromSubscript} instead of \texttt{numel} to achieve specific results without redefining how \texttt{numel} works.

How to Modify Number of Arguments

When a class overloads \texttt{numArgumentsFromSubscript}, MATLAB calls this method instead of \texttt{numel} to compute the number of arguments expected for \texttt{subsref} nargout and \texttt{subsasgn} nargin.

If classes do not overload \texttt{numArgumentsFromSubscript}, MATLAB calls \texttt{numel} to compute the values of nargout or nargin.

MATLAB calls \texttt{numArgumentsFromSubscript} with three input arguments:
function n = numArgumentsFromSubscript(obj,s,indexingContext)
    ... 
end

<table>
<thead>
<tr>
<th>Input Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>Object whose subsref or subsasgn method is called</td>
</tr>
<tr>
<td>s</td>
<td>Indexing structure that contains the indexing type and indices used in the operation</td>
</tr>
<tr>
<td>indexingContext</td>
<td>Context in which the indexing operation occurs: indexed reference used as a statement, index reference used as a function argument, and indexed assignment</td>
</tr>
</tbody>
</table>

MATLAB uses the value returned by numArgumentsFromSubscript for indexed reference and assignment. Determine the context in which the indexing operation executes by testing the value of indexingContext in your implementation of numArgumentsFromSubscript. For example, test for any or all the possible indexing contexts.

function n = numArgumentsFromSubscript(obj,~,indexingContext)
    switch indexingContext
    case matlab.mixin.util.IndexingContext.Statement
        n = ...; % nargout for indexed reference used as statement
    case matlab.mixin.util.IndexingContext.Expression
        n = ...; % nargout for indexed reference used as function argument
    case matlab.mixin.util.IndexingContext.Assignment
        n = ...; % nargin for indexed assignment
    end
end

For more information and examples, see numArgumentsFromSubscript.

**Note**  For MATLAB version R2015b and later releases, overload numArgumentsFromSubscript instead of numel to customize indexing for your class.

**See Also**

**More About**

- “Number of Arguments for subsref and subsasgn” on page 17-6
- “Use of size and numel with Classes” on page 12-77
## Concatenation Methods

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Default Concatenation&quot; on page 17-11</td>
</tr>
<tr>
<td>&quot;Methods to Overload&quot; on page 17-11</td>
</tr>
</tbody>
</table>

### Default Concatenation

You can concatenate objects into arrays. For example, suppose that you have three instances of the class `MyClass`, `obj1`, `obj2`, `obj3`. You can form arrays of these objects using brackets. Horizontal concatenation calls `horzcat`:

```
HorArray = [obj1, obj2, obj3];
```

`HorArray` is a 1-by-3 array of class `MyClass`. You can concatenate the objects along the vertical dimension, which calls `vertcat`:

```
VertArray = [obj1; obj2; obj3]
```

`VertArray` is a 3-by-1 array of class `MyClass`. To concatenate arrays along different dimensions, use the `cat` function. For example:

```
ndArray = cat(3, HorArray, HorArray);
```

`ndArray` is a 1-by-3-by-2 array.

### Methods to Overload

Overload `horzcat`, `vertcat`, and `cat` to produce specialized behaviors in your class. Overload both `horzcat` and `vertcat` whenever you want to modify object concatenation because MATLAB uses both functions for any concatenation operation.

### See Also

**Related Examples**

- "Subclasses of Built-In Types with Properties" on page 12-68
Object Converters

In this section...

“Why Implement Converters” on page 17-12
“Converters for Package Classes” on page 17-12
“Converters and Subscripted Assignment” on page 17-13

Why Implement Converters

You can convert an object of one class to an object of another class. A converter method has the same name as the class it converts to, such as char or double. Think of a converter method as an overloaded constructor method of another class. The converter takes an instance of its own class and returns an object of a different class.

Converters enable you to:

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly
- Control how instances are interpreted in other contexts

Suppose that you define a polynomial class. If you create a double method for the polynomial class, you can use it to call other functions that require inputs of type double.

\[
p = \text{polynomial}(...);
dp = \text{double}(p);
\text{roots}(dp)
\]

\(p\) is a polynomial object, \texttt{double} is a method of the \texttt{polynomial} class, and \texttt{roots} is a standard MATLAB function whose input arguments are the coefficients of a polynomial.

Converters for Package Classes

Classes defined in packages can have names that are a dot-separated list of names. The last name is a class and preceding names are packages. Name the conversion methods using the package qualifiers in the method names. For example, a conversion method to convert objects of \texttt{MyClass} to objects of the \texttt{PkgName.PkgClass} class uses this method name:
```matlab
classdef MyClass
    ...
    methods
        function objPkgClass = PkgName.PkgClass(objMyclass)
            ...
        end
    end
end
```

You cannot define a converter method that uses dots in the name in a separate file. Define package-class converters in the `classdef` file.

### Converters and Subscripted Assignment

When you make a subscripted assignment statement like:

```matlab
A(1) = myobj;
```

MATLAB compares the class of the Right-Side variable to the class of the Left-Side variable. If the classes are different, MATLAB attempts to convert the Right-Side variable to the class of the Left-Side variable. To do this conversion, MATLAB first searches for a method of the Right-Side class that has the same name as the Left-Side class. Such a method is a converter method, which is similar to a typecast operation in other languages.

If the Right-Side class does not define a method to convert from the Right-Side class to the Left-Side class, MATLAB calls the Left-Side class constructor, passing it the Right-Side variable.

For example, suppose that you make the following assignments:

```matlab
A(1) = objA; % Object of class ClassA
A(2) = objB; % Object of class ClassB
```

MATLAB attempts to call a method of `ClassB` named `ClassA`. If no such converter method exists, MATLAB software calls the `ClassA` constructor, passing `objB` as an argument. If the `ClassA` constructor cannot accept `objB` as an argument, then MATLAB returns an error.

Use cell arrays to store objects of different classes.
See Also

Related Examples
- “Converter Methods” on page 10-21
- “The Double Converter” on page 19-15
Object Array Indexing

Default Indexed Reference and Assignment

MATLAB classes support object array indexing by default. Many class designs require no modification to this behavior.

Arrays enable you to reference and assign elements of the array using a subscripted notation. This notation specifies the indices of specific array elements. For example, suppose that you create two arrays of numbers (using `randi` and concatenation).

Create a 3-by-4 array of integers from 1 through 9:

```matlab
A = randi(9,3,4)
```

```matlab
A =

4     8     5     7
4     2     6     3
7     5     7     7
```

Create a 1-by-3 array of the numbers 3, 6, 9:

```matlab
B = [3 6 9];
```

Reference and assign elements of either array using index values in parentheses:

```matlab
B(2) = A(3,4);
```

```matlab
B =

3     7     9
```

When you execute a statement that involves indexed reference:
C = A(3,4);

MATLAB calls the built-in subsref function to determine how to interpret the statement. Similarly, if you execute a statement that involves indexed assignment:

C(4) = 7;

MATLAB calls the built-in subsasgn function to determine how to interpret the statement.

The MATLAB default subsref and subsasgn functions also work with user-defined objects. For example, create an array of objects of the same class:

```matlab
for k=1:3
    objArray(k) = MyClass;
end
```

Referencing the second element in the object array, objArray, returns the object constructed when k = 2:

```matlab
D = objArray(2);
class(D)
```

ans =

MyClass

You can assign an object to an array of objects of the same class, or an uninitialized variable:

```matlab
newArray(3,4) = D;
```

Arrays of objects behave much like numeric arrays in MATLAB. You do not need to implement any special methods to provide standard array behavior with your class.

For general information about array indexing, see “Array Indexing”.

**What You Can Modify**

You can modify your class indexed reference and/or assignment behavior by implementing class methods called subsref and subsasgn. For syntax description, see their respective reference pages.
Once you add a `subsref` or `subsasgn` method to your class, then MATLAB calls only the class method, not the built-in function. Therefore, your class method must implement all the indexed reference and assignment operations that you want your class to support. These operations include:

- Dot notation calls to class methods
- Dot notation reference and assignment involving properties
- Any indexing using parentheses '()' 
- Any indexing using braces '{}'

Implementing `subsref` and `subsasgn` methods gives you complete control over the interpretation of indexing expressions for objects of your class. Implementing the extent of behaviors that MATLAB provides by default is nontrivial.

**When to Modify Indexing Behavior**

Default indexing for object arrays and dot notation for access to properties and methods enables user-defined objects to behave like built-in classes. For example, suppose that you define a class with a property called `Data` that contains an array of numeric data.

This statement:

```
obj.Data(2,3)
```

Returns the value contained in the second row, third column of the array. If you have an array of objects, use an expression like:

```
objArray(3).Data(2,3)
```

This statement returns the value contained in the second row, third column of the third element in the array.

Modify the default indexing behavior when your class design requires behavior that is different from MATLAB default behavior.

**Built-In subsref and subsasgn Called in Methods**

MATLAB does not call class-defined `subsref` or `subsasgn` methods within the overloaded methods. Within class methods, MATLAB always calls the built-in `subsref` and `subsasgn` functions. This behavior occurs within the class-defined `subsref` and `subsasgn` methods too.
For example, within a class method, this dot reference:

```matlab
obj.Prop
```
calls the built-in `subsref` function. To call the class-defined `subsref` method, use:

```matlab
subsref(obj, substruct('.', 'Prop'))
```

Whenever a method requires the functionality of the class-defined `subsref` or `subsasgn` method, the class must call the overloaded methods as functions. Do not use the operators, `'()'`, `'{}'`, or `'.'`.

For example, suppose that you define a class to represent polynomial. This class has a `subsref` method that evaluates the polynomial with the value of the independent variable equal to the subscript. Assume that this statement defines the polynomial with its coefficients:

```matlab
p = polynom([1 0 -2 -5]);
```

The MATLAB expression for the resulting polynomial is:

```matlab
x^3 - 2*x - 5
```

This subscripted expression returns the value of the polynomial at `x = 3`:

```matlab
p(3)
```

```matlab
ans =
    16
```

Suppose that you want to use this feature in another class method. To do so, call the `subsref` function directly. The `evalEqual` method accepts two `polynom` objects and a value at which to evaluate the polynomials:

```matlab
methods
    function ToF = evalEqual(p1, p2, x)
        % Create arguments for subsref
        subs.type = '()';
        subs.subs = {x};
        % Need to call subsref explicitly
        y1 = subsref(p1, subs);
        y2 = subsref(p2, subs);
        if y1 == y2
            ToF = true;
        else
```

17 Specialize Object Behavior
ToF = false;
end
end
end

This behavior enables you to use standard MATLAB indexing to implement specialized behaviors. See “Class with Modified Indexing” on page 17-39 for examples of how to use both built-in and class-modified indexing.

**Avoid Overriding Access Attributes**

Because `subsref` is a class method, it has access to private class members. Avoid inadvertently giving access to private methods and properties as you handle various types of reference. Consider this `subsref` method defined for a class having private properties, x and y:

```matlab
classdef MyPlot
    properties (Access = private)
        x
        y
    end
    properties
        Maximum
        Minimum
        Average
    end
    methods
        function obj = MyPlot(x,y)
            obj.x = x;
            obj.y = y;
            obj.Maximum = max(y);
            obj.Minimum = min(y);
            obj.Average = mean(y);
        end
        function B = subsref(A,S)
            switch S(1).type
                case '.
                    switch S(1).subs
                        case ' plot'
                            B = plot(A.x,A.y);
                        otherwise
                            % Enable dot notation for all properties and methods
                            B = A.(S.subs);
                        end
                    end
                end
            end
    end
end
```

17-19
This `subsref` enables the use of dot notation to create a plot using the name `plot`. The statement:

```matlab
obj = MyPlot(1:10,1:10);
h = obj.plot;
```
calls the `plot` function and returns the handle to the graphics object.

You do not need to code each method and property name. The `otherwise` code in the inner `switch` block manages any name reference that you do not explicitly specify in `case` statements. However, using this technique exposes any private and protected class members via dot notation. For example, you can reference the private property, `x`, with this statement:

```matlab
obj.x
```

```
ans =
     1     2     3     4     5     6     7     8     9    10
```

The same issue applies to writing a `subsasgn` method that enables assignment to private or protected properties. Your `subsref` and `subsasgn` methods might need to code each specific property and method name explicitly to avoid violating the class design.

### See Also

### Related Examples

- “Code Patterns for subsref and subsasgn Methods” on page 17-21
- “Indexed Reference” on page 17-28
- “Indexed Assignment” on page 17-31
**Code Patterns for subsref and subsasgn Methods**

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Customize Indexed Reference and Assignment” on page 17-21</td>
</tr>
<tr>
<td>“Syntax for subsref and subsasgn Methods” on page 17-21</td>
</tr>
<tr>
<td>“Indexing Structure Describes Indexing Expressions” on page 17-22</td>
</tr>
<tr>
<td>“Values of the Indexing Structure” on page 17-23</td>
</tr>
<tr>
<td>“Typical Patterns for Indexing Methods” on page 17-23</td>
</tr>
</tbody>
</table>

**Customize Indexed Reference and Assignment**

User-defined classes have the same indexing behaviors as that of built-in classes. Classes can customize indexing operations by overloading the functions that MATLAB calls to evaluate indexing expressions. Overload the `subsref` and `subsasgn` functions when you want to define special behaviors for indexed reference and assignment.

For an overview of object indexing, see “Object Array Indexing” on page 17-15.

**Syntax for subsref and subsasgn Methods**

MATLAB calls the `subsref` and `subsasgn` methods of your class with these arguments.

<table>
<thead>
<tr>
<th>Method</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b = subsref(obj,s)</code></td>
<td>• obj — Object or object array used in indexing expression</td>
<td>b — Result of indexing expression</td>
</tr>
<tr>
<td></td>
<td>• s — Indexing structure</td>
<td></td>
</tr>
<tr>
<td><code>obj = subsasgn(obj,s,b)</code></td>
<td>• obj — Object or object array used in indexing expression</td>
<td>obj — Object or object array after assignment</td>
</tr>
<tr>
<td></td>
<td>• s — Indexing structure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• b — Value being assigned</td>
<td></td>
</tr>
</tbody>
</table>

**Modifying Number of Arguments**

If your class design requires that indexing operations return or assign a different number of values than the number defined by the default indexing operation, overload the
numArgumentsFromSubscript function to control nargout for subsref and nargin for subsasgn. For more information and examples, see numArgumentsFromSubscript.

**Indexing Structure Describes Indexing Expressions**

The indexing structure contains information that describes the indexing expression. Class methods use the information in the indexing structure to evaluate the expression and implement custom behavior.

For example, the CustomIndex class defines a property that you can use in indexing expressions.

```matlab
classdef CustomIndex
    properties
        DataArray
    end
end
```

Create an object and assign a 5-by-5 matrix created by the `magic` function to the `DataArray` property.

```matlab
a = CustomIndex;
a.DataArray = magic(5);
```

This subscripted reference expression returns the first row of the 5-by-5 matrix.

```matlab
a.DataArray(1,:)
an =
    17     24     1     8    15
```

This expression assigns new values to the first row of the array stored in the `DataArray` property.

```matlab
a.DataArray(1,:) = [1 2 3 4 5];
```

This assignment statement uses:

- A `'.'` type reference
- A property name following the dot (that is, `DataArray`)
- A range of indices `(1,:)` within parentheses
The indexing structure contains this information in the type and subs fields.

**Values of the Indexing Structure**

When executing an indexing expression, MATLAB calls the class `subsref` or `subsasgn` method, if the class overloads these functions. One of the arguments passed to the method is the indexing structure. The indexing structure has two fields:

- **type** — One of the three possible indexing types: '. ', '() ', '{} '
- **subs** — A char vector with the property name or cell array of the indices used in the expression, including : and end.

If the indexing expression is a compound expression, then MATLAB passes an array of structures, one struct for each level of indexing. For example, in this expression:

```matlab
a.DataArray(1,:)
```

the indexing structure array `S` has these values:

- `S(1).type` is set to '. ', indicating that the first indexing operation is a dot.
- `S(1).subs` is set to the property name, 'DataArray'

The second level of indexing is in the second element of the indexing structure:

- `S(2).types` is set to '()' indicating the second indexing operation is parentheses indexing
- `S(2).subs` is set to a cell array containing the indices \{[1], [,:]\}

**Typical Patterns for Indexing Methods**

To overload the `subsref` and `subsasgn` functions:

- Determine the full indexing expression using the types and subs fields of the indexing structure.
- Implement the specialized behaviors for the indexing operations supported by the class.
- Return the appropriate values or modified objects in response to the call by MATLAB.

A switch statement is a convenient way to detect the first level of indexing. There are three types of indexing—dot, parentheses, and braces. Each case block in the switch
statement implements all indexing expressions that begin with that first-level type of indexing.

The methods must implement all indexing expressions that the class supports. If you do not customize a particular type of indexing, call the built-in function to handle that expression.

Use the length of the indexing structure array and indexing type define conditional statements for compound indexing expressions.

**Code Framework for `subsref` Method**

The following framework for the `subsref` method shows how to use information in the indexing structure in conditional statements. Your application can involve other expression not shown here.

```matlab
function varargout = subsref(obj,s)
    switch s(1).type
    case '.
        if length(s) == 1
            % Implement obj.PropertyName
        elseif length(s) == 2 && strcmp(s(2).type,'()')
            % Implement obj.PropertyName(indices)
        else
            [varargout{1:nargout}] = builtin('subsref',obj,s);
        end
    case '()
        if length(s) == 1
            % Implement obj(indices)
        elseif length(s) == 2 && strcmp(s(2).type,'.')
            % Implement obj(ind).PropertyName
        elseif length(s) == 3 && strcmp(s(2).type,'.') && strcmp(s(3).type,'()')
            % Implement obj(indices).PropertyName(indices)
        else
            % Use built-in for any other expression
            [varargout{1:nargout}] = builtin('subsref',obj,s);
        end
    case '{}
        if length(s) == 1
            % Implement obj{indices}
        elseif length(s) == 2 && strcmp(s(2).type,'.')
            % Implement obj{indices}.PropertyName
        elseif length(s) == 3 && strcmp(s(2).type,'.') && strcmp(s(3).type,'()')
            % Implement obj{indices}.PropertyName(indices)
        else
            % Use built-in for any other expression
            [varargout{1:nargout}] = builtin('subsref',obj,s);
        end
    case ':'
        if length(s) == 1
            % Implement obj.PropertyName
        elseif length(s) == 2 && strcmp(s(2).type,'()')
            % Implement obj.PropertyName(indices)
        else
            [varargout{1:nargout}] = builtin('subsref',obj,s);
        end
    end
```

17-24
Using `varargout` for the returned value enables the method to work with object arrays. For example, suppose that you want to support the return of a comma-separated list with an expression like this one:

\[ [x_1, \ldots, x_n] = \text{objArray.PropertyName}(\text{Indices}) \]

This expression results in a two-element indexing structure array. The first-level type is dot (\'.\') and the second level is parentheses ('()'). Build the `varargout` cell array with each value in the array.

```matlab
function [varargout{1:nargout}] = builtin('subsref',obj,s);
    end
    otherwise
        error('Not a valid indexing expression')
end
```

**subsasgn Pattern**

The following framework for the `subsasgn` method shows how to use the indexing structure in conditional statements that implement assignment operations.

```matlab
function obj = subsasgn(obj,s,varargin)
    switch s(1).type
        case '.'
            if length(s)==2 && strcmp(s(2).type,'()')
                prop = s(1).subs;          % Property name
                n = numel(obj);             % Number of elements in array
                varargout = cell(1,n);      % Preallocate cell array
                for k = 1:n
                    varargout{k} = obj(k).(prop).(s(2).subs);
                end
            end
        end
    end

subsasgn Pattern
```

```matlab
function obj = subsasgn(obj,s,varargin)
    switch s(1).type
        case '.'
            if length(s) == 1
                % Implement obj.PropertyName = varargin{:};
                ...
            elseif length(s) == 2 && strcmp(s(2).type,'()')
                % Implement obj.PropertyName(indices) = varargin{:};
                ...
            else
                % Call built-in for any other case
                obj = builtin('subsasgn',obj,s,varargin{:});
            end
        case '()'
            if length(s) == 1
```
Using `varargin` for the right-side value of the assignment statement enables the method to work with object arrays. For example, suppose that you want to support the assignment of a comma-separated list with an expression like this one:

```matlab
C = {'one';'two';'three'};
[objArray.PropertyName] = C{:}
```

This expression results in an indexing structure with the dot type ('.') indexing. The cell array `C` on the right side of the assignment statement produces a comma-separated list. This code assigns one list item to each property in the object array.

```matlab
C = {'one';'two';'three'};
[objArray.PropertyName] = C{:}
```
See Also

Related Examples
- “Class with Modified Indexing” on page 17-39
- “Class to Represent Hardware” on page 12-83
- “Subclasses of Built-In Types with Properties” on page 12-68
Indexed Reference

In this section...

“How Indexed Reference Works” on page 17-28
“Compound Indexed References” on page 17-29

How Indexed Reference Works

Object indexed references are in three forms — parentheses, braces, and dot-name:

A(I)
A{I}
A.name

Each of these statements results in a call by MATLAB to the `subsref` of class A, or a call to the built-in `subsasgn` function if the class of A does not implement a `subsasgn` method.

MATLAB passes two arguments to `subsref` and requires `subsref` to return the result of the indexed reference:

\[ B = \text{subsref}(A,S) \]

The first argument is the object being referenced, A. The second argument, S, is a `substruct` with two fields:

- `S.type` is a `char` vector containing `'()'`, '{ }', or '.' specifying the indexing type used.
- `S.subs` is a cell array or `char` vector containing the actual index or name. A colon used as an index is passed in the cell array as the colon character `':'`. Ranges specified using a colon (e.g., `2:5`) are expanded to `2 3 4 5`.

For example, the expression:

\[ A(1:4,:) \]

Causes MATLAB to call `subsref(A,S)`, where S is a 1-by-1 structure with a two-element cell array. The cell array contains the numbers 1, 2, 3, 4, and the colon character `:`.

\[ S.type = '()' \]
\[ S.subs = \{1:4,':'\} \]
Returning the contents of each cell of `S.subs` gives the index values for the first
dimension and a `char` vector `':'` for the second dimension:

```matlab
S.subs{:}
```

```matlab
ans =
    1     2     3     4
ans =
    :
```

The default `subsref` returns all array elements in rows 1 through 4 and all the columns
in the array.

Similarly, this expression:

```matlab
A{1:4}
```

Uses a cell array containing the numbers 1, 2, 3, 4.

```matlab
S.type = '{}'
S.subs = {1:4}
```

The default `subsref` returns the contents of all cell array elements in rows 1 through 4
and all the columns in the array.

This expression:

```matlab
A.Name
```

Calls `subsref(A,S)`, where the `struct` `S` has these values:

```matlab
S.type = '.'
S.subs = 'Name'
```

**Compound Indexed References**

These simple calls are combined for more complicated indexing expressions. In such
cases, `length(S)` is the number of indexing levels. For example,

```matlab
A(1,2).PropertyName(1:4)
```

calls `subsref(A,S)`, where `S` is a 3-by-1 array of `structs` with the values:
S(1).type = '()'  S(2).type = '.'  S(3).type = '()'  
S(1).subs = {1,2}  S(2).subs = 'PropertyName'  S(3).subs = {1:4}

See Also

Related Examples

- “Indexed Assignment” on page 17-31
- “Number of Arguments for subsref and subsasgn” on page 17-6
- “Modify nargout and nargin for Indexing Methods” on page 17-9
- “Code Patterns for subsref and subsasgn Methods” on page 17-21
Indexed Assignment

In this section...

“How Indexed Assignment Works” on page 17-31
“Indexed Assignment to Objects” on page 17-33
“Compound Indexed Assignments” on page 17-33

How Indexed Assignment Works

Object indexed assignments are in three forms — parentheses, braces, and dot-name:

\[
A(I) = B \\
A\{I\} = B \\
A.name = B
\]

Each of these statements results in a call by MATLAB to the \texttt{subsasgn} method of class \texttt{A}, or a call to the built-in \texttt{subsasgn} function if the class of \texttt{A} does not implement a \texttt{subsasgn} method.

MATLAB passes three arguments to \texttt{subsasgn} and requires \texttt{subsasgn} to return the result of the assignment:

\[
A = \texttt{subsasgn}(A,S,B)
\]

The first argument, \texttt{A}, is the object being assigned the value in the third argument \texttt{B}.

The second argument is the indexing structure, \texttt{substruct}. \texttt{S} has two fields:

- \texttt{S.type} is a char vector containing '()', '{}', or '.' specifying the indexing type used.
- \texttt{S.subs} is a cell array or character array containing the actual indices or field name. A colon used as an index is passed in the cell array as the character ':'. Ranges specified using a colon (e.g., 2:5) are expanded to 2 \ 3 \ 4 \ 5.

For example, the assignment statement:

\[
A(2,3) = B;
\]

generates a call to \texttt{subsasgn}:

\[
A = \texttt{subsasgn}(A,S,B)
\]
S contains:

S.type = '()' 
S.subs = {2,3}

The built-in subsasgn:

- Determines the class of A. If B is not the same class as A, then MATLAB tries to construct an object of the same class as A using B as an input argument. If this attempt fails, MATLAB returns an error.
- If A and B are, or can be made, into the same class, then MATLAB assigns the value of B to the array element at row 2, column 3.
- If A does not exist before you execute the assignment statement, then MATLAB initializes the five array elements that come before A(2,3) with default objects of class B.

Similarly, this expression

A{2,3} = B

Uses these values for S:

S.type = '{}'
S.subs = {2,3}

The built-in subsasgn:

- Assigns B to the cell array element at row 2, column 3.
- If A does not exist before you execute the assignment statement, MATLAB initializes the five cells that come before A(2,3) with []. The result is a 2-by-3 cell array.

This expression:

A.Name = B

Calls A = subsasgn(A,S,B) where the struct S has these values:

S.type = '.'
S.subs = 'Name'

The built-in subsasgn:

- Assigns B to the struct field Name.
• If A does not exist before you execute the assignment statement, MATLAB creates a `struct` variable, A with field `Name` and assigns the value of B to this field location.

• If `struct` A exists, but has no field `Name`, then MATLAB adds the field `Name` and assigns the value of B to the new field location.

• If `struct` A exists and has a `Name` field, then MATLAB assigns the value of B to `Name`.

You can redefine all or some of these assignment behaviors by implementing a `subsasgn` method for your class.

**Indexed Assignment to Objects**

If A is an object, this expression:

```matlab
A.Name = B
```

Calls `A = subsasgn(A,S,B)` where, S has these values:

```matlab
S.type = '.
S.subs = 'Name'
```

The default `subsasgn`:

• Attempts to assign B to the `Name` property.

• If the class of A does not have a `Name` property, MATLAB returns an error.

• If the `Name` property has restricted access (private or protected), MATLAB determines if the assignment is allowed based on the context in which the assignment is made.

• If the class of A defines a set method for property `Name`, MATLAB calls the set method.

• MATLAB applies all other property attributes before determining whether to assigning B to the property `Name`.

**Compound Indexed Assignments**

These simple calls are combined for more complicated indexing expressions. In such cases, `length(S)` is the number of indexing levels. For example,

```matlab
A(1,2).PropertyName(1:4) = B
```

calls `subsasgn(A,S,B)`, where S is a 3-by-1 array of structures with the values:
S(1).type = '()'    S(2).type = '.'              S(3).type = '()'
S(1).subs = {1,2}   S(2).subs = 'PropertyName'   S(3).subs = {1:4}

See Also

Related Examples

•    “Indexed Reference” on page 17-28
•    “Number of Arguments for subsref and subsasgn” on page 17-6
•    “Modify nargout and nargin for Indexing Methods” on page 17-9
•    “Code Patterns for subsref and subsasgn Methods” on page 17-21
Define end Indexing for an Object

When you use end in an object indexing expression, such as \texttt{A(4:end)}, the end function returns the index value corresponding to the last element in that dimension.

Classes can overload the end function to implement specialized behavior. If your class defines an end method, MATLAB calls that method to determine how to interpret the expression.

The end method has the calling syntax:

\[
\text{ind} = \text{end}(A, k, n)
\]

The arguments are described as follows:

- \texttt{A} is the object
- \texttt{k} is the index in the expression using the end syntax
- \texttt{n} is the total number of indices in the expression
- \texttt{ind} is the index value to use in the expression

For example, consider the 3-by-5 array \texttt{A}. When MATLAB encounters the expression:

\[
A(\text{end}-1,:) 
\]

MATLAB calls the end method defined for the object \texttt{A} using the arguments:

\[
\text{ind} = \text{end}(A, 1, 2)
\]

These arguments mean that the end statement occurs in the first index and there are two indices. The end class method returns the index value for the last element of the first dimension (from which 1 is subtracted in this case). The original expression is evaluated as:

\[
A(3-1,:) 
\]
If your class implements an end method, ensure that it returns a value appropriate for the class.

**The end Method**

The end method for the `MyDataClass` example (see “Class with Modified Indexing” on page 17-39) operates on the contents of the `Data` property. The objective of this method is to return a value that can replace `end` in any indexing expression, such as:

```matlab
obj(4:end)
obj.Data(2,3:end)
```

This end method determines a positive integer value for `end`. The method returns the value so that MATLAB can use it in the indexing expression.

```matlab
function ind = end(obj,k,n)
    szd = size(obj.Data);
    if k < n
        ind = szd(k);
    else
        ind = prod(szd(k:end));
    end
end
```

**See Also**

**Related Examples**

- “Class with Modified Indexing” on page 17-39
- “Objects in Index Expressions” on page 17-37
Objects in Index Expressions

In this section...

“Objects Indexes” on page 17-37
“Ways to Implement Objects as Indices” on page 17-37
“subsindex Implementation” on page 17-37

Objects Indexes

MATLAB can use objects as indices in indexed expressions. The rules of array indexing apply — indices must be positive integers. Therefore, MATLAB must be able to derive a value from the object that is a positive integer for use in the indexed expression.

Indexed expressions like \( X(A) \), where \( A \) is an object, cause MATLAB to call the \texttt{subsindex} function. However, if an indexing expression results in a call to an overloaded \texttt{subsref} or \texttt{subsasgn} method defined by the class of \( X \), then MATLAB does not call \texttt{subsindex}.

Ways to Implement Objects as Indices

There are several ways to implement indexing of one object by another object, \( X(A) \):

- Define a \texttt{subsindex} method in the class of \( A \) that converts \( A \) to an integer. MATLAB calls \( A \)'s \texttt{subsindex} method to perform indexing operations when the class of \( X \) does not overload \texttt{subsref} or \texttt{subsasgn}.
- If the class of \( X \) overloads \texttt{subsref} or \texttt{subsasgn}, these methods can call the \texttt{subsindex} method of \( A \) explicitly. The class of \( A \) must implement a \texttt{subsindex} method that returns an appropriate value.
- If the class of \( X \) overloads \texttt{subsref} or \texttt{subsasgn}, these methods can contain code that determines an integer index value. In this case, the class of \( A \) does not implement a \texttt{subsindex} method.

subsindex Implementation

\texttt{subsindex} must return the value of the object as a zero-based integer index value in the range 0 to \texttt{prod(size(X))-1}. 
Suppose that you want to use object A to index into object B. B can be a single object or an array, depending on the class designs.

\[ C = B(A); \]

Here are two examples of `subsindex` methods. The first assumes you can convert class A to a `uint8`. The second assumes class A stores an index value in a property.

- The `subsindex` method implemented by class A can convert the object to numeric format to be used as an index:

  ```matlab
  function ind = subsindex(obj)
  ind = uint8(obj);
  end
  ```

  The class of `obj` implements a `uint8` method to provide the conversion from the object to an integer value.

- Class A implements `subsindex` to return a numeric value that is stored in a property:

  ```matlab
  function ind = subsindex(obj)
  ind = obj.ElementIndex;
  end
  ```

**Note** `subsindex` values are 0-based, not 1-based.

**See Also**

`numArgumentsFromSubscript` | `subsasgn` | `subsref`

**Related Examples**

- “end as Object Index” on page 17-35

**More About**

- “Modify nargout and nargin for Indexing Methods” on page 17-9
Class with Modified Indexing

In this section...

“How to Modify Class Indexing” on page 17-39
“Class Description” on page 17-39
“Specialize Subscripted Reference — subsref” on page 17-40
“Specialize Subscripted Assignment — subsasgn” on page 17-42
“Implement Addition for Object Data — double and plus” on page 17-43
“MyDataClass.m” on page 17-44

How to Modify Class Indexing

This example defines a class that modifies the default indexing behavior by implementing subsref and subsasgn methods. The class also implements type conversion and addition by implementing a double converter method and a plus method.

The objective of the class design is to:

• Enable you to treat an object of the class as a numeric array
• Be able to contain nonnumeric and numeric data in an object of the class

Class Description

The class has three properties:

• Data — numeric test data
• Description — description of test data
• Date — date test was conducted

Assume that you have the following random data (randi):

```matlab
d = randi(9,3,4)
d =
```

```plaintext
8  9  3  9
9  6  5  2
2  1  9  9
```
Create an instance of the class:

```matlab
obj = MyDataClass(d, 'Test001')
```

```matlab
obj =

MyDataClass with properties:

   Data: [3x4 double]
   Description: 'Test001'
   Date: [2012 1 7 9 32 34.5190]
```

The constructor arguments pass the values for the Data and Description properties. The `clock` function assigns the value to the Date property from within the constructor. This approach captures the time and date information when each instance is created.

Here is the preliminary code listing without the `subsref`, `subsasgn` double, and plus methods.

```matlab
classdef MyDataClass
    properties
        Data
        Description
    end
    properties (SetAccess = private)
        Date
    end
    methods
        function obj = MyDataClass(data, desc)
            if nargin > 0
                obj.Data = data;
            end
            if nargin > 1
                obj.Description = desc;
            end
            obj.Date = clock;
        end
    end
end
```

**Specialize Subscripted Reference — subsref**

Implement a `subsref` method to support both the default and a specialized type of indexing.
• The default indexed reference behavior for scalar objects:

```matlab
obj.Data(2,3)
```

```matlab
ans =

5
```

• And to add the functionality to index into the `Data` property with an expression like this statement:

```matlab
obj(2,3)
```

If you redefine '()' indexing to support access to the `Data` property, you cannot create arrays of `MyDataClass` objects and use '()' indexing to access individual objects. You can reference only scalar objects.

To achieve the design goals, the `subsref` method must handle all indexing types. The `subsref` method:

• Calls the `builtin` `subsref` function for '.' indexing
• Returns an error for '{}' indexing
• Defines its own version of '()' indexing.

The result: `obj(i)` is equivalent to `obj.Data(i)`.

```matlab
function sref = subsref(obj,s)
    % obj(i) is equivalent to obj.Data(i)
    switch s(1).type
    case '.
        sref = builtin('subsref',obj,s);
    case '()'
        if length(s) < 2
            sref = builtin('subsref',obj.Data,s);
            return
        else
            sref = builtin('subsref',obj,s);
        end
    case '{}'
        error('MYDataClass:subsref','...
            'Not a supported subscripted reference')
    end
end
```
Specialize Subscripted Assignment — subsasgn

To support the equivalent of the indexed reference behavior with indexed assignment, implement a subsasgn method.

- Support the default indexed assignment:

  \[
  \text{obj.Data(2,3) = 9;}
  \]

- Add the functionality to assign values to the Data property with an expression like this statement:

  \[
  \text{obj(2,3) = 9;}
  \]

Like the subsref method, the subsasgn method:

- Calls the builtin subsasgn function for '. ' indexing
- Returns an error for '{} indexing
- Defines its own version of '()' indexing.

The substruct function redefines the index type and index subscripts structure that MATLAB passes to subsref and subsasgn.

```matlab
function obj = subsasgn(obj,s,val)
    if isempty(s) && isa(val,'MyDataClass')
        obj = MyDataClass(val.Data,val.Description);
    end
    switch s(1).type
        case '. '
        obj = builtin('subsasgn',obj,s,val);
        case '()'
            if length(s)<2
                if isa(val,'MyDataClass')
                    error('MyDataClass:subsasgn',...'
                        'Object must be scalar')
                elseif isa(val,'double')
                    % Redefine the struct s to make the call: obj.Data(i)
                    snew = substruct('.','Data','()',s(1).subs(:));
                    obj = subsasgn(obj,snew,val);
                end
            end
        case '{}'
            error('MyDataClass:subsasgn',...'
                Not a supported subscripted assignment')
    end
end
```
Implement Addition for Object Data — double and plus

First, implement a `double` method that converts an object to an array of doubles. By implementing a `double` converter method, it is possible to add a `MyDataClass` object to another class of object. However, the other class must implement a `double` method that also returns an array of doubles. For more information on type conversion, see “Object Converters” on page 17-12.

Allow direct addition of the `Data` property values by implementing a `plus` method. Implementing a `plus` method enables the use of the `+` operator for addition of `MyDataClass` objects.

Because the `plus` method implements addition by adding double arrays, MATLAB:

- Apply the rules of addition when adding `MyDataClass` objects
- Returns errors for any condition that can cause errors in default numeric addition. For example, dimension mismatch.

The `plus` method uses the `double` method to convert the object to numeric values before performing the addition:

```matlab
function a = double(obj)
    a = obj.Data;
end

function c = plus(obj,b)
    c = double(obj) + double(b);
end
```

For example, the `plus` method enables you to add a scalar number to the object `Data` array.

Here are the values of the `Data`, displayed using indexed reference:

```matlab
obj(:,:,:)
ans =
     8     9     3     9
```
Add 7 to the array contained in the Data property:

\[ \text{obj} + 7 \]

\[ \text{ans} = \]

\[
\begin{bmatrix}
15 & 16 & 10 & 16 \\
16 & 13 & 16 & 9 \\
9 & 8 & 16 & 16
\end{bmatrix}
\]

**MyDataClass.m**

This definition for MyDataClass includes the end indexing method discussed in “end as Object Index” on page 17-35.

```matlab
classdef MyDataClass
    % Example for "A Class with Modified Indexing"
    properties
        Data
        Description
    end
    properties (SetAccess = private)
        Date
    end
    methods
        function obj = MyDataClass(data,desc)
            % Support 0-2 args
            if nargin > 0
                obj.Data = data;
            end
            if nargin > 1
                obj.Description = desc;
            end
            obj.Date = clock;
        end
        function sref = subsref(obj,s)
            % obj(i) is equivalent to obj.Data(i)
            switch s(1).type
                case '.'
                    sref = builtin('subsref',obj,s);
            end
    end
end
```
case '()'
    if length(s)<2
        sref = builtin('subsref',obj.Data,s);
        return
    else
        sref = builtin('subsref',obj,s);
    end
case '{}'
    error('MyDataClass:subsref',...'
        Not a supported subscripted reference')
end
double(obj) = obj.Data;
plus(obj,b) = double(obj) + double(b);
function ind = end(obj,k,n)
    szd = size(obj.Data);
    if k < n
        ind = szd(k);
    else
        ind = prod(szd(k:end));
    end
end

See Also

Related Examples
- “end as Object Index” on page 17-35
- “Number of Arguments for subsref and subsasgn” on page 17-6
Operator Overloading

Why Overload Operators

By implementing operators that are appropriate for your class, you can integrate objects of your class into the MATLAB language. For example, objects that contain numeric data can define arithmetic operations like +, *, - so that you can use these objects in arithmetic expressions. By implementing relational operators, you can use objects in conditional statements, like switch and if statements.

How to Define Operators

You can implement MATLAB operators to work with objects of your class. To implement operators, define the associated class methods.

Each operator has an associated function (e.g., the + operator has an associated plus.m function). You can implement any operator by creating a class method with the appropriate name. This method can perform whatever steps are appropriate for the operation being implemented.

For a list of operators and associated function names, see “MATLAB Operators and Associated Functions” on page 17-50.

Object Precedence in Operations

User-defined classes have a higher precedence than built-in classes. For example, suppose q is an object of class double and p is a user-defined class. Both of these expressions generate a call to the plus method in the user-defined class, if it exists:

\[
\begin{align*}
q + p \\
p + q
\end{align*}
\]
Whether this method can add objects of class `double` and the user-defined class depends on how you implement the method.

When `p` and `q` are objects of different classes, MATLAB applies the rules of precedence to determine which method to use.

“Object Precedence in Method Invocation” on page 9-46 provides information on how MATLAB determines which method to call.

**Operator Precedence**

Overloaded operators retain the original MATLAB precedence for the operator. For information on operator precedence, see “Operator Precedence”.

**Sample Implementation — Addable Objects**

The `Adder` class implements addition for objects of this class by defining a `plus` method. `Adder` defines addition of objects as the addition of the `NumericData` property values. The `plus` method constructs and returns an `Adder` object whose `NumericData` property value is the result of the addition.

The `Adder` class also implements the less than operator (`<`) by defining a `lt` method. The `lt` method returns a logical value after comparing the values in each object `NumericData` property.

```matlab
classdef Adder
    properties
        NumericData
    end
    methods
        function obj = Adder(val)
            obj.NumericData = val;
        end
        function r = plus(obj1,obj2)
            a = double(obj1);
            b = double(obj2);
            r = Adder(a + b);
        end
        function d = double(obj)
            d = obj.NumericData;
        end
        function tf = lt(obj1,obj2)
            if obj1.NumericData < obj2.NumericData
```
Using a double converter enables you to add numeric values to `Adder` objects and to perform addition on objects of the class.

```
a = Adder(1:10)
a =
    Adder with properties:
        NumericData: [1 2 3 4 5 6 7 8 9 10]
```

Add two objects:
```
a + a
ans =
    Adder with properties:
        NumericData: [2 4 6 8 10 12 14 16 18 20]
```

Add an object with any value that can be cast to double:
```
b = uint8(255) + a
b =
    Adder with properties:
        NumericData: [256 257 258 259 260 261 262 263 264 265]
```

Compare objects `a` and `b` using the `<` operator:
```
a < b
ans =
    1
```
Ensure that your class provides any error checking required to implement your class design.

**MATLAB Operators and Associated Functions**

The following table lists the function names for MATLAB operators. Implementing operators to work with arrays (scalar expansion, vectorized arithmetic operations, and so on), can also require modifying indexing and concatenation. Use the links in this table to find specific information on each function.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Method to Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>plus(a,b)</td>
<td>Binary addition</td>
</tr>
<tr>
<td>a - b</td>
<td>minus(a,b)</td>
<td>Binary subtraction</td>
</tr>
<tr>
<td>-a</td>
<td>uminus(a)</td>
<td>Unary minus</td>
</tr>
<tr>
<td>+a</td>
<td>uplus(a)</td>
<td>Unary plus</td>
</tr>
<tr>
<td>a.*b</td>
<td>times(a,b)</td>
<td>Element-wise multiplication</td>
</tr>
<tr>
<td>a*b</td>
<td>mtimes(a,b)</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>a./b</td>
<td>rdivide(a,b)</td>
<td>Right element-wise division</td>
</tr>
<tr>
<td>a.</td>
<td>ldivide(a,b)</td>
<td>Left element-wise division</td>
</tr>
<tr>
<td>a/b</td>
<td>mrdivide(a,b)</td>
<td>Matrix right division</td>
</tr>
<tr>
<td>a</td>
<td>mldivide(a,b)</td>
<td>Matrix left division</td>
</tr>
<tr>
<td>a.^b</td>
<td>power(a,b)</td>
<td>Element-wise power</td>
</tr>
<tr>
<td>a^b</td>
<td>mpower(a,b)</td>
<td>Matrix power</td>
</tr>
<tr>
<td>a &lt; b</td>
<td>lt(a,b)</td>
<td>Less than</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>gt(a,b)</td>
<td>Greater than</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>le(a,b)</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>ge(a,b)</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>a ~= b</td>
<td>ne(a,b)</td>
<td>Not equal to</td>
</tr>
<tr>
<td>a == b</td>
<td>eq(a,b)</td>
<td>Equality</td>
</tr>
<tr>
<td>a &amp; b</td>
<td>and(a,b)</td>
<td>Logical AND</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>or(a,b)</td>
</tr>
<tr>
<td>~a</td>
<td>not(a)</td>
<td>Logical NOT</td>
</tr>
<tr>
<td>Operation</td>
<td>Method to Define</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>a:d:b</td>
<td>colon(a,d,b)</td>
<td>Colon operator</td>
</tr>
<tr>
<td>a:b</td>
<td>colon(a,b)</td>
<td></td>
</tr>
<tr>
<td>a'</td>
<td>ctranspose(a)</td>
<td>Complex conjugate transpose</td>
</tr>
<tr>
<td>a.'</td>
<td>transpose(a)</td>
<td>Matrix transpose</td>
</tr>
<tr>
<td>[a b]</td>
<td>horzcat(a,b,...)</td>
<td>Horizontal concatenation</td>
</tr>
<tr>
<td>[a; b]</td>
<td>vertcat(a,b,...)</td>
<td>Vertical concatenation</td>
</tr>
<tr>
<td>a(s1,s2,...sn)</td>
<td>subsref(a,s)</td>
<td>Subscripted reference</td>
</tr>
<tr>
<td>a(s1,...,sn) = b</td>
<td>subsasgn(a,s,b)</td>
<td>Subscripted assignment</td>
</tr>
<tr>
<td>b(a)</td>
<td>subsindex(a)</td>
<td>Subscript index</td>
</tr>
</tbody>
</table>

**See Also**

**Related Examples**
- “Define Arithmetic Operators” on page 19-21
- “Methods That Modify Default Behavior” on page 17-2
Customizing Object Display

- “Custom Display Interface” on page 18-2
- “How CustomDisplay Works” on page 18-7
- “Role of size Function in Custom Displays” on page 18-10
- “Customize Display for Heterogeneous Arrays” on page 18-12
- “Class with Default Object Display” on page 18-14
- “Choose a Technique for Display Customization” on page 18-19
- “Customize Property Display” on page 18-23
- “Customize Header, Property List, and Footer” on page 18-26
- “Customize Display of Scalar Objects” on page 18-32
- “Customize Display of Object Arrays” on page 18-37
- “Overloading the disp Function” on page 18-42
Custom Display Interface

In this section...

“Default Object Display” on page 18-2
“CustomDisplay Class” on page 18-3
“Methods for Customizing Object Display” on page 18-3

Default Object Display

MATLAB adds default methods named disp and display to all MATLAB classes that do not implement their own methods with those names. These methods are not visible, but create the default simple display.

The default simple display consists of the following parts:

• A header showing the class name, and the dimensions for nonscalar arrays.
• A list of all nonhidden public properties, shown in the order of definition in the class.

The actual display depends on whether the object is scalar or nonscalar. Also, there are special displays for a scalar handle to a deleted object and empty object arrays. Objects in all of these states are displayed differently if the objects have no properties.

The details function creates the default detailed display. The detailed display adds these items to the simple display:

• Use of fully qualified class names
• Link to handle class, if the object is a handle
• Links to methods, events, and superclasses functions executed on the object.

See “Class with Default Object Display” on page 18-14 for an example of how MATLAB displays objects.

Properties Displayed by Default

MATLAB displays object properties that have public get access and are not hidden (see “Property Attributes” on page 8-9). Inherited abstract properties are excluded from display. When the object being displayed is scalar, any dynamic properties attached to the object are also included.
CustomDisplay Class

The `matlab.mixin.CustomDisplay` class provides an interface that you can use to customize object display for your class. To use this interface, derive your class from `CustomDisplay`:

```matlab
classdef MyClass < matlab.mixin.CustomDisplay
```

The `CustomDisplay` class is `HandleCompatible`, so you can use it in combination with both value and handle superclasses.

**Note** You cannot use `matlab.mixin.CustomDisplay` to derive a custom display for enumeration classes.

disp, display, and details

The `CustomDisplay` interface does not allow you to override `disp`, `display`, and `details`. Instead, override any combination of the customization methods defined for this purpose.

Methods for Customizing Object Display

There are two groups of methods that you use to customize object display for your class:

- Part builder methods build the strings used for the standard display. Override any of these methods to change the respective parts of the display.
- State handler methods are called for objects in specific states, like scalar, nonscalar, and so on. Override any of these methods to handle objects in a specific state.

All of these methods have protected access and must be defined as protected in your subclass of `CustomDisplay` (that is, `Access = protected`).

Parts of an Object Display

There are three parts that makeup the standard object display — header, property list, and footer

For example, here is the standard object display for a `containers.Map` object:
The default object display does not include a footer. The detailed display provides more information:

```plaintext
>> map1 = containers.Map({'Apr', 'Jul', 'Nov'}, [4, 7, 11])

map1 =

Map with properties:

    Count: 3
    KeyType: char
    ValueType: double

Method | Purpose | Default
-------|---------|---------
getHeader | Create the text used for the header. | Returns the char vectors, [class(obj), ' with properties:']] linking the class name to a help popup
getPropertyGroups | Define how and what properties display, including order, values, and grouping. | Returns an array of PropertyGroup objects, which determines how to display the properties
```

**Part Builder Methods**

Each part of the object display has an associated method that assembles the respective part of the display.
### Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFooter</td>
<td>Create the text used for the footer.</td>
<td>There are two footers:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Simple display — Returns an empty <code>char</code> vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Detailed display — Returns linked calls to methods, events, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>superclasses for this class</td>
</tr>
</tbody>
</table>

### Object States That Affect Display

There are four object states that affect how MATLAB displays objects:

- Valid scalar object
- Nonscalar object array
- Empty object array
- Scalar handle to a deleted object

### State Handler Methods

Each object state has an associated method that MATLAB calls whenever displaying objects that are in that particular state.

<table>
<thead>
<tr>
<th>State Handler Method</th>
<th>Called for Object in This State</th>
</tr>
</thead>
<tbody>
<tr>
<td>displayScalarObject</td>
<td><code>(isa(obj,'handle') &amp;&amp; isvalid(obj)) &amp;&amp; prod(size(obj)) == 1</code></td>
</tr>
<tr>
<td>displayNonScalarObject</td>
<td><code>prod(size(obj)) &gt; 1</code></td>
</tr>
<tr>
<td>displayEmptyObject</td>
<td><code>prod(size(obj)) == 0</code></td>
</tr>
<tr>
<td>displayScalarHandleToDeletedObject</td>
<td><code>isa(obj,'handle') &amp;&amp; isscalar(obj) &amp;&amp; ~isvalid(obj)</code></td>
</tr>
</tbody>
</table>

### Utility Methods

The `CustomDisplay` class provides utility methods that return strings that are used in various parts of the different display options. These static methods return text that simplifies the creation of customized object displays.

If the computer display does not support hypertext linking, the strings are returned without the links.
### See Also

#### Related Examples
- “How CustomDisplay Works” on page 18-7
How CustomDisplay Works

Steps to Display an Object

When displaying an object, MATLAB determines the state of the object and calls the appropriate method for that state (see “Object States That Affect Display” on page 18-5).

For example, suppose obj is a valid scalar object of a class derived from CustomDisplay. If you type obj at the command line without terminating the statement with a semicolon:

```matlab
>> obj
```

The following sequence results in the display of obj:

1. MATLAB determines the class of obj and calls the disp method to display the object.
2. disp calls size to determine if obj is scalar or nonscalar.
3. When obj is a scalar handle object, disp calls isValid to determine if obj is the handle of a deleted object. Deleted handles in nonscalar arrays do not affect the display.
4. disp calls the state handler method for an object of the state of obj. In this case, obj is a valid scalar that results in a call to:
   ```matlab
displayScalarObject(obj)
   ```
5. displayScalarObject calls the display part-builder methods to provide the respective header, property list, and footer.

```matlab
... header = getHeader(obj); disp(header)
...
... groups = getPropertyGroups(obj) displayPropertyGroups(obj,groups) ...
... footer = getFooter disp(footer)
```
MATLAB follows a similar sequence for nonscalar object arrays and empty object arrays.

In the case of scalar handles to deleted objects, `disp` calls the `displayScalarHandleToDelete` method, which displays the default text for handles to deleted objects without calling any part-builder methods.

**Methods Called for a Given Object State**

The following diagram illustrates the methods called to display an object that derives from `CustomDisplay`. The `disp` method calls the state handler method that is appropriate for the state of the object or object array being displayed.

Only an instance of a handle class can be in a state of scalar handle to a deleted object.
See Also

Related Examples

- “Class with Default Object Display” on page 18-14
Role of size Function in Custom Displays

In this section...

“How size Is Used” on page 18-10
“Precautions When Overloading size” on page 18-10

How size Is Used

In the process of building the custom display, CustomDisplay methods call the size function at several points:

- `disp` calls `size` to determine which state handler method to invoke.
- The default `getHeader` method calls `size` to determine whether to display a scalar or nonscalar header.
- The default `displayPropertyGroups` method calls `size` to determine if it should look up property values when the property group is a cell array of property names. By default, only scalar objects display the values of properties.

Precautions When Overloading size

If your class overloads the `size` function, then MATLAB calls the overloading version. You must ensure that the implementation of `size` is consistent with the way you want to display objects of the class.

An unusual or improper implementation of `size` can result in undesirable display behavior. For example, suppose a class overloads `size` reports an object as scalar when it is not. In this class, a property list consisting of a cell array of strings results in the property values of the first object of the array being displayed. This behavior can give the impression that all objects in the array have the same property values.

However, reporting an object as scalar when in fact the object is empty results in the object displaying as an empty object array. The default methods of the CustomDisplay interface always determine if the input is an empty array before attempting to access property values.

As you override CustomDisplay methods to implement your custom object display, consider how an overloading size method can affect the result.
See Also

Related Examples
• “Methods That Modify Default Behavior” on page 17-2
Customize Display for Heterogeneous Arrays

You can call only sealed methods on nonscalar heterogeneous arrays. If you want to customize classes that are part of a heterogeneous hierarchy, you must override and declare as Sealed all the methods that are part of the CustomDisplay interface.

The versions of disp and display that are inherited from matlab.mixin.CustomDisplay are sealed. However, these methods call all of the part builder (“Part Builder Methods” on page 18-4) and state handler methods (“State Handler Methods” on page 18-5).

To use the CustomDisplay interface, the root class of the heterogeneous hierarchy can declare these methods as Sealed and Access = protected.

If you do not need to override a particular method, then call the superclass method, as shown in the following code.

For example, the following code shows modifications to the getPropertyGroups and displayScalarObject methods, while using the superclass implementation of all others.

```matlab
classdef RootClass < matlab.mixin.CustomDisplay & matlab.mixin.Heterogeneous
    %...
    methods (Sealed, Access = protected)
        function header = getHeader(obj)
            header = getHeader@matlab.mixin.CustomDisplay(obj);
        end

        function groups = getPropertyGroups(obj)
            % Override of this method
            % ...
        end

        function footer = getFooter(obj)
            footer = getFooter@matlab.mixin.CustomDisplay(obj);
        end

        function displayNonScalarObject(obj)
            displayNonScalarObject@matlab.mixin.CustomDisplay(obj);
        end

        function displayScalarObject(obj)
            % Override of this method
            % ...
        end

        function displayEmptyObject(obj)
            displayEmptyObject@matlab.mixin.CustomDisplay(obj);
        end
end
```

function displayScalarHandleToDeletedObject(obj)
    displayScalarHandleToDeletedObject@matlab.mixin.CustomDisplay(obj);
end
end
end

You do not need to declare the inherited static methods as Sealed.

See Also

Related Examples
- “Designing Heterogeneous Class Hierarchies” on page 10-24
Class with Default Object Display

In this section...

"The EmployeeInfo Class" on page 18-14
"Default Display — Scalar" on page 18-15
"Default Display — Nonscalar" on page 18-15
"Default Display — Empty Object Array" on page 18-16
"Default Display — Handle to Deleted Object" on page 18-17
"Default Display — Detailed Display" on page 18-17

The EmployeeInfo Class

The EmployeeInfo class defines a number of properties to store information about company employees. This simple class serves as the example class used in display customization sample classes.

EmployeeInfo derives from the matlab.mixin.CustomDisplay class to enable customization of the object display.

EmployeeInfo is also a handle class. Therefore instances of this class can be in the state referred to as a handle to a deleted object. This state does not occur with value classes (classes not derived from handle).

classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
    properties
        Name
        JobTitle
        Department
        Salary
        Password
    end
    methods
        function obj = EmployeeInfo
            obj.Name       = input('Name: ');
            obj.JobTitle   = input('Job Title: ');
            obj.Department = input('Department: ');
            obj.Salary     = input('Salary: ');
            obj.Password   = input('Password: ');
        end
end
The `matlab.mixin.CustomDisplay` is handle compatible. Therefore, superclasses can be either handle or value classes.

**Default Display — Scalar**

Here is the creation and display of a scalar `EmployeeInfo` object. By default, MATLAB displays properties and their values for scalar objects.

Provide inputs for the constructor:

```matlab
>> Emp123 = EmployeeInfo;
Name: 'Bill Tork'
Job Title: 'Software Engineer'
Department: 'Product Development'
Salary: 1000
Password: 'bill123'
```

Display the object:

```matlab
>> Emp123
```

```matlab
Emp123 =

EmployeeInfo with properties:

   Name: 'Bill Tork'
   JobTitle: 'Software Engineer'
   Department: 'Product Development'
   Salary: 1000
   Password: 'bill123'
```

**Testing for Scalar Objects**

To test for scalar objects, use `isscalar`.

**Default Display — Nonscalar**

The default display for an array of objects does not show property values. For example, concatenating two `EmployeeInfo` objects generates this display:
>>[Emp123,Emp124]
ans

    1x2 EmployeeInfo array with properties:

            Name
            JobTitle
            Department
            Salary
            Password

Testing for Nonscalar Objects

To test for nonscalar objects, use a negated call to isscalar.

Default Display — Empty Object Array

An empty object array has at least one dimension equal to zero.

>> Empt = EmployeeInfo.empty(0,5)

Empt =

    0x5 EmployeeInfo array with properties:

            Name
            JobTitle
            Department
            Salary
            Password

Testing for Empty Object Arrays

Use isempty to test for empty object arrays. An empty object array is not scalar because its dimensions can never be 1-by-1.

>> emt = EmployeeInfo.empty

emt =

    0x0 EmployeeInfo array with properties:

            Name
            JobTitle
Default Display — Handle to Deleted Object

When a handle object is deleted, the handle variable can remain in the workspace.

```matlab
>> delete(Emp123)
>> Emp123
Emp123 =
    handle to deleted EmployeeInfo
```

Testing for Handles to Deleted Objects

To test for a handle to a deleted object, use `isvalid`.

**Note** `isvalid` is a handle class method. Calling `isvalid` on a value class object causes an error.

Default Display — Detailed Display

The details method does not support customization and always returns the standard detailed display:

```matlab
details(Emp123)
EmployeeInfo handle with properties:

    Name: 'Bill Tork'
    JobTitle: 'Software Engineer'
    Department: 'Product Development'
    Salary: 1000
    Password: 'bill123'
```

Methods, Events, Superclasses
See Also

Related Examples

•  “Custom Display Interface” on page 18-2
Choose a Technique for Display Customization

In this section...

“Ways to Implement a Custom Display” on page 18-19
“Sample Approaches Using the Interface” on page 18-20

Ways to Implement a Custom Display

The way you customize object display using the `matlab.mixin.CustomDisplay` class depends on:

- What parts of the display you want to customize
- What object states you want to use the custom display

If you are making small changes to the default layout, then override the relevant part builder methods ("Part Builder Methods" on page 18-4). For example, suppose you want to:

- Change the order or value of properties, display a subset of properties, or create property groups
- Modify the header text
- Add a footer

If you are defining a nonstandard display for a particular object state (scalar, for example), then the best approach is to override the appropriate state handler method ("State Handler Methods" on page 18-5).

In some cases, a combination of method overrides might be the best approach. For example, your implementation of `displayScalarObject` might

- Use some of the utility methods ("Utility Methods" on page 18-5) to build your own display strings using parts from the default display
- Call a part builder method to get the default text for that particular part of the display
- Implement a completely different display for scalar objects.

Once you override any `CustomDisplay` method, MATLAB calls your override in all cases where the superclass method would have been called. For example, if you override the
getHeader method, your override must handle all cases where a state handler method calls getHeader. (See “Methods Called for a Given Object State” on page 18-8)

**Sample Approaches Using the Interface**

Here are some simple cases that show what methods to use for the particular customized display.

**Change the Display of Scalar Objects**

Use a nonstandard layout for scalar object display that is fully defined in the displayScalarObject method:

```matlab
classdef MyClass < matlab.mixin.CustomDisplay
    ...
    methods (Access = protected)
        function displayScalarObject(obj)
            % Implement the custom display for scalar obj
            end
    end
end
classdef MyClass < matlab.mixin.CustomDisplay
    properties
        Prop1
end
```

**Custom Property List with Standard Layout**

Use standard display layout, but create a custom property list for scalar and nonscalar display:

```matlab
classdef MyClass < matlab.mixin.CustomDisplay
    ...
    methods (Access = protected)
        function groups = getPropertyGroups(obj)
            % Return PropertyGroup instances
            end
    end
end
classdef MyClass < matlab.mixin.CustomDisplay
    properties
        ...
    end
```

**Custom Property List for Scalar Only**

Use standard display layout, but create a custom property list for scalar only. Call the superclass getPropertyGroups for the nonscalar case:

```matlab
classdef MyClass < matlab.mixin.CustomDisplay
    properties
        ...
    end
```
function groups = getPropertyGroups(obj)
if isscalar(obj)
    % Scalar case: change order
    propList = {'Prop2','Prop1','Prop3'};
    groups = matlab.mixin.util.PropertyGroup(propList)
else
    % Nonscalar case: call superclass method
    groups = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
end
end

Custom Property List with Modified Values

Change the values displayed for some properties in the scalar case by creating property/value pairs in a struct. This getPropertyGroups method displays only Prop1 and Prop2, and displays the value of Prop2 as Prop1 divided by Prop3.

classdef MyClass < matlab.mixin.CustomDisplay
    properties
        Prop1
        Prop2
        Prop3
    end
    methods(Access = protected)
        function groups = getPropertyGroups(obj)
            if isscalar(obj)
                % Specify the values to be displayed for properties
                propList = struct('Prop1',obj.Prop1,...
                                  'Prop2',obj.Prop1/obj.Prop3);
                groups = matlab.mixin.util.PropertyGroup(propList)
            else
                % Nonscalar case: call superclass method
                groups =getPropertyGroups@matlab.mixin.CustomDisplay(obj);
            end
        end
    end
end

Complete Class Definitions

For complete class implementations, see these sections:

- “Customize Property Display” on page 18-23
- “Customize Header, Property List, and Footer” on page 18-26
Customize Property Display

**Objective**

Change the order and number of properties displayed for an object of your class.

**Change the Property Order**

Suppose your class definition contains the following property definition:

```matlab
properties
    Name
    JobTitle
    Department
    Salary
    Password
end
```

In the default scalar object display, MATLAB displays all the public properties along with their values. However, you want to display only Department, JobTitle, and Name, in that order. You can do this by deriving from CustomDisplay and overriding the `getPropertyGroups` method.

Your override

- Defines method `Access` as `protected` to match the definition in the `CustomDisplay` superclass
- Creates a cell array of property names in the desired order
- Returns a `PropertyGroup` object constructed from the property list cell array

```matlab
methods (Access = protected)
    function propgrp = getPropertyGroups(~)
        proplist = {'Department','JobTitle','Name'};
        propgrp = matlab.mixin.util.PropertyGroup(proplist);
end
```
When you create a PropertyGroup object using a cell array of property names, MATLAB automatically

- Adds the property values for a scalar object display
- Uses the property names without values for a nonscalar object display (including empty object arrays)

The `getPropertyGroups` method is not called to create the display for a scalar handle to a deleted object.

### Change the Values Displayed for Properties

Given the same class properties used in the previous section, you can change the value displayed for properties by building the property list as a struct and specifying values for property names. This override of the `getPropertyGroups` method uses the default property display for nonscalar objects by calling the superclass `getPropertyGroups` method. For scalar objects, the override:

- Changes the value displayed for the `Password` property to a '*' character for each character in the password.
- Displays the text 'Not Available' for the `Salary` property.

```matlab
methods (Access = protected)
    function propgrp = getPropertyGroups(obj)
        if ~isscalar(obj)
            propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
        else
            pd(1:length(obj.Password)) = '*';
            propList = struct('Department',obj.Department,...
                               'JobTitle',obj.JobTitle,...
                               'Name',obj.Name,...
                               'Salary','Not available',...
                               'Password',pd);
            propgrp = matlab.mixin.util.PropertyGroup(propList);
        end
    end
end
```

The object display looks like this:

```
EmployeeInfo with properties:

   Department: 'Product Development'
```
JobTitle: 'Software Engineer'
       Name: 'Bill Tork'
       Salary: 'Not available'
       Password: '*******'

Full Class Listing

classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
   properties
      Name
      JobTitle
      Department
      Salary
      Password
   end
   methods
      function obj = EmployeeInfo
         obj.Name       = input('Name: ');
         obj.JobTitle   = input('Job Title: ');
         obj.Department = input('Department: ');
         obj.Salary     = input('Salary: ');
         obj.Password   = input('Password: ');
      end
   end
   methods (Access = protected)
      function propgrp = getPropertyGroups(obj)
         if ~isscalar(obj)
            propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
         else
            pd(1:length(obj.Password)) = '*';
            propList = struct('Department',obj.Department,...
                              'JobTitle',obj.JobTitle,...
                              'Name',obj.Name,...
                              'Salary','Not available',...
                              'Password',pd);
            propgrp = matlab.mixin.util.PropertyGroup(propList);
         end
      end
   end
end

See Also

Related Examples

•   “Choose a Technique for Display Customization” on page 18-19
Customize Header, Property List, and Footer

<table>
<thead>
<tr>
<th>In this section…</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Objective” on page 18-26</td>
</tr>
<tr>
<td>“Design of Custom Display” on page 18-26</td>
</tr>
<tr>
<td>“getHeader Method Override” on page 18-28</td>
</tr>
<tr>
<td>“getPropertyGroups Override” on page 18-29</td>
</tr>
<tr>
<td>“getFooter Override” on page 18-29</td>
</tr>
</tbody>
</table>

**Objective**

Customize each of the three parts of the display — header, property groups, and footer.

**Design of Custom Display**

*Note* This example uses the EmployeeInfo class described in the “Class with Default Object Display” on page 18-14 section.

For the header:

- Use default header for nonscalar object arrays.
- Build header text with linked class name and department name (from Department property)

For properties:

- Nonscalar object arrays display a subset of property names in a different order than the default.
- Scalar objects create two property groups that have titles (Public Info and Personal Info).

For the footer:

- Add a footer to the display, only when the object is a valid scalar that displays property values.
Here is the customized display of an object of the `EmployeeInfo` class.

```
Emp123 =

EmployeeInfo Dept: Product Development

  Public Info
  Name: 'Bill Tork'
  JobTitle: 'Software Engineer'

  Personal Info
  Salary: 1000
  Password: 'bill123'

Company Private
```

Here is the custom display of an array of `EmployeeInfo` objects:

```
[Emp123,Emp124]
ans =
```

```
1x2 EmployeeInfo array with properties:
  Department
  Name
  JobTitle
```

Here is the display of an empty object array:

```
>> EmployeeInfo.empty(0,5)
ans =
```

```
0x5 EmployeeInfo array with properties:
  Department
  Name
  JobTitle
```

Here is the display of a handle to a delete object (`EmployeeInfo` is a handle class):

```
>> delete(Emp123)
>> Emp123
```
Emp123 =
    handle to deleted EmployeeInfo

Implementation

The EmployeeInfo class overrides three matlab.mixin.CustomDisplay methods to implement the display shown:

- getHeader
- getPropertyGroups
- getFooter

Each method must produce the desired results with each of the following inputs:

- Scalar object
- Nonscalar object array
- Empty object array

**getHeader Method Override**

MATLAB calls getHeader to get the header text. The EmployeeInfo class overrides this method to implement the custom header for scalar display. Here is how it works:

- Nonscalar (including empty object) arrays call the superclass getHeader, which returns the default header.
- Scalar handles to deleted objects do not result in a call to getHeader.
- Scalar inputs build a custom header using the getClassnameForHeader static method to return linked class name text, and the value of the Department property.

Here is the EmployeeInfo override of the getHeader method. The required protected access is inherited from the superclass.

```matlab
methods (Access = protected)
    function header = getHeader(obj)
        if ~isscalar(obj)
            header = getHeader@matlab.mixin.CustomDisplay(obj);
        else
            className = matlab.mixin.CustomDisplay.getClassnameForHeader(obj);
            newHeader = [className,' Dept: ',obj.Department];
            header = sprintf('%s
',newHeader);
        end
```
getPropertyGroups Override

MATLAB calls `getPropertyGroups` to get the `PropertyGroup` objects, which control how properties are displayed. This method override defines two different property lists depending on the object’s state:

- For nonscalar inputs, including empty arrays and arrays containing handles to deleted objects, create a property list as a cell array to reorder properties.
  
  By default, MATLAB does not display property values for nonscalar inputs.
- For scalar inputs, create two property groups with titles. The scalar code branch lists properties in a different order than the nonscalar case and includes `Salary` and `Password` properties. MATLAB automatically assigns property values.
- Scalar handles to deleted objects do not result in a call to `getPropertyGroups`.

Both branches return a `matlab.mixin.util.PropertyGroup` object, which determines how to displays the object properties.

Here is the `EmployeeInfo` override of the `getPropertyGroups` method. The protected access is inherited from the superclass.

```matlab
methods (Access = protected)
    function propgrp = getPropertyGroups(obj)
        if ~isscalar(obj)
            propList = {'Department','Name','JobTitle'};
            propgrp = matlab.mixin.util.PropertyGroup(propList);
        else
            gTitle1 = 'Public Info';
            gTitle2 = 'Personal Info';
            propList1 = {'Name','JobTitle'};
            propList2 = {'Salary','Password'};
            propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
            propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
        end
    end
end
```

getFooter Override

MATLAB calls `getFooter` to get the footer text. The `EmployeeInfo` `getFooter` method defines a footer for the display, which is included only when the input is a valid scalar object. In all other cases, `getFooter` returns an empty `char` vector.
Scalar handles to deleted objects do not result in a call to `getFooter`.

```matlab
function footer = getFooter(obj)
if isscalar(obj)
    footer = sprintf('%s\n','Company Private');
else
    footer = '';
end
end
```

### Complete Class Listing

```matlab
classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
    properties
        Name
        JobTitle
        Department
        Salary
        Password
    end
    methods
        function obj = EmployeeInfo
            obj.Name       = input('Name: ');
            obj.JobTitle   = input('Job Title: ');
            obj.Department = input('Department: ');
            obj.Salary     = input('Salary: ');
            obj.Password   = input('Password: ');
        end
    end
    methods (Access = protected)
        function header = getHeader(obj)
            if ~isscalar(obj)
                header = getHeader@matlab.mixin.CustomDisplay(obj);
            else
                className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
                newHeader = [className,' Dept: ',obj.Department];
                header = sprintf('%s\n',newHeader);
            end
        end
        function propgrp = getPropertyGroups(obj)
            if ~isscalar(obj)
                propList = {'Department','Name','JobTitle'};
                propgrp = matlab.mixin.util.PropertyGroup(propList);
            else
                gTitle1 = 'Public Info';
                gTitle2 = 'Personal Info';
                propList1 = {'Name','JobTitle'};
                propList2 = {'Salary','Password'};
                propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
                propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
            end
        end
end
```
propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
end
end

function footer = getFooter(obj)
    if isscalar(obj)
        footer = sprintf('%s
','Company Private');
    else
        footer = '';  
    end
end
end

See Also

Related Examples
• “Choose a Technique for Display Customization” on page 18-19
Customize Display of Scalar Objects

In this section...

“Objective” on page 18-32
“Design Of Custom Display” on page 18-32
“displayScalarObject Method Override” on page 18-33
“getPropertyGroups Override” on page 18-34

Objective

Customize the display of scalar objects.

Design Of Custom Display

Note  This example uses the EmployeeInfo class described in the “Class with Default Object Display” on page 18-14 section.

The objective of this customized display is to:

- Modify the header to include the department name obtained from the Department property
- Group properties into two categories titled Public Info and Personal Info.
- Modify which properties are displayed
- Modify the values displayed for Personal Info category
- Use the default displayed for nonscalar objects, including empty arrays, and scalar deleted handles

For example, here is the customized display of an object of the EmployeeInfo class.

Emp123 =

EmployeeInfo Dept: Product Development

Public Info
    Name: 'Bill Tork'
    JobTitle: 'Software Engineer'
Personal Info
   Salary: 'Level: 10'
   Password: '*******'

Implementation

The EmployeeInfo class overrides two matlab.mixin.CustomDisplay methods to implement the display shown:

- displayScalarObject — Called to display valid scalar objects
- getPropertyGroups — Builds the property groups for display

displayScalarObject Method Override

MATLAB calls displayScalarObject to display scalar objects. The EmployeeInfo class overrides this method to implement the scalar display. Once overridden, this method must control all aspects of scalar object display, including creating the header, property groups, and footer, if used.

This implementation:

- Builds a custom header using the getClassNameForHeader static method to return linked class name text and the value of the Department property to get the department name.
- Uses sprintf to add a new line to the header text
- Displays the header with the built-in disp function.
- Calls the getPropertyGroups override to define the property groups (see following section).
- Displays the property groups using the displayPropertyGroups static method.

Here is the EmployeeInfo override of the displayScalarObject method. The required protected access is inherited from the superclass.

```matlab
methods (Access = protected)
    function displayScalarObject(obj)
        className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
        scalarHeader = [className,' Dept: ',obj.Department];
        header = sprintf('
%s',scalarHeader);
        disp(header)
        propgroup = getPropertyGroups(obj);
        matlab.mixin.CustomDisplay.displayPropertyGroups(obj,propgroup)
    end
end
```
getPropertyGroups Override

MATLAB calls `getPropertyGroups` when displaying scalar or nonscalar objects. However, MATLAB does not call this method when displaying a scalar handle to a deleted object.

The `EmployeeInfo` class overrides this method to implement the property groups for scalar object display.

This implementation calls the superclass `getPropertyGroups` method if the input is not scalar. If the input is scalar, this method:

- Defines two titles for the two groups
- Creates a cell array of property names that are included in the first group. MATLAB adds the property values for the display
- Creates a struct array of property names with associated property values for the second group. Using a struct instead of a cell array enables you to replace the values that are displayed for the `Salary` and `Password` properties without changing the personal information stored in the object properties.
- Constructs two `matlab.mixin.util.PropertyGroup` objects, which are used by the `displayScalarObject` method.

Here is the `EmployeeInfo` override of the `getPropertyGroups` method. The required protected access is inherited from the superclass.

```matlab
methods (Access = protected)
function propgrp = getPropertyGroups(obj)
    if ~isscalar(obj)
        propgrp = getPropertyGroups(matlab.mixin.CustomDisplay(obj));
    else
        gTitle1 = 'Public Info';
        gTitle2 = 'Personal Info';
        propList1 = {'Name','JobTitle'};
        pd(1:2:length(obj.Password)) = '*';
        level = round(obj.Salary/100);
        propList2 = struct('Salary',...
            ['Level: ',num2str(level)],...
            'Password',pd);
        propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
        propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
    end
end
end
```
Complete Class Listing

classdef EmployeeInfo4 < handle & matlab.mixin.CustomDisplay
    properties
        Name
        JobTitle
        Department
        Salary
        Password
    end
    methods
        function obj = EmployeeInfo4
            obj.Name       = input('Name: ');
            obj.JobTitle   = input('Job Title: ');
            obj.Department = input('Department: ');
            obj.Salary     = input('Salary: ');
            obj.Password   = input('Password: ');
        end
    end
    methods (Access = protected)
        function displayScalarObject(obj)
            className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
            scalarHeader = [className,' Dept: ',obj.Department];
            header = sprintf('%s\n',scalarHeader);
            disp(header)
            propgroup = getPropertyGroups(obj);
            matlab.mixin.CustomDisplay.displayPropertyGroups(obj,propgroup)
        end
        function propgrp = getPropertyGroups(obj)
            if ~isscalar(obj)
                propgrp = getPropertyGroups@matlab.mixin.CustomDisplay(obj);
            else
                % property groups for scalars
                gTitle1 = 'Public Info';
                gTitle2 = 'Personal Info';
                propList1 = {'Name','JobTitle'};
                pd(1:length(obj.Password)) = '*';
                level = round(obj.Salary/100);
                propList2 = struct('Salary',
                    ['Level: ',num2str(level)],...
                    'Password',pd);
                propgrp(1) = matlab.mixin.util.PropertyGroup(propList1,gTitle1);
                propgrp(2) = matlab.mixin.util.PropertyGroup(propList2,gTitle2);
            end
        end
    end
end
See Also

Related Examples

• “Choose a Technique for Display Customization” on page 18-19
**Customize Display of Object Arrays**

**In this section...**

| “Objective” on page 18-37 |
| “Design of Custom Display” on page 18-37 |
| “The displayNonScalarObject Override” on page 18-38 |
| “The displayEmptyObject Override” on page 18-39 |

**Objective**

Customize the display of nonscalar objects, including empty object arrays.

**Design of Custom Display**

**Note** This example uses the EmployeeInfo class described in the “Class with Default Object Display” on page 18-14 section.

The objective of this customized display is to:

- Construct a custom header using some elements of the default header
- Display a subset of property-specific information for each object in the array.
- List handles to deleted objects in the array using a char vector with links to documentation for handle objects and the class.
- Display empty objects with a slight modification to the default header

Here is the customized display of an array of three EmployeeInfo objects:

1x3 EmployeeInfo array with members:
1. Employee:
   - Name: 'Bill Tork'
   - Department: 'Product Development'

2. Employee:
   - Name: 'Alice Blackwell'
   - Department: 'QE'
3. Employee:
   Name: 'Nancy Green'
   Department: 'Documentation'

Deleted object handles in the array indicate their state:

1x3 EmployeeInfo members:

1. Employee:
   Name: 'Bill Tork'
   Department: 'Product Development'

2. handle to deleted EmployeeInfo

3. Employee:
   Name: 'Nancy Green'
   Department: 'Documentation'

To achieve the desired result, the EmployeeInfo class overrides the following methods of the matlab.mixin.CustomDisplay class:

• displayNonScalarObject — Called to display nonempty object arrays
• displayEmptyObject — Called to display empty object arrays

The **displayNonScalarObject Override**

MATLAB calls the **displayNonScalarObject** method to display object arrays. The override of this method in the EmployeeInfo class:

• Builds header text using **convertDimensionsToString** to obtain the array size and **getClassNameForHeader** to get the class name with a link to the help for that class.
• Displays the modified header text.
• Loops through the elements in the array, building two different subheaders depending on the individual object state. In the loop, this method:
  • Detects handles to deleted objects (using the **isValid** handle class method). Uses **getDeletedHandleText** and **getClassNameForHeader** to build text for array elements that are handles to deleted objects.
  • Builds a custom subheader for valid object elements in the array
• Creates a PropertyGroup object containing the **Name** and **Department** properties for valid objects
• Uses the `displayPropertyGroups` static method to generate the property display for valid objects.

Here is the implementation of `displayNonScalarObjects`:

```matlab
methods (Access = protected)
    function displayNonScalarObject(objAry)
        dimStr = matlab.mixin.CustomDisplay.convertDimensionsToString(objAry);
        cName = matlab.mixin.CustomDisplay.getClassNameForHeader(objAry);
        headerStr = [dimStr,' ',cName,' members:'];
        header = sprintf('%s
',headerStr);
        disp(header)
        for ix = 1:length(objAry)
            o = objAry(ix);
            if ~isvalid(o)
                str1 = matlab.mixin.CustomDisplay.getDeletedHandleText;
                str2 = matlab.mixin.CustomDisplay.getClassNameForHeader(o);
                headerInv = [str1,' ',str2];
                tmpStr = [num2str(ix),'. ',headerInv];
                numStr = sprintf('%s
',tmpStr);
                disp(numStr)
            else
                numStr = [num2str(ix),'. Employee:'];
                disp(numStr)
                propList = struct('Name',o.Name,...
                    'Department',o.Department);
                propgrp = matlab.mixin.util.PropertyGroup(propList);
                matlab.mixin.CustomDisplay.displayPropertyGroups(o,propgrp);
            end
        end
    end
end
```

**The displayEmptyObject Override**

MATLAB calls the `displayEmptyObject` method to display empty object arrays. The implementation of this method in the `EmployeeInfo` class builds a custom header for empty objects following these steps:

• Gets the array dimensions in character format using the `convertDimensionsToString` static method.
• Gets text with the class name linked to the `helpPopup` function using the `getClassNameForHeader` static method.
• Builds and displays the custom text for empty arrays.

```matlab
methods (Access = protected)
    function displayEmptyObject(obj)
        dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj);
        className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
```
emptyHeader = [dimstr,' ',className,' with no employee information'];
header = sprintf('%s
',emptyHeader);
disp(header)
end

For example, an empty EmployeeInfo object displays like this:

Empt = EmployeeInfo.empty(0,5)

Empt =

0x5 EmployeeInfo with no employee information

Complete Class Listing

classdef EmployeeInfo < handle & matlab.mixin.CustomDisplay
properties
    Name
    JobTitle
    Department
    Salary
    Password
end
methods
    function obj = EmployeeInfo
        obj.Name       = input('Name: ');
        obj.JobTitle   = input('Job Title: ');
        obj.Department = input('Department: ');
        obj.Salary     = input('Salary: ');
        obj.Password   = input('Password: ');
    end
end
methods (Access = protected)
    function dispNonScalarObject(objAry)
        dimStr = matlab.mixin.CustomDisplay.convertDimensionsToString(objAry);
        cName = matlab.mixin.CustomDisplay.getClassNameForHeader(objAry);
        headerStr = [dimStr,' ',cName,' members:'
        header = sprintf('%s\n',headerStr);
        disp(header)
        for ix = 1:length(objAry)
            o = objAry(ix);
            if ~isvalid(o)
                str1 = matlab.mixin.CustomDisplay.getDeletedHandleText;
                str2 = matlab.mixin.CustomDisplay.getClassNameForHeader(o);
                headerInv = [str1,' ',str2];
                tmpStr = [num2str(ix),'. ',headerInv];
                numStr = sprintf('%s\n',tmpStr);
                disp(numStr)
            else
                numStr = [num2str(ix),'. Employee';
                disp(numStr)
                propList = struct('Name',o.Name,'
                    'Department',o.Department);
propgrp = matlab.mixin.util.PropertyGroup(propList);
matlab.mixin.CustomDisplay.displayPropertyGroups(o,propgrp);
end
end
end

function displayEmptyObject(obj)
dimstr = matlab.mixin.CustomDisplay.convertDimensionsToString(obj);
className = matlab.mixin.CustomDisplay.getClassNameForHeader(obj);
emptyHeader = [dimstr, ',className, ' with no employee information'];
header = sprintf('%s
',emptyHeader);
disp(header)
end
end

See Also

Related Examples

• "Choose a Technique for Display Customization" on page 18-19
Overloading the disp Function

In this section...

| “Display Methods” on page 18-42 |
| “Overloaded disp” on page 18-42 |
| “Relationship Between disp and display” on page 18-42 |

Display Methods

Subclassing `matlab.mixin.CustomDisplay` is the best approach to customizing object display. However, if you do not derive your class from `matlab.mixin.CustomDisplay`, overload the `disp` function to change how MATLAB displays objects of your class.

MATLAB calls the `display` function whenever an object is referred to in a statement that is not terminated by a semicolon. For example, the following statement creates the variable `a`. MATLAB calls `display`, which displays the value of `a` in the command line.

```matlab
a = 5
```

```matlab
a = 
5
```

display then calls `disp`.

Overloaded disp

The built-in `display` function prints the name of the variable that is being displayed, if an assignment is made, or otherwise uses `ans` as the variable name. Then `display` calls `disp` to handle the actual display of the values.

If the variable that is being displayed is an object of a class that overloads `disp`, then MATLAB always calls the overloaded method. MATLAB calls `display` with two arguments and passes the variable name as the second argument.

Relationship Between disp and display

MATLAB invokes the built-in `display` function when:
• MATLAB executes a statement that returns a value and is not terminated with a semicolon.
• There is no left-side variable, then MATLAB prints `ans =` followed by the value.
• Code explicitly invokes the `display` function.

When invoked `display`:
• If the input argument is an existing variable, `display` prints the variable name and equal sign, followed by the value.
• If the input is the result of an expression, `display` does not print `ans =`.

MATLAB invokes the built-in `disp` function when:
• The built-in `display` function calls `disp`.
• Code explicitly invokes `disp`.

For empty built-in types (numeric types, `char`, `struct`, and `cell`) the `display` function displays:

• `[]` — for numeric types
• "0x0 struct array with no fields." — for empty structs.
• "Empty cell array: 0-by-1" — for empty cell arrays.
• "" — for empty char arrays

disp differs from display in these ways:
• `disp` does not print the variable name or `ans`.
• `disp` prints nothing for built-in types (numeric types, `char`, `struct`, and `cell`) when the value is empty.

See Also

Related Examples
• “Custom Display Interface” on page 18-2
• “Overload disp for DocPolynom” on page 19-17
Defining Custom Data Types
Class Design for Polynomials

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Object Requirements” on page 19-2</td>
</tr>
<tr>
<td>“DocPolynom Class Members” on page 19-2</td>
</tr>
<tr>
<td>“DocPolynom Class Synopsis” on page 19-4</td>
</tr>
<tr>
<td>“The DocPolynom Constructor” on page 19-13</td>
</tr>
<tr>
<td>“Remove Irrelevant Coefficients” on page 19-14</td>
</tr>
<tr>
<td>“Convert DocPolynom Objects to Other Types” on page 19-15</td>
</tr>
<tr>
<td>“Overload disp for DocPolynom” on page 19-17</td>
</tr>
<tr>
<td>“Display Evaluated Expression” on page 19-18</td>
</tr>
<tr>
<td>“Redefine Indexed Reference” on page 19-19</td>
</tr>
<tr>
<td>“Define Arithmetic Operators” on page 19-21</td>
</tr>
</tbody>
</table>

Object Requirements

This example implements a class to represent polynomials in the MATLAB language. The design requirements are:

- Value class behavior—a polynomial object should behave like MATLAB numeric variables when copied and passed to functions.
- Specialized display and indexing
- Objects can be scalar only. The specialization of display and indexing functionality preclude normal array behavior.
- Arithmetic operations
- Double converter simplifying the use of polynomial object with existing MATLAB functions that accept numeric inputs.

DocPolynom Class Members

The class definition specifies a property for data storage and defines a folder (@DocPolynom) that contains the class definition.

The following table summarizes the properties defined for the DocPolynom class.
**DocPolynom Class Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coef</td>
<td>double</td>
<td>[]</td>
<td>Vector of polynomial coefficients [highest order ... lowest order]</td>
</tr>
</tbody>
</table>

The following table summarizes the methods for the DocPolynom class.

**DocPolynom Class Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocPolynom</td>
<td>Class constructor</td>
</tr>
<tr>
<td>double</td>
<td>Converts a DocPolynom object to a double (that is, returns its coefficients in a vector)</td>
</tr>
<tr>
<td>char</td>
<td>Creates a formatted display of the DocPolynom object as powers of x and is used by the disp method</td>
</tr>
<tr>
<td>disp</td>
<td>Determines how MATLAB displays DocPolynom objects on the command line</td>
</tr>
<tr>
<td>subsref</td>
<td>Enables you to specify a value for the independent variable as a subscript, access the coef property with dot notation, and call methods with dot notation.</td>
</tr>
<tr>
<td>plus</td>
<td>Implements addition of DocPolynom objects</td>
</tr>
<tr>
<td>minus</td>
<td>Implements subtraction of DocPolynom objects</td>
</tr>
<tr>
<td>mtimes</td>
<td>Implements multiplication of DocPolynom objects</td>
</tr>
</tbody>
</table>

**Using the DocPolynom Class**

The following examples illustrate basic use of the DocPolynom class.

Create DocPolynom objects to represent the following polynomials. The argument to the constructor function contains the polynomial coefficients $f(x) = x^3 - 2x - 5$ and $f(x) = 2x^4 + 3x^2 + 2x - 7$.

```matlab
p1 = DocPolynom([1 0 -2 -5])
p1 =
    x^3 - 2*x - 5
p2 = DocPolynom([2 0 3 2 -7])
```
p2 = 
  2*x^4 + 3*x^2 + 2*x - 7

Find the roots of the polynomial by passing the coefficients to the roots function.

roots(p1.coef)

ans =
  2.0946 + 0.0000i
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i

Add the two polynomials p1 and p2.

MATLAB calls the plus method defined for the DocPolynom class when you add two DocPolynom objects.

p1 + p2

ans =
  2*x^4 + x^3 + 3*x^2 - 12

**DocPolynom Class Synopsis**

<table>
<thead>
<tr>
<th>Example Code</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>classdef DocPolynom</td>
<td>Value class that implements a data type for polynomials.</td>
</tr>
</tbody>
</table>
| properties 
  coef 
end | Vector of polynomial coefficients [highest order ... lowest order] |
| methods | For general information about methods, see “Ordinary Methods” on page 9-8 |
### Example Code

<table>
<thead>
<tr>
<th>Function</th>
<th>Discussion</th>
</tr>
</thead>
</table>
| *function* obj = DocPolynom(c)*  
  *if* nargin > 0  
  *if* isa(c,'DocPolynom')  
  obj.coef = c.coef;  
  *else*  
  obj.coef = c(:).';  
  *end*  
  *end* | Class constructor creates objects using:  
  • Coefficient vector of existing object  
  • Coefficient vector passed as argument  
  See “The DocPolynom Constructor” on page 19-13 |
| *function* obj = set.coef(obj,val)  
  *if* ~isa(val,'double')  
  error('Coefficients must be doubles')  
  *end*  
  ind = find(val(:).'==0);  
  *if* ~isempty(ind);  
  obj.coef = val(ind(1):end);  
  *else*  
  obj.coef = val;  
  *end*  
  *end* | Set method for coef property:  
  • Allows coefficients only of type double  
  • Removes leading zeros from the coefficient vector.  
  See “Remove Irrelevant Coefficients” on page 19-14 |
| *function* c = double(obj)  
  c = obj.coef;  
  *end* | Convert DocPolynom object to double by returning the coefficients.  
  See “Convert DocPolynom Objects to Other Types” on page 19-15 |
**Example Code**

```matlab
function str = char(obj)
    if all(obj.coef == 0)
        s = '0';
        str = s;
        return
    else
        d = length(obj.coef)-1;
        s = cell(1,d);
        ind = 1;
        for a = obj.coef;
            if a ~= 0;
                if ind ~= 1
                    if a > 0
                        s(ind) = {' + '};
                        ind = ind + 1;
                    else
                        s(ind) = {' - '};
                        a = -a;
                        ind = ind + 1;
                    end
                end
                if a ~= 1 || d == 0
                    if a == -1
                        s(ind) = {'-'};
                        ind = ind + 1;
                    else
                        s(ind) = {num2str(a)};
                        ind = ind + 1;
                        if d > 0
                            s(ind) = {'*'};
                            ind = ind + 1;
                        end
                    end
                end
            end
        end
        if d >= 2
            s(ind) = {[char('x'^ int2str(d))]
        elseif d == 1
            s(ind) = {'x'};
            ind = ind + 1;
        end
    end
end
```

**Discussion**

Convert DocPolynom object to char that represents the expression:

\[ y = f(x) \]

See “Convert DocPolynom Objects to Other Types” on page 19-15.
<table>
<thead>
<tr>
<th>Example Code</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = d - 1;</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>str = [s{:}];</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td><strong>function</strong> disp(obj)</td>
<td>Overload disp function. Display objects as output of char method.</td>
</tr>
<tr>
<td>c = char(obj);</td>
<td>For information about this code, see “Overload disp for DocPolynom” on page 19-17</td>
</tr>
<tr>
<td><strong>if</strong> iscell(c)</td>
<td></td>
</tr>
<tr>
<td>disp(['     ' c{:}])</td>
<td></td>
</tr>
<tr>
<td><strong>else</strong></td>
<td></td>
</tr>
<tr>
<td>disp(c)</td>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
<tr>
<td>dispPoly(obj,x)</td>
<td>Return evaluated expression with formatted output.</td>
</tr>
<tr>
<td>p = char(obj);</td>
<td>Uses output of char method to evaluate polynomial at specified values of independent variable.</td>
</tr>
<tr>
<td>e = @(x)eval(p);</td>
<td>For information about this code, see “Display Evaluated Expression” on page 19-18</td>
</tr>
<tr>
<td>y = zeros(length(p));</td>
<td></td>
</tr>
<tr>
<td>disp(['y = ',p])</td>
<td></td>
</tr>
<tr>
<td><strong>for</strong> k = 1:length(x)</td>
<td></td>
</tr>
<tr>
<td>y(k) = e(x(k));</td>
<td></td>
</tr>
<tr>
<td>disp(['   ',num2str(y(k)),...</td>
<td></td>
</tr>
<tr>
<td>' = f(x = ',...</td>
<td></td>
</tr>
<tr>
<td>num2str(x(k)),',')]);</td>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
<tr>
<td>Example Code</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td></td>
</tr>
</tbody>
</table>
| `function b = subsref(a,s)`  
  `switch s(1).type`  
  `case '()'`  
  `ind = s.subs{;;};`  
  `b = polyval(a.coef,ind);`  
  `case '.'`  
  `switch s(1).subs`  
  `case 'coef'`  
  `b = a.coef;`  
  `case 'disp'`  
  `disp(a)`  
  `otherwise`  
  `if length(s)>1`  
  `b = a.(s(1).subs)(s(2).subs{;;});`  
  `else`  
  `b = a.(s.subs);`  
  `end`  
  `end`  
<table>
<thead>
<tr>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redefine indexed reference for DocPolynom objects.</td>
</tr>
<tr>
<td>For information about this code, see “Redefine Indexed Reference” on page 19-19</td>
</tr>
</tbody>
</table>
### Example Code

<table>
<thead>
<tr>
<th>Example Code</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>function r = plus(obj1,obj2)</strong></td>
<td>Define three arithmetic operators:</td>
</tr>
<tr>
<td>obj1 = DocPolynom(obj1);</td>
<td>• Polynomial addition</td>
</tr>
<tr>
<td>obj2 = DocPolynom(obj2);</td>
<td>• Polynomial subtraction</td>
</tr>
<tr>
<td>k = length(obj2.coef) - length(obj1.coef);</td>
<td>• Polynomial multiplication</td>
</tr>
<tr>
<td>zp = zeros(1,k);</td>
<td>For information about this code, see “Define Arithmetic Operators” on page 19-21.</td>
</tr>
<tr>
<td>zm = zeros(1,-k);</td>
<td>For general information about defining operators, see “Operator Overloading” on page 17-47.</td>
</tr>
<tr>
<td>r = DocPolynom([zp,obj1.coef] + [zm,obj2.coef]);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td><strong>function r = minus(obj1,obj2)</strong></td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>obj1 = DocPolynom(obj1);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>obj2 = DocPolynom(obj2);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>k = length(obj2.coef) - length(obj1.coef);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>zp = zeros(1,k);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>zm = zeros(1,-k);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>r = DocPolynom([zp,obj1.coef] - [zm,obj2.coef]);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td><strong>function r = mtimes(obj1,obj2)</strong></td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>obj1 = DocPolynom(obj1);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>obj2 = DocPolynom(obj2);</td>
<td>end statements for methods and for classdef.</td>
</tr>
<tr>
<td>r = DocPolynom(conv(obj1.coef,obj2.coef));</td>
<td>end statements for methods and for classdef.</td>
</tr>
</tbody>
</table>

### Expand for Class Code

classdef DocPolynom
    % Documentation example
    % A value class that implements a data type for polynomials
    % See Implementing a Class for Polynomials in the
    % MATLAB documentation for more information.

    properties
        coef
    end

    % Class methods
    methods
        function obj = DocPolynom(c)

end
if nargin > 0
    if isa(c,'DocPolynom')
        obj.coef = c.coef;
    else
        obj.coef = c(:).';
    end
end
end % DocPolynom

function obj = set.coef(obj,val)
    if ~isa(val,'double')
        error('Coefficients must be doubles')
    end
    % Remove leading zeros
    ind = find(val(:).'~=0);
    if ~isempty(ind);
        obj.coef = val(ind(1):end);
    else
        obj.coef = val;
    end
end % set.coef

function c = double(obj)
    c = obj.coef;
end % double

function str = char(obj)
    % Created a formatted display of the polynom
    % as powers of x
    if all(obj.coef == 0)
        s = '0';
        str = s;
        return
    else
        d = length(obj.coef)-1;
        s = cell(1,d);
        ind = 1;
        for a = obj.coef;
            if a ~= 0;
                if ind ~= 1
                    if a > 0
                        s(ind) = {' + '};
                        ind = ind + 1;
                    else
                        s(ind) = {' - '};
                    end
                end
            end
        end
        str = s;
    end
end % char
function disp(obj)
    % DISP Display object in MATLAB syntax
    c = char(obj);
    if iscell(c)
        disp(['    ' c{:}])
    else
        disp(c)
    end
end % disp

function dispPoly(obj,x)
    % evaluate obj at x
    p = char(obj);
e = @(x)eval(p);
y = zeros(length(x));
disp(['y = ',p])
for k = 1:length(x)
    y(k) = e(x(k));
    disp(['  ',num2str(y(k)),...
        ' = f(x = ',...
        num2str(x(k)),')]})
end

function b = subsref(a,s)
% SUBSREF Implementing the following syntax:
% obj([1 ...])
% obj.coef
% obj.disp
% out = obj.method(args)
% out = obj.method
switch s(1).type
    case '()'
        ind = s.subs{:};
        b = polyval(a.coef,ind);
    case '.'
        switch s(1).subs
            case 'coef'
                b = a.coef;
            case 'disp'
                disp(a)
            otherwise
                if length(s)>1
                    b = a.(s(1).subs)(s(2).subs{:});
                else
                    b = a.(s.subs);
                end
            end
        end
    otherwise
        error('Specify value for x as obj(x)')
end
end % subsref

function r = plus(obj1,obj2)
% PLUS  Implement obj1 + obj2 for DocPolynom
obj1 = DocPolynom(obj1);
obj2 = DocPolynom(obj2);
k = length(obj2.coef) - length(obj1.coef);
zp = zeros(1,k);
zm = zeros(1,-k);
r = DocPolynom([zp,obj1.coef] + [zm,obj2.coef]);
end % plus

function r = minus(obj1,obj2)
    % MINUS Implement obj1 - obj2 for DocPolynoms.
    obj1 = DocPolynom(obj1);
    obj2 = DocPolynom(obj2);
    k = length(obj2.coef) - length(obj1.coef);
zp = zeros(1,k);
zm = zeros(1,-k);
r = DocPolynom([zp,obj1.coef] - [zm,obj2.coef]);
end % minus

function r = mtimes(obj1,obj2)
    % MTIMES Implement obj1 * obj2 for DocPolynoms.
    obj1 = DocPolynom(obj1);
    obj2 = DocPolynom(obj2);
    r = DocPolynom(conv(obj1.coef,obj2.coef));
end % mtimes
end % methods
end % classdef

The DocPolynom Constructor

The following function is the DocPolynom class constructor, which is in the file @DocPolynom/DocPolynom.m:

methods
    function obj = DocPolynom(c)
        if isa(c,'DocPolynom')
            obj.coef = c.coef;
        else
            obj.coef = c(:).';
        end
    end
end

Constructor Calling Syntax

It is possible to all the DocPolynom constructor with two different arguments:
• Input argument is a `DocPolynom` object — If you call the constructor function with an input argument that is already a `DocPolynom` object, the constructor returns a new `DocPolynom` object with the same coefficients as the input argument. The `isa` function checks for this input.

• Input argument is a coefficient vector — If the input argument is not a `DocPolynom` object, the constructor attempts to reshape the values into a vector and assign them to the `coef` property.

The `coef` property set method restricts property values to doubles. See “Remove Irrelevant Coefficients” on page 19-14 for a description of the property set method.

An example use of the `DocPolynom` constructor is the statement:

```matlab
p = DocPolynom([1 0 -2 -5])
p =
x^3 - 2*x -5
```

This statement creates an instance of the `DocPolynom` class with the specified coefficients. Note that the display of the object shows the equivalent polynomial using MATLAB language syntax. The `DocPolynom` class implements this display using the `disp` and `char` class methods.

### Remove Irrelevant Coefficients

MATLAB software represents polynomials as row vectors containing coefficients ordered by descending powers. Zeros in the coefficient vector represent terms that drop out of the polynomial. Leading zeros, therefore, can be ignored when forming the polynomial.

Some `DocPolynom` class methods use the length of the coefficient vector to determine the degree of the polynomial. It is useful, therefore, to remove leading zeros from the coefficient vector so that its length represents the true value.

The `DocPolynom` class stores the coefficient vector in a property that uses a set method to remove leading zeros from the specified coefficients before setting the property value.

```matlab
methods
function obj = set.coef(obj,val)
    if ~isa(val,'double')
        error('Coefficients must be doubles')
    end
    ind = find(val(:).~=0);
    if ~isempty(ind);
```
Convert DocPolynom Objects to Other Types

The DocPolynom class defines two methods to convert DocPolynom objects to other classes:

- **double** — Converts to the double numeric type so functions can perform mathematical operations on the coefficients.
- **char** — Converts to characters used to format output for display in the command window

The Double Converter

The double converter method for the DocPolynom class simply returns the coefficient vector:

```plaintext
methods
    function c = double(obj)
        c = obj.coef;
    end
end
```

For the DocPolynom object `p`:

```plaintext
p = DocPolynom([1 0 -2 -5]);
```

the statement:

```plaintext
c = double(p)
```

returns:

```plaintext
c = 1     0    -2    -5
```

which is of class `double`:
The Character Converter

The `char` method produces a `char` vector that represents the polynomial displayed as powers of \( x \). The `char` vector returned is a syntactically correct MATLAB expression.

The `char` method uses a cell array to collect the `char` vector components that make up the displayed polynomial.

The `disp` method uses the `char` method to format the `DocPolynom` object for display. The `evalPoly` method uses `char` to create the MATLAB expression to evaluate.

Users of `DocPolynom` objects are not likely to call the `char` or `disp` methods directly, but these methods enable the `DocPolynom` class to behave like other data classes in MATLAB.

Here is the `char` method.

```matlab
methods
    function str = char(obj)
        if all(obj.coef == 0)
            s = '0';
            str = s;
            return
        else
            d = length(obj.coef)-1;
            s = cell(1,d);
            ind = 1;
            for a = obj.coef;
                if a ~= 0;
                    if ind ~= 1
                        if a > 0
                            s(ind) = {' + '};
                            ind = ind + 1;
                        else
                            s(ind) = {' - '};
                            a = -a;
                            ind = ind + 1;
                        end
                    end
                end
            end
        end
        if a ~= 1 || d == 0
            if a == -1
                s(ind) = {' - '};
                ind = ind + 1;
            else
                s(ind) = {num2str(a)};
                ind = ind + 1;
            end
        end
    end
end
```
if d > 0
    s(ind) = {'*'};
    ind = ind + 1;
end
end
end
if d >= 2
    s(ind) = {{'x'^ int2str(d)};}
    ind = ind + 1;
elseif d == 1
    s(ind) = {'x'};
    ind = ind + 1;
end
end
d = d - 1;
end
str = [s{:}];
end
end

Overload disp for DocPolynom

To provide a more useful display of DocPolynom objects, this class overloads disp in the class definition.

This disp method relies on the char method to produce a text representation of the polynomial, which it then displays on the screen.

The char method returns a cell array or the character '0' if the coefficients are all zero.

methods
    function disp(obj)
        c = char(obj);
        if iscell(c)
            disp(['     ' c{:}])
        else
            disp(c)
        end
    end
end

When MATLAB Calls the disp Method

The statement:

p = DocPolynom([1 0 -2 -5])
creates a DocPolynom object. Because the statement is not terminated with a semicolon, the resulting output is displayed on the command line:

\[
p = x^3 - 2x - 5
\]

**Display Evaluated Expression**

The `char` converter method forms a MATLAB expression for the polynomial represented by a DocPolynom object. The `dispPoly` method evaluates the expression returned by the `char` method with a specified value for `x`.

```matlab
methods
    function dispPoly(obj,x)
        p = char(obj);
        e = @(x)eval(p);
        y = zeros(length(x));
        disp(['y = ',p])
        for k = 1:length(x)
            y(k) = e(x(k));
            disp([' ',num2str(y(k)),...
                ' = f(x = ',...
                num2str(x(k)),')]'))
        end
    end
end
```

Create a DocPolynom object `p`:

\[
p = \text{DocPolynom([1 0 -2 -5])}
\]

\[
p = x^3 - 2x - 5
\]

Evaluate the polynomial at `x` equal to three values, \([3 5 9]\):

\[
dispPoly(p,[3 5 9])
\]

\[
y = x^3 - 2x - 5
\]

\[
16 = f(x = 3)
\]

\[
110 = f(x = 5)
\]

\[
706 = f(x = 9)
\]
Redefine Indexed Reference

The DocPolynom class redefines indexed reference to support the use of objects representing polynomials. In the DocPolynom class, a subscripted reference to an object causes an evaluation of the polynomial with the value of the independent variable equal to the subscript.

For example, given the following polynomial:

\[ f(x) = x^3 - 2x - 5 \]

Create a DocPolynom object \( p \):

\[
p = \text{DocPolynom}([1 \ 0 \ -2 \ -5])
\]

\[
p = \quad x^3 - 2x - 5
\]

The following subscripted expression evaluates the value of the polynomial at \( x = 3 \) and at \( x = 4 \), and returns the resulting values:

\[
p([3 \ 4])
\]

\[
as = \quad 16 \quad 51
\]

Indexed Reference Design Objectives

Redefine the default subscripted reference behavior by implementing a \texttt{subsref} method.

If a class defines a \texttt{subsref} method, MATLAB calls this method for objects of this class whenever a subscripted reference occurs. The \texttt{subsref} method must define all the indexed reference behaviors, not just a specific case that you want to change.

The DocPolynom \texttt{subsref} method implements the following behaviors:

- \( p(x = [a_1...a_n]) \) — Evaluate polynomial at \( x = a \).
- \( p.coef \) — Access \texttt{coef} property value
- \( p.disp \) — Display the polynomial as a MATLAB expression without assigning an output.
- \( obj = p.method(args) \) — Use dot notation to call methods arguments and return a modified object.
• obj = p.method — Use dot notation to call methods without arguments and return a modified object.

**subsref Implementation Details**

The `subsref` method overloads the `subsref` function.

For example, consider a call to the `polyval` function:

```matlab
p = DocPolynom([1 0 -2 -5])
p =
   x^3 - 2*x - 5
polyval(p.coef,[3 5 7])
an =
   16 110 324
```

The `polyval` function requires the:

- Polynomial coefficients
- Values of the independent variable at which to evaluate the polynomial

The `polyval` function returns the value of \( f(x) \) at these values. `subsref` calls `polyval` through the statements:

```matlab
    case '()'
        ind = s.subs{:};
        b = polyval(a.coef,ind);
```

When implementing `subsref` to support method calling with arguments using dot notation, both the `type` and `subs` structure fields contain multiple elements.

The `subsref` method implements all subscripted reference explicitly, as show in the following code listing.

```matlab
methods
    function b = subsref(a,s)
        switch s(1).type
            case '()'
                ind = s.subs{:};
                b = polyval(a.coef,ind);
            case '.'
                switch s(1).subs
                    case 'coef'
                        b = a.coef;
```

19-20
Define Arithmetic Operators

Several arithmetic operations are meaningful on polynomials. The DocPolynom class implements these methods:

<table>
<thead>
<tr>
<th>Method and Syntax</th>
<th>Operator Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus(a,b)</td>
<td>Addition</td>
</tr>
<tr>
<td>minus(a,b)</td>
<td>Subtraction</td>
</tr>
<tr>
<td>mtimes(a,b)</td>
<td>Matrix multiplication</td>
</tr>
</tbody>
</table>

When overloading arithmetic operators, consider the data types you must support. The plus, minus, and mtimes methods are defined for the DocPolynom class to handle addition, subtraction, and multiplication on DocPolynom — DocPolynom and DocPolynom — double combinations of operands.

Define + Operator

If either p or q is a DocPolynom object, this expression:

\[ p + q \]

Generates a call to a function @DocPolynom/plus, unless the other object is of higher precedence.

The following method overloads the plus (+) operator for the DocPolynom class:

```matlab
function r = plus(obj1,obj2)
```
Here is how the function works:

- Ensure that both input arguments are `DocPolynom` objects so that expressions such as `p + 1` that involve both a `DocPolynom` and a `double`, work correctly.
- Access the two coefficient vectors and, if necessary, pad one of them with zeros to make both the same length. The actual addition is simply the vector sum of the two coefficient vectors.
- Call the `DocPolynom` constructor to create a properly typed object that is the result of adding the polynomials.

**Define - Operator**

Implement the `minus` operator (`-`) using the same approach as the `plus` (`+`) operator.

The `minus` method computes `p - q`. The dominant argument must be a `DocPolynom` object.

```matlab
methods
function r = minus(obj1, obj2)
    obj1 = DocPolynom(obj1);
    obj2 = DocPolynom(obj2);
    k = length(obj2.coef) - length(obj1.coef);
    zp = zeros(1, k);
    zm = zeros(1, -k);
    r = DocPolynom([zp, obj1.coef] + [zm, obj2.coef]);
end
end
```

**Define the * Operator**

Implement the `mtimes` method to compute the product `p*q`. The `mtimes` method implements *matrix* multiplication since the multiplication of two polynomials is the convolution (`conv`) of their coefficient vectors:
methods
    function r = mtimes(obj1,obj2)
        obj1 = DocPolynom(obj1);
        obj2 = DocPolynom(obj2);
        r = DocPolynom(conv(obj1.coef,obj2.coef));
    end
end

Using the Arithmetic Operators

Given the DocPolynom object:

\[ p = \text{DocPolynom}([1 0 -2 -5]); \]

The following two arithmetic operations call the DocPolynom plus and mtimes methods:

\[ q = p+1; \]
\[ r = p*q; \]

to produce

\[ q = x^3 - 2x - 4 \]
\[ r = x^6 - 4x^4 - 9x^3 + 4x^2 + 18x + 20 \]
Designing Related Classes
A Class Hierarchy for Heterogeneous Arrays

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Interfaces Based on Heterogeneous Arrays” on page 20-2</td>
</tr>
<tr>
<td>“Define Heterogeneous Hierarchy” on page 20-2</td>
</tr>
<tr>
<td>“Assets Class” on page 20-5</td>
</tr>
<tr>
<td>“Stocks Class” on page 20-7</td>
</tr>
<tr>
<td>“Bonds Class” on page 20-9</td>
</tr>
<tr>
<td>“Cash Class” on page 20-10</td>
</tr>
<tr>
<td>“Default Object” on page 20-12</td>
</tr>
<tr>
<td>“Operating on an Assets Array” on page 20-14</td>
</tr>
</tbody>
</table>

Interfaces Based on Heterogeneous Arrays

A heterogeneous class hierarchy lets you create arrays containing objects of different classes that are related though inheritance. You can define class methods that operate on these heterogeneous arrays as a whole.

A class design based on heterogeneous arrays provides a more convenient interface than, for example, extracting elements from a cell array and operating on these elements individually. For more information on the design of class hierarchies that support heterogeneous arrays, see “Designing Heterogeneous Class Hierarchies” on page 10-24.

All heterogeneous hierarchies derive from `matlab.mixin.Heterogeneous`.

Define Heterogeneous Hierarchy

Note This example does not use valid terminology or techniques for managing financial assets. The purpose of this example is only to illustrate techniques for defining heterogeneous class hierarchies.

This example implements a system of classes to represent financial assets, such as stocks, bonds, and cash. Classes to represent categories of assets have certain common requirements. Each instance has one of the following:
• Textual description
• Type (stock, bond, or cash)
• Means to determine the current value of the asset

Heterogeneous arrays of these objects need methods that can operate on the whole array. These operations include:

• Creating a table of information about all assets contained in the array
• Graphing the relative contribution of each asset type contained in the array

These requirements are factored into the class that is the root of the hierarchy. The root class derives from `matlab.mixin.Heterogeneous`. In the following diagram, the `Assets` class is the root of the hierarchy. The `Stocks`, `Bonds`, and `Cash` classes provide the specialization required for each type of asset.
Designing Related Classes

```
matlab.mixin.Heterogeneous

cat
getDefaultScalarElement
horzcat
vertcat

Assets

Description: char
Type

getCurrentValue
getDefaultScalarElement
pie
makeReport

Stocks

NumShares: double
Symbol: string
SharePrice: double
Type: string

Stock
gGetCurrentValue
getSharePrice
getPPS

Bonds

FaceValue: double
Yield: double
CurrentYield: double
Type: string

Bonds
gGetCurrentYield
gGetCurrentValue
gRate

Cash

Amount: double
Type: string

Cash
gGetCurrentValue
save
spend
```
Assets Class

The Assets class:

- Derives directly from `matlab.mixin.Heterogeneous`
- Is the root of the heterogeneous hierarchy
- Is abstract
- Is the class of heterogeneous arrays composed of any mixture of Stock, Bond, and Cash objects

Properties

The Assets class defines two properties:

- Description — A general description of the individual asset constrained to be of class `char`.
- Type — The type of asset defined as an abstract property that each subclass implements.

Methods

The Assets class defines these methods:

- `pie` — A sealed method that creates a pie chart showing the relative mix of asset types.
- `makeReport` — A sealed method that creates a report listing the assets.
- `getCurrentValue` — An abstract method that each concrete subclass must implement to return the current value of the asset.
- `getDefaultScalarElement` — `matlab.mixin.Heterogeneous` class method overridden in the Assets class to specify a default object. The Assets class is abstract so it cannot be used as the default object. For more information, see “Default Object” on page 20-12.

Methods in Heterogeneous Hierarchies

Methods defined by the Assets class are either:

- Concrete methods (fully implemented) that subclasses do not override
- Abstract methods (signatures only) that subclasses implement
Concrete methods defined by superclasses in a heterogeneous hierarchy must specify the **Sealed** attribute. Sealing these methods prevents subclasses from overriding methods implemented by the superclass. When calling methods on a heterogeneous array, MATLAB calls the methods defined by the class of the array (**Assets** in this example).

The **pie** and **makeReport** methods are examples of sealed methods that operate on heterogeneous arrays composed of **Stock**, **Bond**, and **Cash** objects.

Abstract methods defined by the superclasses in a heterogeneous hierarchy must specify the **Abstract** attribute. Defining an abstract method in a superclass ensures that concrete subclasses have an implementation for that exact method name. Use these methods element-wise so that each object calls its own method.

The **getCurrentValue** method is an example of an abstract method that is implemented by each subclass to get the current value of each asset.

Each type of subclass object calculates its current value in a different way. If you add another category of asset by adding another subclass to the hierarchy, this class must implement its own version of a **getCurrentValue** method. Because all subclasses implement a **getCurrentValue** method, the **pie** and **makeReport** methods work with newly added subclasses.

For more information on the **Sealed** and **Abstract** method attributes, see “Method Attributes” on page 9-5.

**Assets Class Code**

The **Assets** class and other classes in the hierarchy are contained in a package called **financial**.

```matlab
classdef Assets < matlab.mixin.Heterogeneous
    % file: +financial.@Assets/Assets.m
    properties
        Description char = 'Assets'
    end
    properties (Abstract, SetAccess = private)
        Type
    end
    methods (Abstract)
        % Not implemented by Assets class
        value = getCurrentValue(obj)
    end
    methods (Static, Sealed, Access = protected)
```
function defaultObject = getDefaultScalarElement
    defaultObject = financial.DefaultAsset;
end
end
methods (Sealed)
    % Implemented in separate files
    % +financial.@Assets/pie.m
    % +financial.@Assets/makeReport.m
    pie(assetArray)
    makeReport(assetArray)
end
end

For code listings for pie and makeReport, see “Operating on an Assets Array” on page 20-14.

**Stocks Class**

The Stocks class represents a specific type of financial asset. It is a concrete class that implements the abstract members defined by the Assets class, and defines class properties and methods specific to this type of asset.

**Properties**

The Stocks class defines these properties:

- **NumShares** — The number of shares held for this asset.
- **Symbol** — The ticker symbol corresponding to this stock.
- **Type** — Stocks class implementation of the abstract property defined by the Assets class. This concrete property must use the same attributes as the abstract version (that is, `SetAccess private`).
- **SharePrice** — Dependent property for the price per share. The `get.SharePrice` method obtains the current share price from web services when this property is queried.

**Methods**

The Stocks class defines these methods:

- **Stocks** — The constructor assigns property values and supports a default constructor called with no input arguments.
• **getCurrentValue** — This method is the Stocks class implementation of the abstract method defined by the Assets class. It returns the current value of this asset.

• **get.SharePrice** — The property get method for the dependent SharePrice property returns the current share price of this stock. For information on how to access web services from MATLAB, see the `webread` function.

**Stocks Class Code**

```matlab
classdef Stocks < financial.Assets
    properties
        NumShares double = 0
        Symbol string
    end
    properties (SetAccess = private)
        Type = "Stocks"
    end
    properties (Dependent)
        SharePrice double
    end
    methods
        function sk = Stocks(description,numshares,symbol)
            if nargin == 0
                description = '';  
                numshares = 0;
                symbol = ''; 
            end
            sk.Description = description;
            sk.NumShares = numshares;
            sk.Symbol = symbol;
        end
        function value = getCurrentValue(sk)
            value = sk.NumShares*sk.SharePrice;
        end
        function pps = get.SharePrice(sk)
            % Implement web access to obtain
            % Current price per share
            % Returning dummy value
            pps = 1;
        end
    end
end
```

20-8 Designing Related Classes
Bonds Class

The Bonds class represents a specific type of financial asset. It is a concrete class that implements the abstract members defined by the Assets class and defines class properties and methods specific to this type of asset.

Properties

The Bonds class defines these properties:

- **FaceValue** — Face value of the bond.
- **Yield** — Annual interest rate of the bond.
- **Type** — Bonds class implementation of the abstract property defined by the Assets class. This concrete property must use the same attributes as the abstract version (that is, SetAccess private).
- **CurrentYield** — Dependent property for the current yield, The `get.CurrentYield` property get method obtains the value from web services.

Methods

The Bonds class defines these methods:

- **Bonds** — The constructor assigns property values and supports a default constructor called with no input arguments.
- **getCurrentValue** — This method is the Bonds class implementation of the abstract method defined by the Assets class. It returns the current value of this asset.
- **get.CurrentYield** — The property get method for the dependent CurrentYield property returns the current yield on this bond. For information on how to access web serviced from MATLAB, see the `webread` function.

Bonds Class Code

```matlab
classdef Bonds < financial.Assets
    properties
        FaceValue double = 0
        Yield double = 0
    end
    properties (SetAccess = private)
        Type = "Bonds"
    end
    properties (Dependent)
```
CurrentYield double = 0
end
methods
function b = Bonds(description,facevalue,yield)
    if nargin == 0
        description = '';
        facevalue = 0;
        yield = 0;
    end
    b.Description = description;
    b.FaceValue = facevalue;
    b.Yield = yield;
    b.Type = AssetTypes.Bonds;
end
function mv = getCurrentValue(b)
y = b.Yield;
cy = b.CurrentYield;
if cy <= 0 || y <= 0
    mv = b.FaceValue;
else
    mv = b.FaceValue*y/cy;
end
end
function r = get.CurrentYield(b)
    % Implement web access to obtain
    % Current yield for this bond
    % Returning dummy value
    r = 0.24;
end
end
end

Cash Class

The Cash class represents a specific type of financial asset. It is a concrete class that implements the abstract members defined by the Assets class and defines class properties and methods specific to this type of asset.

Properties

The Cash class defines these properties:

- Amount — The amount of cash held in this asset.
• **Type** — Cash class implementation of the abstract property defined by the Assets class. This concrete property must use the same attributes as the abstract version (that is, SetAccess private).

**Methods**

The Cash class defines these methods:

• **Cash** — The constructor assigns property values and supports a default constructor called with no input arguments.

• **getCurrentValue** — This method is the Cash class implementation of the abstract method defined by the Assets class. It returns the current value of this asset.

• **save** — This method adds the specified amount of cash to the existing amount and returns a new Cash object with the current amount.

• **spend** — This method deducts the specified amount from the current amount and returns a new Cash object with the current amount.

**Cash Class Code**

```plaintext
classdef Cash < financial.Assets
  properties
    Amount double = 0
  end
  properties (SetAccess = private)
    Type = "Cash"
  end
  methods
    function c = Cash(description,amount)
      if nargin == 0
        description = ''; amount = 0;
      end
      c.Description = description;
      c.Amount = amount;
    end
    function value = getCurrentValue(c)
      value = c.Amount;
    end
    function c = save(c,amount)
      newValue = c.Amount + amount;
      c.Amount = newValue;
    end
    function c = spend(c,amount)
  end
end
```

A Class Hierarchy for Heterogeneous Arrays

20-11
newValue = c.Amount - amount;
if newValue < 0
    c.Amount = 0;
    disp('Your balance is $0.00')
else
    c.Amount = newValue;
end
end

Default Object

The design of this class hierarchy uses an abstract root class (Assets). Therefore, the Assets class must specify a concrete class to use as a default object by overriding getDefaultScalarElement. In this case, options include:

• Use one of the existing concrete classes for the default object.
• Define a concrete class in the hierarchy to use for the default object.

This implementation adds the DefaultAsset class to the hierarchy as a subclass of the Assets class. MATLAB creates objects of this class when:

• Creating arrays using indexed assignment with gaps in index numbers
• Loading heterogeneous arrays from MAT-files when MATLAB cannot find the class of an array element.

This diagram shows the addition of the DefaultAsset class:
DefaultAsset Class Code

classdef DefaultAsset < financial.Assets
    % file: +financial.@DefaultAsset/DefaultAsset.m
    properties (SetAccess = private)
        Type = "DefaultAsset"
    end
    methods
        function obj = DefaultAsset
            obj.Description = 'Place holder';
        end
        function value = getCurrentValue(~)
            value = 0;
        end
    end
end

Operating on an Assets Array

The Assets class defines these methods to operate on heterogeneous arrays of asset objects:

- **pie** — Creates a pie chart showing the mix of asset types in the array.
- **makeReport** — Uses the MATLAB table object to display a table of asset information.

To operate on a heterogeneous array, a method must be defined for the class of the heterogeneous array and must be sealed. In this case, the class of heterogeneous arrays is always the Assets class. MATLAB does not use the class of the individual elements of the heterogeneous array when dispatching to methods.

makeReport Method Code

The Assets class makeReport method builds a table using the common properties and getCurrentValue method for each object in the array.

function makeReport(obj)
    numMembers = length(obj);
    descrs = cell(1,numMembers);
    types(numMembers) = "";
    values(numMembers) = 0;
    for k = 1:numMembers
        descrs{k} = obj(k).Description;
        types{k} = obj(k).Type;
        values{k} = obj(k).getCurrentValue(~);
    end
    % ...
types(k) = obj(k).Type;
values(k) = obj(k).getCurrentValue;
end

t = table;
t.Description = descs';
t.Type = types';
t.Value = values';
disp(t)
end

The Assets class pie method calls the getCurrentValue method element-wise on objects in the array to obtain the data for the pie chart.

**pie Method Code**

```matlab
function pie(assetArray)
    stockAmt = 0; bondAmt = 0; cashAmt = 0;
    for k=1:length(assetArray)
        if isa(assetArray(k), 'financial.Stocks')
            stockAmt = stockAmt + assetArray(k).getCurrentValue;
        elseif isa(assetArray(k), 'financial.Bonds')
            bondAmt = bondAmt + assetArray(k).getCurrentValue;
        elseif isa(assetArray(k), 'financial.Cash')
            cashAmt = cashAmt + assetArray(k).getCurrentValue;
        end
    end
    k = 1;
    if stockAmt ~= 0
        label(k) = {'Stocks'};
        pieVector(k) = stockAmt;
        k = k +1;
    end
    if bondAmt ~= 0
        label(k) = {'Bonds'};
        pieVector(k) = bondAmt;
        k = k +1;
    end
    if cashAmt ~= 0
        label(k) = {'Cash'};
        pieVector(k) = cashAmt;
    end
    pie(pieVector, label)
    tv = stockAmt + bondAmt + cashAmt;
stg = {["Total Value of Assets: ",num2str(tv,'%0.2f')]};
```
Create an Assets Array

These statements create a heterogeneous array by concatenating the Stocks, Bonds, and Cash objects. Calling the makeReport and pie methods creates the output shown.

```matlab
s = financial.Stocks('Acme Motor Company',100,string('A'));
b = financial.Bonds('3 Month T',700,0.3);
c(1) = financial.Cash('Bank Account',500);
c(2) = financial.Cash('Gold',500);
assetArray = [s,b,c];
makeReport(assetArray)
pie(assetArray)
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Acme Motor Company'</td>
<td>Stocks</td>
<td>1232.5</td>
</tr>
<tr>
<td>'3 Month T'</td>
<td>Bonds</td>
<td>807.69</td>
</tr>
<tr>
<td>'Bank Account'</td>
<td>Cash</td>
<td>500</td>
</tr>
<tr>
<td>'Gold'</td>
<td>Cash</td>
<td>500</td>
</tr>
</tbody>
</table>
See Also

Related Examples
- “Designing Heterogeneous Class Hierarchies” on page 10-24
- “Validate Property Values” on page 8-24
- “Set and Get Methods for Dependent Properties” on page 8-61