MATLAB® 7
Classes and Object-Oriented Programming

MATLAB®

The MathWorks™

Accelerating the pace of engineering and science
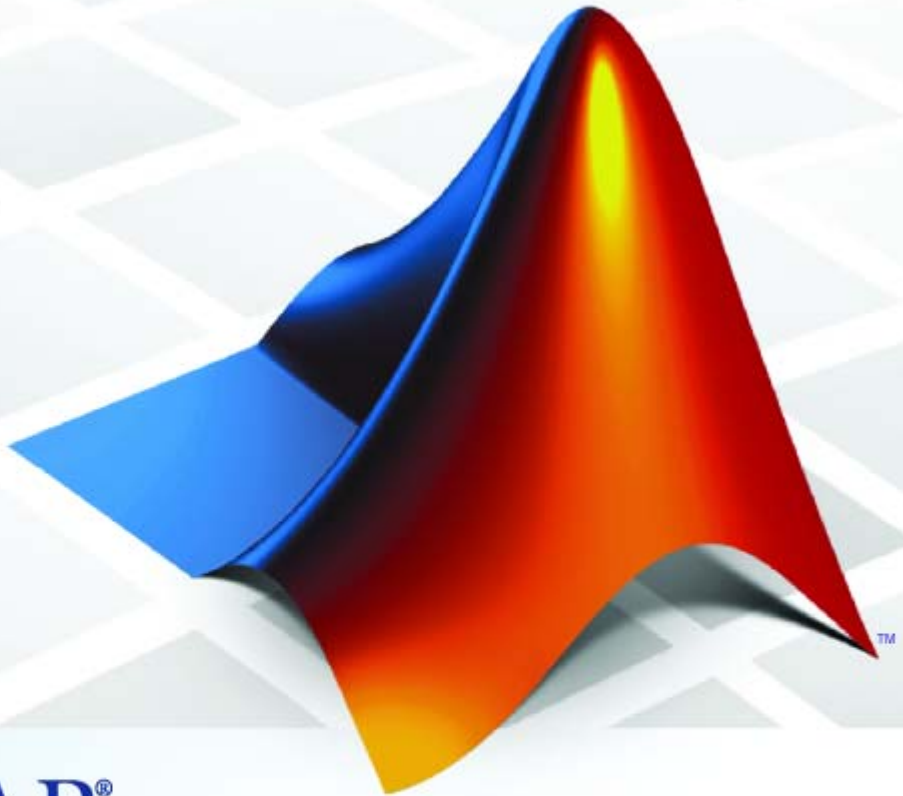
How to Contact The MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Classes and Object-Oriented Programming

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

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

March 2008	Online only	New for MATLAB 7.6 (Release 2008a)

Contents

Working with Classes

3

Value or Handle Class — Which to Use

4

Building on Other Classes

5

Properties — Storing Class Data

6

Methods — Defining Class Operations

7

Events — Sending and Responding to Messages

8

Implementing a Class for Polynomials

9

Designing Related Classes

10

Getting Started

Where to Begin

In this section...
"Video Demo of MATLAB® Classes" on page 1-2
"MATLAB® Programmer Without Object-Oriented Programming Experience" on page 1-2
"MATLAB® Programmer with Object-Oriented Programming Experience" on page 1-2

Video Demo of MATLAB® Classes

You can watch a brief presentation on MATLAB® class development by clicking this link:

Play demo

MATLAB® Programmer Without Object-Oriented Programming Experience

If you create MATLAB programs, but are not defining classes to accomplish your tasks, start with the following sections:

- "Why Use Object-Oriented Design" on page 1-4

- "MATLAB® Classes" on page 2-2

- "Examples to Get Started" on page 2-6

- "Learning Object-Oriented Programming" on page 2-7

MATLAB® Programmer with Object-Oriented Programming Experience

If you are experienced with both MATLAB programming and object-oriented techniques, start with the following sections:

- "Defining Classes — Syntax Overview" on page 2-11

- "Detailed Information and Examples" on page 2-8

- "Compatibility with Previous Versions " on page 2-36
- "MATLAB® and Other OO Languages" on page 2-40

Why Use Object-Oriented Design

In this section...
"Approaches to Writing MATLAB® Programs" on page 1-4
"When Should You Start Creating Object-Oriented Programs" on page 1-8

Approaches to Writing MATLAB® Programs

Creating software applications typically involves designing how to represent the application data and determining how to implement 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 seamlessly in your programs.

Procedural Programming Design

In procedural programming, your design focuses on steps that must be executed to achieve a desired state. You typically represent data as individual variables or fields of a structure and 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 perhaps many operations on the data.

Object-Oriented Programming 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 (an object of class `double` might have a value of 5). The functions defined by the class (called methods) are what implement object behaviors that are common to all objects of a class (you can add two doubles regardless of their values).

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 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 a function (`surf` in this case) is called without the necessary arguments:

```
try
    surf
catch ME
    disp(ME.message)
end
Not enough input arguments.
```

In this code, `ME` is an object of the `MException` class, which is returned by the `catch` statement to the function's workspace. Displaying the value of the object's `message` property returns information about the error (the `surf`

function requires input arguments). However, this is not all the information available in the MException object.

You can list the public properties of an object with the properties function:

```
properties(ME)
Properties for class MException:
    identifier
    message
    cause
    stack
```

Objects Organize Data

The information returned in an MException object is stored in properties, which are much like structure fields. You reference a property using dot notation, as in ME.message. This reference returns the value of the property. For example,

```
class(ME.message)
ans =
char
```

shows that the value of the message property is an array of class char (a text string). The stack property contains a MATLAB struct:

```
ME.stack
ans =
    file: 'U:\bat\A\perfect\matlab\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 50
```

You can simply treat the property reference, ME.stack as a structure and reference its fields:

```
ME.stack.file
ans =
D:\myMATLAB\matlab\toolbox\matlab\graph3d\surf.m
```

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

```
class(ME.stack.file)
ans =
char
```

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

```
strcmp(ME.stack.name,'surf')
ans =
     1
```

Object properties can contain any class of value and can even determine their value dynamically. This provides more flexibility than a structure and is easier to investigate than a cell array, which lacks fieldnames and requires indexing into various cells using array dimensions.

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 quite 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 needed to change, you would have to update the functions you have written to use the new data.

Objects provide an advantage in that objects define their own operations. A requirement of the MException object is that it can generate its own report. The methods that implement an object's operations are part of the object definition (i.e., specified by the class that defines the object). The object definition might be modified many times, but the interface your program (and others) use does not change. Think of your program as a client of the object, which isolates your code from the object's code.

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

```
methods(ME)
Methods for class MException:

addCause     eq           isequal      rethrow
ctranspose   fieldnames   ne           throw
disp         fields       permute      throwAsCaller
display      getReport    reshape      transpose
```

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

```
ME.getReport
ans =
??? Error using ==> surf at 50
Not enough input arguments.
```

Objects often have methods that overload (redefined for the particular object) MATLAB functions (e.g., `isequal`, `fieldnames`, etc.). This 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 doubles. If `ME` and `ME2` are `MException` objects, you can compare them with this statement:

```
isequal(ME,ME2)
```

However, what really happens in this case is MATLAB calls the `MException` `isequal` method because you have passed `MException` objects to `isequal`.

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

```
ME == ME2
```

Of course, 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 Start Creating Object-Oriented Programs

Objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

Simple programming tasks are easily implemented as simple functions, but as the magnitude and complexity of your tasks increase, functions become more complex and difficult to manage.

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

Understanding a Problem in Terms of Its Objects

Thinking in terms of things or objects is simpler and more natural for some problems. You might think of the nouns in your problem statement as the objects you need to define and the verbs as the operations you must perform.

For example, consider performing an analysis of economic institutions. It would be difficult to represent the various institutions as procedures even though they are all actors in the overall economy. Consider banks, mortgage companies, credit unions. You can represent each institution as an object that performs certain actions and contains certain data. The process of designing the objects involves identifying the characteristics of these institutions that are important to your application.

Identify Commonalities. All of these institutions belong in the general class of lending institutions, so all objects might provide a `loan` operation and have a `Rate` property that stores the current interest rate.

Identify Differences. You must also consider how each institution differs. A mortgage company might provide only home mortgage loans. Therefore, the `loan` operation might need be specialized for mortgage companies to provide `fixRateLoan` and `varRateLoan` methods to accommodate two loan types.

Consider Interactions. Institutions can interact, as well. For example, a mortgage company might sell a mortgage to a bank. To support this activity, the mortgage company object would support a `sellMortgage` operation and the bank object would support a `buyMortgage` operation.

You might also define a loan object, which would represent a particular loan. It might need `Amount`, `Rate`, and `Lender` properties. When the loan is sold to another institution, the `Lender` property could be changed, but all other information is neatly packaged within the loan object.

Add Only What Is Necessary. It is likely that these institutions engage in many activities that are not of interest to your application. During the design phase, you need to determine what operations and data an object needs to contain based on you problem definition.

Managing Data. Objects encapsulate the model of what the object represents. If the object represents a kind of lending institution, all the behaviors of lending institutions that are necessary for your application are contained by this object. This approach simplifies the management of data that is necessary in a typical procedural program.

Objects Manage Internal State

In the simplest sense, objects are data structures that encapsulate some internal state, which you access via its methods. When you invoke a method, it is the object that determines exactly what code to execute. In fact, two objects of the same class might execute different code paths for the same method invocation because their internal state is different. The internal workings of the object need not be of concern to your program — you simply use the interface the object provides.

Hiding the internal state from general access leads to more robust code. If a loan object's `Lender` property can be changed only by the object's `newLender` method, then inadvertent access is less likely than if the loan data were stored in a cell array where an indexing assignment statement could damage the data.

Objects provide a number of useful features not available from structures and cell arrays. For example, objects provide the ability to:

- Constrain the data assigned to any given property by executing a function to test values whenever an assignment is made

- Calculate the value of a property only when it is queried and thereby avoid storing data that might be dependent on the state of other data

- Broadcast notices when any property value is queried or changed, to which any number of listeners can respond by executing functions,

- 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 you need to 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

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

An object-oriented design could result in a simpler solution by factoring 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, it is inherited by the various classes you derive from the base class. This 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 is often called an interface class. Incorporating this kind of class into your program design enables you to:

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

For example, suppose you are creating an object to return information about errors that occur during the execution of specific blocks of code. There might be functions that return special types of information that you want to include in an error report only when the error is generated by these functions.

The interface class, from which all error objects are derived, could specify that all error objects must support a getReport method, but not specify how to implement that method. The class of error object created for the functions returning special information could implement its version of the getReport method to handle the different data.

The requirement defined by the interface class is that all error objects be able to display an error report. All programs that use this feature can rely on it being implement in a consistent way.

All of the classes derived from the interface class can create a method called getReport without any name conflicts because it is the class of the object that determines which getReport is called.

Reducing Complexity

Objects reduce complexity by reducing what you need to know to use a component or part of a system. This happens in a couple of ways:

- Implementation details are hidden behind the interfaces defined by objects.

- Rules controlling how objects interact are enforced by object design and, therefore, not left to object users to enforce.

To illustrate these advantages, consider the implementation of a data structure called a doubly linked list.
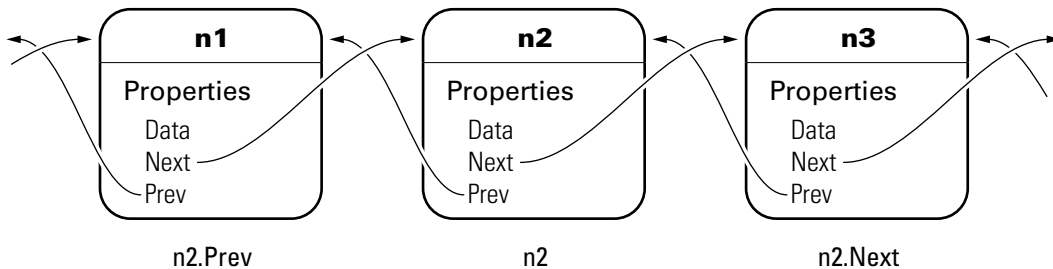
To add a new node to the list, the following steps must occur. First disconnect the nodes:

1 Unlink `n2.Prev` from `n1`

2 Unlink `n2.Next` from `n3`

3 Unlink `n3.Prev` from `n2`

4 Unlink `n1.Next` from `n2`

Disconnect the nodes
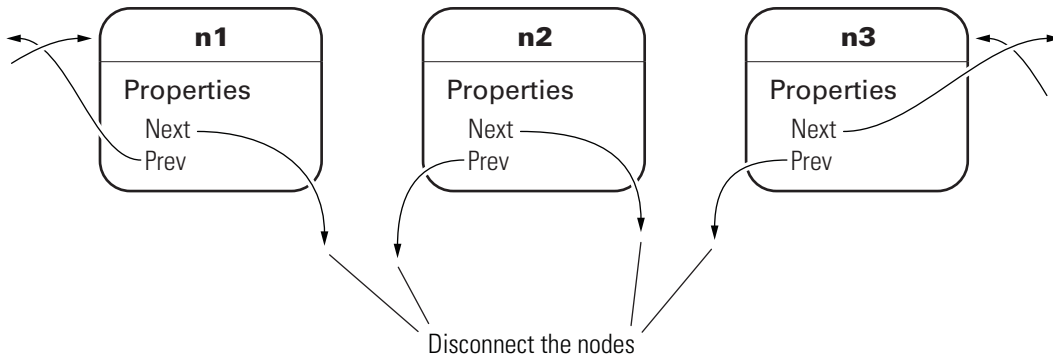
Now connect the new node and renumber:

5 Link new.Prev to n1

6 Link new.Next to n3 (was n2)

7 Link n1.Next to new (will be n2)

8 Link n3.Prev to new (will be n2)

Newly inserted node
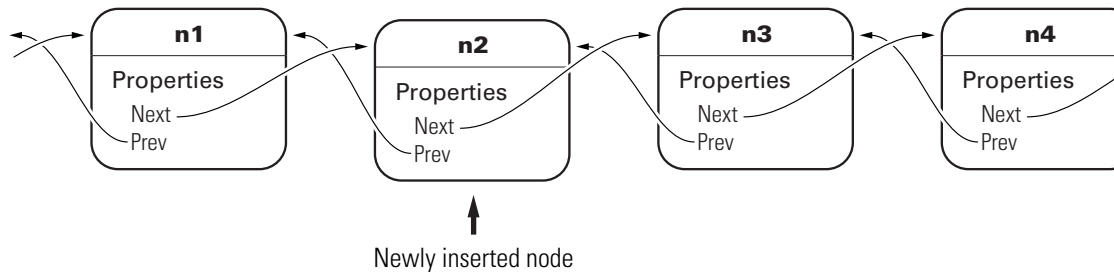
The act of inserting a new node in an existing doubly linked list requires a number of steps. Any application code using a data structure like this can perform these operations. However, defining the linked list nodes as objects enables all the operations like added, deleting, and rearranging nodes to be encapsulated in methods. The code that implements these operations can be well tested, implemented in an optimal way, always up to date with the current version of the class, and can even automatically update old-versions of the objects when they are loaded from MAT-files.

The objects enforce the rules for how the nodes interact by implementing methods to carry out these operations. A single addNode method would perform all the steps listed above. This removes the responsibility for enforcing constraints from the applications that use the objects. It also means the application is less likely to generate errors in its own implementation of the process.

This approach can reduce the complexity of your application code, provide greater consistency across applications, and reduce testing and maintenance requirements.

Fostering Modularity

As you decompose a system into objects (car –> engine –> fuel system –> oxygen sensor), you form modules around natural boundaries. These objects provide interfaces by which they interact with other modules (which might be other objects or functions). Often the data and operations behind the interface are hidden from other modules to segregate implementation from interface.

Classes provide three levels of control over code modularity:

- Public — Any code can access this particular property or call this method.

- Protected — Only the object's own methods and those of the object's whose class has been 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.

Reduce Code Redundancy

Suppose your application requires a number of dialog windows to interact with users. By defining a class containing all the common aspects of the

dialog windows, and then deriving the specific dialog classes from this base class, you can:
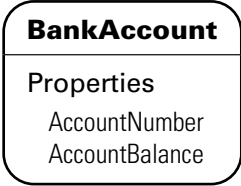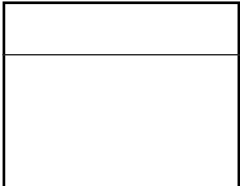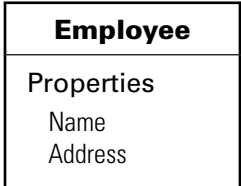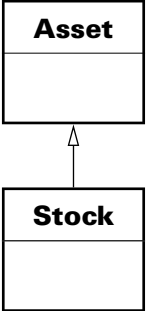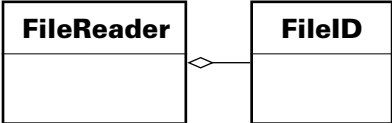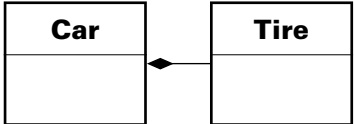
- Reuse code that is common to all dialog window implementations
- Reduce code testing effort due to common code
- Provide a common interface to dialog developers
- Enforce a consistent look and feel
- Apply global changes to all dialog windows more easily

Learning More

See "MATLAB® Classes" on page 2-2 to learn more about writing object-oriented MATLAB programs.

Class Diagrams Used in This Documentation

The diagrams representing classes that appear in this documentation follow the conventions described in the following legend.

Concept	Graphical representation	Example
Object		**BankAccount** Properties AccountNumber AccountBalance
Class		**Employee** Properties Name Address
is_a		**Asset** ↑ **Stock**
has_a	(aggregation)	**FileReader** ◇— **FileID**
	(composition)	**Car** ◆— **Tire**

2

MATLAB® Classes Overview

MATLAB® Classes

Classes in the MATLAB® Language

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:

```
>> a = 7;
>> b = 'some string';
>> whos
  Name      Size            Bytes  Class

  a         1x1                 8  double
  b         1x11               22  char
```

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 might be double, single, or other types of numbers. Some variables can contain different classes of values like `cells`.

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. However, 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 would know how to add polynomial objects because the polynomial class defines this operation.

When you define a class, you overload special MATLAB functions (`plus.m` for the addition operator) that are called by the MATLAB runtime when those operations are applied to an object of your class.

See "Example — A Polynomial Class" on page 9-2 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 that are 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 (i.e., subclasses).

- Packages — Directories that define a scope for class and function naming

These are general descriptions of these components and concepts. This documentation describes all of these components in detail.

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 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 defined by the superclass. A subclass is more specific than its superclass and might add new properties, methods, and events to those 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 Positive numbers. All three sets are subsets of Real numbers, 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.
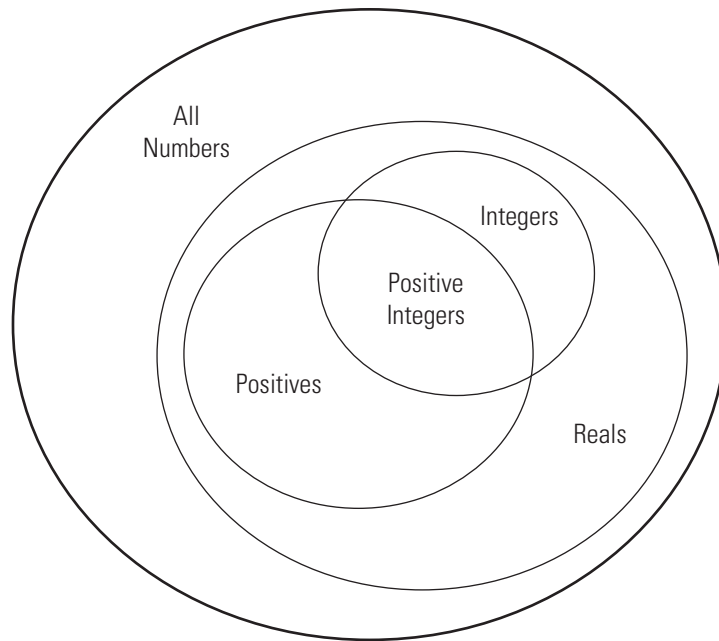
The "is a" relationship is a good way to determine if it is appropriate to define a particular subset in terms of existing supersets. For example, each of the following statements makes senses:

- 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 a 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 very 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 of 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 needed 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.

Examples to Get Started

The following examples illustrate some basic features of MATLAB classes.

"Developing Classes — Typical Workflow" on page 2-23 — applies object-oriented thinking to a familiar concept to illustrate the process of designing classes.

"Using Objects to Write Data to a File" on page 2-30 — shows advantages of using objects to define certain operations and how smoothly object fit in a function-oriented workflow.

"Example — Representing Structured Data" on page 2-43 — shows the application of object-oriented techniques to managing data.

"Example — Implementing Linked Lists" on page 2-52 — using a handle class to implement a doubly linked list.

Learning Object-Oriented Programming

The following references can help you develop a basic understanding of object-oriented design and concepts.

- Shalloway, A., J. R. Trott, *Design Patterns Explained A New Perspective on Object-Oriented Design.*. Boston, MA: Addison-Wesley 2002.

- Gamma, E., R. Helm, R. Johnson, J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley 1995.

- Freeman, E., Elisabeth Freeman, Kathy Sierra, Bert Bates, *Head First Design Patterns*. Sebastopol, CA 2004.

- See Wikipedia® :Object Oriented Programming

Detailed Information and Examples

Rapid Access to Information

This section provides a gateway to both conceptual information and example implementations. It enables you to scan the information available for broad topics

Topic	Background Information and Discussion	Code Examples
Classes		
	List of all class member attributes: Attribute Tables "MATLAB® Classes" on page 2-2 for an introduction to object-oriented programming concepts. "Class Overview" on page 3-2 for an overview of classes features.	"Developing Classes — Typical Workflow" on page 2-23 for a simple example "Example — Representing Structured Data" on page 2-43 "Example — Implementing Linked Lists" on page 2-52 "Example — A Polynomial Class" on page 9-2 "Example — A Simple Class Hierarchy" on page 10-2 "Example — Containing Assets in a Portfolio" on page 10-18
Attributes	"Class Attributes" on page 3-5 for a list of class attributes	
	"Hierarchies of Classes — Concepts" on page 5-2 describes how classes can be built on other classes	"Example — A Simple Class Hierarchy" on page 10-2
	"Creating Subclasses — Syntax and Techniques" on page 5-6 "Modifying Superclass Methods and Properties" on page 5-8	"Specializing the dlnode Class" on page 2-59

(Continued)

Topic	Background Information and Discussion	Code Examples
Kinds of classes	"Comparing Handle and Value Classes" on page 4-2 "The Handle Base Class" on page 4-10 — a detailed description of the abstract class.	"Example — Implementing Linked Lists" on page 2-52
Properties		
	"Defining Properties" on page 6-4 for an overview of what properties are and how to use them	
	"Property Definition Block" on page 6-4 shows how to specify initial values	"Restricting Properties to Specific Values" on page 2-46
Attributes	"Specifying Property Attributes" on page 6-6 for a list of property attributes	"Dependent Properties" on page 2-48
	"Dynamic Properties — Adding Properties to an Instance" on page 6-17	"Attaching Data to the Object" on page 6-18
Methods		
	"Class Methods" on page 7-2 for an overview of methods	
Attributes	"Method Attributes" on page 7-4 for a list of method attributes	
	"Class Constructor Methods" on page 7-14 for information about constructor methods	"Simplifying the Interface with a Constructor" on page 2-47
	"Handle Class Delete Methods" on page 4-13	

(Continued)

Topic	Background Information and Discussion	Code Examples
	"Controlling Property Access" on page 6-10	"Restricting Properties to Specific Values" on page 2-46
	"Implementing a Set/Get Interface for Properties" on page 4-18	
Events		
	"Events and Listeners — Concepts" on page 8-2 for an overview of how events work "Defining Events and Listeners — Syntax and Techniques" on page 8-8 for the syntax used to define events and listeners	"Example — Using Events to Update Graphs" on page 8-16 for a complete example that uses events and listeners, including a property listener

Defining Classes — Syntax Overview

Class Building Blocks

The basic components in the class definition are blocks describing the whole class and specific aspects of its definition:

- *classdef block* contains the class definition within a file that starts with the `classdef` keyword and terminates with the `end` keyword.

- *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.

- *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.

- *events block* (one for each unique set of attribute specifications) contains the names of events that are declared by this class. The events blocks starts with the `events` keyword and terminates with the `end` keyword.

Note that `properties`, `methods`, and `events` are keywords only within a `classdef` block.

More In Depth Information

"Defining Classes — Syntax" on page 3-4 for more detail on class syntax.

"Defining Properties" on page 6-4 for information on specifying properties.

"Class Methods" on page 7-2 for information on specifying methods.

"Defining Events and Listeners — Syntax and Techniques" on page 8-8 for information on the use of events.

Attribute Tables for a list of all attributes.

Class Directories

There are two different types of class directories:

- A directory on the MATLAB® path containing class definition M-files. The name of each file must match the class (and constructor) name. You must define the class entirely in this file, but you can put multiple classes in this single directory.

- An @*class_name* directory that resides inside a directory on the MATLAB path. You can define only one class in the @*class_name* directory, but can use multiple files.

See "Methods in Separate Files" on page 7-7 for more information.

More Information on Class Directories

See "Organizing Classes in Directories" on page 3-8 for more information on class directories and see "Scoping Classes with Packages" on page 3-13 for information on using classes from package directories.

The Classdef Block

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.

Assigning Class Attributes

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 class_name
    ...
end
```

One or more attribute values assigned:

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

See "Class Attributes" on page 3-5 for a list of attributes and a discussion of the behaviors they control.

Specifying Superclasses

To define a class in terms of one or more other classes by specifying the superclasses on the `classdef` line:

```
classdef class_name < superclass_name
    ...
end
```

See "Creating Subclasses — Syntax and Techniques" on page 5-6 for more information.

Specifying Properties

You can control aspects of property definition in the following ways:

- Specifying an initial value for each property individually

- Assigning attribute values on a per block basis

- Defining methods that execute when the property is set or queried

Defining an Initial Value

Within a properties block, you can control the individual property's initial value. Initial values can be constant values or MATLAB expressions. Expressions cannot reference variables. For example:

```
classdef class_name
   properties
      PropertyName % No initial value assigned
      PropertyName = 'some text';
      PropertyName = sin(pi/12); % Expression returns initial value
   end
end
```

The evaluation of initial values occurs only once before the class is first instantiated. Property values that are not specified in the definition are set to empty ([]).

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 class_name
   properties
      PropertyName % No initial value assigned
      PropertyName = sin(pi/12); % Expression returns initial value
   end
   properties (SetAccess = private, GetAccess = private)
      Stress
      Strain
   end
end
```

In this case, both the Stress and Strain properties can be modified only by methods in the same class definition as this properties block (i.e., the properties set access is private).

"Specifying Property Attributes" on page 6-6 provides a description of property attributes.

Property Access Methods

You can define a method that is called whenever a property is set (a set access method) or a method that is called whenever a property is queried (a get access method). Define these methods in methods blocks that specify no attributes and have the following syntax:

```
methods
   function value = get.PropertyName(object)
      ...
   end
   function obj = set.PropertyName(obj,value)
      ...
   end
end
```

"Property Access Methods" on page 6-10 for more information on these methods.

"Defining Properties" on page 6-4 for information on properties.

Specifying Methods

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. Assignment to property values can be made at that time. You must terminate all method functions with an end statement.

```
classdef class_name
   methods
      function obj = class_name(arg1,arg2,...)
         obj.Prop1 = arg1;
         ...
      end
      function normal_method(obj,arg2,...)
         ...
      end
   end
   methods (Static = true)
      function normal_method(arg1,...)
         ...
```

```
            end
        end
    end
```

Methods In Separate Files

To define a method in a separate file in the class @-directory, create the function in a separate M-file, but do not use a method block in that file. Name the M-file with the function's name, as with any M-file function.

If you want to specify method attributes for a method in a separate file, you must include the method's signature in a methods block in the classdef block. For example, the following code shows a method that is declared with Access private in the classdef block, but is implemented in a separate file. Do not include the function or end keywords in the methods block.

```
classdef class_name
    methods (Access = private)
        output = myFunc(obj,arg1,arg2)
    end
    ...
end
```

In an M-file named myFunc.m, in the @-directory, define the function:

```
function output = myFunc(obj,arg1,arg2)
    ...
end
```

You need to include the method signature in the file with the classdef block only if you want to specify attributes for that method. Otherwise, you can just implement the method as an M-file function in the @-directory.

Defining Private Methods

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

See "Method Attributes" on page 7-4 for a list of method attributes.

More Detailed Information On Methods

See "Class Methods" on page 7-2 for more information about methods.

Specifying Events and Listeners

To define an event, you simply declare a name for the event in the `events` block. Then one of the class's methods triggers the event using the `notify` method, which is inherited from the `handle` class. Note that only classes derived from the `handle` class can define events. For example, the following class:

- Defines an event named `StateChange`

- Triggers the event using the inherited `notify` method.

```
classdef class_name < handle % Subclass handle
    events % Define an event called StateChange
        StateChange
    end
    ...
    methods
        function upDataGUI(obj)
            ...
            % Broadcast notice that StateChange event has occured
            notify(obj,'StateChange');
        end
    end
end
```

Listening for the Event

Any number of objects might be listening for the `StateChange` event to occur. When `notify` executes, the MATLAB runtime calls all registered listener callbacks and passes the handle of the object generating the event and an event structure to these functions. To register a listener callback, use the `addlistener` method of the handle class.

See "Defining Events and Listeners — Syntax and Techniques" on page 8-8

Specifying Attributes

For a quick reference to all attributes, see Attribute Tables

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` (the default):

```
properties (SetAccess = private)
    ScreenSize = getScreenSize;
end
```

All class definition blocks (`classdef`, `properties`, `methods`, and `events` support specific attributes, all of which have default values. You need to specify attribute values only in cases where you want to change from the default value to another predefined value.

Attribute Descriptions

For lists of supported attributes see:

- "Class Attributes" on page 3-5

- "Specifying Property Attributes" on page 6-6

- "Method Attributes" on page 7-4

- "Event Attributes" on page 8-7

Specifying Attribute Values

When you specify attribute values, all the components defined within the definition block are affected by the attribute specification. 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`.

Note that defining properties with different attribute settings require multiple `properties` blocks.

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

You can specified multiple attributes as a comma-separated list, as shown above.

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

```
classdef (Sealed = true) myclass
   ...
end
```

Simpler Syntax for true/false Attributes

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

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

is the same as:

```
methods (Static = true)
   ...
end
```

Use the `not` operator before an attribute name to define it as `false`:

```
methods (~Static)
   ...
```

```
    end
```

is the same as:

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

Sample Class Syntax

The following code shows the syntax of a typical class definition. Note that this is not a functioning class because it references functions that have not been implemented. The purpose of this section is to illustrate a variety of syntactic constructions.

```
classdef (ConstructOnLoad) employee < handle
    % Class help goes here
    properties
        Name % Property help goes here
    end

    properties (Dependent)
        JobTitle
    end

    properties (Transient)
        OfficeNumber
    end

    properties (SetAccess = protected, GetAccess = private)
        EmpNumber
    end

    events
        BackgroundAlert
    end

    methods
        function Eobj = employee(name)
            % Method help here
                Eobj.Name = name;
```

```
                 Eobj.EmpNumber = employee.getEmpNumber;
        end

        function result = backgroundCheck(obj)
           result = queryGovDB(obj.Name,obj.SSNumber);
             if result == false
                notify(obj,'BackgroundAlert');
             end
        end

        function jobt = get.JobTitle(obj)
           jobt = currentJT(obj.EmpNumber);
        end

        function set.OfficeNumber(obj,setvalue)
           if isInUse(setvalue)
              error('Not available')
           else
              obj.OfficeNumber = setvalue;
           end
        end
    end

    methods (Static)
        function num = getEmpNumber
           num = queryDB('LastEmpNumber') + 1;
        end
    end
 end
```

Functions Used with Objects

The MATLAB language provides the follow functions that enable you to work more effectively with objects.

Function	Purpose
class	Return class of object
events	Display class event names
isa	Determine whether argument is object of specific class

Function	Purpose
isobject	Determine whether input is MATLAB object
methods	Information on class methods
methodsview	Information on class methods in separate window
properties	Display class property names

Developing Classes — Typical Workflow

In this section...

Formulating a Class

This example discusses the design and implementation of a simple class. 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 current status (open, closed, etc.)

You need to perform certain operations on a bank account:

- Deposit money
- Withdraw money

You might also want the bank account to send a notice if the balance is too low and an attempt is made to withdraw money. When this event occurs, the bank account can broadcast a notice and other entities that are designed to listen for these notices, such as an account manager program, can take action. In this case, the status of all bank accounts is determined by an account manager program that looks at 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.

In MATLAB® classes, data is stored in properties, operations are implemented with methods, and notifications are supported with events and listeners. Therefore, the bank account class needs the components discussed in the following sections.

Defining Class Data

The class needs to define properties to store the account number and the account balance:

- `AccountNumber` — This property is assigned a value when you create an instance of the class.

- `AccountBalance` — This property is modified by the class operation of depositing and withdrawing money.

- `AccountStatus` — This property is set to an initial value when an instance of the class is created. It is then changed by methods from the `AccountManager` class whenever the value of the `AccountBalance` falls below `0`.

The first two properties contain information that only the class should be able to change, so the `SetAccess` attribute should be set to `private` (only class methods can set these values). The `AccountStatus` is determined by an external program that needs access to the property, so its `SetAccess` attribute is left as the default, which is `public` (any code can access this property value).

Defining Class Operations

There are three operations that the class must be able to perform, so there needs to be three methods:

- `deposit` — Update the `AccountBalance` property when a deposit transaction occurs

- `withdraw` — Update the `AccountBalance` property when a withdrawal transaction occurs

- `BankAccount` — Create an initialized instance of the class

Defining Class Events

Bank accounts with negative balances have their status changed by the account manager program, as described above. To implement this action, the BankAccount class triggers an event when a withdrawal causes a negative balance to occur. Therefore, the triggering of the InsufficientsFunds event occurs from within the withdraw method.

To define an event, you simply define a name within an events block. The event is triggered by a call to the notify handle class method. Note that this is not a predefined event; it could be named with any string and you can trigger this event with any action you choose.

Implementing the BankAccount Class

It makes sense for there to be only one set of data associated with any instance 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, the BankAccount class should be implemented as a handle class. All copies of a given handle object refer to the same data.

Display Fully Commented Example Code

You can display the code for this example in a popup window that contains detailed comments and links to related sections of the documentation:

BankAccount class

AccountManager class

You can open both class files in your editor by clicking this link:

Open in editor

Class Definition

```
classdef BankAccount < handle
   properties (Hidden)
      AccountStatus = 'open';
   end
% The following properties can be set only by class methods
```

```matlab
    properties (SetAccess = private)
        AccountNumber
        AccountBalance = 0;
    end
% Define an event called InsufficientFunds
    events
        InsufficientFunds
    end
    methods
        function BA = BankAccount(AccountNumber,InitialBalance)
            BA.AccountNumber = AccountNumber;
            BA.AccountBalance = InitialBalance;
% Calling a static method requires the class name
% addAccount registers the InsufficientFunds listener on this instance
            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
            newbal = BA.AccountBalance - amt;
            BA.AccountBalance = newbal;
% If a withdrawal results in a negative balance,
% trigger the InsufficientFunds event using notify
            if newbal < 0
                notify(BA,'InsufficientFunds')
            end
        end % withdraw
    end % methods
end % classdef
```

Implementing the AccountManager Class

The `AccountManager` class provides two methods that implement and register a listener for the `InsufficientsFunds` event, which is defined for all `BankAccount` objects. The `BankAccount` class constructor method calls `addAccount` to register the listener for the instance being created.

Class Definition

```
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 addAccount(BA)
% Call the handle addlistener method
% Object BA is a handle class
         addlistener(BA, 'InsufficientFunds', ...
            @(src, evnt)AccountManager.assignStatus(src));
      end
   end
end
```

Note that the `AccountManager` class is never instantiated. It serves as a container for the event listener used by all `BankAccount` objects.

Using the BankAccount Class

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

```
BA = BankAccount(1234567,500);
BA.AccountNumber
ans =
```

```
1234567
BA.AccountBalance
ans =
    500
BA.AccountStatus
ans =
open
```

Now suppose you make a withdrawal of $600, which results in a negative account balance:

```
BA.withdraw(600)
BA.AccountBalance
ans =
  -100
BA.AccountStatus
ans =
overdrawn
```

When the $600 withdrawal occurred, the `InsufficientsFunds` event was triggered. Because the `AccountBalance` is not less than –$200, the `AccountStatus` was set to `overdrawn`:

```
BA.withdraw(200)
BA.AccountBalance
ans =
  -300
BA.AccountStatus
ans =
closed
```

Now the `AccountStatus` has been set to `closed` by the listener and further attempts to make withdrawals are blocked:

```
BA.withdraw(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:

```
BA.deposit(700)
```

```
BA.AccountStatus
ans =
open
BA.withdraw(100)
BA.AccountBalance
ans =
   300
```

Using Objects to Write Data to a File

In this section...
"Flexible Workflow" on page 2-30
"Performing a Task with an Object" on page 2-30
"Using Objects in Functions" on page 2-32

Flexible Workflow

The MATLAB® language does not require you to define classes for all the code you write. You can use objects along with ordinary functions. This section illustrates the use of an object that implements the basic task of writing text to a file. Then this object is used in a function to write a text file template for a class definition.

Performing a Task with an Object

One of the advantages of defining a class instead of simply writing a function to perform a task is that classes provide better control over related data. For example, consider the task of writing data to a file. It involves the following steps:

- Opening a file for writing and saving the file identifier

- Using the file identifier to write data to the file

- Using the file identifier to close the file

The Filewriter Class

This simple class definition illustrates how you might create a class to write text to a file. It shows how you can use a class definition to advantage by:

- Hiding private data — The caller does not need to manage the file identifier.

- Ensuring only one file identifier is in use at any time — Copies of handle objects reference the same file identifier as the original.

- Providing automatic file closing when the object is deleted — the object's `delete` method takes care of cleanup without needing to be called explicitly.

This class is derived from the `handle` class so that a `Filewriter` object is a handle object. All copies of handle objects reference the same internal data so there will be only one file identifier in use, even if you make copies of the object. Also, handle classes define a `delete` method which is called automatically when a handle object is destroyed. This example overrides the `delete` method to close the file before the file identifier is lost and the file is left open.

```
classdef Filewriter < handle
 % Property data is private to the class
    properties (SetAccess = private, GetAccess = private)
        FileID
    end % properties

    methods
 % Construct an object and
 % save the file ID
        function obj = Filewriter(filename)
            obj.FileID = fopen(filename,'a');
        end

        function writeToFile(obj,text_str)
            fprintf(obj.FileID,'%s\n',text_str);
        end
 % Delete methods are always called before a object
 % of the class is distroyed
        function delete(obj)
            fclose(obj.FileID);
        end
    end   % methods
end % class
```

Using a Filewriter Object

Note that the user provides a file name to create a `Filewriter` object, and then uses the class `writeToFile` method to write text to the file. The following statements create a file named `mynewclass.m` and write one line to it. The `clear all` command deletes the `Filewriter` object, which causes its `delete` method to be called and the file is closed.

```
>> fw = Filewriter('mynewclass.m');
```

```
>> fw.writeToFile('classdef mynewclass < handle')
>> clear fw
>> type mynewclass

classdef mynewclass < handle
```

Using Objects in Functions

Filewriter objects provide functionality that you can use from functions and within other classes. You can create an ordinary function that uses this object, as the writeClassFile function does below.

This example creates only one simple class template, but another version might accept a cell array of attribute name/value pairs, method names, and so on.

```
function writeClassFile(classname,superclass)
 % Use a Filewriter object to write text to a file
    fw = Filewriter([classname '.m']);
    if nargin > 1
        fw.writeToFile(['classdef ' classname ' < ' superclass])
    else
        fw.writeToFile(['classdef ' classname])
    end
    fw.writeToFile('    properties ')
    fw.writeToFile(' ')
    fw.writeToFile('    end % properties')
    fw.writeToFile(' ')
    fw.writeToFile('    methods ')
    fw.writeToFile(['       function obj = ' classname '()'])
    fw.writeToFile(' ')
    fw.writeToFile('       end')
    fw.writeToFile('    end % methods')
    fw.writeToFile('end % classdef')
    delete(fw) % Delete object, which closes file
end
```

To create a class file template, call writeClassFile with the name of the new class and its superclass. Use the type command to display the contents of the file:

```
>> writeClassFile('myNewClass','handle')
>> type myNewClass

classdef myNewClass < handle
    properties

    end % properties

    methods
        function obj = myNewClass()

        end
    end % methods
end % classdef
```

More Information on These Techniques

"The Handle Base Class" on page 4-10

"Handle Class Delete Methods" on page 4-13

Using the Editor and Debugger with Classes

Referring to Class Files

Classes are defined in M-files just like scripts and functions. To use the editor or debugger with a class file, you must use the full class name. For example, suppose the M-file for a class, myclass.m is in the following location:

```
+PackDir1/+PackDir2/@myclass/myclass.m
```

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

```
edit PackDir1.PackDir2.myclass
```

You could also use path notation:

```
edit +PackDir1/+PackDir2/@myclass/myclass
```

If myclass.m is not in an @-directory, then you can type:

```
edit +PackDir1/+PackDir2/myclass
```

To refer to functions inside a package directory you can use dot or path separators:

```
edit PackDir1.PackDir2.packFunction
edit +PackDir1/+PackDir2/packFunction
```

To refer to a function defined in its own M-file inside of a class @-directory, you can use the following:

```
edit +PackDir1/+PackDir2/@myclass/myMethod
```

Debugging Class Files

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

Modifying Classes

If you modify a class definition (i.e., change attributes or the specification of `classdef`, `properties`, `methods`, or `events` blocks), then these changes might not take affect immediately unless there are no instances of the class. MATLAB does not use a modified class definition while any instance of a previous definition exists in MATLAB. Instead, MATLAB continues to use the old definition until all instances are cleared.

You can use the `clear` command:

```
clear classes
```

to clear the current workspace of all variables, clear all functions (which might have persistent variables holding class instances), and clear all classes that are not instantiated. Note that clearing classes clears all variables in the workspace.

However, you are not required to issue the `clear classes` command after changing the code in a method's function definition. MATLAB executes the changed method the next time the method is called.

Compatibility with Previous Versions

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 that is terminated with an `end` statement.

- 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 prior to Version 7.6 and current class definitions (i.e., using `classdef`). Therefore, you cannot subclass an old class to create a new version.

Only One @-Directory per Class

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

Private Methods

You do not need to define private directories in class directories 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:

- The `class` function is not used.

- It must call the superclass constructor only if you need 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 Prior to Version 7.6

```
    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);
```

The same `Stock` class constructor is now written as shown below. The inheritance is defined on the `classdef` line and the constructor is defined within a `methods` block.

Constructor Function for Version 7.6

```
classdef Stock < Asset
   ...
   methods

      function s = Stock(description,num_shares,share_price)
% Call superclass constructor to pass arguments
```

2-37

```
            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: "How to Use Properties" on page 6-2

- Handle classes: "Comparing Handle and Value Classes" on page 4-2

- Events and listeners: "Events and Listeners — Concepts" on page 8-2

- Class member attributes: Attribute Tables

- Abstract classes: "Abstract Classes and Interfaces" on page 5-21

- Dynamic properties: "Dynamic Properties — Adding Properties to an Instance" on page 6-17

- Ability to subclass MATLAB built-in classes: "Creating Subclasses — Syntax and Techniques" on page 5-6

- Packages for scoping functions and classes: "Scoping Classes with Packages" on page 3-13. Packages are not supported for classes created prior to MATLAB Version 7.6 (i.e., classes that do not use classdef).

Examples of Old and New

The MATLAB Version 7.6 implementation of classes uses significantly different syntax from previous releases. However, classes written in previous versions continue to work in this and future versions. 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 re-implement examples using the latest syntax. These same classes were implemented in the original MATLAB Classes and Objects documentation and provide a means for comparison.

"Example — A Polynomial Class" on page 9-2

"Example — A Simple Class Hierarchy" on page 10-2

"Example — Containing Assets in a Portfolio" on page 10-18

Obsolete Documentation

Documentation for MATLAB Classes and Objects prior to Version 7.6 is available here.

MATLAB® and Other OO Languages

In this section...
"Some Differences from C++ and Sun™ Java™ Code" on page 2-40
"Common Object-Oriented Techniques" on page 2-42

Some Differences from C++ and Sun™ Java™ Code

If you are accustomed to programming in other object-oriented languages, such as C++ or the Java™ language, you will find that the MATLAB® programming language differs from these languages 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 property values are assigned or queried by code external to object methods. For example, the following statement:

```
myobj.Material = 'plastic';
```

assigns the string `plastic` to the `Material` property of `myobj`. However, before making the actual assignment, `myobj` executes a method called `set.Material` (assuming the class of `myobj` defines this method) which can perform any operations on the data to check its validity, set other values, and so on. See "Controlling Property Access" on page 6-10 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 6-7 for a full list of property attributes.

Pass By Reference

In MATLAB classes, variables are not passed by reference. However, MATLAB classes support two kinds of classes that behave in different ways with respect to referenced data: value classes and handle classes.

Value Classes. When writing methods that update an object, you must pass back the updated object and use an assignment statement. For example, a value class implementation of a set method for a property requires the object to be returned with the new value set.

```
A = set(A,'PropertyName',PropertyValue);
```

Handle Classes. If A is a handle object, then there is only one copy of the data so setting the value of a property changes the data that is referred to by the handle variable. For example, a handle class behaves like Handle Graphics® objects:

```
set(A,'PropertyName',PropertyValue);
```

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 software uses the left-most object to select the method to call. However, MATLAB can dispatch to a method of an argument in any position within an argument list if the class of that argument is superior to the other arguments.

See "Specifying Class Precedence" on page 3-11 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.

Common Object-Oriented Techniques

This table provides links to sections where object-oriented techniques commonly used by other object-oriented languages are discussed.

Technique	How to Use in MATLAB
Operator overloading	"Overloading Operators" on page 7-31
Multiple inheritance	"Using Multiple Inheritance" on page 5-12
Subclassing	"Creating Subclasses — Syntax and Techniques" on page 5-6
Destructor	"Handle Class Delete Methods" on page 4-13
Data member scoping	"Property Attributes" on page 6-7
Packages (scoping classes)	"Scoping Classes with Packages" on page 3-13
Named constants	"Defining Named Constants" on page 3-25
Static members	"Static Methods" on page 7-22
Constructor	"Class Constructor Methods" on page 7-14
Copy constructor	No direct equivalent
Reference/reference classes	"Comparing Handle and Value Classes" on page 4-2
Abstract class/Interface	"Abstract Classes and Interfaces" on page 5-21
Garbage collection	"Object Lifecycle" on page 4-14
Instance properties	"Dynamic Properties — Adding Properties to an Instance" on page 6-17
Importing classes	"Importing Classes" on page 3-23
Events and Listeners	"Events and Listeners — Concepts" on page 8-2

Example — Representing Structured Data

In this section...

Display Fully Commented Example Code

Open class code in a popup window — Use this link if you want to see the final code for this class annotated with links to descriptive sections.

Open class definition file in the MATLAB® editor. — Use this link if you want to save and modify your version of the class.

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

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. While a MATLAB `struct` with field names describing the particular data element is a useful way to organize data, the use of a class to define both the data storage (properties) and operations you can perform on that data (methods) provides advantages, as this example illustrates.

Concepts on Which This Example Is Based.

For purposes of this example, the data represents tensile stress/strain measurements, which 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

The following table describes the structure of the data.

Data	Description
Material	Character string identifying the type of material tested
Sample number	Number of a particular test sample
Stress	Vector of doubles representing the stress applied to the sample during the test.
Strain	Vector of doubles representing the strain at the corresponding values of the applied stress.
Modulus	Double defining an elastic modulus of the material under test, which is calculated from the stress and strain data

Defining the TensileData Class

This class is designed to store data, so it defines a property for each of the data elements. The following class block defines five properties and specifies their initial values according to the type of data each will contain. Defining initial values is not required, but can be useful if a property value is not assigned during object creation.

Note that 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.

```
classdef TensileData
   properties
      Material = '';
      SampleNumber = 0;
      Stress
      Strain
      Modulus = 0;
   end
end
```

Creating an Instance and Assigning Data

Create a TensileData object and assign data to it with the following statements:

```
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 Array

You can treat the TensileData object (td in the statements above) much as you would any MATLAB structure array. However, defining a data structure as a class has advantages:

- Users cannot accidently misspell a field name without getting an error. For example, typing the following:

  ```
  >>td.Modulis = ...
  ```

  would simply add a new field to a structure array, 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 whos 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.

The next section describes how to add type checking and how to restrict property access in the TensileData class.

Restricting Properties to Specific Values

You can restrict the values to which a property can be set by defining a property set access method. MATLAB software then calls this function whenever a value is set for a property, including when creating the object.

Defining the Material Property Set Function

The property set method restricts the assignment of the Material 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 = 'aluminum';
        SampleNumber = 0;
        Stress
        Strain
        Modulus = 0;
    end% properties

    methods
        function obj = set.Material(obj,material)
            if ~(strcmpi(material,'aluminum') ||...
                strcmpi(material,'stainless steel') ||...
                strcmpi(material,'carbon steel'))
                error('Material must be aluminum, stainless steel, or carbon steel')
```

```
            end
        obj.Material = material;
        end % set.Material
    end% methods
end% classdef
```

When an attempt is made to set the `Material` property, the MATLAB runtime passes the object and the specified value to the property's `set.Material` function (the `obj` and the `material` input arguments). In this case, if the value does not match the acceptable values, the function returns an error. Otherwise, the specified value is used to set the property. Only the set method can directly access the property in the object (without calling the property set method).

For example:

```
>>td = TensileData;
>>td.Material = 'composite';
??? Error using ==> TensileData.TensileData>Material_set__
Material must be aluminum, stainless steel, or carbon steel
```

Simplifying the Interface with a Constructor

You can simplify the interface to the `TensileData` class by adding a constructor function 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.

```
function td = TensileData(material,samplenum,stress,strain)
    if nargin > 0 % Support calling with 0 arguments
        td.Material = material;
        td.SampleNumber = samplenum;
        td.Stress = stress;
        td.Strain = strain;
    end
end % TensileData
```

Using the constructor, you can create a `TensileData` object fully populated with data using the following statement:

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

Calculating Modulus

Note that the constructor function does not have an input argument for the value of the `Modulus` property. This is because the value of the `Modulus`:

- Is easy to calculate from the `Stress` and `Strain` property values
- Needs to 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. You can do this with a property `get` access method, which is described in the next section.

Dependent Properties

`TensileData` objects do not store the value of the `Modulus` property; instead this value is calculated whenever it is requested. This approach enables you to update the `Stress` and `Strain` property data at any time without having to recalculate the value of the `Modulus` property.

Defining the Modulus Property Get Function

The `Modulus` property depends on `Stress` and `Strain`, so its `Dependent` attribute is set to logical `true`. To do this, add another `properties` keyword before the `Modulus` property specification to specify the `Dependent` attribute. Define the property's get method in a methods block.

```
properties (Dependent = true)
    Modulus
end % Modulus get method

methods
    function modulus = get.Modulus(obj)
        ind = find(obj.Strain > 0); % Find nonzero strain
        modulus = mean(obj.Stress(ind)./obj.Strain(ind));
```

```
    end % Modulus get method
  end % methods
```

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

The MATLAB runtime 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
```

Displaying TensileData Objects

The `TensileData` class can implement a `disp` method that controls what is displayed when an object of this class is shown on the command line (for example, by an assignment statement not terminated by a semicolon).

The `TensileData` `disp` method displays the value of the `Material`, `SampleNumber`, and `Modulus` properties. It does not display the `Stress` and `Strain` property data since these properties contain raw data that is not easily viewed in the command window. The `plot` method (described in the next section) provides a better way to display stress and strain data.

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

```
methods
   function disp(td)
      fprintf(1,'Material: %s\nSample Number: %g\nModulus: %1.5g\n',...
      td.Material,td.SampleNumber,td.Modulus);
   end % disp
end % methods
```

A 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. A `TensileData` object contains the stress and strain data so it is useful to define a class method that is designed to plot this data.

The `TensileData` 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
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 % plot
```

The first argument to this method is a `TensileData` object, which contains the data and is used by the MATLAB runtime to dispatch to the `TensileData` class `plot` method and not the built-in `plot` function.

The variable list of arguments that follow are passed directly to the built-in `plot` function from within the method. This enables the `TensileData` plot method to behave like the built-in `plot` function, which allows you to pass line specifier arguments or property name/value pairs along with the data.

For example, plotting the following object:

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

produces this graph.
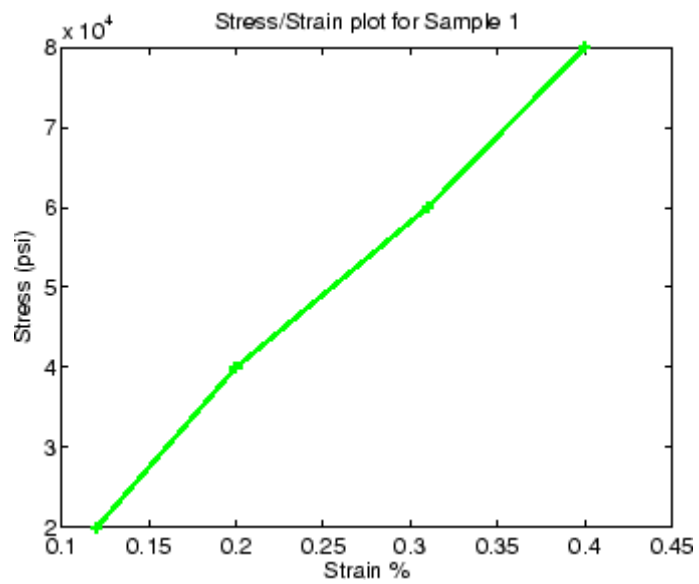
Example — Implementing Linked Lists

In this section...
"Displaying Fully Commented Example Code" on page 2-52
"Important Concepts Demonstrated" on page 2-52
"dlnode Class Design" on page 2-53
"Creating Doubly Linked Lists" on page 2-54
"Why a Handle Class for Doubly Linked Lists?" on page 2-55
"Defining the dlnode Class" on page 2-56
"Specializing the dlnode Class" on page 2-59

Displaying Fully Commented Example Code

Open class code in a popup window — Use this link if you want to see the code for this class annotated with links to descriptive sections.

Open class definition file in the MATLAB® editor. — Use this link if you want to save and modify your version of the class.

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

Important Concepts Demonstrated

Encapsulation

This example shows how classes can encapsulate the internal structure used to implement the design goal—a mechanism to implement doubly linked lists. Encapsulation conceals the internal workings of the class from other code and provides a stable interface to programs that use this class. It also prevents client code from misusing the class because only class methods can access certain class data.

Class methods define the operations that can be performed on nodes of this class. These methods hide the potentially confusing process of inserting and removing nodes, while at the same time providing and interface that simply performs the operations you need to perform:

- Creating a node by passing the constructor a data value
- Inserting nodes with respect to other nodes in the list (before or after)
- Removing nodes from the list

See "Defining the dlnode Class" on page 2-56 for the implementation details.

Handle Class Behavior

This example shows an application of a handle class and explains why this is the best choice for the class. See "Why a Handle Class for Doubly Linked Lists?" on page 2-55

dlnode Class Design

This example defines a class for creating the nodes of doubly linked lists in which each node contains:

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

Each node has methods that enables the node to be:

- Disconnected from a linked list
- Connected before a specified node in a linked list
- Connected after a specific node in a linked list

Class Properties

Each node is implemented 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 three-node object n1, n2, and n3. It also shows how the nodes reference the next and previous node.
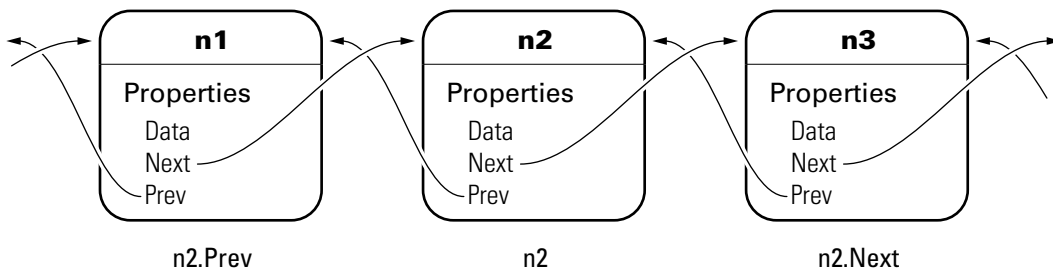
Class Methods

The dlnode class implements the following methods:

- dlnode — Constructs a node and assigns the value passed as input to the Data property

- insertAfter — Inserts this node after the specified node

- insertBefore — Inserts this node before the specified node

- disconnect — Removes this node from the list

- disp — Overloads default disp function so that only the Data property is displayed on the command line

- delete — Removed this node from the list before it is destroyed

Creating Doubly Linked Lists

You create a node by passing the node's data to the dlnode class constructor. For example, these statements create three nodes with sequential integer data just for simplicity:

```
n1=dlnode(1);
```

```
n2=dlnode(2);
n3=dlnode(3);
```

You build these nodes into a doubly linked list using the class methods:

```
n2.insertAfter(n1)
n3.insertAfter(n2)
```

Now the three nodes are linked. The dlnode disp method returns the data for the node referred to:

```
n1.Next % Points to n2
ans =
Doubly-linked list node with data:
     2
n2.Next.Prev % Points back to n2
ans =
Doubly-linked list node with data:
     2
n1.Next.Next % Points to n3
ans =
Doubly-linked list node with data:
     3
n3.Prev.Prev % Points to n1
ans =
Doubly-linked list node with data:
     1
```

Why a Handle Class for Doubly Linked Lists?

Each node is unique in that no two nodes can be previous to or next to the same node. Suppose 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 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. Then x must point to the same node as n2. This means there has to be a way for multiple variables to refer to the same instance. The MATLAB handle class provides a means for both x and n2 to refer to the same node. All instances of the handle class are handles that exhibit the copy behavior described above.

Note that the handle class defines the eq method, which enables the use of the == operator with all handle objects.

See "Comparing Handle and Value Classes" on page 4-2 for more information on kinds of MATLAB classes.

See "The Handle Base Class" on page 4-10 for more information about the handle class.

Defining the dlnode Class

Assume the following doubly linked list is used in the following examples:

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

Class Properties

The dlnode class is itself a handle class because it is derived from the handle class. Note that the Next and Prev properties can be set only by class methods (SetAccess = private) to prevent client code from performing any incorrect

operation with these properties. The dlnode class defines methods that perform all the operations that are allowed on these nodes.

```
classdef dlnode < handle
   properties
      Data
   end
   properties (SetAccess = private)
      Next
      Prev
   end
```

Creating a Node Object

To create a node object, you need to specify only the node's data. When the node is added to a list, the Next and Prev properties are set by the class methods that perform the insertion.

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

Disconnecting Nodes

The disconnect method removes a node from a list and heals the list by connecting the appropriate nodes. Note that disconnect is always called on a node before attempting to connect it to a linked list to ensure a known state for the Next and Prev properties.

```
function disconnect(node)
   Prev = node.Prev;
   Next = node.Next;
   if ~isempty(Prev)
      Prev.Next = Next;
   end
   if ~isempty(Next)
      Next.Prev = Prev;
   end
```

```
        node.Next = [];
        node.Prev = [];
    end
```

First the list is repaired. Assume this example is using n2 from the three-node list discussed above:

```
n1 = n2.Prev;
n3 = n2.Next;
if n1 exists, then
    n1.Next = n3;
if n3 exists, then
    n3.Prev = n1
```

Now the list is rejoined because n1 connects to n3 and n3 connects to n1. The final step is to ensure that n2.Next and n2.Prev are both empty (i.e., n2 is not connected).

Inserting Nodes

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

```
methods
    function insertAfter(newNode, nodeBefore)
        disconnect(newNode);
        newNode.Next = nodeBefore.Next;
        newNode.Prev = nodeBefore;
        if ~isempty(nodeBefore.Next)
            nodeBefore.Next.Prev = newNode;
        end
        nodeBefore.Next = newNode;
    end
```

The first step is always to ensure the new node is not connected to any other nodes; hence, disconnect is called first. Assuming you are calling insertAfter to insert n2 back into the list after n1:

```
n2.insertAfter(n1)
```

```
n2.Next = n1.Next; % n1.Next is still n3, set n2.Next to n3
n2.Prev = n1;
if n1.Next is not empty, then
    n1.Next.Prev = n2; % n1.Next is n3, so n3.Prev is set to n2
n1.Next = n2; % n1.Next is now set to n2
```

Displaying a Node on the Command Line

All objects call a default disp function, which displays all object properties and their values on the command line. That is not a useful approach in this case because the Next and Prev properties contain other node objects. Therefore, the dlnode class overloads disp to display only a text message and the value of the Data property.

```
function disp(node)
    disp('Doubly-linked list node with data:')
    disp(node.Data);
end
```

Deleting a Node Object

When you define a delete method for a handle class, it is called before destroying the object. dlnode objects are nodes in a doubly linked list, so it is important to disconnect the node before destroying it. Otherwise the links to that node are broken. The disconnect method performs the necessary steps, so the delete method can simply call disconnect:

```
function delete(node)
    disconnect(node);
end
```

Specializing 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 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 (i.e., subclass dlnode) to create a new class that has all the features of dlnode and more. And because dlnode is a handle class, this new class is a handle class too.

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

```
classdef NamedNode < dlnode
   properties
      Name = '';
   end
   methods
      function n = NamedNode (name,data)
         if nargin == 0   % to handle the no argument case
            name = '';
            data = [];
         end
         n = n@dlnode(data); % Initialize a dlnode object
         n.Name = name;
      end
      function disp(n) % Override the dlnode disp method
         disp(['Node Name: ' n.Name])
         disp(['Node Data: ' num2str(n.Data)])
      end
   end % methods
end % classdef
```

The `NamedNode` class adds a `Name` property to store the node name and overrides the `disp` method defined in the `dlnode` class. The constructor calls the class constructor for the `SimpleNode` class, and then assigns a value to the `Name` property.

Note that the constructor method is designed to handle the zero input argument case by creating an empty cell array, which contains the arguments if there are any and remains empty if there are not. See "Basic Structure of Constructor Methods" on page 7-18 for more information on defining class constructor methods and why a constructor method should be able to handle the case where it is called with no input arguments.

Using the NamedNode Class to Create a Doubly Linked List

Use the `NamedNode` class in a way very similar to the `SimpleNode` class. For example:

```
n1=NamedNode('First Node',100);
```

```
n2=NamedNode('Second Node',200);
n3=NamedNode('Third Node',300);
```

Now use the insert methods inherited from dlnode to build the list:

```
n2.insertAfter(n1)
n3.insertAfter(n2)
```

The nodes display their name and data when querying the properties:

```
>> n1.Next
ans =
Node Name: Second Node
Node Data: 200
>> n1.Next.Next
ans =
Node Name: Third Node
Node Data: 300
>> n3.Prev.Prev
ans =
Node Name: First Node
Node Data: 100
```

Example — Class for Graphing Functions

The *class block* is the code that starts with the classdef key word and terminates with the end key word. The following example illustrated a simple class definition that uses:

- Handle class

- Property set and get functions

- Use of a delete method for the handle object

- Static method syntax

Display Fully Commented Example Code

You can display this class definition in a separate window that contains links to related sections in the documentations by clicking this link:

Example with links

Class Definition Block

The following code defines a class called topo. It is derived from handle so it is a handle class, which means it references the data it contains. See "Using the topo Class" on page 2-64 for information on how this class behaves.

```
classdef topo < handle                 % topo is a subclass of handle
   properties
      FigHandle = [];
      FofXY
      Lm = [-2*pi 2*pi];                % Initial property value
   end % properties
```

```matlab
    properties (Dependent = true)        % Do not store the Data property
        Data
    end % properties Dependent = true

    methods
        function obj = topo(fnc,limits)    % Constructor returns object
            obj.FofXY = fnc;                   % Assign property values
            obj.Lm = limits;
        end % topo

        function lm = set.Lm(obj,lim)      % Lm property set function
            if  ~(lim(1) < lim(2))
                error('Limits must be monotonically increasing')
            else
                    obj.Lm = lim;
            end
        end % set.Lm

        function data = get.Data(obj)       % get function calculates Data
            [x,y] = topo.grid(obj.Lm);          % Use class name to call static method
            matrix = obj.FofXY(x,y);
            data.X = x;
            data.Y = y;
            data.Matrix = matrix;               % Return value of property
        end % get.Data

        function surflight(obj)                % Graph function as surface
            surf(obj.Data.X,obj.Data.Y,obj.Data.Matrix,...
                'FaceColor',[.8 .8 0],'EdgeColor',[0 .2 0],...
                'FaceLighting','phong');
            camlight left; material shiny; grid off
            colormap copper
            obj.FigHandle = gcf;
        end % surflight method

        function delete(obj)
            h = obj.FigHandle;
            if ishandle(h)
                delete(h); % Delete the figure
            else
```

2-63

```
                return
          end
      end % delete
   end % methods

   methods (Static = true)            % Define static method
     function [x,y] = grid(lim)
        inc = (lim(2)-lim(1))/35;
        [x,y] = meshgrid(lim(1):inc:lim(2));
     end % grid
   end % methods Static = true
end % topo class
```

Using the topo Class

This class is designed to display a combination surface/contour graph of mathematical functions of two variables evaluated on a rectangular domain of x and y. For example, any of the following functions can be evaluated over the specified domain (note that x and y have the same range of values in this example just for simplicity).

```
x.*exp(-x.^2 - y.^2); [-2 2]
sin(x).*sin(y); [-2*pi 2*pi]
sqrt(x.^2 + y.^2); [-2*pi 2*pi]
```

To create an instance of the class, passing a function handle and a vector of limits to the constructor. The easiest way to create a function handle for these functions is to use an anonymous function:

```
tobj = topo(@(x,y) x.*exp(-x.^2-y.^2),[-2 2]);
```

The class surflight method uses the object to create a graph of the function. The actual data required to create the graph is not stored. When the surflight method accesses the Data property, the property's get function performs the evaluation and returns the data in the Data property structure fields. This data is then plotted. The advantage of not storing the data is the reduced size of the object.

Behavior of the Handle Class

The topo class is defined as a handle class. This means that instances of this class are handle objects that reference the underlying data store created by constructing the object. For example, suppose you create an instance of the class and create a copy of the object:

```
tobj = topo(@(x,y) x.*exp(-x.^2-y.^2),[-2 2]);
a = tobj;
surflight(a) % Call class method to create a graph
```
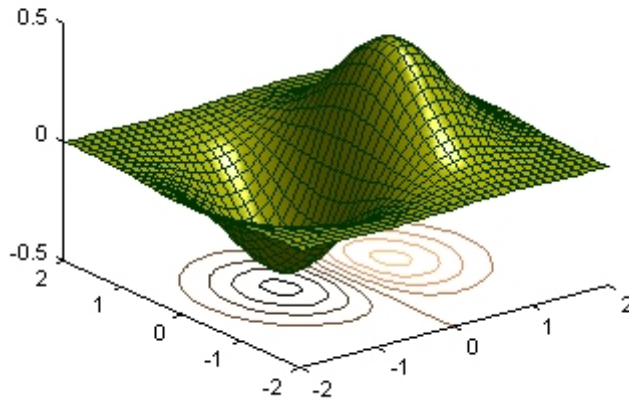

Now suppose you change the FofXY property so that it contains a function handle that points to another function:

```
tobj.FofXY = @(x,y) y.*exp(-x.^2-y.^2); % now multiply
exp by y instead of x
surflight(a)
```
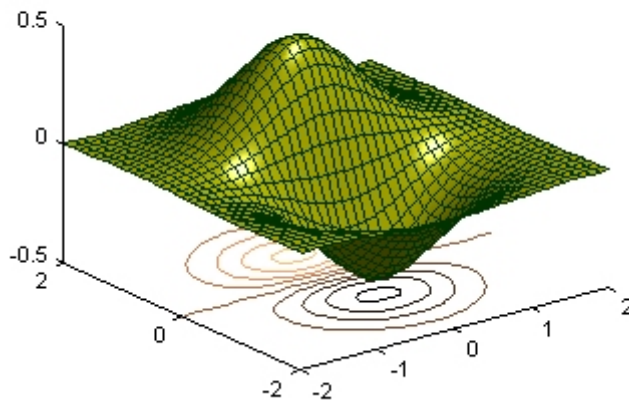
Because a is a copy of the handle object tobj, changes to the data referenced by tobj also change the data referenced by a.

How a Value Class Differs

If topo were a value class, the objects tobj and a would not share data; each would have its own copy of the property values.

3

Working with Classes

Class Overview

MATLAB® User-Defined Classes

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.

MATLAB classes are defined in code blocks, with sub-blocks delineating the definitions of various class members. See "classdef Syntax" on page 3-4 for details on the `classdef` block.

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 abstract, and so on. The following sections describe these attributes:

- "Class Attributes" on page 3-5

- "Method Attributes" on page 7-4

- "Specifying Property Attributes" on page 6-6

- "Event Attributes" on page 8-7

Class definitions can provide information, such as inheritance relationships or the names of class members without actually constructing the class. See "Obtaining Information About Classes with Meta-Classes" on page 3-26.

See "Specifying Attributes" on page 3-6 for more on attribute syntax.

Kinds of Classes

There are two kinds of MATLAB classes—handle and value classes.

- Handle classes create objects that reference the data contained. Copies refer to the same data.

- Value classes make copies of the data whenever the object is copied or passed to a function. MATLAB numeric types are value classes.

See "Comparing Handle and Value Classes" on page 4-2 for a more complete discussion.

Creating Class Hierarchies

For more information on how to define class hierarchies, see Chapter 5, "Building on Other Classes ".

Defining Classes — Syntax

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 classes definition.

The following diagram shows the syntax of a classdef block. Only comments and blank lines can precede the classdef key word.

classdef keyword begins definition block.

```
classdef (ConstructOnLoad = true) PositiveIntegers < Integers & Positives

...

end
```

classdef block

Class attribute   Attribute value   Class name   Super classes
(logical true)

end keyword terminates definition block.
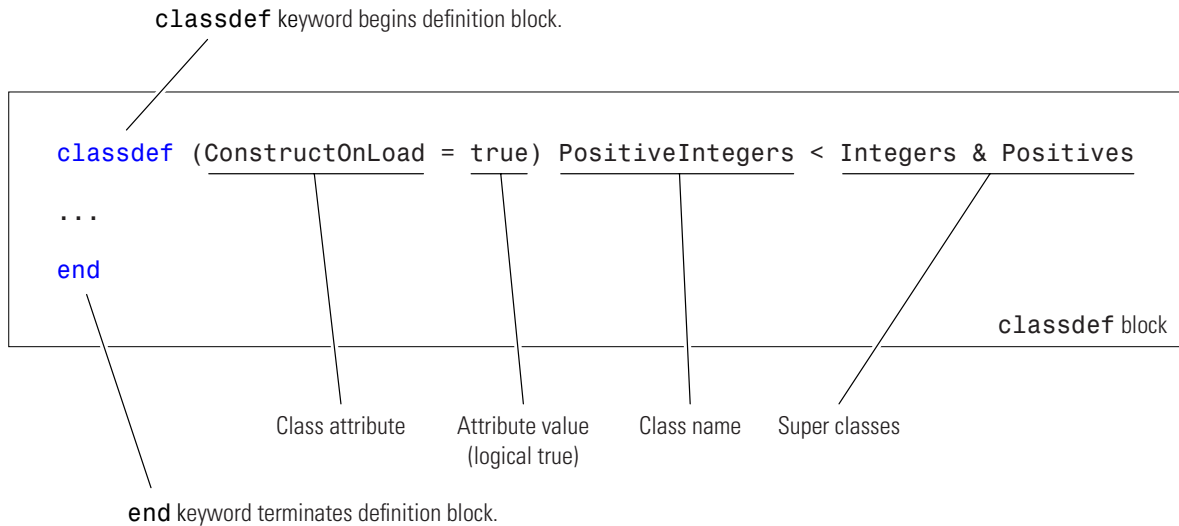
Examples of Class Definitions

See the following links for examples of class definitions:

- "Example — Representing Structured Data" on page 2-43

- "Example — Implementing Linked Lists" on page 2-52

- "Developing Classes — Typical Workflow" on page 2-23

- "Example — A Polynomial Class" on page 9-2

Class Attributes

In this section...
"Table of Class Attributes" on page 3-5
"Specifying Attributes" on page 3-6

Table of 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.

Attribute Name	Class	Description
Hidden	logical (default = false)	If set to true, the class does not appear in the output of MATLAB® commands or tools that display class names.
InferiorClasses	cell (default = {})	Use this attribute to establish a precedence relationship among classes. Specify a cell array of meta.class objects using the ? operator. The built-in classes double, single, char, logical, int64, uint64, int32, uint32, int16, uint16, int8, uint8, cell, struct, and function_handle are always inferior to user-defined classes and do not show up in this list. See "Specifying Class Precedence" on page 3-11
ConstructOnLoad	logical (default = false)	If true, the class constructor is called automatically when loading an object from a MAT-file. Therefore, the construction must be implemented so that calling it with no arguments does not produce an error. See "Saving and Loading Objects" on page 3-17
Sealed	logical (default = false)	If true, the class can be not be subclassed.

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. This means that, for example, you might use multiple `properties` definition blocks so you can apply different attribute setting to different properties.

Inheritance of Superclass Attributes

Class attributes are not inherited. Therefore, superclass attributes do not affect subclasses, with the exception of `InferiorClasses` attribute. See "Specifying Class Precedence" on page 3-11 for more information on this attribute.

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 key word, as shown below:

```
classdef (attribute-name = expression, ...) ClassName

    properties (attribute-name = expression, ...)
      ...
    end
    methods (attribute-name = expression, ...)
      ...
    end
    events (attribute-name = expression, ...)
      ...
    end
end
```

Evaluating Expressions in Attribute Assignments

The term *expression* in the statements above is used to mean any legal MATLAB expression that evaluates to a single array. The expression is evaluated when the class definition is first needed and the resulting value is treated as a constant.

The MATLAB language evaluates the expression in its own context and cannot reference variables or the class being defined. Attribute expressions can reference static methods and static properties of other classes and use the MATLAB path context of the class at the time of definition. This means the expression cannot access variables in the MATLAB workspace, but can access any function and static method or property that is on your path at the time the class definition is evaluated.

For example, in the following `classdef` line, the `Sealed` attribute is set to logical `true` only if a certain directory is on the MATLAB path.

```
classdef (Sealed = isdir('myDir'))
```

Organizing Classes in Directories

In this section...
"Options for Class Directory" on page 3-8
"@-Directories" on page 3-8
"Path Directories" on page 3-9
"Class Precedence and MATLAB® Path" on page 3-9

Options for Class Directory

There are two types of directories that can contain class definitions. Each behave differently in a number of respects.

- @-directories — Directory name begins with "@" and is not on the MATLAB® path, but its parent directory is on the path. Use this type of directory when you want to use multiple files for one class definition. There can be only one class per directory and the name of the class must match the name of the directory, without the "@" symbol.

- path directories — Directory name does not use @ character and is itself on the MATLAB path. Use this type of directory when you want multiple classes in one directory.

See the `path` function for information about the MATLAB path.

@-Directories

An @-directory is contained by a path directory, but is not itself on the MATLAB path. You place the class definition file inside the @-directory, which can also contain method files. The class definition file must have the same name as the @-directory (without the @-sign) and the class definition (beginning with the `classdef` key word) must appear in the file before any other code (white space and comments do not constitute code). The name of the class must match the name of the file that contains the class definition.

You must use an @-directory if you want to use more than one file for your class definition. Methods defined in separate files match the file name to the function name.

Path Directories

You can locate class definition files in directories that are on the MATLAB path. These classes are visible on the path like any ordinary function. Class definitions placed in path directories 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.

The name of the file must match the name of the class, as specified with the `classdef` key word. Using a path directory eliminates the need to create a separate @-directory for each class. However, the entire class definition must be contained within a single file.

Class Precedence and MATLAB® Path

When multiple class definition files with the same name exist, the precedence of a given file is determined by its location on the MATLAB path. All class definition files before it on the path (whether in an @-directory or not) take precedence and it takes precedence over all class definition files occurring later on the path.

For example, consider a path with the following directories, containing the files indicated:

```
dir1/foo.m          % defines class foo
dir2/foo.m          % defines function foo
dir3/@foo/foo.m     % defines class foo
dir4/@foo/bar.m     % defines method bar
dir5/foo.m          % defines class foo
```

The MATLAB language applies the logic in the following list to determine which version of `foo` to call:

- Class `dir1/foo.m` takes precedence over the class `dir3/@foo` because it is before `dir3/@foo` on the path.

- Class `dir3/@foo` takes precedence over function `dir2/foo.m` because it is a class in an @-directory and `dir2/foo.m` is not a class (@-directory classes take precedence over functions).

- Function `dir2/foo.m` takes precedence over class `dir5/foo.m` because it comes before class `dir5/foo.m` on the path and because class `dir5/foo.m`

is not in an @-directory. Classes not defined in @-directories abide by path order with respect to functions.

- Class `dir3/@foo` takes precedence over class `dir4/@foo`; therefore, the method `bar` is not recognized as part of the `foo` class (which is defined only by `dir3/@foo`).

- If `dir3/@foo/foo.m` does not contain a `classdef` keyword (i.e., it is a MATLAB class prior to Version 7.6), then `dir4/@foo/bar.m` becomes a method of the `foo` class defined in `dir3/@foo`.

Previous Behavior of Classes Defined in @-Directories

In MATLAB Versions 5 through 7, @-directories do not shadow other @-directories having the same name, but residing in later path directories. Instead, the class is defined by the combination of methods from all @-directories having the same name. This is no longer true.

Note that for backward compatibility, classes defined in @-directories always take precedence over functions and scripts having the same name, even those that come before them on the path.

Specifying Class Precedence

InferiorClasses Attribute

You can specify the relative precedence of user-defined classes using the class
InferiorClasses attribute. Assign a cell array of class names (represented
as metaclass objects) to this attribute to specify classes that are inferior to
the class you are defining. For example, the following classdef declares that
myClass is superior to class1 and class2.

```
classdef (InferiorClasses = {?class1,?class2}) myClass
    ...
end
```

The ? operator combined with a class name creates a metaclass object. This
syntax enables you to create a meta-class object without requiring you to
construct an actual instance of the class.

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.

More Information

See "Determining Which Method Is Invoked" on page 7-8 for more on how the
MATLAB classes dispatch when evaluating expressions containing objects.

See "Class Precedence and MATLAB® Path" on page 3-9 for information on
how the location of a class definition on the MATLAB path determines its
precedence.

See "Obtaining Information About Classes with Meta-Classes" on page 3-26
for information on meta-class objects.

Inheriting Inferior Classes

The inferior classes specified by the InferiorClasses attribute is inherited
by subclasses. If the subclass also defines the InferiorClasses attribute,

the true list of inferior classes is the union of the inferior classes defined by the subclass and all superclasses.

Scoping Classes with Packages

Package Directory

Packages are special directories that can contain class directories, functions, and other packages. Packages define a scope (sometimes called a namespace) for the contents of the package directory. This means function and class names need to be unique only within the package. Using a package provides a means to organize classes and functions and to select names for these components that can be reused in other packages.

Note Packages are not supported for classes created prior to MATLAB® Version 7.6 (i.e., classes that do not use `classdef`).

Package directories always begin with the + character. For example,

```
+mypack
+mypack/pkfcn.m % a package function
+mypack/@myClass % class in a package
```

The package directory's parent directory must be on the MATLAB path.

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 "Importing Classes" on page 3-23.) For example, call a package function with this syntax:

```
z = mypack.pkfcn(x,y);
```

Note that 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)
```

Similarly, a package class would be defined with only the class name:

```
classdef myClass
```

but would be called with the package prefix:

```
obj = mypack.myClass(arg1,arg2,...);
```

Calling class methods does not require the package name because you have an instance of the class:

```
obj.myMethod(arg) or
myMethod(obj,arg)
```

A static method requires the full class name:

```
mpack.myClass.stMethod(arg)
```

Referencing Package Members from Outside the Package

Because functions, classes, and other packages contained in a package are scoped to that package, to reference any of the package members, you must 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 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 in `mysubpack`:

```
mypack.mysubpack.myfcn(arg1,arg2);
```

If `mypack.myfirstclass` has a method called `myfcn`, it is called as any method call on an object:

```
obj = mypack.myfirstclass;
myfcn(obj,arg);
```

If `mypack.myfirstclass` has a property called `MyProp`, it can be assigned using dot notation and the object:

```
obj = mypack.myfirstclass;
obj.MyProp = some_value;
```

Packages and MATLAB® Path

You cannot add package directories to the MATLAB path, but you must add the package's parent directory to the path. Even if a package directory is the current directory, its parent directory must still be on the MATLAB path or the package members are not accessible.

Package members remain scoped to the package even if the package directory is the current directory. You must, therefore, always refer to the package members using the package name.

Package directories do not shadow other package directories that are positioned later on the path, unlike classes, which do shadow other classes.

Resolving Redundant Names

Suppose a package and a class have the same name. For example:

```
dir1/+foo
dir2/@foo/foo.m
```

A call to `which foo` returns the path to the executable class constructor:

```
>> which foo
dir2/@foo/foo.m
```

A function and a package can have the same name. However, a package name by itself is not an identifier so 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:

```
dir1/+foo/bar.m % bar is a function in package foo
dir2/@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
dir2/@foo/bar.m
```

In cases where a path directory contains both package and class directories with the same name, the class static method takes precedence over the package method:

```
dir1/@foo/bar.m % bar is a static method of class foo
dir1/+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
dir1/@foo/bar.m
```

Saving and Loading Objects

In this section...

Modifying the Save and Load Process

You can save objects to a MAT-file and then reload them into the workspace. In most cases, saving and loading objects from MAT-files is no different from working with other variables. However, you can modify the save and load process to better support class backward and forward compatibility.

Class saveobj and loadobj Methods

You can define methods for your class that are executed when you call `save` or `load` on an object:

- The `saveobj` method is called by `save` before performing the save operation. The `save` function then saves the value returned by the object's `saveobj` method. You can use the `saveobj` method to return a modified object or any other type of variable, such as a `struct` array.

- The `loadobj` method is called by `load` before performing the load operation. The `load` function loads into the workspace the value returned by the object's `loadobj` method. Defining a `loadobj` method gives you the opportunity to modify the object being returned.

If you implement a `saveobj` method that modifies the object being saved, then you probably need to implement a `loadobj` method to return the object to its proper state when reloading it. For example, you might want to store an object's data in a `struct` array and reconstruct the object when reloaded to guard against the possibility that the class definition might change.

What Information Is Saved

Saving objects in MAT-files saves the following data:

- The names and values of all properties not marked as `Transient` or `Constant` (See "Specifying Property Attributes" on page 6-6 for a description of these attributes.) However, to decrease file size, property values are saved only if they differ from the default values specified in the class definition.

- All property initial values

- The full name of the object's class, including any package qualifiers.

Even if your `saveobj` method saves only the object's data in an array and not the object itself, the object's class name is saved so that `load` can determine which `loadobj` method to call.

How Property Data is Loaded

If the class's `ConstructOnLoad` attribute is set to `true`, then the load process calls the class constructor with no arguments and creates a new object of the class using the data stored in the saved object. When loading objects from MAT-files the `load` function does the following:

- First reassigns the initial values that were stored in the MAT-file

- Then assigns saved values to those properties whose values were not the same as their initial values at the time the objects was saved.

- Each of these assignments (initial value and saved value) calls the property set method, if one is defined for that property.

It is possible for an initial value to cause an error in a set property method (for example, the class definition might have changed). When an error occurs while an object is being loaded from a file, the MATLAB® runtime returns the saved values in a MATLAB `struct`.

When an error occurs, your `loadobj` method is passed the `struct` instead of an object. The `loadobj` method must use the values in the `struct` to call the class constructor and create a new object.

You Must Implement loadobj as a Static Method

You must implement the loadobj method as a static method because loadobj can actually be called with a struct or other data instead of an object of the class. The saveobj method is implemented as an ordinary method (i.e., calling it requires an instance of the class).

Code for These Examples

The following discussion on saving and loading objects refers to a BankAccount class. Use the following link to see the full code for this class.

Open in editor

Processing Objects During Load Operations

When you load an object that has been saved to a MAT-file using the load function, the object's loadobj method is called, if it exists. Implementing a loadobj method enables you to apply some processing to the object before it is loaded into the workspace. This might be necessary in cases such as the following:

• The class definition has changed since the object was saved and you need to modify the object before reloading.

• A saveobj method modified the object during the save operation, perhaps saving data in an array to be more efficient, and the loadobj method must reconstruct the object based on what saveobj did.

Updating an Object Property When Loading with loadboj

In the following example, the loadobj method checks to see if the object to be loaded has an old, shorter account number and calls a function to return an updated account number if necessary. After updating the object's AccountNumber property, loadobj returns the object to be loaded into the workspace.

```
methods (Static = true)
    function obj = loadobj(a)
        accnb = a.AccountNumber;
        if length(num2str(accnb)) < 12
            a.AccountNumber = updateAccountNumber(accnb); % update object
```

```
            end
            obj = a;  % return the updated object
        end
    end
```

In this case, you do not need to implement a saveobj method and are using loadobj only to ensure older saved objects are brought up to date before loading.

The next section provides an example in which loadobj performs specific operations to recreate an object based on the data returned by saveobj during the save operation.

Processing Objects During Save and Load Operations

Suppose that you know class definitions are likely to change over time, but you want to be able to reload older data and create objects according to the most current class definition. One approach is to save the object's data in a struct, but not the object itself.

To use this approach, create a saveobj method to extract the data you want to save from the object and write this data into a struct, which saveobj returns to the save function. You must also implement a loadobj method to extract the data from the reloaded variable and use this data to create a new object.

Saving Object Data Only with saveobj

The following example saves the data from a BankAccount object having two properties. saveobj saves the values of the AccountNumber and the AccountBalance properties in the struct variable A, which has field names that match the property names. saveobj then returns the variable A to be saved in the MAT-file by the save function.

```
methods
    function A = saveobj(obj)
        A.AccountNumber = obj.AccountNumber;
        A.AccountBalance = obj.AccountBalance;
    end
end
```

Reconstructing Objects with loadobj

Suppose that a new property is added to the `BankAccount` class and all saved `BankAccount` objects must be recreated using the new class constructor. To create a valid object, call the constructor using the data saved in the variable `A` and pass any other arguments required. Because you are calling the latest version of the constructor in the `loadobj` method, the very latest version of the objects is created.

In this example, a new property called `AccountStatus` was added to the `BankAccount` class. If the account balance is greater than zero, `AccountStatus` is set to `open`. If the account balance is zero or less, `AccountStatus` is set to `overdrawnor` to `frozen`.

This `loadobj` method calls the class constructor with the appropriate value for the new property after testing the data saved from the `AccountBalance` property.

```
methods (Static = true)
    function obj = loadobj(a)
        if A.AccountBalance > O
            obj = BankAccount(a.AccountNumber,a.AccountBalance,'open');
        else
            obj = BankAccount(a.AccountNumber,a.AccountBalance,'overdrawn');
        end
    end
end
```

saveobj Method Expanded

Using `saveobj` and `loadobj` to manage changes to class definitions probably requires you to update these methods with each modification to the class. For example, you might need to accommodate a mix of old and new `BankAccount` objects with the `saveobj` method.

```
methods
    function A = saveobj(obj)
        A.AccountNumber = obj.AccountNumber;
        A.AccountBalance = obj.AccountBalance
        if BankAccount.isPropertyOf(obj,'AccountStatus')
            A.AccountStatus = obj.AccountStatus;
```

```
            end
        end
    end
```

Does an Object Have a Particular Property

This class defines a static method that determines if a property exists for an object. isPropertyOf is used to determine if the object being loaded has the new AccountStatus property.

```
methods (Static = true)
    function tf = isaPropertyOf(anyobj,PropName)
        if ~isa(anyobj,'meta.class')
            mobj = metaclass(anyobj);
        else
            mobj = anyobj;
        end
        tf = false;
        for k=1:length(mobj.Properties)
            if strcmp(mobj.Properties{k}.Name,PropName)
                tf = true;
            end
        end
    end
end
```

Note that you can also use this method to determine if a class defines a particular property without creating an instance. For example, this statement determines if the BankAccount class defines a property called AccountType:

```
BankAccount.isaPropertyOf(?BankAccount,'AccountType')
```

The ?BankAccount argument creates a meta.class object from the class name, and then checks the class definition to determine if the class currently defines the named property.

See "Obtaining Information About Classes with Meta-Classes" on page 3-26 for more information on meta-class objects.

Importing Classes

Syntax for Importing Classes

You can import classes into a function to simplify access to class members. For example, suppose there is a package that contains a number of classes, but you need to use only one of these classes in your function, or perhaps even just a static method from that class. You can use the import command as follows:

```
function myFunc
    import pkg.cls1
    obj = cls1(arg,...); % call cls1 constructor
end
```

Note that you do not need to reference the package name (pkg) once you have imported the class (cls1). You can also import all classes in a package using the syntax pkg.*, where * indicates all classes in the package. For example,

```
function myFunc
    import pkg.*
    obj1 = cls1(arg,...); % call pkg.cls1 constructor
    obj2 = cls2(arg,...); % call pkg.cls2 constructor
end
```

Importing Class Members

You can import the static methods, constant properties, and enumeration names defined in a class so that you do not need to use the package and class name in the importing function. Use the class notation to import class members.

```
import class package.class.*
```

For example, myFunc is importing all the members of class pkg.cls1 and the constant property C from class pkg.consts.

```
function myFunc
   import class pkg.cls1.*
   import class pkg.consts.C
   obj = cls1(arg,...); % call pkg.cls1 constructor
   m = calcMass;        % call pkg.cls1 static method
   e = m*C^2;           % constant property pkg.const.C
end
```

Defining Named Constants

Creating a Class for Named Constants

There are situations where it is useful to define a collection of constants whose values can be accessed by name. To do this, create a class having properties with their Constant attribute set to true. Note that setting the Constant attribute to true effectively sets the SetAccess to private, so that the values of the properties cannot be changed outside of the class. You might define a package of classes defining various sets of constants and import these classes into any function that needs them.

A Package for Constants

To create a library for constant values that you can, first create a package directory, and then define the various classes to organize the constants you want to provide. For example, to implement a set of constants used for making astronomical calculations, you can define a AstroConstants class in the Constants package:

```
+Constants/@AstroConstants/AstroConstants.m
```

The class defines on a set of Constant properties with initial values assigned.

```
classdef AstroConstants
  properties (Constant = true)
  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, you would need to reference them with a fully qualified class name. For example, the following function uses some of the constants defined in AstroContants:

```
function E = energyToOrbit(m,r)
  E = Constants.AstroConstants.G * Constants.AstroConstants.Me * m * ...
    (1/Constants.AstroConstants.Re-0.5*r);
end
```

3-25

Obtaining Information About Classes with Meta-Classes

What Are Meta-Classes

Meta-classes are classes that contain information about class definitions. Each block in a class definition has an associated meta-class that defines the attributes for that block. Each attribute corresponds to a property in the meta-class. An instance of a meta-class has values assigned to each property that correspond to the values of the attributes of the associated class block.

Meta-classes enable introspection of class definitions, which is useful for programmatic inspection of classes and objects. Such techniques are used by tools such as property inspectors, debuggers, and so on.

The Meta Package

The meta package is a package of meta-classes that are involved in the definition of classes and class components. The class name indicates the component described by the meta-class:

```
meta.package
meta.class
meta.property
meta.method
meta.event
```

Each meta-class has properties, methods, and events that contain information about the class or class component. See `meta.package`, `meta.class`, `meta.property`, `meta.method` and `meta.event` for more information on these meta-classes.

Creating Meta-Class Objects

You can create meta-class objects from class instances or from the class definition directly.

- ? operator — Returns a meta-class object for the named class

- `metaclass` — Returns a meta-class object for the class instance

```
mobj = ?classname; % create meta-class object from class name
obj = myClass;
mobj = metaclass(obj); % create meta-class object
from class instance
```

Note that the `metaclass` function returns the `meta.class` object (i.e., an object of the `meta.class` class). You can obtain other meta-class objects (`meta.property`, `meta.method`, etc.) from the `meta.class` object.

Note Meta-class is a term used to describe a kind of class. `meta.class` is a class in the `meta` package whose instances contain information about MATLAB® classes.

Using Meta-Class Objects

Here is how you can use meta-class objects:

- Obtain a `meta.class` object from a class definition (using ?) or from a class instance (using `metaclass`).

- Use the `meta.class` properties, methods, and events to obtain information about the class or class instance from which you obtained the `meta.class` object, including getting other meta-class objects, such as the `meta.properties` objects defined for each of the class's properties.

The following examples show these techniques.

Inspecting a Class

Consider the following class and the information you can obtain from its meta-class instances. The EmployeeData class is a handle class with two properties, one of which defines a set access function. The capability to glean information about the class programmatically without requiring an instance of the class can be useful when implementing tools, such as inspectors, code debuggers, and so on.

```matlab
classdef EmployeeData < handle
   properties
      EmployeeName
   end
   properties (GetAccess = private)
      EmployeeNumber
   end
   methods
      function obj = EmployeeData(name,ss)
         obj.EmployeeName = name;
         obj.EmployeeNumber = ss;
      end
      function obj = set.EmployeeName(obj,name)
         if ischar(name)
            obj.EmployeeName = name;
         else
            error('Employee name must be a text string')
         end
      end
   end
end
```

Inspecting the Class Definition

Using the `EmployeeData` class defined above, you can create a `meta.class` object using the ? operator:

```matlab
mboj = ?EmployeeData;
```

You can determine what classes `EmployeeData` is derived from:

```matlab
a = mobj.SuperClasses; % a is cell array of meta.class objects
a{1}.Name
ans =
   handle
```

Inspecting Properties. Suppose you want to know the names of the properties defined by this class. First obtain a cell array of `meta.properties` objects from the `meta.class` Properties property.

```matlab
mpCell = mobj.Properties;
```

The length of `mpCell` indicates there are two `meta.property` objects, one for each property:

```
length(mpCell)
ans =
     2
```

Now get a `meta.property` object from the cell array:

```
prop1 = mpCell{1}
prop1 =
   meta.property % prop1 is a meta.property object
prop1.Name
ans =
   EmployeeName % first object is EmployName property's meta.property
```

You can now query any of the `meta.property` object's properties for the `EmployName` class. You can determine the setting of all property attributes and even obtain a function handle to the property's set access function:

```
setmeth = prop1.SetMethod
setmeth =
   @D:\MyDir\@EmployeeData\EmployeeData.m>EmployeeData.set.EmployeeName
```

Querying the `meta.property` class `SetMethod` property returns a function handle to the set access method defined in the `EmployeeData` class.

Inspecting an Instance of a Class

Suppose you create a `EmployeeData` object:

```
EdObj = EmployeeData('My Name',1234567);
mEdObj = metaclass(EdObj);
mpCell = mEdobj.Properties;
eval(['EdObj.',mpCell{1}.Name])
ans =
   My Name
eval(['EdObj.',mpCell{2}.Name])
??? Getting the 'EmployeeNumber' property of the 'EmployeeData' class is
not allowed.
mpCell{2}.GetAccess
ans =
```

`private`

4

Value or Handle Class —
Which to Use

Comparing Handle and Value Classes

In this section...
"Why Select Value or Handle" on page 4-2
"Behavior of MATLAB® Built-In Classes" on page 4-2
"Behavior of User-Defined Classes" on page 4-3

Why Select Value or Handle

MATLAB® classes support two kinds of classes — value classes and handle classes. The kind of class you select to use depends on the desired behavior of the class instances and which features you want to use. For example, do you want to use events and listeners, define dynamic properties, and so on.

The section "Which Kind of Class to Use" on page 4-8 describes how to select the kind of class to use for your application.

Behavior of MATLAB® Built-In Classes

If you create an object of the class int32 and make a copy of this object, the result is two independent objects having no data shared between them. The following code example creates an object of class int32 and assigns it to variable a, which is then copied to b. When you raise a to the fourth power and assign the value again to the variable a, the MATLAB language creates a new object with the new data and assigns it to the variable a, overwriting the previous assignment. The value of b does not change.

```
a = int32(7);
b = a;
a = a^4;
b
    7
```

The value of a is copied to b and results in two independent versions of the original object. This is typical of MATLAB numeric classes.

Handle Graphics® classes return a handle to the object created. A *handle* is a variable that references an instance of a class. If you copy the handle, you

have another variable that refers to the same object. There is still only one version of the object's data. For example, if you create a Handle Graphics line object and copy its handle to another variable, you can set the properties of the same line using either copy of the handle.

```
x = 1:10; y = sin(x);
h1 = line(x,y);
h2 = h1;

>>set(h2,'Color','red') % line is red
>>set(h1,'Color','green') % line is green
>>delete(h2)
>>set(h1,'Color','blue')
??? Error using ==> set
Invalid handle object.
```

Note also, if you delete one handle, all copies are now invalid because you have deleted the single object that all copies point to.

Behavior of User-Defined Classes

Value class instances behave like built-in numeric classes and handle class instances behave like Handle Graphics objects, as illustrated in "Behavior of MATLAB® Built-In Classes" on page 4-2.

Value Classes

Objects of value classes are permanently associated with the variables to which they are assigned. When a value object is copied, the object's data is also copied and the new object is independent of changes to the original object. Instances behave like standard MATLAB numeric and `struct` classes.

Value Class Behavior

Value classes are useful in cases when both assigning an object to a variable and passing an object to a function should make a copy of the object. Value objects are always associated with one workspace or temporary variable and go out of scope when that variable goes out of scope or is cleared. There are no references to value objects, only copies which are themselves objects.

For example, suppose you define a `polynomial` class whose `Coefficients` property stores the coefficients of the polynomial. Note how copies of these value-class objects are independent of each other:

```
p = polynomial([1 0 -2 -5]);
p2 = p;
p.Coefficients = [2 3 -1 -2 -3];
p2.Coefficients
ans =
   1 0 -2 -5
```

Creating a Value Class

All classes that are not subclasses of the `handle` class are value classes. Therefore, the following `classdef` creates a value class named `myValueClass`:

```
classdef myValueClass
   ...
end
```

Handle Classes

Objects of handle classes use a handle to reference objects of the class. A handle is a variable that identifies a particular instance of a class. When a handle object is copied, the handle is copied, but not the data stored in the object's properties. The copy refers to the same data as the original—if you change a property value on the original object, the copied object reflects the same change.

All handle classes are derived from the abstract `handle` class. In addition to providing handle copy semantics, deriving from the `handle` class enables your class to:

- Inherit a number of useful methods ("Handle Class Methods" on page 4-11)

- Define events and listeners ("Defining Events and Listeners — Syntax and Techniques" on page 8-8)

- Define dynamic properties ("Dynamic Properties — Adding Properties to an Instance" on page 6-17)

- Implement Handle Graphics type set and get methods ("Dynamic Properties — Adding Properties to an Instance" on page 6-17)

Creating a Handle Class

You have to explicitly subclass the `handle` class to create a handle class:

```
classdef myClass < handle
    ...
end
```

See "The Handle Base Class" on page 4-10 for more information on the handle class and its methods.

Subclasses of Handle Classes

If you subclass a class that is itself a subclass of the `handle` class, your subclass is also a handle class. You do not need to explicitly specify the handle superclass in your class definition. For example, the following `employee` class is defined as a handle class:

```
classdef employee < handle
...
end
```

Suppose you want to create a subclass of the `employee` class for engineer employees, which must also be a handle class. You do not need to specify `handle` as a superclass in the `classdef`:

```
classdef engineer < employee
...
end
```

Handle Class Behavior

A handle is an object that references its data indirectly. When constructing a handle, the MATLAB runtime creates an object with storage for property values and the constructor function returns a handle to this object. When the handle is assigned to a variable or when the handle is passed to a function, the handle is copied, but not the underlying data.

For example, suppose you have defined a `handle` class that stores data about company employees, such as the department in which they work:

```
classdef employee < handle
   properties
      Name = ''
      Department = '';
   end
   methods
      function e = employee(name,dept)
         e.Name = name;
         e.Department = dept;
      end % employee
      function transfer(obj,newDepartment)
         obj.Department = newDepartment;
      end % transfer
   end
end
```

The `transfer` method in the above code changes the employee's department (the `Department` property of an `employee` object). In the following statements, `e2` is a copy of the handle object `e`. Notice that when you change the `Department` property of object `e`, the property value also changes in object `e2`.

```
e = employee('Fred Smith','QE');
e2 = e; % Copy handle object
transfer(e,'Engineering')
e2.Department
ans =
Engineering
```

The variable `e2` is an alias for `e` and refers to the same property data storage as `e`.

If the `employee` class was a value class, the `transfer` method would modify only its local copy of the `employee` object and would have to return the modified object to copy over the existing object:

```
e = transfer(e,'Engineering');
```

However, in a value class, the preceding statement would have no effect on the variable e2, which is a different employee object.

Deleting Handles

You can destroy handle objects before they become unreachable by explicitly calling the `delete` function. Deleting the handle of a handle class object makes all handles invalid. For example:

```
delete(e2)
e.Department
??? Invalid or deleted object.
```

Calling the `delete` function on a handle object invokes the destructor function(s) for that object. See "Handle Class Delete Methods" on page 4-13 for more information.

Which Kind of Class to Use

In this section...
"Examples of Value and Handle Classes" on page 4-8
"When to Use Handle Classes" on page 4-8
"When to Use Value Classes" on page 4-9

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 semantics as MATLAB® numeric classes.

"Example — A Polynomial Class" on page 9-2 and "Example — Representing Structured Data" on page 2-43 provides examples of value classes.

Handle classes enable you to create objects that can be shared by more than one function or object. Handle objects allow more complex interactions among objects because they allow objects to reference each other.

"Example — Implementing Linked Lists" on page 2-52 and "Developing Classes — Typical Workflow" on page 2-23 provides examples of a handle class.

When to Use Handle Classes

You should use a handle class when:

- No two instances of a class can have exactly 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, so the two objects are not identical.
  - Nodes in lists or trees having specific connectivity to other nodes—no two nodes can have the same connectivity.

- The class represents physical and unique objects like serial ports or printers where the entity or state cannot exist in a MATLAB variable, but a handle to such entity can be a variable.

- 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 is derived from the `dynamicprops` class (a subclass of `handle`) so that instances can define dynamic properties.

- The class is derived from the `hgsetget` class (a subclass of `handle`) so that it can implement a Handle Graphics™ style set/get interface.

When to Use Value Classes

Value class instances behave like normal MATLAB variables. A typical use of value classes is to define data structures. For example, suppose you want to define a class to represent polynomials. This class can define a property to contain a list of coefficients for the polynomial and implement methods that enable you to perform various common operations on the polynomial object, such as 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. See "Subclassing MATLAB® Built-In Classes" on page 5-13 for more information on value classes.

The Handle Base Class

In this section...
"Building on the Handle Class" on page 4-10
"Handle Class Methods" on page 4-11
"Relational Methods" on page 4-11
"Testing Handle Validity" on page 4-12
"Handle Class Delete Methods" on page 4-13

Building on the Handle Class

The handle class is an abstract class, which means you cannot create an instance of this class directly. Instead, you use this class as a superclass when you implement your own class. The handle class is the foundation of all classes that are themselves handle classes. When you define a class that is a subclass of handle, you have created a handle class. Therefore, all classes that follow handle semantics are subclasses of the handle class.

Handle Subclasses

There are two subclasses of the handle class that provide additional features when you derive your class from these subclasses:

- hgsetget — Provides set and get methods that enable you to implement a Handle Graphics™ style interface. See "Implementing a Set/Get Interface for Properties" on page 4-18 for information on subclassing hgsetget.

- dynamicprops — Provides the ability to define instance properties. See "Dynamic Properties — Adding Properties to an Instance" on page 6-17 for information on subclassing dynamicprops.

  Deriving from subclasses of the handle class means that your class is a handle class and inherits all the handle class methods, plus the special features provided by these subclasses.

Handle Class Methods

While the handle class defines no properties, it does define the methods discussed in this section. Whenever you create a handle class (i.e., subclass the handle class), your subclass inherits these methods.

You can 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
```

"Defining Events and Listeners — Syntax and Techniques" on page 8-8 provides information on how to use the notify and addlistener methods, which are related to the use of events.

"Creating Subclasses — Syntax and Techniques" on page 5-6 provides general information on defining subclasses.

Relational Methods

```
function TF = eq(H1,H2)
function TF = ne(H1,H2)
function TF = lt(H1,H2)
function TF = le(H1,H2)
function TF = gt(H1,H2)
function TF = ge(H1,H2)
```

The handle class overloads these functions with implementations that allow for equality tests and sorting on handles. For each pair of input arrays, a logical array of the same size is returned where 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.

Testing Handle Validity

The `isvalid` method enables you to determine if you have a valid handle.

```
function 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. If `H` is an array, then `B` is also an array. If `H` is not an array of handle objects, then every element in `B` is false.

Determining If a Handle Belongs to the Handle Class

You can use the `isa` function to determine if a handle belongs to the class `handle`, as opposed to being a Sun™ Java™ or Handle Graphics® handle. For example, suppose you have the following class:

```
classdef button < handle
   properties
      UiHandle
   end
   methods
      function obj = button(pos))
         obj.UiHandle = uicontrol('Position',pos);
      end
   end
end
```

Create a `button` object

```
h = button([50 20 50 20]);
```

The difference between the Handle Graphics object handle, which is stored in the `UiHandle` property and the `handle` class handle, `h`.

```
>> isa(h,'handle')
ans =
     1

>> isa(h.UiHandle,'handle')
ans =
     0
```

```
>> ishandle(h)
ans =
     0

>> ishandle(h.UiHandle)
ans =
     1
```

Handle Class Delete Methods

The MATLAB® runtime destroys objects in the workspace of a function when:

- A variable is reassigned to a new value

- A variable is not used in the remainder of a function

- A function ends

When MATLAB destroys an object, values stored in the object's properties are also destroyed and any computer memory associated with the object is returned to MATLAB or the operating system. While you do not need to be concerned with freeing memory in handle classes, there might be other operations that you want to perform when an object is destroyed, such as closing a file or shutting down an external program that was started from the object's constructor function.

Once an object is deleted, any handles to that object in any workspace become invalid, which means they produce an error if an attempt is made to access their contents. Variables previously referencing the deleted value become unassigned and inaccessible.

When to Define a Delete Method for Your Class

You should perform any necessary cleanup operations in a special optional method having the name delete (sometimes called a destructor method). The delete method's signature is,

```
function delete(h)
```

where h is a scalar handle.

For example, you might want to close a file that you have opened for writing in your object's delete method. This function calls fclose on a file identifier that is stored in the object's FileID property:

```
function delete(obj)
    fclose(obj.FileID);
end
```

"Using Objects to Write Data to a File" on page 2-30 presents an example that uses this delete method.

Object Lifecycle

The MATLAB runtime invokes the delete method only when an object's lifecycle ends. An object's lifecycle 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 is defined to exist from the time the variable is assigned until the time it is reassigned, cleared, or no longer referenced within that function or any handle array. If the variable is not cleared or reassigned, the lifecycle of an object ends sometime between the time its variable is no longer used inside the function and the end of the function. Within the time between a variable's last use and the end of function execution, the MATLAB runtime destroys the object to which the variable refers.

Sequence During Handle Object Destruction

When an object is destroyed (explicitly or because its lifecycle ended), MATLAB invokes the delete methods in the following sequence:

1 The delete method for the object's class

2 The delete method of each base class, starting with the immediate base classes and working up the hierarchy to the most general base classes

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` so the delete
function of `supclass1` is called before the delete function of `supclass2`.

```
classdef myClass < supclass1 < supclass2
```

Superclass `delete` methods cannot call methods or access properties
belonging to a subclass.

After each `delete` method has been called, the property values belonging
exclusively to the class whose method was just called are destroyed. The
destruction of property values that contain other handle objects causes the
`delete` methods for those objects to be called.

Restricting Object Deletion

A class can prevent explicit destruction of objects by setting its `delete`
method's `Access` attribute to `private`. An error is issued if you call `delete` on
a handle object whose `delete` method is `private` unless the call to `delete`
is made from within one of its own class methods.

Similarly, if the class `delete` method's `Access` attribute is set to `protected`,
explicitly deleting objects of that class is allowed only by methods of the class
and any subclasses.

Finding Handle Objects and Properties

In this section...
"Finding Handle Objects" on page 4-16
"Finding Handle Object Properties" on page 4-16

Finding Handle Objects

The findobj method enables you to locate handle objects that meet certain conditions.

```
function HM = findobj(H,<conditions>)
```

The findobj method uses the same syntax as the MATLAB® findobj function, except that the first argument must be a handle array. This method returns an array of handles matching the conditions specified.

Finding Handle Object Properties

The findprop method returns the meta.property object for the specified object and property.

```
function mp = findprop(h,'PropertyName')
```

The findprop method returns the meta.property object associated with the *PropertyName* property defined by the class of h, or it can be a dynamic property created by the addprop method of the dynamicprops class.

You can use the returned meta.property object to obtain information about the property, such as querying the settings of any of its attributes. For example, the following statements determine that the setting of the AccountStatus property's Dependent attribute is false.

```
ba = BankAccount(007,50,'open');
mp = findprop(ba,'AccountStatus'); % get meta.property object
mp.Dependent
ans =
     0
```

"Obtaining Information About Classes with Meta-Classes" on page 3-26 provides more information on meta-classes.

Implementing a Set/Get Interface for Properties

In this section...
"The Standard Set/Get Interface" on page 4-18
"Property Get Method" on page 4-18
"Property Set Method" on page 4-19
"Subclassing hgsetget" on page 4-19

The Standard Set/Get Interface

The MATLAB® Handle Graphics® system implements an interface based on set and get methods that enable you to set or query the value of graphics object properties. The hgsetget subclass of the handle class provides implementations of these methods that your class can inherit and provide the same functionality for instances of your class.

Note The set and get methods referred to in this section are different from the property set access and property get access methods that you can implement to control what happens when any attempt is made to set or query a property value. See "Controlling Property Access" on page 6-10 for information on property access methods.

Property Get Method

The get method returns property values from a handle array.

```
function SV = get(H)
function CV = get(H,prop)
```

- If you do not specify property names, get returns a struct array in which each element corresponds to the element in H. Each field in the struct corresponds to a property defined by the class of H. The value of each field is the value of the corresponding property.

- If you specify prop as a char array, then it is interpreted as a property name and get returns the value of that property if H is scalar, or returns a cell array of property values if H is an array of handles. The cell array is

always a column vector regardless of the shape of H. If prop is a cell array of string property values, then get returns a cell array of values where each row in the cell corresponds to an element in H and each column in the cell corresponds to an element in prop.

Property Set Method

The set method assigns values to properties for handles in array H.

```
function S = set(H)
function info = set(H, prop)
function set(H,'PropertyName',PropertyValue)
```

If you do not specify property values, then set returns a cell array of possible values for each requested property, when the property value is restricted to a finite enumeration of possible values.

- If you specify only H, set returns a struct with one field for each property in the class of H and with each field containing either an empty cell array or a cell array of possible property values (if such a finite set exists).

- If you specify prop as a string containing a property name, then set returns either a cell array of possible values or an empty cell.

- If you specify prop as a cell array of property names, then set returns a cell column vector in which each cell corresponds to a property in prop and each cell's value is a cell array of possible values or the empty cell if there is no finite enumeration of possible values.

You can also pass property-value pairs to set using cell arrays and structures as described for the built–in set function.

Subclassing hgsetget

This example creates a class with the set/get interface and illustrates the behavior of the inherited methods:

```
classdef myAccount < hgsetget % subclass hgsetget
    properties
        AccountNumber
        AccountBalance = 0;
        AccountStatus
```

```
        end % properties
        methods
           function obj = myAccount(actnum,intamount,status)
              obj.AccountNumber = actnum;
              obj.AccountBalance = intamount;
              obj.AccountStatus = status;
           end % myAccount
           function obj = set.AccountStatus(obj,val)
              if ~(strcmpi(val,'open') ||...
                 strcmpi(val,'deficit') ||...
                 strcmpi(val,'frozen'))
                 error('Invalid value for AccountStatus ')
              end
           obj.AccountStatus = val;
           end % set.AccountStatus
        end % methods
     end % classdef
```

Create an instance of the class and save its handle:

```
h = myAccount(1234567,500,'open');
```

You can query the value of any object property using the inherited get method:

```
get(h,'AccountBalance')
ans =

   500
```

You can set the value of any property using the inherited set method:

```
set(h,'AccountStatus','closed')
??? Error using ==> myAccount.myAccount>myAccount.set.AccountStatus at 19
Invalid value for AccountStatus
```

The property set function (set.AccountStatus) is called when you use the set method:

```
set(h,'AccountStatus','frozen')
```

Listing All Properties

The standard set/get interface enables you to display all object properties and their current values using get with no output argument and only a handle as input. For example,

```
>> get(h)
    AccountNumber: 1234567
    AccountBalance: 500
    AccountStatus: 'open'
```

Similarly, you can list the object's setable properties using set:

```
>> set(h)
    AccountNumber: {}
    AccountBalance: {}
    AccountStatus: {}
```

Customizing the Property List

You can customize the way property lists are displayed by redefining the following methods in your subclass:

- setdisp — Called by set when you call set with no output arguments and a single input parameter containing the handle array.

- getdisp — Called by get when you call get with no output arguments and a single input parameter containing the handle array.

4-21

5

Building on Other Classes

Hierarchies of Classes — Concepts

Classification

Organizing classes into hierarchies facilitates the reuse of code and, thereby, the reuse of solutions to design problems that have already been solved. You might 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.

Sales People and Engineers are
subsets of Employees

Base class
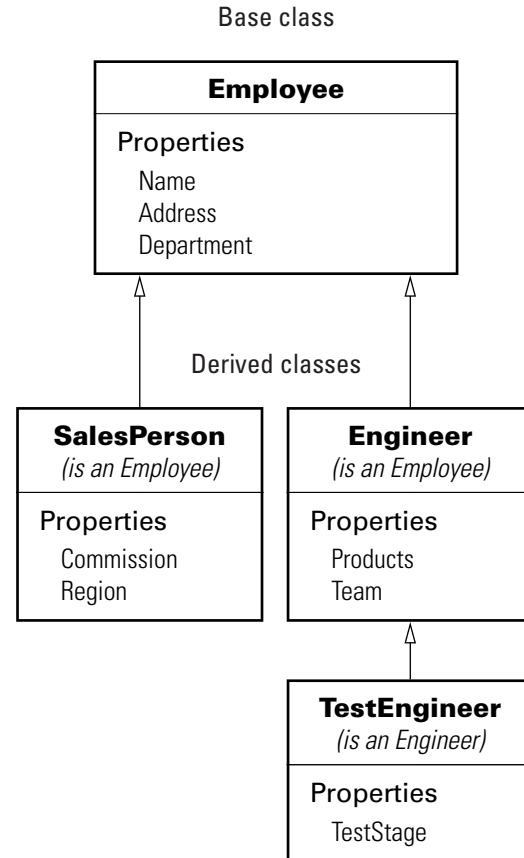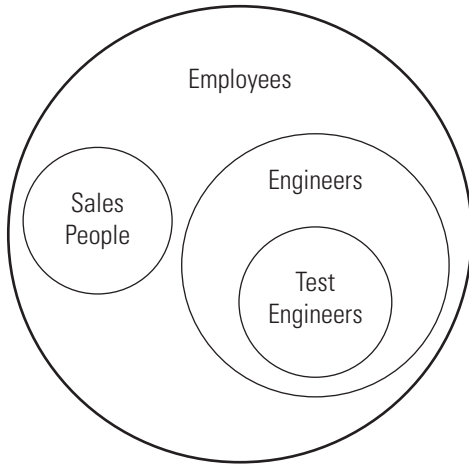
At the root of the hierarchy is the Employee class. It contains data and
operations that apply to the set of all employees. Contained in the set
of employees are subsets whose members, while still employees, are also
members of sets that more specifically define the type of employee. These
subsets are represented by subclasses like TestEngineer.

Developing the Abstraction

Classes are representations of real world concepts or things. When designing
a class, you need to 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 might be what all employees have in common.

When designing classes, your abstraction should contain only those elements that are necessary. For example, the employee's hair color and shoe size certainly characterize the employee, but are probably not relevant to the design of this employee class. Their sales region might be relevant only to some employee so this characteristic belongs in a subclass.

Designing Class Hierarchies

As you design a system of classes, you should place common data and functionality in a superclass, which you can then use to derive subclasses classes. The subclasses inherit all the data and functionality of the superclass and contain only aspects that are unique to their particular purposes. This approach provides advantages:

- You avoid duplicating code that can be common to all classes.

- You can add or change subclasses at any time without modifying the superclass or affecting other derived classes.

- If a change is made to the superclass (e.g., all employees are assigned a badge number), then these changes are automatically picked up by the subclasses.

Super and Subclass Behavior

Subclass objects behave like objects of the super class because they are specializations of the super class. This fact facilitates the development of related classes that behave similarly, but are implemented differently.

A Subclass Object Is A Superclass Object

You usually can 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 of the superclass, as well as any new features defined by the subclass.

A Subclass Object Can be Treated Like a Superclass Object

You can pass a subclass object to a super class method, but you can access only those properties that are defined in the super class. This behavior enables you to modify the subclasses without affecting the super class.

Two points about super and subclass behavior to keep in mind are:

• Methods defined in the super class can operate on objects belonging to the subclass.

• Methods defined by the subclass cannot operate on objects belonging to the super class.

Therefore, you can treat an `Engineer` object like any other `Employee` object, but an `Employee` object cannot pass for an `Engineer` object.

Creating Subclasses — Syntax and Techniques

In this section...

"Defining a Subclass" on page 5-6

"Referencing Superclasses from Subclasses" on page 5-6

"Creating an Alias for an Existing Class" on page 5-7

Defining a Subclass

To define a class that is a subclass of another class, add the superclass to the classdef line after a < character:

```
classdef classname < superclassname
```

When inheriting from multiple classes, use the & character to indicate the combination of the superclasses:

```
classdef classname < super1 & super2
```

See "Class Member Compatibility" on page 5-11 for more information on class intersection.

Class Attributes

Subclasses do not inherit superclass attributes, with the exception of the InferiorClasses attribute. If both superclasses and the subclass define the InferiorClasses attribute, the true list of inferior classes is the union of the inferior classes defined by the subclass and all superclasses.

Referencing Superclasses from Subclasses

When you construct an instance of a subclass, you must use the *obj@baseclass1(args)* syntax to initialize the object for each superclass. For example, the following segment of a class definition shows a class called stock that is a subclass of a class called asset.

```
classdef stock < asset
   methods
       function s = stock(...)
```

```
            s = s@asset(...) % call asset constructor
            ...
        end
    end
end
```

"Constructing Subclasses" on page 7-17 provides more information on creating subclass constructor methods.

Creating an Alias for an Existing Class

You can create an alias for a class that enables you to refer to the class using a different name. This technique is similar to the C++ typedef concept. To create an alias, create an empty subclass:

```
classdef newclassname < oldclassname
end
```

This technique might be useful when reloading objects that were saved using the old class name. However, the class of the object reflects the new name. For example,

```
class(obj)
```

returns the new class name.

Modifying Superclass Methods and Properties

In this section...

Modifying Superclass Methods

An important concept to keep in mind when designing classes is that a subclass object is also an object of its superclass. Therefore, you should be able to pass a subclass object to a superclass method and have the method execute properly. At the same time, you might need to apply special processing to the unique aspects of the subclass. Some useful techniques to do this include:

- Calling a superclass method from within a subclass method
- Redefining in the subclass protected methods called from within a public superclass method
- Defining the same named methods in both super and subclass, but using different implementations

Extending Superclass Methods

Subclass methods can call superclass methods of the same name. This enables you to extend a superclass method in a subclass without completely redefining the superclass method. For example, suppose both superclass and subclass define a method called `foo`. The method names are the same so the subclass method can call the superclass method, but the subclass method can also perform other steps before and after calling the call to the superclass method to operate on the specialized parts to the subclass that are not part of the superclass.

For example this subclass defines a `foo` method, which calls the superclass `foo` method

```
classdef sub < super
   methods
      function foo(obj)
         preprocessing steps
```

```
            foo@super(obj); % Call superclass foo method
            postprocessing steps
        end
    end
end
```

See "Invoking Superclass Methods in Subclass Methods" on page 7-12 for more on this syntax.

Completing Superclass Methods

A superclass method can define a process that is executed in a series of steps using a protected method for each step (`Access` attribute set `protected`). Subclasses can then create their own versions of the protected methods that implement the individual steps in the process.

The implementation of this technique would works as shown here:

```
classdef super
    methods
        function foo(obj)
            step1(obj)
            step2(obj)
            step3(obj)
        end
    end
    methods (Access = protected)
        function step1(obj)
            superclass version
        end
        etc.
    end
end
```

The subclass would not reimplement the `foo` method, it would reimplement 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® dispatching rules ensure the subclass step methods are called.

```
classdef sub < super
   ...
   methods (Access = protected)
      function step1(obj)
          subclass version
      end
      etc.
   end
end
```

Redefining Superclass Methods

You can completely redefine a superclass method. In this case, both the superclass and the subclass would define the same named method.

Modifying Superclass Properties

There are only two conditions that allow you to redefine superclass properties:

- The superclass property Abstract attribute is set to true

- The superclass property has both the SetAccess and GetAccess attributes set to private

In the first case, the superclass is just requesting that you define a concrete version of this property to ensure a consistent interface. In the second case, only the superclass can access the private property, so the subclass is free to reimplement it in any way.

Subclassing from Multiple Classes

Class Member Compatibility

When you create a subclass derived from multiple classes, the subclass inherits the properties, methods, and events defined by all specified superclasses. If a property, method, or event is defined by more than one superclass, 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.

There are a variety of situations where name and definition conflicts can be resolved, as described in the following sections.

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 `GetAcess` attributes set to `private`

- The properties have the same definition in all super classes (e.g., 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's `Access` attribute is set to `private` so only the defining superclass can access the method.

- The method has the same definition in all derived classes. This 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. This means the superclass methods must not have their `Sealed` attributed set to `true`.

- Only one superclass defines the method as `Sealed`, in which case, the sealed method's definition is adopted by the subclass.

- The superclases define the methods as `Abstract` and rely on the subclass the 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's `ListenAccess` and `NotifyAccess` attributes must be set to `private`.

- The event has the same definition in all superclasses (e.g., when all superclasses inherited the event from a common base class)

Using 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 superclasses are enhanced in future versions and might introduce new conflicts. These potential problems might be reduced if only one superclass is unrestricted and all other superclasses are classes in which all methods are abstract and must be defined by a subclass or inherited from the unrestricted superclass.

In general, when using multiple inheritance, you should ensure that all superclasses remain free of conflicts in definition.

Subclassing MATLAB® Built-In Classes

In this section...
"MATLAB® Built-In Classes" on page 5-13
"Why Subclass Built-In Classes" on page 5-13
"Behavior of Built-In Functions with Subclass Objects" on page 5-15
"Example — Representing Hardware as a Class" on page 5-18

MATLAB® Built-In Classes

Built-in classes represent fundamental kinds of data such as numeric arrays, logical arrays, and character arrays. Other classes tend to combine data belonging to these fundamental classes. `cell` arrays and `struct` arrays are other fundamental classes.

Built-in classes define methods that enable you to perform many operations on objects of these classes. For example, consider the `double` class. The MATLAB® language enables you to create objects of class `double` using an assignment statement, to create arrays of doubles, and to perform complex operations on these arrays such as matrix multiplication, indexing, and so on.

See "Built-In Classes (Data Types)" for more information on these classes.

Why Subclass Built-In Classes

You can derive classes from most MATLAB built-in classes. This is useful when you are creating a class that handles data primarily of a built-in class and you want to be able to use the methods of that class directly with objects of your class. For example, if you subclass the built-in class `double`, you can use an object of your subclass anywhere a double is expected.

Classes that define enumeration can derive from integral classes and thereby 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`).

Built-In Classes You Cannot Subclass

You cannot subclass the following built-in MATLAB classes:

- `char`

- `cell`

- `struct`

- `function_handle`

Extending Double

The MATLAB built-in class `double` defines a wide range of methods to perform arithmetic operations, indexing, matrix operation, and so on. Therefore, subclassing `double` enables you to add specific features without implementing many of the methods that a numeric class needs to function usefully in the MATLAB language.

The following class adds a property to contain descriptive text along with an array of doubles.

```
classdef mydouble < double
   properties
      DataHeader
   end
   methods
      function obj = mydouble(data,str)
         obj = obj@double(data); % initial the base class portion
         obj.DataHeader = str;
      end
      function display(obj)
         disp(obj.DataHeader)
         display(double(obj)) % built-in display function for double
      end
   end
end
```

You can create an instance of the class `mydouble` and call any methods of the `double` class. Note that `mydouble` defines its own display method that calls the built-in `display` function used for doubles:

```
>> md = mydouble(rand(4),'Random numbers')
Random numbers
ans =
    0.4218    0.6557    0.6787    0.6555
    0.9157    0.0357    0.7577    0.1712
    0.7922    0.8491    0.7431    0.7060
    0.9595    0.9340    0.3922    0.0318
```

Calling a method inherited from class `double`, like `sum`, returns a double and therefore uses the `display` method of class `double`:

```
>> sum(md)
ans =
    3.0892    2.4746    2.5718    1.5645
```

The `mydouble` class also inherits indexing capabilities. For example:

```
>> md(1:3,1)
ans =
    0.4218
    0.9157
    0.7922
```

Concatenation is not supported for subclasses that define properties. This is because it is not possible to combine the properties of different objects. For information about the behavior of classes that derive from built-in classes, see "Behavior of Built-In Functions with Subclass Objects" on page 5-15 .

```
md2 = mydouble(rand(4),'More random numbers');
>> [md md2]
??? Error using ==> horzcat
Cannot concatenate objects of class mydouble. When a class inherits from a builtin
type, concatenation is only allowed if it does not define any property.
```

Behavior of Built-In Functions with Subclass Objects

When you create a class that derives from a MATLAB built-in class, the inherited part of the data is treated like a private property of the built-in class. This means that you can access this data only by using methods of the built-in class or by indexing.

"Extending Double" on page 5-14 provides an example that illustrates this type of behavior.

Built-in functions inherited from built-in classes behave in different ways, depending on which particular function you are using. The following categories describe each behavior:

- "Built-In Methods" on page 5-16
- "Indexing Functions" on page 5-16
- "Catenation Functions" on page 5-17

Built-In Methods

Most built-in functions used with built-in classes are actually methods of that class. For example, the `sin` function is a method of the `double` and the `single` classes. All built-in class methods work with subclasses of the built-in class. (This is a basic concept of object-oriented design).

When you call a built-in method on a subclass object, only the built-in parts of the subclass object are used as inputs to the method and the value returned is of the built-in class, not your subclass. For example, using the `mydouble` class defined in the previous section, you can see that calling the `sin` method on a `mydouble` object returns an object of class `double`:

```
md = mydouble(rand(4),'Random numbers');
smd = sin(md);
>> whos
  Name      Size             Bytes  Class
  md        4x4                212  mydouble
  smd       4x4                128  double
```

To list the methods of a built-in class, use the `methods` command. For example:

```
methods double
```

Indexing Functions

Built-in classes use specially implemented versions of the `subsref`, `subsasgn`, and `subsindex` functions 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 `mydouble` subclass object returns the second element in the vector:

```
md = mydouble([1 2 3 4],'Four-element vector');
>> md(2)
ans =
     2
```

Assigning a new value to the second element in the `mydouble` object operates only on the built-in data:

```
>> md(2) = 12
Four-element vector
ans =
     1    12     3     4
```

To access the property value, use dot notation:

```
>> md.DataHeader
ans =
Four-element vector
>> md.DataHeader = 'Non-monotonic vector'
ans =
Non-monotonic vector
```

Catenation Functions

Built-in classes use the functions `horzcat`, `vertcat`, and `cat` to implement concatenation. When you use these function with subclass objects of the same type, the built-in data parts are concatenated to form a new object. For example, you can concatenate objects of the following subclass:

```
classdef simpledouble < double
   methods
      function obj = simpledouble(data)
         obj = obj@double(data); % initial the base class portion
      end
   end
end
```

Concatenate two objects along the second dimension:

```
sd1 = simpledouble(rand(2))
sd2 = simpledouble(rand(2))
c = cat(2,sd1,sd2)
>> disp(c) % simpledouble has no display method
    0.8143    0.9293    0.1966    0.6160
    0.2435    0.3500    0.2511    0.4733
```

cat returns a new simpledouble object:

```
>> whos
  Name      Size            Bytes  Class
  c         2x4               120  simpledouble
  sd1       2x2                88  simpledouble
  sd2       2x2                88  simpledouble
```

Note that you cannot concatenate subclass objects of built-in classes if the subclass defines properties. Such an operation does not make sense if there are properties defined because there is no usable way to combine properties of different objects.

Example — Representing Hardware as a Class

This example shows the implementation of a class to represent an optical multiplex card. These cards typically have a number of input ports, which are represented here by their data rates and their names. There is also an output port. The output rate of a multiplex card is the sum of the input port data rates.

The OMuxCard class defines the output rate as a dependent property, and then defines a get access method for this property. This means the actual output rate is calculated when it is required. See "Property Get Methods" on page 6-13 for more information on this technique.

Why Derive from a Built-In Class

The OMuxCard class is derived from the int32 class because the input port data rates are being represented by 32–bit integers. The OMuxCard class inherits the methods of the int32 class, which simplifies the implementation of this subclass.

Class Definition

Here is the definition of the `OMuxCard` class. Notice that the input port rates are used to initialize the `int32` portion of class.

```
classdef OMuxCard < int32
   properties
      InPutNames % cell array of strings
      OutPutName % a string
   end
   properties (Dependent = true)
      OutPutRate
   end
   methods
      function obj = OMuxCard(inptnames, inptrates, outpname)
         obj = obj@int32(inptrates); % initial the int32 class portion
         obj.InPutNames = inptnames;
         obj.OutPutName = outpname;
      end
      function x = get.OutPutRate(obj)
         x = sum(obj); % calculate the value of the property
      end
   end
end
```

Using the Class with Methods of int32

The constructor takes three arguments:

- `inptnames` — Cell array of input port names

- `inptrates` — Vector of input port rates

- `outpname` — Name for the output port

```
>> omx = OMuxCard({'inp1','inp2','inp3','inp4'},[3
12 12 48],'outp')
ans =
            3          12          12          48
```

You can treat an `OMuxCard` object like an `int32`. For example, this statement accesses the `int32` data in the object to determine the names of the input ports that have a rate of `12`:

```
>> omx.InPutNames(find(omx==12))
ans =
    'inp2'     'inp3'
```

Indexing the `OMuxCard` object accesses the `int32` vector of input port rates:

```
>> omx(1:2)
ans =
          3            12
```

The `OupPutRate` property get access method makes use of `sum` to sum the output port rates:

```
>> omx.OutPutRate
ans =
    75
```

Abstract Classes and Interfaces

Abstract Classes

An *abstract* class serves as a basis (i.e., a superclass) for a group of related subclasses. It forms the abstractions that are common to all subclasses by specifying the common properties and methods that all subclasses must implement. However, the abstract class cannot itself be instantiated. Only the subclasses can be instantiated. These subclasses are sometimes referred to as *concrete* classes.

Abstract classes are useful for describing functionality that is common to a group of classes, but requires unique implementations within each class. This approach is often called an *interface* because the abstract class defines the interface of each subclass without specifying the actual implementation.

Defining Abstract Classes

Define an abstract class by setting the Abstract attribute on one or more methods of a class to true. You do not use a function...end block to define an abstract method, use only the method signature.

For example, the group abstract class defines two methods that take two input arguments and return a result:

```
classdef group
% Both methods must be implemented so that
% the operations are communtative
   methods (Abstract)
       rsult = add(numeric,polynom)
       rsult = times(numeric,polynom)
   end
end
```

The subclasses must implement methods with the same names. The names and number of arguments can be different. However, the abstract class typically conveys details about the expected implementation via its comments and argument naming.

Abstract Properties

For properties that have `Abstract` attributes set to `true`:

- Concrete subclasses must redefine abstract properties without the `Abstract` attribute set to true and must use the same values for `SetAccess` and `GetAccess` attributes as the base class.

- Abstract properties cannot define set or get access methods (see "Controlling Property Access" on page 6-10) and cannot specify initial values. The subclass must define the property and then can create set or get access methods and specify initial values.

Abstract Methods

For methods that have `Abstract` attributes set to `true`:

- Abstract methods have no implementation in the abstract class. The method has a normal function line (without the `function` or `end` key words) that can include input and output argument lists.

- 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.

Interfaces and Abstract Classes

The properties and methods defined by a class form the interface that determines how users interact with objects of the class. When you are creating a group of related classes, it makes sense to define a common interface to all these classes, even though the actual implementations of this interface might differ from one class to another. Abstract properties and methods provide a mechanism to create interfaces for groups of classes.

For example, consider a set of classes designed to represent a variety of graphs (e.g., line plots, bar graphs, pie charts, and so on). Suppose all classes

need to implement a Data property to contain the data used to generate the graph. However, the form of the data might differ considerably from one type of graph to another. Consequently, the way each class implements the Data property needs to be different.

The same differences apply to methods. All classes might 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 enables you to enforce a consistent interface to a group of related objects. As more classes are added in the future, the interface is maintained.

Example — Interface for Classes Implementing Graphs

This example creates an interface for classes used to display specialized graphs. The interface is an abstract class that defines properties and methods that the subclasses need to implement, but does not specify how to implement these components. This enforces the use of a consistent interface while at the same time provides the necessary flexibility to implement the internal workings of each specialized graph subclass differently.

In this example, the interface, derived subclasses, and a utility function are contained in a package directory:

```
+graphics/graph.m       % abstract interface class
+graphics/linegraph.m   % concrete subclass
+graphics/addButtons.m  % static method of graph class
```

Interface Properties and Methods

The graph class specifies the following properties to be defined by subclasses:

- Primitive — Handle of the Handle Graphics® object used to implement the specialized graph. The user has no need to access these objects directly so this is property has protected SetAccess and GetAccess.

- AxesHandle — Handle of the axes used for the graph. The specialized graph objects might set axes object properties and also limit this properties SetAccess and GetAccess to protected.

- Data — All specialized graph objects need to store data, but the type of data varies so each subclass defines the way the data is stored. Subclass users can change the data so this property has public access rights.

The graph class names three abstract methods that subclasses need to implement. The graph class also suggests in comments that each subclass constructor needs to accept the data to be plotted 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. This method is used as a callback for the zoom buttons created by the addButtons static method.

- updateGraph — Method called by the set.Data method to update the plotted data whenever the Data property is changed.

Interface Guides Class Design

The package of classes that derive from the graph abstract class implement the following behaviors:

- Creating an instance of a specialized graph object (subclass object) without rendering the plot

- Specifying any or none of the object's properties when you create a specialized graph object

- Changing any object property automatically updates the currently displayed plot

- Allowing each specialized graph object to implement whatever additional properties it needs to give users control over those characteristics.

Defining the Interface

The graph class is an abstract class that defines the methods and properties used by the derived classes. Comments in the abstract class suggest the intended implementation:

```matlab
classdef graph < 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 % HG primitive handle
        AxesHandle % Axes handle
    end
    properties % Public access
        Data
    end
    methods (Abstract)
        draw(obj)
        % Use a line, surface,
        % or patch HG primitive
        zoom(obj,factor)
        % Change the CameraViewAngle
        % for 2D and 3D views
        % use camzoom for consistency
        updateGraph(obj)
        % Called by the set.Data method
        % to update the drawing primitive
        % whenever the Data property is changed
    end
    methods
        function set.Data(obj,newdata)
            obj.Data = newdata;
            updateGraph(obj)
        end
    end
    methods (Static)
        addButtons(obj)
    end
end
```

The `graph` 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. However, by defining the set access method that calls an abstract method (`updateGraph`, which must be implemented by each subclass), the `graph` interface more effectively imposes a specific design on the whole package of classes, without limiting flexibility.

Static Method to Work with All Subclasses

The `addButtons` method simply adds push buttons for the `zoom` methods that each subclass must implement. Using a static method instead of an ordinary function enables the `addButtons` method to access the protected class data (the axes handle). Notice how the particular object's `zoom` method is defined as the buttons' callback.

```
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
```

Deriving a Concrete Class — linegraph

Note Display the fully commented code for the `linegraph` class by clicking this link: linegraph class.

This example defines only a single subclass used to represent a simple line graph. It derives from `graph`, but provides implementations for the abstract methods `draw`, `zoom`, `updateGraph`, and its own constructor. The base class (`graph`) and subclass are all contained in a package (`graphics`), which must be used to reference the class name:

```
classdef linegraph < graphics.graph
```

Adding Properties

The linegraph class implements the interface defined in the graph class and adds two additional properties—LineColor and LineType. This class defines initial values for each property in case a linegraph object is created without specifying these properties (which might happen when reloading an object from a MAT-file). Specifying property values in the constructor is optional. Data is required to produce a plot, but not to create a linegraph object):

```
properties
    LineColor = [0 0 0];
    LineType = '-';
end
```

The linegraph Constructor

The constructor accepts a struct with x and y coordinate data, as well as property name/property value pairs:

```
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
```

Implementing the draw Method

The linegraph draw method uses property values to create a line object. The line handle is stored as protected class data. The linegraph class does not create the line primitive and display its plot until the draw method is called. To support the use of no input arguments for the constructor, draw checks the Data property to determine if it is empty before proceeding:

```
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 = get(h,'Parent');
end
```

Implementing the zoom Method

The linegraph zoom method follows the directions in the graph class and uses the camzoom function, which provides a convenient interface to zooming and operates correctly with the push buttons created by the addButtons method.

Defining the Property Set Methods

Property set methods provide a convenient way to execute code automatically when the value of a property is changed or assigned for the first time in a constructor (see "Property Set Methods" on page 6-11). 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, which 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 might define different properties unique to their specialization (e.g., FaceColor).

The Data property set method is implemented in the graph class. However, the graph class requires each subclass to define a method called updateGraph, which handles the update of plot data for the specific drawing primitive used.

Using the linegraph Class

The linegraph class defines the simple API specified by the graph base class and implements its specialized type of graph:

```
d.x = 1:10;
d.y = rand(10,1);
lg = graphics.linegraph(d,'LineColor','b','LineType',':');
```

```
lg.draw;
graphics.graph.addButtons(lg);
```

Clicking the **Zoom In** button shows the zoom method providing the callback for the button.
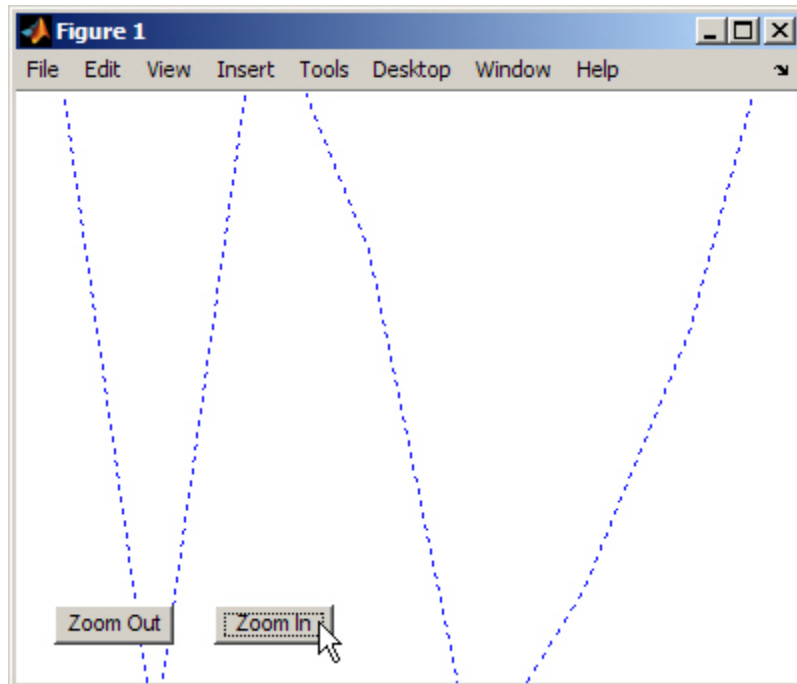
Changing properties updates the graph:

```
d.y = rand(10,1); % new set of random data for y
lg.Data = d;
lg.LineColor = [.9 .1 .6]; % LineColor can be char or double
```

Now click **Zoom Out** and see the new results:

6
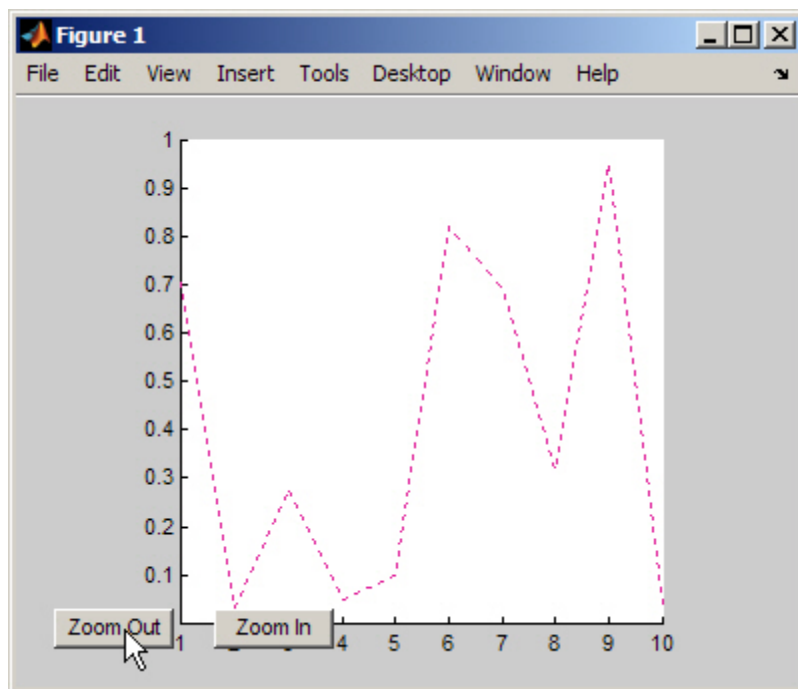
Properties — Storing Class Data

How to Use Properties

In this section...
"What Are Properties" on page 6-2
"Types of Properties" on page 6-2

What Are Properties

Properties encapsulate the data that belongs to instances of classes. Data contained in properties can be public, protected, or private, it can be a fixed set of constant values, or it can be dependent 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 that determine what happens when a property value is specified or queried.

See "Property Attributes" on page 6-7 for a summary of property attributes.

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 cannot be changed outside the class definition. See "Defining Named Constants" on page 3-25

- Calculate its value based on the current value of other data. See "Property Get Methods" on page 6-13

- Execute a function to determine if an attempt to assign a value meets a certain criteria. See "Property Set Methods" on page 6-11

- Trigger an event notification when any attempt is made to get or set its value. See "Property-Set and Query Events" on page 8-4

- Restrict access by other code to the property value. See the `SetAccess` and `GetAccess` attributes "Property Attributes" on page 6-7

- Control whether its value is saved with the object in a MAT-file. See

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 value is calculated by a get access method 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 no access methods are required

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 can be calculated only when needed.

When to Use Dependent Properties

Define properties as dependent when you need to:

- Compute the value of a property from other values (e.g., area is computed from `Width` and `Height` properties).
- Provide a value in different formats depending on other values (e.g., size of a push button in values determined by the current setting of its `Units` property).
- Provide a standard interface where a particular property might or might not be used depending on other values (e.g., different computer platforms might have different components on a toolbar).

"Controlling Property Access" on page 6-10 provides information on defining property access methods.

Defining Properties

Property Definition Block

The following illustration shows a typical property specification. The property 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 = [];

end
```

properties block

Property name   Initial value
(empty)

end keyword terminates definition block.
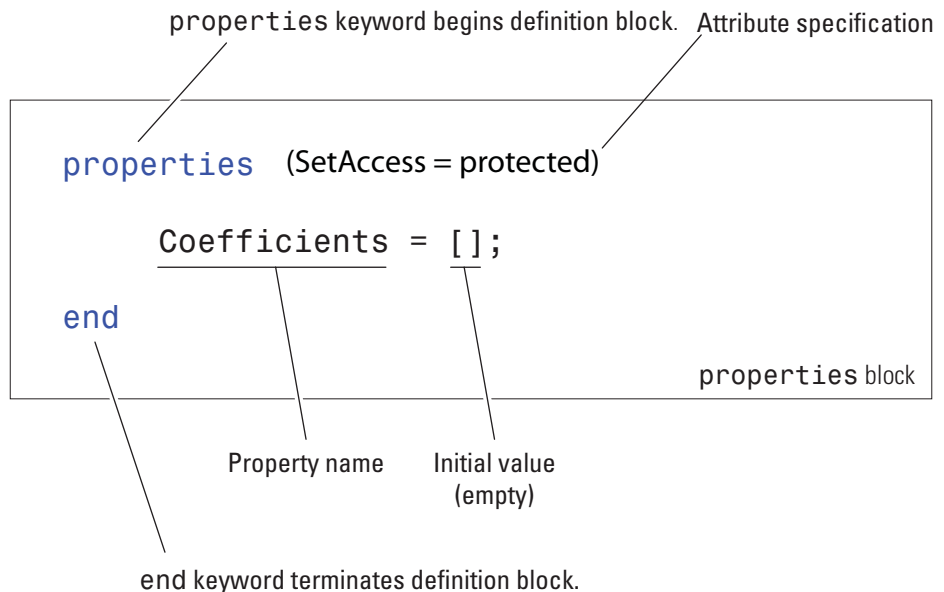
Assigning an Initial Value

The example above shows the `Coefficients` property specified as having an initial value that is an empty array of doubles.

You can initialize property values with MATLAB® expressions, but these expressions cannot refer to the class being defined 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 the class is first used.

Accessing Property Values

Property access syntax is similar to MATLAB structure field syntax. For example, assume there is a polynomial class called `polyno` that defines a `Coefficients` property. If you created a `polyno` object p:

```
p = polyno([1 0 -2 -3]); % Create an instance p (this
code does not execute)
```

you can access this property as follows:

```
c = p.Coefficients; % Assign the current property value to c
p.Coefficients = [4 0 -2 3 5]; % Assign new property values
```

When you access a property, the MATLAB runtime performs any operations that are required by the property, such as executing a property set or get access method and triggering property access events.

See "Implementing a Set/Get Interface for Properties" on page 4-18 for information on how to define `set` and `get` methods for properties.

Inheritance of Properties
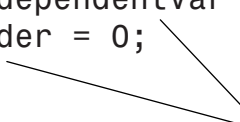
When you derive one class from another class, the derived (subclass) class inherits all the properties of the superclass. In general, subclasses should define only properties that are unique to that particular class. Superclasses should define properties that are used by more than one subclass.

Specifying Property Attributes

Attributes specified with the `properties` key word 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:

```
properties
     Coefficients = [];
end
properties (SetAccess = 'private')
     IndependentVar = 0;
     Order = 0;
end
```

These properties (and any others placed in this block) have private set access

Property Attributes

Table of Property Attributes

All properties support the attributes listed in the following table. Attributes enable you to modify the behavior of properties. Attribute values apply to all properties defined within the `properties` block that specifies the nondefault values.

Attribute Name	Class	Description
Abstract	logical default = false	If `true`, the property has no implementation, but a concrete subclass must redefine this property without Abstract being set to `true`. • Abstract properties cannot define set or get access methods. See "Controlling Property Access" on page 6-10. • Abstract properties cannot define initial values. See "Assigning an Initial Value" on page 6-5. • All subclasses must specify the same values as the superclass for the property SetAccess and GetAccess attributes. • Abstract=true should be used with the class attribute Sealed=false (the default).

(Continued)

Attribute Name	Class	Description
Constant	logical default = false	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 "Defining Named Constants" on page 3-25 for more information.
Dependent	logical default = false	If `false`, property value is stored in object. If `true`, property value is not stored in object and the set and get functions cannot access the property by indexing into the object using the property name. See "Property Get Methods" on page 6-13
GetAccess	char default = public	`public` — unrestricted access `protected` – access from class or derived classes `private` — access by class members only
GetObservable	logical default = false	If `true`, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are queried. See "Property-Set and Query Events" on page 8-4

(Continued)

Attribute Name	Class	Description
Hidden	logical default = false	Determines whether the property should be shown in a property list (e.g., Property Inspector, call to set or get, etc.).
SetAccess	char default = public	public — unrestricted access protected – access from class or derived classes private — access by class members only
SetObservable	logical default = false	If true, and it is a handle class property, then listeners can be created for access to this property. The listeners are called whenever property values are modified. See "Property-Set and Query Events" on page 8-4
Transient	logical default = false	If true, property value is not saved when object is saved to a file. See "Saving and Loading Objects" on page 3-17 for more about saving objects.

Controlling Property Access

Property Access Methods

Property access methods enable you to execute code whenever properties values are referenced or assigned a new value. These methods enable you to:

- Execute code before an assignment is made to property values to impose range restrictions, check for proper types and dimensions, provide error handling, and so on.

- Execute code before returning the current value of properties to calculate the value of properties that do not store values, change the value of other properties, trigger events, and so on.

Property access methods automatically execute whenever object properties are queried or set. However, you can define property access only for concrete properties (i.e., properties that are not defined as abstract).

Note Property set and get access methods are not equivalent to user-callable set and get methods used to access property values from an instance of the class. See "Implementing a Set/Get Interface for Properties" on page 4-18 for information on user-callable set and get methods.

Access methods have special names that include the property's name. Therefore, get.*PropertyName* is executed whenever *PropertyName* is

referenced and `set.`*`PropertyName`* is executed whenever *`PropertyName`* is assigned a new value.

You must define property access methods in a methods block that specifies no attributes. You cannot call these methods, but they are called when properties are accessed. Therefore, 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's `Methods` property. However, the `meta.property` object's `SetMethod` property contains a function handle to the property's set method and the `GetMethod` property contains a function handle to the property's get method.

For example, if the class `myClass` defines a set function for its `Text` property, you can obtain a function handle to this method from the `meta.class` object:

```
m = ?myClass;
m.Properties{1}.SetMethod % Assuming Text is the first property in the cell array
ans =
   @\mydir\@myClass\myClass.m>myClass.set.Text % This is a function handle
```

The `meta.class` object (m) contains `meta.property` objects corresponding to each class property in its `Properties` property. This example assumes that the `Text` property corresponds to the first `meta.property` object in the cell array of `meta.property` objects. The order of the class properties in the `meta.class` `Properties` property is the same as the order in which the properties are defined in the class definition.

"Obtaining Information About Classes with Meta-Classes" on page 3-26 provides more information on using meta classes.

"Function Handles" discusses the use of function handles.

Property Set Methods

Property set methods have the following syntax, where *`PropertyName`* is the name of the property.

```
methods % No method attributes
    function obj = set.PropertyName(obj,value)
end
```

Here `obj` is the object whose property is being assigned a value and `value` is the new value that is to be assigned to the property.

Value class set functions must return the object with the new value for the property assigned. Value classes replace the object whose property is being assigned with the object returned by the set method. Handle classes do not need to return the modified object.

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.

```
function obj = set.PropertyName(obj,value)
   if ~(value > 0)
      error('Property value must be positive')
   else
      obj.PropertyName = value;
   end
end
```

See "Restricting Properties to Specific Values" on page 2-46 for an example of a property set method.

Set Method Behavior
MATLAB® software calls a property set method whenever a property value is assigned, if a set method for that property exists. However, property set methods are NOT called in the following cases:

- Assigning a value to a property from within its own property set method, to prevent recursive calling of the set method

- Assigning a property to its default initial value

- Initial values specified in class definitions do not invoke the set method

- Copying a value object (i.e., not a handle object). Neither the set or get method is called when copying property values from one object to another.

It is possible for a set method from one property to assign values to other properties of the object. However, assignments made from property set methods cause the execution of any set methods defined for those properties.

When a property assignment takes place, modifications to the object that has been passed to the set method are reflected in the calling function's copy of the object. Therefore, an assignment to even a single property is able to affect the whole object. This enables set a method to change other properties in the object as well as its designated property. For example, a graphics window object might have a `Units` property and a `Size` property. Changing the `Units` property might also require a change to the values of the `Size` property to reflect the new units.

Property Get Methods

A property's get method is called whenever the property value is queried. For example, passing a property value in the following statement causes the method `get.XYData` to execute, if it exists.

```
plot(obj.XYData)
```

Property get methods have the following syntax, where *PropertyName* is the name of the property. Note that the function must return the property value.

```
methods % No method attributes
    function value = get.PropertyName(obj)
end
```

Dependent Properties — Values Not Stored

One application of a property get method is to determine the value of a property only when it is needed, and thereby avoid storing the value. To do this, set the property's `Dependent` attribute to `true`:

```
properties (Dependent = true)
    PropertyName
end
```

Now the get method for the *PropertyName* property determines the value of that property and assigns it to the object from within the method:

```
function value = get.PropertyName(obj)
    value = calculateValue;
    ...
end
```

This get method calls a function or static method `calculateValue` to calculate the property value and returns `value` to the code accessing the property. The property get method can take whatever action is necessary within the method to produce the output value.

"Dependent Properties" on page 2-48 provide an example of a property get method.

Set and Get Method Execution and Property Events

MATLAB software generates events before and after set and get operations that can be used to inform listeners that property values have been referenced or assigned. The timing of event generation is as follows:

- `PreGet` — Before the property's get method is called
- `PostGet` — After the property's get method has returned its value

If a property value is computed (`Dependent = true`), then the behavior of its set events are similar to the get events:

- `PreSet` — Before the property's set method is called
- `PostSet` — After the property's set method is called

If a property is not computed (`Dependent = false`, the default), then the events are generated by the actual assignment statement within the set method:

- `PropertyPreSet` — Before the property's new value is assigned within the set method
- `PropertyPostSet` — After the property's new value is assigned within the set method

"Events and Listeners — Concepts" on page 8-2 provides general information about events and listeners.

"Creating Property Listeners" on page 8-14 provides information about using property events.

"Implementing the PostSet Property Event and Listener" on page 8-27 shows an example of a property listener.

Access Methods and Subscripted Reference and Assignment

You can use subscripting as a way to reference or assign property values (e.g., `a = obj.prop(6)` or `obj.prop(6) = a`) without interfering with property set and get methods. In the case of subscripted reference, the get method returns the whole property value and MATLAB software then accesses the value referenced by subscripting on that object.

For subscripted assignment, the get method is invoked to get the property value, the subscripted assignment is carried out into the returned property, and then the new property value is passed to the set method.

MATLAB software always passes scalar objects to set and get methods. When reference or assignment occurs on an object array, the set and get methods are called in a loop.

Performing Additional Steps with Property Access Methods

Property access methods are useful in cases where you need 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, apply 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:

```
classdef Testpoint
  properties
    expectedResult = [];
  end
  properties(Constant,SetAccess = private,GetAccess = private)
    scalingFactor = 0.001;
  end
  methods
    function obj = set.expectedResult(obj,erIn)
```

```matlab
                    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/scalingFactor;
                end
            end
        end
```

Dynamic Properties — Adding Properties to an Instance

In this section...
"What Are Dynamic Properties" on page 6-17
"Defining Dynamic Properties" on page 6-17

What Are Dynamic Properties

Use dynamic properties to associated data with instances of classes without modifying the class definition. This is useful for attaching temporary data to objects. You can access dynamic properties like any class-defined properties to set and query their values. You can also set property attributes, add property set and get methods, and define listeners to respond to property change events.

It is possible for more than one program to define dynamic properties on the same object so you should take care to avoid name conflicts.

Defining 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

Setting Dynamic Property Attributes

Use the meta.Dynamicproperty object associated with the dynamic property to set property attributes. For example:

```
P.Hidden = true;
```

You can remove the dynamic property by deleting its `meta.Dynamicproperty` object:

```
delete(P);
```

The property attributes `Constant`, `Sealed`, and `Abstract` have no meaning for dynamic properties and setting the value of these attributes to `true` has no effect.

Attaching Data to the Object

Suppose, for example, you are using a predefined set of GUI widget classes (e.g., buttons, sliders, check boxes, etc.) and you want to store the location on a grid of each instance of the widget class. The widget classes might not be designed to store location data for your particular layout scheme and you want to avoid creating a map or hash table to maintain this information separately.

Assuming the `button` class is a subclass of `dynamicprops`, you could add a dynamic property to store your layout data. Here is a very simple class to create a uicontrol button:

```
classdef button < dynamicprops
   properties
      UiHandle
   end
   methods
      function obj = button(pos)
         obj.UiHandle = uicontrol('Position',pos);
      end
   end
end
```

Create an instance of the `button` class, add a dynamic property, and set its value:

```
b1 = button([20 40 80 20]); % button class uses HG-type position layout
b1.addprop('myCoord');  % Add a dynamic property
b1.myCoord = [2,3];     % Set the property value
```

You can access the dynamic property just like any other property, but only on the instance for which it has been defined:

```
>> b1.myCoord

ans =

    2    3
```

Defining Property Access Methods for Dynamic Properties

Dynamic properties enable you to add properties to class instances without modifying class definitions. You can also define property set access or get access methods without creating new class methods. See "Property Access Methods" on page 6-10 for more on the purpose and techniques of these methods.

Here are the steps for creating a property access methods:

• Define a function that implements the desired operations to be performed before the property set or get occurs.

• Obtain the dynamic property's corresponding `meta.Dynamicproperty` object.

• Assign to the `meta.Dynamicproperty` object's `GetMethod` or `SetMethod` property a function handle pointing to your set or get property function. This function is not a method of the class in this case so you cannot use a naming scheme like `set.PropertyName`. Instead, use any valid function name.

Suppose you want to create a property set function for the `button` class dynamic property `myCoord` created above. The function might be written as follows:

```
function set_myCoord(obj,val)
    if  ~(length(val) == 2) % require two values
        error('myCoords require two values ')
    end
    obj.myCoord = val; % set property value
end
```

Because `button` is a `handle` class, the property set function does not need to return the object as an output argument. You simply assign the value to the property if the value is value is valid.

Use the `handle` class method `findprop` to get the `meta.Dynamicproperty` object:

```
mb1 = b1.findprop('myCoord');
mb1.SetMethod = @set_myCoord;
```

The property set function is now called whenever you set this property:

```
b1.myCoord = [1 2 3] % length must be two
??? Error using ==> set_myCoord at 3
myCoords require two values
```

Dynamic Properties and 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. This means that if you delete a dynamic property, and then create another one with the same name, the listeners do not respond to events generated by the new property even though the property has the same name as the property for which the event was defined.

Having a listener defined for a deleted dynamic property does not cause an error, but the listener callback is never executed.

"Property-Set and Query Events" on page 8-4 provides more information on how to define listeners for these events.

7

Methods — Defining Class Operations

Class Methods

What Are 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 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 enables you to hide data and create special interfaces that must be used to access the data stored in objects.

See "Methods for MATLAB® Classes" on page 7-27 for a discussion of how to create classes that integrate well with the MATLAB® language.

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 7-6.

• *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 7-14

• *Destructor methods* are call 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 Delete Methods" on page 4-13

- *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 6-10

- *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 7-22

- *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 "Converting DocPolynom Objects to Other Types" on page 9-6 for some examples.

- *Abstract methods* serve to define a class that cannot be instantiated itself, but serves as a way to define a common interface used by a number of subclasses. Classes that contain abstract methods are often referred to as interfaces. See "Abstract Classes and Interfaces" on page 5-21 for more information and examples.

Method Attributes

Table of Method Attributes

All methods support the attributes listed in the following table. Attributes enable you to modify the behavior of methods. For example, you can prevent access to a method from outside the class or enable the method to be invoked without a class instance.

Attribute values apply to all methods defined within the methods block that specifies the nondefault values.

```
methods (attribute1=value1,attribute2=value2,...)
   ...
end
```

Attribute Name	Class	Description
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] = myMethod(x,y))
Access	char Default=public	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)

(Continued)

Attribute Name	Class	Description
Hidden	logical Default=false	When false, the method name shows in the list of methods displayed using the methods or methodsview commands. If set to true, the method name is not included in these listings.
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.
Static	logical Default=false	Set to true to define a method that is not depend on an object of the class and does not require an object argument. You must use the class name to call the method: *classname.methodname*
		"Static Methods" on page 7-22 provides more information.

Ordinary Methods

Defining Methods

You can specify methods:

- Inside of a class definition block

- In a separate file in the class @-directory

Methods Inside classdef Block

This example shows the definition of a method (the compute function in this example) within the classdef and methods blocks:

```
classdef ClassName
   methods (AttributeName = value,...)
      function x = compute(obj,inc)
         x = obj.y + inc;
      end % compute method
...
   end % methods block
...
end % classedf
```

Note Nonstatic methods must include an explicit object variable in the function definition. The MATLAB® language does not support an implicit reference in the method function definition.

Either of the following statements is correct syntax for calling a method where obj is an object of the class defining the compute method:

```
obj.compute(inc)
compute(obj,inc)
```

See also "Dot Notation vs. Function Notation" on page 7-9.

Method attributes apply only to that particular methods block, which is terminated by the end statement.

Methods in Separate Files

You can define class methods in separate files within the class @-directory. In this case, you create a function in a separate M-file having the same name as the function (i.e., *functionname*.m). You must declare the method signature within a methods block in the classdef block if you want to specify attribute values for that method. For example:

```
classdef myClass
   methods (AttributeName = value,...)
       tdata = testdata(obj,arg1,arg2)
...
   end % methods
...
end % classdef
```

You do not use the methods block in the separate M-file. You simply define the method as a function. So, using the example above, the file testdata.m, would contain the definition of testdata. Note that the signatures must match.

```
function tdata = testdata(myClass_object,argument2,argument3)
    ...
end
```

The following limitations apply to methods defined in separate files:

• If you want to specify attributes for a method defined in a separate file, you must declare this method in a methods block (specifying attribute values) within the classdef block.

- The syntax declared in the methods block (if used) must match the method's function line.

- The separate M-file must be in the class @-directory.

- The constructor method must be defined within the classdef block and, therefore, cannot be in a separate file. (See "Class Constructor Methods" on page 7-14 for information on this method.)

- Set and get property access methods must be defined within the classdef block and, therefore, cannot be in separate files. (See "Controlling Property Access" on page 6-10 for information on these methods.)

Determining Which Method Is Invoked

When the MATLAB runtime invokes an ordinary method that has an argument list, it uses the following criteria to determine which method to call

- The class of the left-most 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 left most 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 both classes define a method called combine.

Calling the method with an object of classB and classA:

```
combine(B,A)
```

actually calls the `combine` method of `classA` because A is the dominant argument.

See "Specifying Precedence" on page 7-12 for information on how to define class precedence.

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:

```
X = setColor(X,'red');
```

The equivalent method call using dot notation is:

```
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.

A Case Where the Result is Different. Here is an example of a case where dot and function notation can give different results. Suppose 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`

```
classdef classB (InferiorClasses = {?classA})
    ...
```

```
   end
```

The methodA method is defined with two input arguments, one of which is an object of classB:

```
classdef classA
methods
   function methodA(obj,obj_classB)
      ...
   end
end
```

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 string 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 string 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` (see "Using Dynamic Field Names" for information on MATLAB structures).

As an example, suppose an object has methods corresponding to each day of the week and these methods have the same names as the days of the week (`Monday`, `Tuesday`, and so on). Also, the methods take as string input arguments, the current day of the month (i.e., the date). Now suppose you write a function in which you want to call the correct method for the current day. You can do this using an expression created with the `date` and `datestr` functions:

```
obj.(datestr(date,'dddd'))(datestr(date,'dd'))
```

The expression `datestr(date,'dddd')` returns the current day as a string. For example:

```
datestr(date,'dddd')

ans =

Tuesday
```

The expression `datestr(date,'dd')` returns the current date as a string. For example:

```
datestr(date,'dd')

ans =
```

```
11
```

Therefore, the expression using dot-parentheses (called on Tuesday the 11th) is the equivalent of:

```
obj.Tuesday('11')
```

Specifying Precedence

"Specifying Class Precedence" on page 3-11 provides information on how you can specify the relative precedence of user-define classes.

Controlling Access to Methods

There might 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.

Local and nested functions inside the method files have the same access as the method. Note that 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. In some cases, the subclass method might need to execute some additional code instead of completely replacing the superclass method. To do this, 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:

```
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\n',...
            s.NumShares,s.SharePrice);
        end % disp
    end
end
```

See "The DocStock disp Method" on page 10-9 for more information on this example.

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

Class Constructor Methods

In this section...
"Rules for Constructors" on page 7-14
"Initializing an Instance Within a Constructor" on page 7-15
"Constructing Subclasses" on page 7-17
"Errors During Class Construction" on page 7-18
"Basic Structure of Constructor Methods" on page 7-18

Rules for Constructors

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 always return an initialized object.

- The constructor has the same name as the class.

- The first output argument from a constructor is always the object constructed.

- If the class being created is a subclass, the MATLAB® class system 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, you must call them from the subclass constructor explicitly.

- A class does not need to define a constructor method unless it is a subclass of a superclass whose constructor requires arguments. In this case, you must explicitly call the superclass constructor with the required arguments. See "Constructing Subclasses" on page 7-17

- If a class does not define a constructor, the MATLAB class system supplies a constructor that takes no arguments and returns a scalar object whose properties are initialized to empty or the values specified as defaults in the property definitions. The constructor supplied by MATLAB also calls all superclass constructors with no arguments.

- If you create a class constructor, you should implement class constructors so that they can be called with no input arguments, in addition to whatever

arguments are normally required See "Supporting the No Input Argument Case" on page 7-16 and "Basic Structure of Constructor Methods" on page 7-18.

• Constructors must always return objects of their own class. A superclass constructor cannot return an object of a subclass.

• Calls to superclass constructors cannot be conditonal This means superclass construction calls cannot be placed in loops, conditions, switches, try/catch, or nested functions. See "No Conditional Calls to Superclass Constructors" on page 7-17 for more information.

• You can restrict access to constructors using method attributes, as with any method.

Initializing an Instance Within a Constructor

Constructor functions must return an initialized object as the first output argument. The first 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.

```
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. See "Property Definition Block" on page 6-4 for a description of this syntax.

For example, the following code calls the class method CalculateValue to assign the value of the property Value.

```
function obj = myClass(a,b,c)
   obj.Value = obj.CalculateValue(a,b);
```

```
      ...
   end
```

Referencing the Object in a Constructor

When initializing the object by assigning values to properties, and so on, you must 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
```

Supporting the No Input Argument Case

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;`)

If there are no input arguments, the constructor creates an object using only default properties values. A good practice is to always add a check for zero arguments to the class constructor to prevent an error if either of the two cases above occur:

```matlab
function obj = myClass(a,b,c)
   if  nargin > 0
      obj.A = a;
      obj.B = b;
      obj.C = c;
         ...
```

```
        end
    end
```

See "Basic Structure of Constructor Methods" on page 7-18 for ways to handle superclass constructors.

Constructing Subclasses

Subclass constructor functions must explicitly call superclass constructors if the superclass constructors require input arguments. The subclass constructor must specify these arguments in the call to the superclass constructor using the following syntax:

```
function obj = myClass(arg)
    obj = obj@SuperClassName(ArgumentList)
        ...
end
```

Any uncalled constructors are called in the left-to-right order that they are specified, but no arguments are passed to these functions.

No Conditional Calls to Superclass Constructors

Calls to superclass constructors must be unconditional and you can have only one call for any given superclass. You must initialize the superclass portion of the object by calling the superclass constructors before you can use the object (for example., to assign property values or call class methods).

In cases where you need to call superclass constructors with different arguments, depending on some condition, you can conditionally build a cell array of arguments and provide one call to the constructor.

For example, in the following example the superclass `shape` constructor is called using some default values when the `cube` constructor has been called with no arguments:

```
classdef cube < shape
    properties
        SideLength = 0;
        Color = [0 0 0];
    end
```

```matlab
methods
    function obj = cube(length,color,upvector,viewangle)
        if nargin == 0 % Provide default values if called with no arguments
            args{1} = [0 0 1];
            args{2} = 10;
        else
            args{1} = upvector;
            args{2} = viewangle;
        end
        obj = obj@shape(args{:});
        if nargin > 0 % Use value if provided
            obj.SideLength = length;
            obj.Color = color;
        end
    ...
    end
    ...
end
```

More on Subclasses

See "Creating Subclasses — Syntax and Techniques" on page 5-6 for information on creating subclasses.

Errors During Class Construction

If an error occurs during the construction of a handle class, the MATLAB class system calls the class destructor on the object along with the destructors for any objects contained in properties and any initialized base classes.

See "Handle Class Delete Methods" on page 4-13 for information on how objects are destroyed.

Basic Structure of Constructor Methods

It is important to consider the state of the object under construction when writing your constructor method. Constructor methods can be structured into three basic sections:

- Pre-initialization — Compute arguments for superclass constructors.

- 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:

```matlab
classdef myClass < baseClass1
   methods
      function obj = myClass(a,b,c)

% Pre Initialization
% Any code not using first output argument (obj)
         if nargin == 0
            args{1} = x;args{2} = y;
            a = z;
         end
         compvalue = myClass.staticMethod(a);

% Object Initialization
% Call base-class constructor before accessing object
         obj = obj@baseClass1(args{:});

% Post Initialization
% Any code, including access to object
         obj.classMethod(...);
         obj.Property = compvalue;
         ...
      end
      ...
   end
   ...
end
```

See for information on creating object arrays in subclass constructors.

Creating Object Arrays

Building Arrays in the Constructor

A constructor function can return an object array by making the necessary assignments when initializing the output argument. For example, the following function creates an m-by-n object array and initializes the Value property of each object to the corresponding element in the input argument f:

```
function obj = obj_class(f)
   m = size(f,1);
   n = size(f,2);
   for i = 1:m
      for j = 1:n
         obj(i,j).Value = f(i,j);
      end
   end
end
```

Initial Values of Object Arrays

When creating an object array, all property values are assigned whatever values are specified in the property definitions, or default to the value of empty double (i.e., [ ]).

Creating Empty Arrays

All classes have a static method named empty that creates an empty array belonging to the class. Empty arrays have no instance data. This method enables you to specify the dimensions of the output array, but at least one of the dimensions must be 0. For example:

```
emptyArray = myClass.empty(5,0);
```

creates a 5–by-0 empty array of `myClass` objects. Calling `empty` with no arguments returns a 0–by-0 empty array.

Static Methods

Why Define Static Methods

Static methods are associated with a class, but not with specific instances of that class. These methods do not perform operations on individual objects of a class and, therefore, do not require an instance of the class as an input argument, like ordinary methods.

Static methods are useful when you do not want to first 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 needed to create class instances.

Suppose 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 a Static Method

To define a method as static, set the methods block Static attribute to true. For example:

```
classdef myClass
    ...
    methods(Static)
        function p = pi(tol)
            [n d] = rat(pi,tol);
            p = n/d;
        end
    end
end
```

"Example — Using Events to Update Graphs" on page 8-16 provides an example that uses a static method to create a set of objects representing graphs.

Calling Static Methods

You invoke a static method using the name of the class followed by dot, and then the name of the method:

```
classname.staticMethodName(args,...)
```

Calling the pi method discussed above would require this statement:

```
value = myClass.pi(.OO1);
```

createViews static method provides an example of a static method.

Inheriting Static Methods

Subclasses can redefine static methods unless the method's Sealed attribute is also set to true in the superclass.

Overloading MATLAB® Functions

Overloading Existing MATLAB® Functions

Class methods can provide implementations of MATLAB® functions that operate only on instances of the class. This is possible because MATLAB software can always identify which class an object belongs to. The dominant argument is used to determine which version of a function to call. If the argument is an object, then MATLAB calls the method defined by the object's class, if there is one.

In cases where a class defines a function with the same name as a global function, the class's implementation of the function is said to *overload* the original global implementation.

Using MATLAB® Functions in Conversion Methods

You might want to overload a number of MATLAB functions to work with an object of your class. Often, a simple solution to providing a full set of MATLAB functionality for a class involves creating methods that convert the data contained in your object to a type that existing MATLAB functions can use.

For example, suppose you define a class to represent polynomials that can have only `single` precision coefficients. You want a `roots` method to work on objects of your new class, but want to use the existing MATLAB `roots` function, which accepts a row vector of doubles that are the coefficients of a polynomial, ordered in descending powers.

The following method accesses a class property, `coefficients`, that contains the polynomial's coefficients, converts them to type `double`, and then passes the vector of doubles to the built–in version of the `roots` function.

```
methods
   function rts = roots(polyobject)
      % Extract data for MATLAB version of roots function
```

```
        coef = double(polyobject.coefficients);
        rts = roots(coef);
    end
end
```

"Overloading MATLAB® Functions for the DocPolynom Class" on page 9-14 provides examples.

"Methods for MATLAB® Classes" on page 7-27 provides a discussion of methods that you might typically implement for MATLAB classes.

Implementing MATLAB® Operators

Classes designed to implement new MATLAB data types typically overload certain operators, such as addition, indexed assignment, and so on.

For example, the addition + (plus) function 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:

```
p1 + p2
```

"Implementing Operators for Your Class" on page 7-31 provides information on methods to overload.

"Defining Arithmetic Operators for DocPolynom" on page 9-11 provides examples.

Rules for Naming to Avoid Conflicts

The names of methods, properties, and events are scoped to the class. Therefore, you should 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 subfunction 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.

Methods for MATLAB® Classes

In this section...
"Supporting Typical MATLAB® Behavior" on page 7-27
"Caution When Overloading MATLAB® Functions" on page 7-27
"Displaying Objects in the Command Window" on page 7-28
"Converting to Another Class" on page 7-29

Supporting Typical MATLAB® Behavior

There are some standard methods that enable user-defined objects to behave like built-in MATLAB® classes. The following table lists some of the methods you might implement for MATLAB classes.

Class Method	Description
disp	Called when you enter disp(obj) on the command line.
display	Called when statements are not terminated by semicolons. disp is often used to implement display methods.
converters like double and char	Convert an object to a MATLAB built-in class
subsref and subsasgn	Enables you to create nonstandard indexed reference and indexed assignment
end	Supports end syntax in indexing expressions using an object; e.g., A(1:end)
subsindex	Supports using an object in indexing expressions

Caution When Overloading MATLAB® Functions

Many built-in MATLAB functions depend on the behavior of other built-in functions, like size. Therefore you must be careful to ensure that what

is returned by an overloaded version of `size` is a correct and accurate representation of the size of an object.

You might also need to overload `numel` to restore proper behavior when you have overloaded `size` to do something other than returning the true size of an object.

Displaying Objects in the Command Window

MATLAB calls a method named `disp` 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` and calls the MATLAB `disp` method for class `double`. This method simply displays the value of `a` in the command line.

```
a = 5
a =
    5
```

All MATLAB classes inherit default `disp` and `display` methods.

You can define a `disp` method for your classes if you want MATLAB to display more useful information on the command line when referring to objects from your class. In many classes, `disp` can simply print the variable name, and then use the `char` converter method to print the contents of the variable. You need to define the `char` method to convert the object's data to a character string because MATLAB displays output as character strings.

You might also use `sprintf` or other data formatting function to implement the `disp` method for your class.

Examples of disp Methods

See the following sections for examples of `disp` methods:

"Displaying TensileData Objects" on page 2-49

"The DocPolynom disp Method" on page 9-9

"The DocAsset Display Method" on page 10-6

Relationship Between disp and display

MATLAB invokes the built-in `display` function when:

- Code executes a MATLAB statement that is not terminated with a semicolon and that returns a value.

- Code explicitly invokes the `display` function.

MATLAB invokes the built-in `disp` function in the following cases:

- The built-in `display` function calls `disp`.

- Code explicitly invokes `disp`.

The built-in `display` function prints the name of the variable being displayed, if an assignment is made, or otherwise uses `ans` as the variable name. `display` then calls `disp` to handle the actual display of the values.

If the variable being displayed is an object of a class that overloads `disp` as a class method, then MATLAB always calls the overloaded method. You should overload `disp` or `disp` and `display` to customize the display of objects. Overloading only `display` is not adequate to property implement a custom display for your class.

Converting to Another Class

It is sometimes useful to 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`.

Converters enable you to:

- Use methods defined for another class

- Ensure that expressions involving objects of mixed class types execute properly

For example, suppose you have defined a `polynomial` class, but you have not overloaded any of the MATLAB functions that operate on the coefficients

of polynomials, which are doubles. If you create a `double` method for the `polynomial` class, you can use it to call other functions that require inputs of type `double`.

You can use the `double` method to call other functions:

```
roots(double(p))
```

where `p` is a polynomial object, `double` is a method of the `polynomial` class, and `roots` is a standard MATLAB function whose input arguments are the coefficients of a polynomial.

Examples of Converter Methods

See the following sections for examples of converter methods:

- "The DocPolynom to Double Converter" on page 9-6
- "The DocPolynom to Character Converter" on page 9-7

Implementing Operators for Your Class

Overloading Operators

You can implement MATLAB® operators (+, *, >, etc.) to work with objects of your class. Do this by defining the relevant functions as class methods.

Each built-in MATLAB operator has an associated function (e.g., the + operator has an associated `plus.m` function). You can overload any operator by creating a class method with the appropriate name.

Overloading enables operators to handle different types and numbers of input arguments and perform whatever operation is appropriate for the highest precedence object:

Object Precedence

User-defined classes have a higher precedence than built-in classes. For example, if `q` is an object of class `double` and `p` is a user-defined class, `polynomial`, the expression

```
q + p
```

generates a call to the `plus` method in the `@polynomial` directory, if it exists. Whether this method can add objects of class `double` and class `polynomial` depends on how you implement it.

When `p` and `q` are objects of different classes, MATLAB applies the rules of precedence to determine which method to use.

"Object Precedence in Expressions Using Operators" on page 7-34 provides information on how MATLAB determines which overloaded method to call.

Examples of Overloaded Operators

"Defining Arithmetic Operators for DocPolynom" on page 9-11 provides examples of overloaded operators.

MATLAB® Operators and Associated Functions

The following table lists the function names for common MATLAB operators.

Operation	Method to Define	Description
a + b	plus(a,b)	Binary addition
a - b	minus(a,b)	Binary subtraction
-a	uminus(a)	Unary minus
+a	uplus(a)	Unary plus
a.*b	times(a,b)	Element-wise multiplication
a*b	mtimes(a,b)	Matrix multiplication
a./b	rdivide(a,b)	Right element-wise division
a.\b	ldivide(a,b)	Left element-wise division
a/b	mrdivide(a,b)	Matrix right division
a\b	mldivide(a,b)	Matrix left division
a.^b	power(a,b)	Element-wise power
a^b	mpower(a,b)	Matrix power
a < b	lt(a,b)	Less than
a > b	gt(a,b)	Greater than
a <= b	le(a,b)	Less than or equal to
a >= b	ge(a,b)	Greater than or equal to
a ~= b	ne(a,b)	Not equal to
a == b	eq(a,b)	Equality

Operation	Method to Define	Description
a & b	and(a,b)	Logical AND
a | b	or(a,b)	Logical OR
~a	not(a)	Logical NOT
a:d:b	colon(a,d,b)	Colon operator
a:b	colon(a,b)	
a'	ctranspose(a)	Complex conjugate transpose
a.'	transpose(a)	Matrix transpose
command window output	display(a)	Display method
[a b]	horzcat(a,b,...)	Horizontal concatenation
[a; b]	vertcat(a,b,...)	Vertical concatenation
a(s1,s2,...sn)	subsref(a,s)	Subscripted reference
a(s1,...,sn) = b	subsasgn(a,s,b)	Subscripted assignment
b(a)	subsindex(a)	Subscript index

Object Precedence in Expressions Using Operators

Establishing an object precedence enables the MATLAB® runtime to determine which of possibly many versions of an operator or function to call in a given situation.

For example, consider the expression

```
objectA + objectB
```

Ordinarily, objects have equal precedence and the method associated with the left-most object is called. However, there are two exceptions:

- User-defined classes have precedence over MATLAB built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using the InferiorClasses attribute.

In "Example — A Polynomial Class" on page 9-2, the polynom class defines a plus method that enables the addition of DocPolynom objects. Given the object p:

```
p = DocPolynom([1 0 -2 -5])
p =
    x^3-2*x-5
```

the expression:

```
1 + p
ans =
    x^3-2*x-4
```

calls the DocPolynom plus method (which converts the double, 1, to a DocPolynom object and then implements the addition of two polynomials). The user-defined 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 using 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 "Rules for Naming to Avoid Conflicts" on page 7-25 for related information.

Class Methods for Graphics Callbacks

Callback Arguments

You can use class methods as callbacks for Handle Graphics® objects by specifying the callback as an anonymous function. Anonymous functions enable you to pass the arguments required by methods (i.e., the first argument is a class object) and graphics object callbacks (i.e., the event source and the event data), as well as any other arguments you want to pass to the function.

The following links provide general information on graphics object callbacks and anonymous functions.

Background Information

- Function Handle Callbacks — Information on graphics object callbacks

- Anonymous Functions — Information about using anonymous functions

General Syntax for Callbacks

The basic syntax for a function handle that you assign to the graphic object's Callback property includes the object as the first argument:

```
@(src,event)method_name(object,src,event,additional_arg,...)
```

You must define the callback method with the following signature:

```
method_name(object,src,event)
```

Object Scope and Anonymous Functions

Anonymous functions take a snapshot of the argument values when you define the function handle. You must, therefore, consider this scoping when assigning the `Callback` property. The following two sections provide examples.

Using Value Classes

Consider the following snippet of a value class definition:

```
classdef SeaLevelAdjuster
   properties
      Slider
   end
   methods
      function seal = SeaLevelAdjuster
          ...
        seal.Slider = uicontrol('Style','slider');
        set(seal.Slider,'Callback',@(src,event)slider_cb(seal,src,event)
       end
   end
end
```

This class assigns the `Callback` property in a separate set statement so that the value object's (`seal`) `Slider` property has been defined when you create the function handle. Otherwise, Handle Graphics freezes `seal` before the uicontrol's handle is assigned to the `Slider` property.

Using Handle Classes

The difference in behavior between a handle object and a value object is important in this case. If you defined the class as a handle class, the object is a reference to the underlying data. Therefore, when the MATLAB® runtime resolves the function handle, the contents of the object reflects assignments made after the function handle is defined:

```
classdef SeaLevelAdjuster < handle
   ...
   properties
      Slider
   end
   methods
```

```
            function seal = SeaLevelAdjuster
                ...
                seal.Slider = uicontrol('Style','slider',...
                'Callback',@(src,event)slider_cb(seal,src,event));
            end
        end
    end
```

Example — Class Method as a Slider Callback

This example defines a slider that varies the color limits of an indexed image to give the illusion of varying the sea level.

Displaying the Class Files

Open the SeaLevelAdjuster class definition file in the MATLAB editor.

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

Class Properties

The class defines properties to store graphics object handles and the calculated color limits:

```
classdef SeaLevelAdjuster < handle
    properties
        Figure = [];
        Axes = [];
        Image = [];
        CLimit = [];
        Slider = [];
    end
end
```

Class Constructor

The class constructor creates the graphics objects and assigns the slider callback (last line in code snippet):

```matlab
methods
    function seal = SeaLevelAdjuster(x,map)
    seal.Figure = figure('Colormap',map,'Position',[100 100 560 580]);
    seal.Axes = axes('DataAspectRatio',[1 1 1],...
        'XLimMode','manual','YLimMode','manual',...
        'DrawMode','fast',...
        'Parent',seal.Figure);
    seal.Image = image(x,'CDataMapping','scaled','Parent',seal.Axes);
    seal.CLimit = get(seal.Axes,'CLim');
    seal.Slider = uicontrol('Style','slider',...
        'Parent',seal.Figure,...
        'Max',seal.CLimit(2),...
        'Min',seal.CLimit(1)-1,...
        'Value',seal.CLimit(1),...
        'Position',[520 100 20 400],...
        'SliderStep',[.005 .002],...
        'Callback',@(src,event)slider_cb(seal,src,event));
    end % SeaLevelAdjuster
end % methods
```

The callback function for the slider is defined to accept the three required arguments — a class instance, the handle of the event source, and the event data:

```matlab
methods
    function slider_cb(seal,src,event)
        min_val = get(seal.Slider,'Value');
        max_val = max(max(get(seal.Image,'CData')));
        set(seal.Axes,'CLim',[min_val max_val])
        drawnow
    end % slider_cb
end % methods
```

Using the SeaLevelAdjuster Class

The class is designed to be used with the cape image that is included with the MATLAB product. To obtain the image data, use the load command:

```matlab
load cape
```

After loading the data, create a `SeaLevelAdjuster` object for the image:

```
seal = SeaLevelAdjuster(X,map)
```

Move the slider to change the apparent sea level and visualize what would happen to Cape Cod if the sea level were to rise.

8
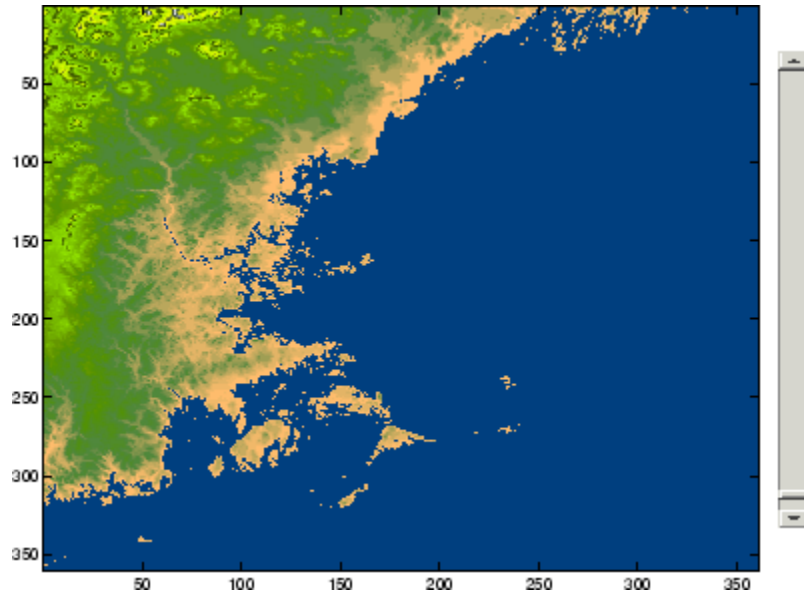
Events — Sending and Responding to Messages

Events and Listeners — Concepts

In this section...
"The Event Model" on page 8-2
"Default Event Data" on page 8-4
"Events Only in Handle Classes" on page 8-4
"Property-Set and Query Events" on page 8-4
"Listeners" on page 8-6

The Event Model

Events represent changes or actions that occur within class instances. For example,

- Modification of class data
- Execution of a method
- Querying or setting a property value
- Destruction of an object

Basically, any activity that can be detected 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 need to respond to the events. The event model works this way:

- A handle class declares a name used to represent an event. "Naming Events" on page 8-8
- After an instance of the event-declaring class is created, you can attach listener objects to it. "Creating Listeners" on page 8-9
- A class method call broadcasts a notice of the event to listeners and the class user is responsible for determining when to declare that the event has occurred. "Triggering Events" on page 8-8

- Listeners execute a callback function when notified that the event has occurred. "Defining Listener Callback Functions" on page 8-12

- Listeners can be bound to the lifecycle of the object that defines the event or limited to the existence and scope of a listener object. "Creating Listeners" on page 8-9

The following diagram illustrates the event model.

1. The **withdraw** method is called.

```
if AccountBalance <= 0
   notify(obj,'InsufficientFunds');
end
```

BankAccount

Properties
  AccountNumber
  AccountBalance

Methods
  deposit
  withdraw

Events
  InsufficientFunds
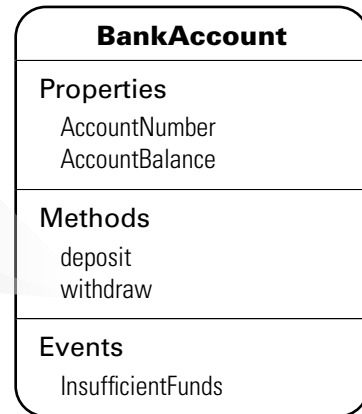
2. The **notify** method triggers an event, and a message is broadcast.

InsufficientFunds

InsufficientFunds

3. Listeners awaiting message execute their callbacks.

   (The broadcasting object does not necessarily know who is listening.)

Listener1

Properties
  EventName = 'InsufficientFunds'
  FunctionHandle = @Callback1

Listener2

Properties
  EventName = 'InsufficientFunds'
  FunctionHandle = @Callback2
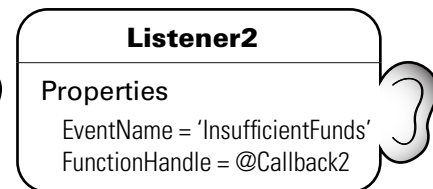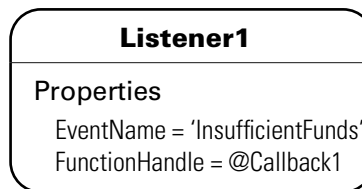
Default Event Data

Events provide information to listener callbacks by passing an event data argument to the callback function. By default, the MATLAB runtime 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

Customizing 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.

"Defining Event-Specific Data" on page 8-9 provides an example showing how to customize this data.

Events Only in Handle Classes

You can define events only in handle classes. This is 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, but this is not generally useful since it cannot access the current state of the object that triggered the event or effect any changes in that object.

"Comparing Handle and Value Classes" on page 4-2 provides general information on handle classes.

"Defining Events and Listeners — Syntax and Techniques" on page 8-8 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 triggered by the property objects and, therefore, are not defined in the class's event 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

This object has one property, `AffectedObject`, that contains the object whose property is the source for this event. (i.e., `AffectedObject` is the object whose property was either accessed or modified.)

You can define your own property-change event data by subclassing the `event.EventData` class. Note that the `event.PropertyEvent` class is a sealed subclass of `event.EventData`.

See "Listening for Changes to Property Values" on page 8-14 for a description of the process for creating property listeners.

See "Implementing the PostSet Property Event and Listener" on page 8-27 for an example.

See "Controlling Property Access" on page 6-10 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 with an enabled listener receives event notification

- `Enabled` — Callback function executes only when `Enabled` is `true`. See "Enabling and Disabling the Listeners" on page 8-30 for an example.

- `Recursive` — Can listener cause same event that triggered the execution of the callback

  `Recursive` is `true` by default. It is possible create a situation where infinite recursion reaches the recursion limit and eventually triggers an error. If you set `Recursive` to `false`, the listener cannot execute recursively if the callback triggers its own event.

"Creating Listeners" on page 8-9 provides more specific information.

Listener Order of Execution

The order in which listeners execute after the firing of an event is undefined. However, all listener callbacks execute synchronously with the event firing.

Event Attributes

Table of 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 key word. For example, all the events defined in the following `events` block have private `ListenAccess` and `NotifyAccess` attributes.

```
events (ListenAccess = 'private', NotifyAccess = 'private')
   anEvent
   anotherEvent
end
```

To define other events in the same class definition that have different attribute settings, create another `events` block.

Attribute Name	Class	Description
Hidden	logical Default=false	If `true`, event does not appear in list of events returned by `events` function (or other event listing functions or viewers).
ListenAccess	char Default=public	Determines where you can create listeners for the event. • `public` — Unrestricted access • `protected` — Access from methods in class or derived classes • `private` — Access by class methods only (not from derived classes)
NotifyAccess	char Default=public	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)

Defining Events and Listeners — Syntax and Techniques

In this section...

Naming Events

You define an event by declaring an event name inside an events block, typically in the class that generates the event. For example, the following class creates an event called `ToggledState`, which the class triggers whenever the toggle button's state changes.

```
classdef ToggleButton < handle
    properties
        State = false
    end

    events
        ToggledState
    end
end
```

Triggering Events

At this point, the `ToggleButton` class has simply defined a name that will be associated with the toggle button state changes—toggling on and toggling off. However, the actual firing of the events must be controlled by a class method. To accomplish this, the `ToggleButton` class adds a method to trigger the events:

```
classdef ToggleButton < handle
    ...
    methods
        function OnStateChange(obj,newState)
```

```matlab
                if newState ~= obj.State
                    notify(obj,'ToggledState'); % Broadcast notice of event
                    obj.State = newState;
                end
            end
        end
    end
```

The `OnStateChange` method calls `notify` to trigger the event, using the handle of the `ToggleButton` object that owns the event and the string name of the event.

Defining Event-Specific Data

Suppose that you want to pass to the listener callback the state of the toggle button as a result of the event. You can add more data to the default event data by subclassing the `event.EventData` class and adding a property to contain this information. You then can pass this object to the `notify` method.

```matlab
classdef ToggleEventData < event.EventData
    properties
        NewState
    end

    methods
        function data = ToggleEventData(newState)
            data.NewState = newState;
        end
    end
end
```

The call to `notify` uses the `ToggleEventData` constructor to create the necessary argument.

```matlab
notify(obj,'ToggledState',ToggleEventData(newState));
```

Creating Listeners

When you call the `notify` method, the MATLAB® runtime sends the event data to all registered listener callbacks. There are two ways to create a listener:

- Use the `addlistener` method, which binds the listener to the lifecycle of the object(s) that will generate the event. The listener object persists until the object it is attached to is destroyed.

- Use the `event.listener` class constructor. In this case, the listeners you create are not tied to the lifecycle of the object(s) being listened to. Instead the listener is active so long as the listener object remains in scope and is not deleted.

Attach Listener to Event Generator — Using addlistener

The following code defines a listener for the `ToggleState` event:

```
lh = addlistener(obj,'ToggleState',@CallbackFunction)
```

The arguments are:

- `obj` — The object that generates the event
- `ToggleState` — The event name passed as a string
- `@CallbackFunction` — A function handle to the callback function

The listener callback function must accept at least two arguments, which are automatically passed by the MATLAB runtime to the callback. The arguments are:

- The source of the event (that is, `obj` in the call to `addlistener`)
- An `event.EventData` object, or a subclass of `event.EventData`, such as the `ToggleEventData` object described earlier "Defining Event-Specific Data" on page 8-9.

The callback function must be defined to accept these two arguments:

```
function CallbackFunction(src,evnt)
    ...
end
```

In cases where the event data (`evnt`) object is user defined, it must be constructed and passed as an argument to the `notify` method. For example, the following statement constructs a `ToggleEventData` object and passes it to `notify` as the third argument:

```
notify(obj,'ToggledState',ToggleEventData(newState));
```

"Defining Listener Callback Functions" on page 8-12 provides more information on callback syntax.

Limiting Listener Scope — Constructing event.listener Objects Directly

You can create listeners by calling the event.listener class constructor directly. When you call the constructor instead of using addlistener to create a listener, the listener exists only while the listener object you create is in scope (e.g., within the workspace of an executing function). It is not tied to the event-generating object's existence.

The event.listener constructor requires the same arguments as used by addlistener—the event-naming object, the event name, and a function handle to the callback:

```
lh = event.listener(obj,'ToggleState',@CallbackFunction)
```

If you want the listener to persist beyond the normal variable scope, you should use addlistener to create it.

Temporarily Deactivating 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:

```
lh.Enabled = false;
```

To re-enable the listener, set Enabled to true.

"Enabling and Disabling the Listeners" on page 8-30 provides an example.

Permanently Deleting Listeners

Calling delete on a listener object destroys it and permanently removes the listener:

```
delete(lh) % Listener object is removed and destroyed
```

Defining Listener Callback Functions

Callbacks are functions that execute when notification of an event is received by the listener. Typically, you define a method in the class that creates the listener as the callback function. You must pass a function handle that references the method to `addlistener` or the `event.listener` constructor when creating the listener.

`function_handle` provides more information on function handles).

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 (see "Defining Event-Specific Data" on page 8-9 for an example that extends this class).

Adding Arguments to a Callback Function

Ordinary class methods (i.e., not static methods) require a class object as the first argument, so you need to add another argument to the callback function definition. The best way to do this is by using an anonymous function.

"Anonymous Functions" provides general information on anonymous functions

For example, suppose you create a method to use as your callback function and want to reference this method as a function handle, as required by `addlistener` or the `event.listener` constructor. Use the following syntax:

```
hlistener = addlistener(eventSourceObj,'MyEvent',@(src,evnt)listenMyEvent(obj,src,evnt))
```

You then would define the method in a method block as usual:

```
methods
   function listenMyEvent(obj,src,evnt)
      % obj - instance of this class
      % src - object generating event
      % evnt - the event data
      ...
   end
end
```

"Variables Used in the Expression" provides information on variables used in anonymous functions.

Listening for Changes to Property Values

Creating Property Listeners

You can listen to the predeclared property events (named: `PreSet`, `PostSet`, `PreGet`, and `PostGet`) by creating a listener for those named events:

- Set the `SetObservable` property attribute of the property to `true`

- Define a callback function

- Create a property listener by including the name of the property as well as the event in the call to `addlistener` (see "Add a Listener to the Property" on page 8-15)

- Optionally subclass `event.data` to create a specialized event data object to pass to the callback function

The following code segments show how to create a `PostSet` property listener for a property called `Limits`.

Set Property Attributes to Enable Property Events

In the properties block, set the `Limits` property `SetObservable` to `true`:

```matlab
properties (SetObservable = true) % Can define
property listeners
   Limits = [-2*pi 2*pi];
   ...
end
```

Define a Callback Function for the Property Event

Define a callback function, passing the required arguments. For example, assume the `handlePropertyEvents` function is a method of the class whose property is being set. This method requires three arguments:

- The first argument (`obj`) is the class instance containing the property that triggers the `PostSet` event.

- The second argument is the event source (the `Limits` property in this case).

- The third argument is the `event.PropertyEvent` object.

```
function handlePropertyEvents(obj,src,evnt)
   switch src.Name % switch on the property name
      case 'Limits'
         obj.calculateData(obj.Limits) % Call class method passing new value for property
   ...
end
```

The property (`obj.Limits`) has the new value because this is a `PostSet` event. Note that `src` is a `meta.property` object for the `Limits` property.

"Obtaining Information About Classes with Meta-Classes" on page 3-26 provides more information about the `meta.property` class.

Add a Listener to the Property

The `addlistener` 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`:

```
lh =
addlistener(obj,'Limits','PostSet',@(src,evnt)handlePropertyEvents(obj,src,evnt));
```

Example — Using Events to Update Graphs

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 values 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.
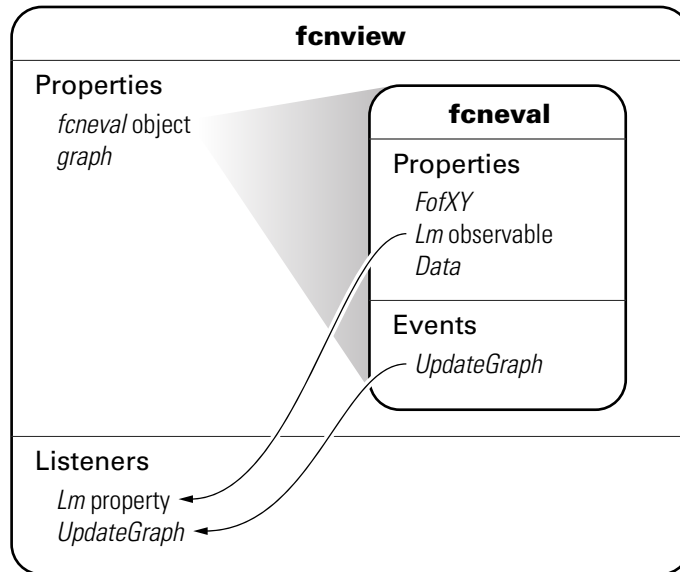
Access Fully Commented Example Code

You can display the code for this example in a popup window that contains detailed comments and links to related sections of the documentation by clicking these links:

`fcneval` class

`fcnview` class

`createViews` static method

You can open all files in your editor by clicking this link:

Open in editor

To use the classes, save the files in directories with the following names:

- @fcneval/fcneval.m
- @fcnview/fcnview.m
- @fcnview/createViews.m

The @-directory's parent directory must be on the MATLAB path.

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 is designed to evaluate a MATLAB expression over a specified range of two variables. It is the source of the data that is graphed as a surface by instances of the `fcnview` class. It is the source of the events used in this example.

Property	Value	Purpose
FofXY	function handle	MATLAB expression (function of two variables).
Lm	two-element vector	Limits over which function is evaluated in both variables. `SetObservable` attribute set to `true` to enable property event listeners.
Data	structure with x, y, and z matrices	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.

Event	When Triggered
UpdateGraph	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.

Method	Purpose
fcneval	Class constructor. Inputs are function handle and two-element vector specifying the limits over which to evaluate the function.
set.FofXY	FofXY property set function. Called whenever property value is set, including during object construction.
set.Lm	Lm property set function. Used to test for valid limits.
get.Data	Data property get function. This method calculates the values for the Data property whenever that data is queried (by class members or externally).
grid	A static method (Static attribute set to true) used in the calculation of the data.

Summary of fcnview Class

Instances 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.

Property	Value	Purpose
FcnObject	fcneval object	This object contains the data that is used to create the function graphs.
HAxes	axes handle	Each instance of a fcnview object stores the handle of the axes containing its subplot.

Property	Value	Purpose
HLUpdateGraph	event.listener object for UpdateGraph event	Setting the event.listener object's Enabled property to true enables the listener; false disables listener.
HLLm	event.listener object for Lm property event	Setting the event.listener object's Enabled property to true enables the listener, false disables listener.
HEnableCm	uimenu handle	Item on context menu used to enable listeners (used to handle checked behavior)
HDisableCm	uimenu handle	Item on context menu used to disable listeners (used to manage checked behavior)
HSurface	surface handle	Used by event callbacks to update surface data.

Method	Purpose
fcnview	Class constructor. Input is fcneval object.
createLisn	Calls addlistener to create listeners for UpdateGraph and Lm property PostSet listeners.
lims	Sets axes limits to current value of fcneval object's Lm property. Used by event handlers.
updateSurfaceData	Updates the surface data without creating a new object. Used by event handlers.
listenUpdateGraph	Callback for UpdateGraph event.
listenLm	Callback for Lm property PostSet event
delete	Delete method for fcnview class.
createViews	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

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 4-11 provides a complete list of methods that are inherited when you subclass the `handle` class.

Methods Inherited from Handle Class	Purpose
addlistener	Register a listener for a specific event and attach listener to event-defining object.
notify	Trigger an event and notify all registered listeners.

Using the fcneval and fcnview Classes

This sections 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 `createViews` 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:

```
feobject = fcneval(@(x,y) x.*exp(-x.^2-y.^2),[-2 2]);
```

Use the `createViews` static method to create the graphs of the function. Note that you must use the class name to call a static function:

```
fcnview.createViews(feobject);
```

The `createView` method generates four views of the function contained in the `fcneval` object.
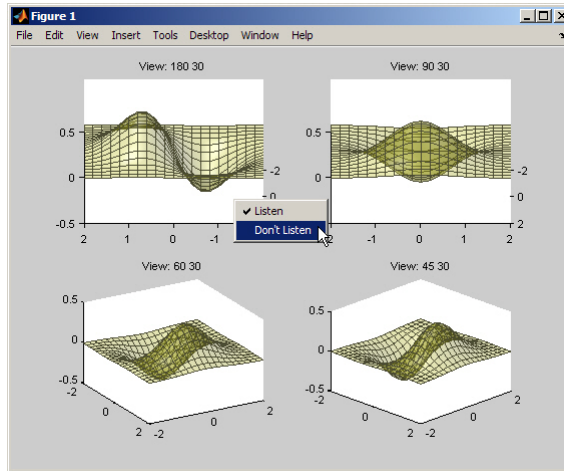
Each subplot defines a context menu that can enable and disable the listeners associated with that graph. For example, if you disable the listeners on subplot 221 (upper left) and change the MATLAB expression contained by the `fcneval` object, only the remaining three subplots update when the `UpdateGraph` event is triggered:

```
feobject.FofXY = @(x,y) x.*exp(-x.^.5-y.^.5);
```
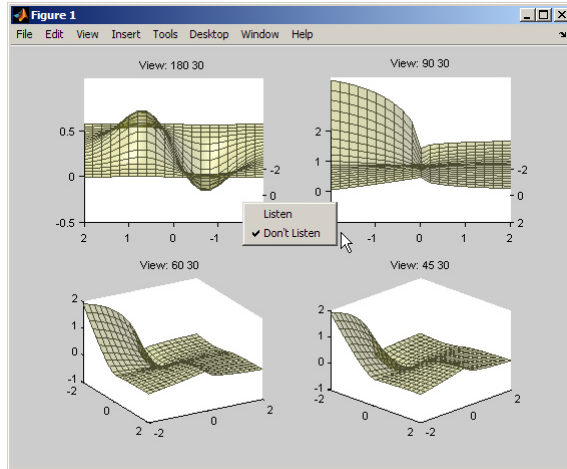
Similarly, if you change the limits by assigning a value to the `feobject.Lm` property, the `feobject` triggers a `PostSet` property event and the listener callbacks update the graph.

```
feobject.Lm = [-8 3];
```
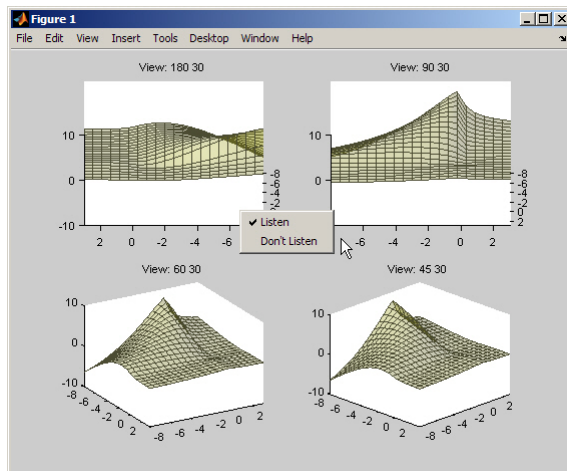

In this figure the listeners are re-enabled via the context menu for subplot 221. Because the listener callback for the property `PostSet` event also updates the surface data, all views are now synchronized

Implementing the 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.

Defining and Firing the 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.

```
obj.FofXY = @(x,y)x^2+y^2
```

2. Setting the property runs a set access method, which, in turn, executes **notify**.

```
function fofxy = set.FofXY(obj,func)
  obj.FofXY = func;
  notify(obj,'UpdateGraph');
end
```

3. The **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.

myfunceval

Properties
 FofXY

Methods
 set.FofXY

Events
 UpdateGraph

UpdateGraph

Listener

Properties
 EventName = 'UpdateGraph'
 FunctionHandle = @listenUpdateGraph

myfuncview

Methods
 listenUpdateGraph
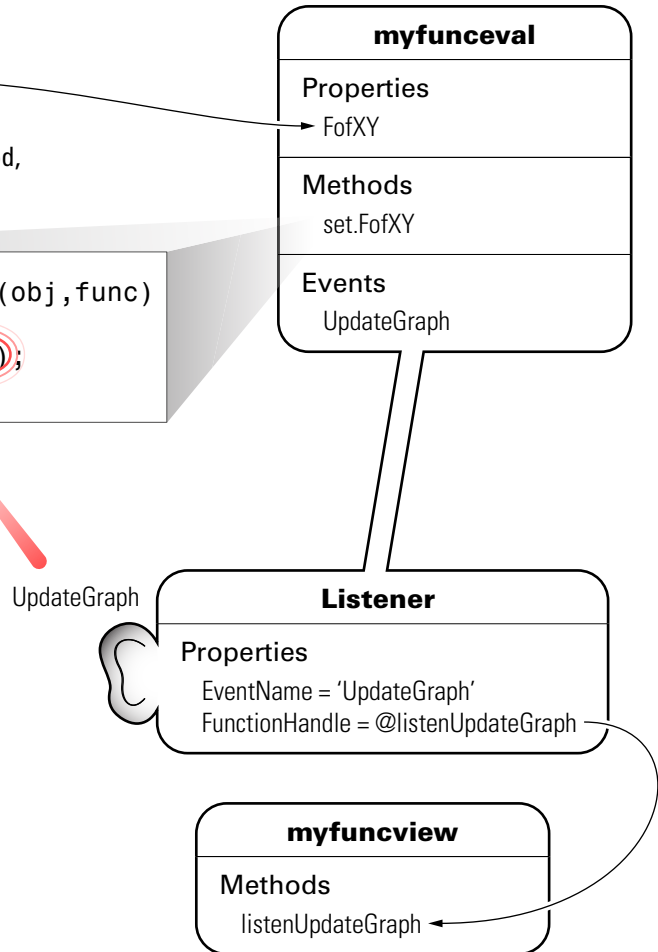
The fcnview class defines a listener for this event. When fcneval triggers the event, the fcnview listener executes a callback function that performs the follow actions:

- Determines if the handle of the surface object stored by the fcnview object is still valid (that is, does the object still exist)

- Updates the surface XData, YData, and ZData by querying the fcneval object's Data property.

The `fcneval` class defines an event name in an `event` block:

```
events
    UpdateGraph
end
```

It is the responsibility of the class to explicitly trigger the event by calling the `notify` method. In this example, `notify` is called from the set function of the property that stores the MATLAB expression for the mathematical function:

```
function fofxy = set.FofXY(obj,func)
    obj.FofXY = func; % Assign property value
    notify(obj,'UpdateGraph'); % Trigger UpdateGraph event
end
```

The class could have implemented a property event for the `FofXY` property and would, therefore, not need to call `notify`. Defining a class event provides more flexibility in controlling when the event is triggered.

Defining the Listener and Callback for the UpdateGraph Event

The `fcnview` class creates a listener for the `UpdateGraph` event using the `addlistener` method:

```
obj.HLUpdateGraph =
addlistener(obj.FcnObject,'UpdateGraph',...
            @(src,evnt)listenUpdateGraph(obj,src,evnt));
% Add obj to argument list
```

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 "Enabling and Disabling the Listeners" on page 8-30).

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 is updated by the callback.

The `listenUpdateGraph` function is defined as follows:

```
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 generally more efficient than creating a new object using the new data:

```
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
```

Implementing the PostSet Property Event and Listener

All properties support the predefined `PostSet` event (See "Property-Set and Query Events" on page 8-4 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.

   `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.

```
                        ┌─────────────────────────┐
                        │       myfunceval         │
                        ├─────────────────────────┤
                        │ Properties (SetObservable)│
                        │   Lm                     │
                        └─────────────────────────┘

            PostSet

                        ┌─────────────────────────┐
                        │        Listener          │
                        ├─────────────────────────┤
                        │ Properties               │
                        │   EventName = 'PostSet'  │
                        │   FunctionHandle = @listenLm│
                        └─────────────────────────┘

                        ┌─────────────────────────┐
                        │       myfuncview         │
                        ├─────────────────────────┤
                        │ Methods                  │
                        │   listenLm               │
                        └─────────────────────────┘
```
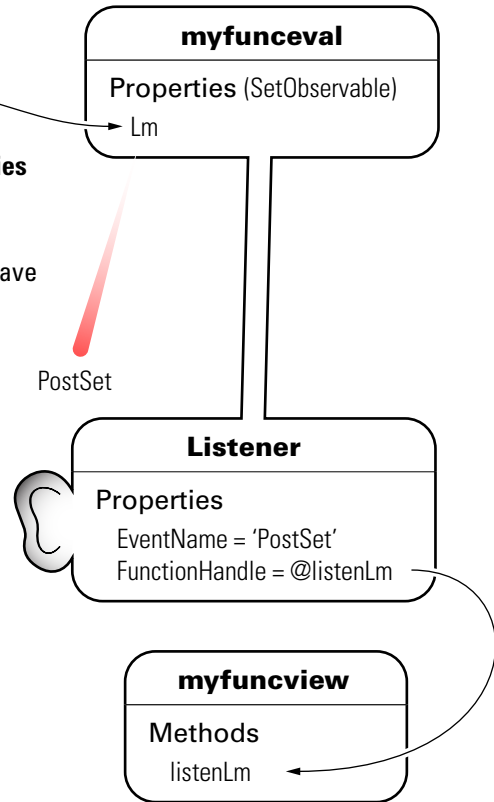
Sequence During the Lm Property Assignment

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, the MATLAB runtime 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.

Enabling the PostSet Property Event

To create a listener for the `PostSet` event, you must set the property's `SetObservable` attribute to `true`:

```
properties (SetObservable = true)
   Lm = [-2*pi 2*pi];  % specifies default value
end
```

The MATLAB runtime automatically triggers the event so it is not necessary to call `notify`.

"Specifying Property Attributes" on page 6-6 provides a list of all property attributes.

Defining the Listener and Callback for the PostSet Event

The `fcnview` class creates a listener for the `PostSet` event using the `addlistener` method:

```
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 "Enabling and Disabling the Listeners" on page 8-30).

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 is updated by the callback.

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:

```
function listenLm(obj,src,evnt)
    if ishandle(obj.HAxes) % If there is an axes
        lims(obj); % Update its limits
        if ishandle(obj.HSurface) % If there is a surface
            obj.updateSurfaceData % Update its data
        end
    end
end
```

Enabling and Disabling the 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.

```
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.

```
function disableLisn(obj,src,evnt)
    obj.HLUpdateGraph.Enabled = false; % Disable listener
    obj.HLLm.Enabled = false; % Disable listener
    set(obj.HEnableCm,'Checked','off') % Unheck Listen
    set(obj.HDisableCm,'Checked','on') % Check Don't Listen
end
```

Implementing a Class for Polynomials

Example — A Polynomial Class

Adding a Polynomial Object to the MATLAB® Language

This example implements a class to represent polynomials in the MATLAB® language. A value class is used because the behavior of a polynomial object within the MATLAB environment should follow the copy semantics of other MATLAB variables. This example also implements for this class, methods to provide enhanced display and indexing, as well as arithmetic operations and graphing.

See "Comparing Handle and Value Classes" on page 4-2 for more information on value classes.

This class overloads a number of MATLAB functions, such as roots, polyval, diff, and plot so that these function can be used with the new polynomial object.

Displaying the Class Files

Open the DocPolynom class definition file in the MATLAB editor.

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

Summary of the DocPolynom Class

The class definition specifies a property for data storage and defines a directory (@DocPolynom) that contains the class definition.

The following table summarizes the properties defined for the DocPolynom class.

DocPolynom Class Properties

Name	Class	Default	Description
coef	double	[ ]	Vector of polynomial coefficients [highest order ... lowest order]

The following table summarizes the methods for the DocPolynom class.

DocPolynom Class Methods

Name	Description
DocPolynom	Class constructor
double	Converts a DocPolynom object to a double (i.e., returns its coefficients in a vector)
char	Creates a formatted display of the DocPolynom object as powers of x and is used by the disp method
disp	Determines how MATLAB displays a DocPolynom objects on the command line
subsref	Enables you to specify a value for the independent variable as a subscript and access the coef property with dot notation
plus	Implements addition of DocPolynom objects

DocPolynom Class Methods (Continued)

Name	Description
minus	Implements subtraction of DocPolynom objects
mtimes	Implements multiplication of DocPolynom objects
roots	Overloads the roots function to work with DocPolynom objects
polyval	Overloads the polyval function to work with DocPolynom objects
diff	Overloads the diff function to work with DocPolynom objects
plot	Overloads the plot function to work with DocPolynom objects

Using the DocPolynom Class

The following examples illustrate basic use of the DocPolynom class.

Create DocPolynom objects to represent the following polynomials. Note that the argument to the constructor function contains the polynomial coefficients $f(x) = x^3 - 2x - 5$ and $f(x) = 2x^4 + 3x^2 + 2x - 7$.

```
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
```

The DocPolynom disp method displays the polynomial in MATLAB syntax.

Find the roots of the polynomial using the overloaded root method.

```
>> roots(p1)

ans =
```

```
   2.0946
  -1.0473 + 1.1359i
  -1.0473 - 1.1359i
```

Add the two polynomials p1 and p2.

The MATLAB runtime 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
```

The sections that follow describe the implementation of the methods illustrated here, as well as other methods and implementation details.

The DocPolynom Constructor Method

The following function is the `DocPolynom` class constructor, which is in the file `@DocPolynom/DocPolynom.m`:

```matlab
function obj = DocPolynom(c)
% Construct a DocPolynom object using the coefficients supplied
    if isa(c,'DocPolynom')
        obj.coef = c.coef;
    else
        obj.coef = c(:).';
    end
end % DocPolynom
```

Constructor Calling Syntax

You can call the `DocPolynom` constructor method 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 situation.

- 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.

  Since the `coef` property definition restricts the values to be doubles, you do not have to check the type of the input argument in the constructor function.

An example use of the `DocPolynom` constructor is the statement:

```
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 how class methods display the equivalent polynomial using MATLAB language syntax. The `DocPolynom` class implements this display using the `disp` and `char` class methods.

Converting DocPolynom Objects to Other Types

The `DocPolynom` class defines two methods to convert `DocPolynom` objects to other classes:

- `double` — Converts to standard MATLAB numeric type so you can perform mathematical operations on the coefficients
- `char` — Converts to string; used to format output for display in the command window

The DocPolynom to Double Converter

The double converter method for the `DocPolynom` class simply returns the coefficient vector, which is a double by definition:

```
function c = double(obj)
% DocPolynom/Double Converter
   c = obj.coef;
end % double
```

For the `DocPolynom` object p:

```
p = DocPolynom([1 0 -2 -5])
```

the statement:

```
c = double(p)
```

returns:

```
c=
    1     0    -2    -5
```

which is of class double:

```
class(c)
ans =
    double
```

The DocPolynom to Character Converter

The char method produces a character string that represents the polynomial displayed as powers of an independent variable, x. Therefore, after you have specified a value for x, the string returned is a syntactically correct MATLAB expression, which you can evaluate.

The disp method uses char to format the DocPolynom object for display.

This function is the char method, which is in the file @DocPolynom/DocPolynom.m.

```matlab
function s = char(p)
% Char(p) is the string representation of p.coef
if all(p.c == 0)
    s = '0';
else
    d = length(p.c) - 1;
    s = [];
    for a = p.c;
        if a ~= 0;
            if ~isempty(s)
                if a > 0
                    s = [s  ' + ' ];
```

```
            else
                s = [s ' - ' ];
                a = -a;
            end
        end
        if a ~= 1 | d == 0
            s = [s num2str(a)];
            if d > 0
                s = [s '*'];
            end
        end
        if d >= 2
            s = [s 'x^' int2str(d)];
        elseif d == 1
            s = [s 'x'];
        end
    end
    d = d - 1;
end
end % char
```

Evaluating the Output

If you create the `DocPolynom` object p:

```
p = DocPolynom([1 0 -2 -5]);
```

and then call the `char` method on p:

```
char(p)
```

the result is:

```
ans =
    x^3 - 2*x - 5
```

The value returned by `char` is a string that you can pass to `eval` after you have defined a scalar value for x. For example:

```
x = 3;
```

```
eval(char(p))
ans =
    16
```

"The DocPolynom subsref Method" on page 9-10 describes a better way to evaluate the polynomial.

The DocPolynom disp Method

The MATLAB disp function enables you to display a character string or the contents of a variable. However, since the disp function has no knowledge of DocPolynom objects, you need to provide an overloaded version in the class definition to handle the case where MATLAB needs to display a DocPolynom object.

This method relies on the char method to produce a string representation of the polynomial, which is then displayed on the screen. The overloaded disp method produces output that is the same as MATLAB output for a built-in class:

```
function disp(obj)
   % disp function for objects
   disp(['     ' char(obj)])
end % disp
```

When MATLAB® Calls the disp Method

The statement:

```
p = DocPolynom([1 0 -2 -5])
```

creates a DocPolynom object. Since the statement is not terminated with a semicolon, the resulting output is displayed on the command line:

```
p =
    x^3 - 2*x - 5
```

The DocPolynom subsref Method

Normally, subscripted assignment is automatically defined by MATLAB. However, in this particular case, the design of the DocPolynom class specifies that a subscripted reference to a DocPolynom object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, given the following polynomial:

$$f(x) = x^3 - 2x - 5$$

evaluates *f(x)*, where *x* is the value of the subscript.

For example, for a DocPolynom object p:

```
p = DocPolynom([1 0 -2 -5])
p =
    x^3 - 2*x - 5
```

the following subscripted expression evaluates the value of the polynomial at x = 3 and x = 4 and returns a vector of resulting values:

```
p([3 4])
ans =
    16    51
```

subsref Implementation Details

The DocPolynom subsref method provides two features:

- The ability to pass a value for the independent variable as a subscripted reference (i.e., p(3) evaluates the polynomial at x = 3)

- Dot notation for accessing the coef property:

```
function b = subsref(a,s)
% Implement a special subscripted assignment
   switch s.type
   case '()'
      ind = s.subs{:};
      for k = 1:length(ind)
         b(k) = eval(strrep(char(a),'x',num2str(ind(k))));
```

```
            end
        case '.'
            switch s.subs
            case 'coef'
                b = a.coef;
            otherwise
                error(['''' s.subs '''' ' is not a property'])
            end
        otherwise
            error('Specify value for x as obj(x)')
        end
    end % subsref
```

For subscripted references using parentheses:

- The DocPolynom class char method produces an expression that can be evaluated.

- The strrep function swaps the passed-in value (the argument a) for the character x.

- The eval function then evaluates the expression and returns the value in the output argument.

For references using dot notation, subsref simply returns the coefficients or an error, if the property name is incorrect.

See the subsref function reference page for more information on implementing subsref methods.

Defining Arithmetic Operators for DocPolynom

Several arithmetic operations are meaningful on polynomials and should be implemented for the DocPolynom class. See "Implementing Operators for Your Class" on page 7-31 for information on overloading other operations that could be useful with this class, such as division, horizontal concatenation, etc.

This section shows how to implement the following methods:

Method and Syntax	Operator Implemented
`plus(a,b)`	Binary addition
`minus(a,b)`	Binary subtraction
`mtimes(a,b)`	Matrix multiplication

When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, 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.

Defining the + Operator

If either `p` or `q` is a `DocPolynom` object, the expression

```
p + q
```

generates a call to a function `@DocPolynom/plus`, unless the other object is of a class of higher precedence. "Object Precedence in Expressions Using Operators" on page 7-34 provides more information.

The following function redefines the `plus` (+) operator for the `DocPolynom` class:

```
function r = plus(obj1,obj2)
% Plus Implement obj1 + obj2 for DocPolynom
    obj1 = DocPolynom(obj1);
    obj2 = DocPolynom(obj2);
    k = length(obj2.coef) - length(obj1.coef);
    r = DocPolynom([zeros(1,k) obj1.coef]+[zeros(1,-k) obj2.coef]);
end % plus
```

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 result.

Defining the - Operator

You can implement the `minus` operator (`-`) using the same approach as the `plus` (`+`) operator.

The MATLAB runtime calls the `DocPolynom` `minus` method to compute `p - q`, where `p`, `q`, or both are `DocPolynom` objects:

```
function r = minus(obj1,obj2)
% MINUS Implement obj1 - obj2 for DocPolynom
    obj1 = DocPolynom(obj1);
    obj2 = DocPolynom(obj2);
    k = length(obj2.coef) - length(obj1.coef);
    r = DocPolynom([zeros(1,k) obj1.coef]-[zeros(1,-k) obj2.coef]);
end % minus
```

Defining the * Operator

The MATLAB runtime calls the `DocPolynom` `mtimes` method to compute the product `p*q`. The `mtimes` method is used to overload *matrix* multiplication since the multiplication of two polynomials is simply the convolution (`conv`) of their coefficient vectors:

```
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
```

Using the Arithmetic Operators

Given the `DocPolynom` object:

```
p = 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 - 2*x - 4

r =
    x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Overloading MATLAB® Functions for the DocPolynom Class

The MATLAB language already has several functions for working with polynomials that are represented by coefficient vectors. You can overload these functions to work with the new `DocPolynom` class.

In the case of `DocPolynom` objects, the overloaded methods can simply apply the original MATLAB function to the coefficients (i.e., the values returned by the `coef` property).

This section shows how to implement the following MATLAB functions.

MATLAB Function	Purpose
`root(obj)`	Calculates polynomial roots
`polyval(obj,x)`	Evaluates polynomial at specified points
`diff(obj)`	Finds difference and approximate derivative
`plot(obj)`	Creates a plot of the polynomial function

Defining the roots Function for the DocPolynom Class

The `DocPolynom` `roots` method finds the roots of `DocPolynom` objects by
passing the coefficients to the overloaded `roots` function:

```
function r = roots(obj)
% roots(obj) returns a vector containing the roots of obj
   r = roots(obj.coef);
end % roots
```

If p is the following `DocPolynom` object:

```
p = DocPolynom([1 0 -2 -5]);
```

then the statement:

```
roots(p)
```

gives the following answer:

```
ans =
    2.0946
    -1.0473 + 1.1359i
    -1.0473 - 1.1359i
```

Defining the polyval Function for the DocPolynom Class

The MATLAB `polyval` function evaluates a polynomial at a given set of
points. The `DocPolynom` `polyval` method uses nested multiplication, or
Horner's method, to reduce the number of multiplication operations used
to compute the various powers of x:

```
function y = polyval(obj,x)
% polyval(obj,x) evaluates obj at the points x
y = 0;
   for a = obj.coef
       y = y.*x + a;
   end
end % polyval
```

Defining the diff Function for the DocPolynom Class

The MATLAB `diff` function finds the derivative of the polynomial. The `DocPolynom diff` method differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree:

```
function q = diff(obj)
% diff(obj) is the derivative of the DocPolynom obj
    c = obj.coef;
    d = length(c) - 1;  % degree
    q = DocPolynom(obj.coef(1:d).*(d:-1:1));
end % diff
```

Defining the plot Function for the DocPolynom Class

The MATLAB `plot` function creates line graphs. The overloaded `plot` function selects the domain of the independent variable to be slightly larger than an interval containing all real roots. Then the `polyval` method is used to evaluate the polynomial at a few hundred points in the domain:

```
function plot(obj)
% plot(obj) plots the DocPolynom obj
    r = max(abs(roots(obj)));
    x = (-1.1:0.01:1.1)*r;
    y = polyval(obj,x);
    plot(x,y);
    title(['y = ' char(obj)])
    xlabel('X')
    ylabel('Y','Rotation',0)
    grid on
end % plot
```

Plotting the two `DocPolynom` objects x and p calls most of these methods:

```
x = DocPolynom([1 0]);
p = DocPolynom([1 0 -2 -5]);
plot(diff(p*p + 10*p + 20*x) - 20)
```

$6*x^5 - 16*x^3 + 9*x$

10
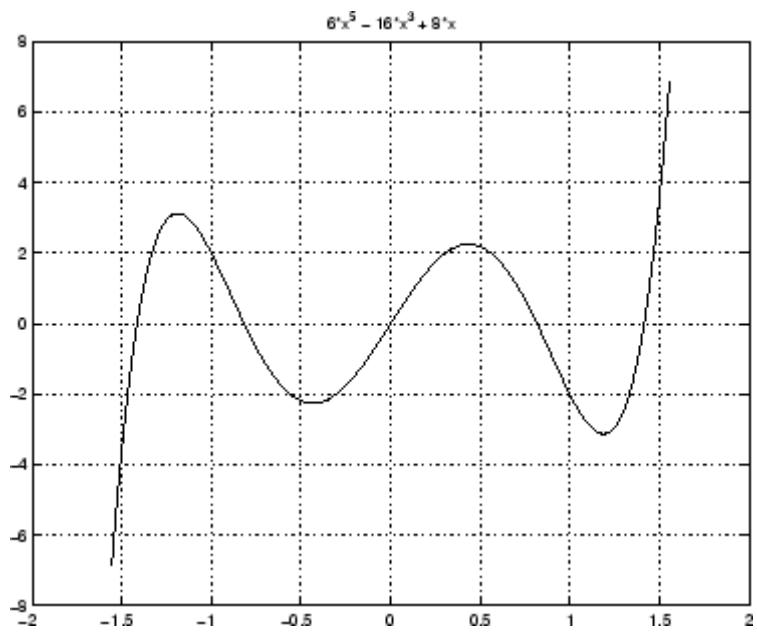
Designing Related Classes

Example — A Simple Class Hierarchy

Shared and Specialized Properties

As an example of how subclasses are specializations of more general classes, consider an asset class that can be used to represent any item that has monetary value. Some examples of assets are stocks, bonds, and savings accounts. This example implements four classes — DocAsset, and the subclasses DocStock, DocBond, DocSavings.

The DocAsset class holds the data that is common to all of the specialized asset subclasses in class properties. The subclasses inherit the super class properties in addition to defining their own properties. The subclasses are all *kinds of* assets.

The following diagram shows the properties defined for the classes of assets.
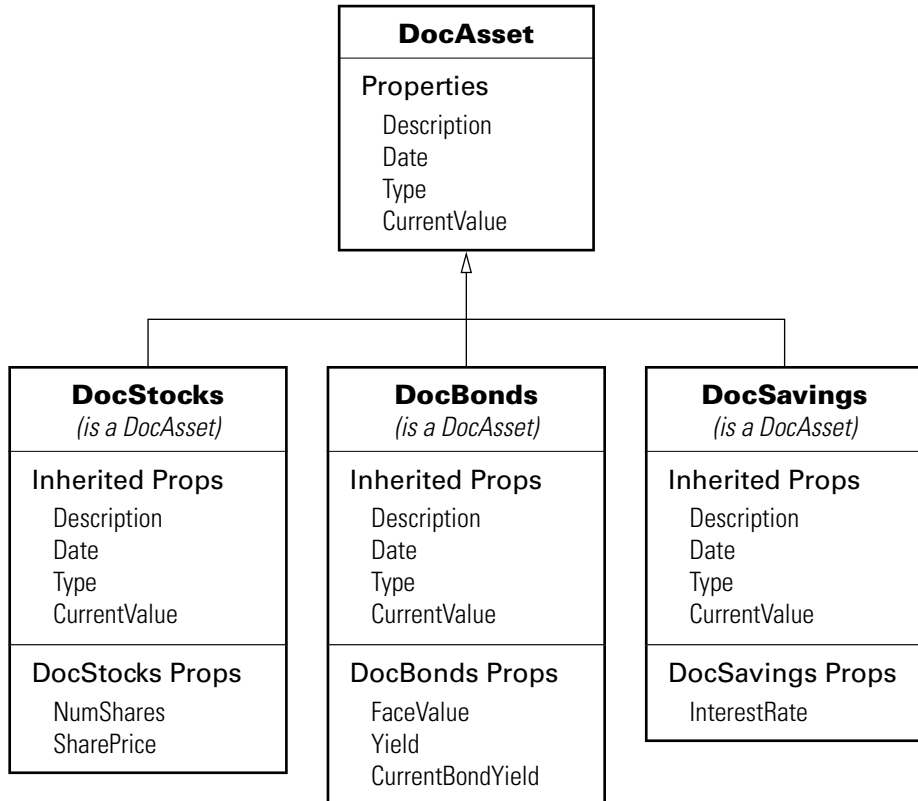
The DocStock, DocBond, and DocSavings classes inherit properties from the DocAsset class. In this example, the DocAsset class provides storage for data common to all subclasses and shares methods with these subclasses.

Designing a Class for Financial Assets

This class provides storage and access for information common to all asset children. It is not intended to be instantiated directly, so it does not require an extensive set of methods. The class contains the following methods:

• Constructor

• A local setter function for one property

Displaying the Class Files

Open the DocAsset class definition file in the MATLAB® Editor.

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

Summary of the DocAsset Class

The class is defined in one file, DocAsset.m, which you must place in an @ directory of the same name. The parent directory of the @DocAsset directory must be on the MATLAB path. See the addpath function for more information.

The following table summarizes the properties defined for the DocAsset class.

DocAsset Class Properties

Name	Class	Default	Description
Description	char	''	Description of asset
CurrentValue	double	0	Current value of asset
Date	char	date	Date when record is created (set by date function)
Type	char	'savings'	Type of asset (stock, bond, savings)

The following table summarizes the methods for the DocAsset class.

DocAsset Class Methods

Name	Description
DocAsset	Class constructor
disp	Displays information about this object
localSetType	Local setter function for Type. Property tests for correct value when property is set.

The DocAsset Constructor Method

This class has four properties that store data common to all of the asset subclasses. All except `Date` are passed to the constructor by a subclass constructor. `Date` is a private property and is set by a call to the `date` function.

- `Description` — A character string that describes the particular asset (e.g., stock name, savings bank name, bond issuer, and so on).

- `Date` — The date the object was created. This property's set access is private and calculated automatically by the `date` command.

- `Type` — The type of asset (e.g., savings, bond, stock). A local set function provides error checking whenever an object is created.

- `CurrentValue` — The current value of the asset.

Property Definition Block

The following code block shows how the properties are defined. Note the `set` function defined for the `Type` property. It restricts the property's values to one of three strings: `bond`, `stock`, or `savings`.

```matlab
properties
    Description = '';
    CurrentValue = 0;
end
properties(SetAccess = private)
    Date = date; % date function sets value
    Type = 'savings';
end
```

Constructor Method Code

Because the `DocAsset` class is not derived from another class, you do not need to construct an object explicitly. You can assign values to the specific output argument (a in the constructor below):

```matlab
function a = DocAsset(description,type,current_value)
% DocAsset constructor function
    a.Description = description;
    a.Date = date;
```

```
            a.Type = type;
            a.CurrentValue = current_value;
         end % DocAsset
```

Set Function for Type Property

In this class design, there are only three types of assets—bonds, stocks, and savings. Therefore, the possible values for the Type property are restricted to one of three possible stings by defining a set function as follows:

```
 function obj = set.Type(obj,type)
    if ~(strcmpi(type,'bond') || strcmpi(type,'stock') || strcmpi(type,'savings'))
       error('Type must be either bond, stock, or savings')
    end
    obj.Type = type;
 end %Type set function
```

The MATLAB runtime calls this function whenever an attempt is made to set the Type property, even from within the class constructor function or by assigning an initial value. Therefore, the following statement in the class definition would produce an error:

```
properties
    Type = 'cash';
end
```

The only exception is the set.Type function itself, where the statement:

```
obj.Type = type;
```

does not result in a recursive call to set.Type.

The DocAsset Display Method

The asset disp method is designed to be called from child-class disp methods. Its purpose is to display the data it stores for the child object. The method simply formats the data for display in a way that is consistent with the formatting of the child's disp method:

```
function disp(a)
% Display a DocAsset object
```

```
        fprintf('Description: %s\nDate: %s\nType: %s\nCurrentValue:%9.2f\n',...
        a.Description,a.Date,a.Type,a.CurrentValue);
    end % disp
```

The DocAsset subclass display methods can now call this method to display the data stored in the parent class. This approach isolates the subclass disp methods from changes to the DocAsset class.

Designing a Class for Stock Assets

Stocks are one type of asset. A class designed to store and manipulate information about stock holdings needs to contain the following information about the stock:

- The number of shares
- The price per share

In addition, the base class (DocAsset) maintains general information including a description of the particular asset, the date the record was created, the type of asset, and its current value.

Displaying the Class Files

Open the DocStock class definition file in the MATLAB Editor.

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

Summary of the DocStock Class

This class is defined in one file, DocStock.m, which you must place in an @ directory of the same name. The parent directory of the @DocStock directory must be on the MATLAB path. See the addpath function for more information.

DocStock is a subclass of the DocAsset class.

The following table summarizes the properties defined for the DocStock class.

DocStock Class Properties

Name	Class	Default	Description
NumShares	double	0	Number of shares of a particular stock
SharePrice	double	0	Current value of asset
Properties Inherited from the DocAsset Class			
Description	char	''	Description of asset
CurrentValue	double	0	Current value of asset
Date	char	date	Date when record is created (set by date function)
Type	char	''	Type of asset (stock, bond, savings)

The following table summarizes the methods for the DocStock class.

DocStock Class Methods

Name	Description
DocStock	Class constructor
disp	Displays information about the object

Specifying the Base Class

The < symbol specifies the DocAsset class as the base class for the DocStock class in the classdef line:

```
classdef DocStock < DocAsset
```

Property Definition Block

The following code shows how the properties are defined:

```
properties
```

```
        NumShares = 0;
        SharePrice = 0;
    end
```

Using the DocStock Class

Suppose you want to create a record of a stock asset for 200 shares of a company called Xdotcom with a share price of $23.47.

Call the `DocStock` constructor function with the following arguments:

- Stock name or description
- Number of shares
- Share price

For example, the following statement:

```
XdotcomStock = DocStock('Xdotcom',200,23.47);
```

creates a `DocStock` object, `XdotcomStock`, that contains information about a stock asset in Xdotcom Corp. The asset consists of 200 shares that have a per share value of $23.47.

The DocStock Constructor Method

The constructor first creates an instance of a `DocAsset` object since the `DocStock` class is derived from the `DocAsset` class (see "The DocAsset Constructor Method" on page 10-5). The constructor returns the `DocStock` object after setting value for its two properties:

```
function s = DocStock(description,num_shares,share_price)
    s = s@DocAsset(description,'stock',share_price*num_shares);
    s.NumShares = num_shares;
    s.SharePrice = share_price;
end % DocStock
```

The DocStock disp Method

When you issue the statement (without terminating with a semicolon):

```
XdotcomStock = DocStock('Xdotcom',100,25)
```

the MATLAB runtime looks for a method in the @DocStock directory called disp. The disp method for the DocStock class produces this output:

```
Description: Xdotcom
Date: 17-Nov-1998
Type: stock
Current Value:  2500.00
Number of shares: 100
Share price: 25.00
```

The following function is the DocStock disp method. When this function returns from the call to the DocAsset disp method, it uses fprintf to display the Numshares and SharePrice property values on the screen:

```
function disp(s)
   disp@DocAsset(s)
   fprintf('Number of shares: %g\nShare price: %3.2f\n',...
   s.NumShares,s.SharePrice);
end % disp
```

Designing a Class for Bond Assets

The DocBond class is similar to the DocStock class in that it is derived from the DocAsset class to represent a specific type of asset.

Displaying the Class Files

Open the DocBond class definition file in the MATLAB Editor.

To use the class, create a directory named @DocBond and save DocBond.m to this directory. The parent directory of @DocBond must be on the MATLAB path. See the addpath function for more information.

Summary of the DocBond Class

This class is defined in one file, DocBond.m, which you must place in an @ directory of the same name. The parent directory of the @DocBond directory must on the MATLAB path.

DocStock is a subclass of the DocAsset class.

The following table summarize the properties defined for the DocBond class

DocBond Class Properties

Name	Class	Default	Description
FaceValue	double	0	Face value of the bond
SharePrice	double	0	Current value of asset
Properties Inherited from the DocAsset Class			
Description	char	''	Description of asset
CurrentValue	double	0	Current value of asset
Date	char	date	Date when record is created (set by date function)
Type	char	''	Type of asset (stock, bond, savings)

The following table summarizes the methods for the DocStock class.

DocBond Class Methods

Name	Description
DocBond	Class constructor
disp	Displays information about this object and calls the DocAsset disp method
calc_value	Utility function to calculate the bond's current value

Specifying the Base Class

The < symbol specifies the DocAsset class as the base class for the DocBond class in the classdef line:

```
classdef DocBond < DocAsset
```

Property Definition Block

The following code block shows how the properties are defined:

```
properties
   FaceValue = 0;
   Yield = 0;
   CurrentBondYield = 0;
end
```

Using the DocBond Class

Suppose you want to create a record of an asset that consists of an xyzbond with a face value of $100 and a current yield of 4.3%. The current yield for the equivalent bonds today is 6.2%, which means that the market value of this particular bond is less than its face value.

Call the DocBond constructor function with the following arguments:

- Bond name or description

- Bond's face value

- Bond's interest rate or yield

- Current interest rate being paid by equivalent bonds (used to calculate the current value of the asset)

For example, this statement:

```
b = DocBond('xyzbond',100,4.3,6.2);
```

creates a DocBond object, b, that contains information about a bond asset xyzbond with a face value of $100, a yield of 4.3%, and also contains information about the current yield of such bonds (6.2% in this case) that is used to calculate the current value.

> **Note** The calculations performed in this example are intended only to illustrate the use of MATLAB classes and do not represent a way to determine the actual value of any monetary investment.

The DocBond Constructor Method

The DocBond constructor method requires four arguments:

```
function b =
DocBond(description,face_value,yield,current_yield)
   market_value = DocBond.calc_value(face_value,yield,current_yield);
   b = b@DocAsset(description,'bond',market_value);
   b.FaceValue = face_value;
   b.Yield = yield;
   b.CurrentBondYield = current_yield;
end % DocBond
```

The calc_value Method

The DocBond class determines the market value of bond assets using a simple formula that scales the face value by the ratio of the bond's interest yield to the current yield for equivalent bonds.

Calculation of the asset's market value requires that the yields be nonzero, and should be positive just to make sense. While the calc_value method issues no errors for bad yield values, it does ensure bad values are not used in the calculation of market value.

The asset's market value is passed to the DocAsset base-class constructor when it is called within the DocBond constructor. calc_value has its Static attribute set to true because it does not accept a DocBond object as an input argument. The output of calc_value is used by the base-class (DocAsset) constructor:

```
methods (Static)
   function market_value = calc_value(face_value,yield,current_yield)
      if current_yield <= 0 || yield <= 0
         market_value = face_value;
```

```
        else
            market_value = face_value*yield/current_yield;
        end
    end % calc_value
end % methods
```

The DocBond disp Method

When you issue this statement (without terminating it with a semicolon):

```
b = DocBond('xyzbond',100,4.3,6.2)
```

the MATLAB runtime looks for a method in the @DocBond directory called
disp. The disp method for the DocBond class produces this output:

```
Description: xyzbond
Date: 17-Nov-1998
Type: bond
Current Value:  $69.35
Face value of bonds: $100
Yield: 4.3%
```

The following function is the DocBond disp method. When this function
returns from the call to the DocAsset disp method, it uses fprintf to display
the FaceValue, Yield, and CurrentValue property values on the screen:

```
function disp(b)
    disp@DocAsset(b) % Call DocAsset disp method
    fprintf('Face value of bonds: $%g\nYield: %3.2f%%\n',...
        b.FaceValue,b.Yield);
end % disp
```

Designing a Class for Savings Assets

The DocSavings class is similar to the DocStock and DocBond class in that it
is derived from the DocAsset class to represent a specific type of asset.

Displaying the Class Files

Open the DocSavings class definition file in the MATLAB Editor.

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

Summary of the DocSavings Class

This class is defined in one file, DocSavings.m, which you must place in an @
directory of the same name. The parent directory of the @DocSavings directory
must on the MATLAB path. See the addpath function for more information.

The following table summarizes the properties defined for the DocSavings
class.

DocSavings Class Properties

Name	Class	Default	Description
InterestRate	double	''	Current interest rate paid on the savings account
Properties Inherited from the DocAsset Class			
Description	char	''	Description of asset
CurrentValue	double	0	Current value of asset
Date	char	date	Date when record is created (set by date function)
Type	char	''	The type of asset (stock, bond, savings)

The following table summarizes the methods for the DocSavings class.

DocSavings Class Methods

Name	Description
DocSavings	Class constructor
disp	Displays information about this object and calls the DocAsset disp method

Specifying the Base Class

The < symbol specifies the DocAsset class as the base class for the DocBond class in the classdef line:

```
classdef DocSavings < DocAsset
```

Property Definition Block

The following code shows how the property is defined:

```
properties
    InterestRate = 0;
end
```

Using the DocSavings Class

Suppose you want to create a record of an asset that consists of a savings account with a current balance of $1000 and an interest rate of 2.9%.

Call the DocSavings constructor function with the following arguments:

- Bank account description
- Account balance
- Interest rate paid on savings account

For example, this statement:

```
sv = DocSavings('MyBank',1000,2.9);
```

creates a `DocSavings` object, `sv`, that contains information about an account in MyBank with a balance of $1000 and an interest rate of 2.9%.

The DocSavings Constructor Method

The savings account interest rate is saved in the `DocSavings` class `InterestRate` property. The asset description and the current value (account balance) are saved in the inherited `DocAsset` object properties.

The constructor calls the base class constructor (`DocAsset.m`) to create an instance of the object. It then assigns a value to the `InterestRate` property.

```
function s = DocSavings(description,balance,interest_rate)
   s = s@DocAsset(description,'savings',balance);
   s.InterestRate = interest_rate;
end % DocSavings
```

The DocSavings disp Method

When you issue this statement (without terminating it with a semicolon):

```
sv = DocSavings('MyBank',1000,2.9)
```

the MATLAB runtime looks for a method in the @DocSavings directory called `disp`. The `disp` method for the `DocSavings` class produces this output:

```
Description: MyBank
Date: 17-Nov-1998
Type: savings
Current Value:  $1000
Interest Rate: 2.9%
```

The following function is the `DocSaving` disp method. When this function returns from the call to the `DocAsset` disp method, it uses `fprintf` to display the `Numshares` and `SharePrice` property values on the screen:

```
function disp(b)
   disp@DocAsset(b)  % Call DocAsset disp method
   fprintf('Interest Rate: %3.2f%%\n',s.InterestRate);
end % disp
```

Example — Containing Assets in a Portfolio

Aggregation is the containment of one class by another class. The basic relationship is that each contained class "is a part of" the container class.

For example, consider a financial portfolio class as a container for a set of assets (stocks, bonds, savings, and so on). It can group, analyze, and return useful information about the individual assets. The contained objects are not accessible directly, but only via the portfolio class methods.

"Example — A Simple Class Hierarchy" on page 10-2 provides information about the assets collected by this portfolio class.

Designing the DocPortfolio Class

The DocPortfolio class is designed to contain the various assets owned by an individual client and to provide information about the status of his or her investment portfolio. This example implements a somewhat over-simplified portfolio class that:

• Contains an individual's assets

• Displays information about the portfolio contents

• Displays a 3-D pie chart showing the relative mix of asset types in the portfolio

Displaying the Class Files

Open the DocPortfolio class definition file in the MATLAB® Editor.

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

Summary of the DocPortfolio Class

This class is defined in one file, DocPortfolio.m, which you must place in an @ directory of the same name. The parent directory of the @DocPortfolio directory must on the MATLAB path. See the addpath function for more information.

The following table summarizes the properties defined for the DocPortfolio class.

DocPortfolio Class Properties

Name	Class	Default	Description
Name	char	''	Name of client owning the portfolio
IndAssets	cell	{}	A cell array containing individual asset objects
TotalValue	double	0	Value of all assets (calculated in the constructor method)

The following table summarizes the methods for the DocPortfolio class.

DocBond Class Methods

Name	Description
DocPortfolio	Class constructor
disp	Displays information about this object and calls the DocAsset disp method
pie3	Overloaded version of pie3 function designed to take a single portfolio object as an argument

Property Definition Block

The following code block shows how the properties are defined:

```
properties
   Name = '';
end
properties (SetAccess = private)
   IndAssets = {};
   TotalValue = 0;
end
```

How Class Properties Are Used

- `Name` — Stores the name of the client as a character string. The client's name is passed to the constructor as an input argument.

- `IndAsset` — A cell array that stores asset objects (i.e., `DocStock`, `DocBond`, and `DocSavings` objects). These asset objects are passed to the `DocPortfolio` constructor as input arguments and assigned to the property from within the constructor function.

- `IndAsset` — The structure of this property is known only to `DocPortfolio` class member functions so the property's `SetAccess` attribute is set to `private`.

- `TotalValue` — Stores the total value of the client's assets. The class constructor determines the value of each asset by querying the asset's `CurrentValue` property and summing the result. Access to the `TotalValue` property is restricted to `DocPortfolio` class member functions by setting the property's `SetAccess` attribute to `private`.

Using the DocPortfolio Class

The `DocPortfolio` class is designed to provide information about the financial assets owned by a client. There are three possible types of assets that a client can own: stocks, bonds, and savings accounts.

The first step is to create an asset object to represent each type of asset owned by the client:

```
XYZStock = DocStock('XYZ Stocks',200,12.34);
USTBonds = DocBond('U.S. Treasury Bonds',1600,3.2,2.8);
SaveAccount = DocSavings('MyBank Acc # 123',2000,6);
VictoriaSelna = DocPortfolio('Victoria Selna',...
   XYZStock,...
   SaveAccount,...
   USTBonds)
```

The `DocPortfolio` object displays the following information:

```
VictoriaSelna =

Assets for Client: Victoria Selna
```

```
Description: XYZ Stocks
Date: 11-Mar-2008
Type: stock
Current Value: $  2468.00
Number of shares: 200
Share price: 12.34
Description: MyBank Acc # 123
Date: 11-Mar-2008
Type: savings
Current Value: $  2000.00
Interest Rate: 6.00%
Description: U.S. Treasury Bonds
Date: 11-Mar-2008
Type: bond
Current Value: $  1828.57
Face value of bonds: $1600
Yield: 3.20%

Total Value: $6296.57
```

"The DocPortfolio pie3 Method" on page 10-22 provides a graphical display of the portfolio.

The DocPortfolio Constructor Method

The DocPortfolio constructor method takes as input arguments a client's name and a variable length list of asset objects (DocStock, DocBond, and DocSavings objects in this example).

The IndAssets property is a cell array used to store all asset objects. From these objects, the constructor determines the total value of the client's assets. This value is stored in the TotalValue property:

```
function p = DocPortfolio(name,varargin)
   p.Name = name;
   for k = 1:length(varargin)
      p.IndAssets{k} = {varargin{k}};
      asset_value = p.IndAssets{k}{1}.CurrentValue;
      p.TotalValue = p.TotalValue + asset_value;
   end
```

```
end % DocPortfolio
```

The DocPortfolio disp Method

The portfolio disp method lists the contents of each contained object by calling the object's disp method. It then lists the client name and total asset value:

```matlab
function disp(p)
   fprintf('\nAssets for Client: %s\n',p.Name);
   for k = 1:length(p.IndAssets)
      disp(p.IndAssets{k}{1}) % Dispatch to corresponding disp
   end
   fprintf('\nTotal Value: $%0.2f\n',p.TotalValue);
end % disp
```

The DocPortfolio pie3 Method

The DocPortfolio class overloads the MATLAB pie3 function to accept a portfolio object and display a 3-D pie chart illustrating the relative asset mix of the client's portfolio. MATLAB calls the @DocPortfolio/pie3.m version of pie3 whenever the input argument is a single portfolio object:

```matlab
function pie3(p)
% Step 1: Get the current value of each asset
   stock_amt = 0; bond_amt = 0; savings_amt = 0;
   for k = 1:length(p.IndAssets)
      if isa(p.IndAssets{k},'DocStock')
         stock_amt = stock_amt + p.IndAssets{k}.CurrentValue;
      elseif isa(p.IndAssets{k},'DocBond')
         bond_amt = bond_amt + p.IndAssets{k}.CurrentValue;
      elseif isa(p.IndAssets{k},'DocSavings')
         savings_amt = savings_amt + p.IndAssets{k}.CurrentValue;
      end % if
   end % for

% Step 2: Create labels and data for the pie graph
   k = 1;
   if stock_amt ~= 0
      label(k) = {'Stocks'};
      pie_vector(k) = stock_amt;
      k = k + 1;
```

```matlab
      end % if
      if bond_amt ~= 0
         label(k) = {'Bonds'};
         pie_vector(k) = bond_amt;
         k = k + 1;
      end % if
      if savings_amt ~= 0
         label(k) = {'Savings'};
         pie_vector(k) = savings_amt;
      end % if

% Step 3: Call pie3, adjust fonts and colors
      pie3(pie_vector,label);set(gcf,'Renderer','zbuffer')
      set(findobj(gca,'Type','Text'),...
      'FontSize',14,'FontWeight','bold')
      colormap prism
      stg(1) = {['Portfolio Composition for ',p.Name]};
      stg(2) = {['Total Value of Assets: $',num2str(p.TotalValue,'%0.2f')]};
      title(stg,'FontSize',10)
   end % pie3
```

There are three parts in the overloaded `pie3` method.

- Step 1 — Get the `CurrentValue` property of each contained asset object and determine the total value in each category.

- Step 2 — Create the pie chart labels and build a vector of graph data, depending on which objects are present in the portfolio.

- Step 3 — Call the MATLAB `pie3` function, make some font and colormap adjustments, and add a title.

Visualizing a Portfolio

You can use a `DocPortfolio` object to present an individual's financial portfolio. For example, given the following assets:

```matlab
XYZStock = DocStock('XYZ Stocks',200,12.34);
USTBonds = DocBond('U.S. Treasury Bonds',1600,3.2,2.8);
SaveAccount = DocSavings('MyBank Acc # 123',2000,6);
VictoriaSelna = DocPortfolio('Victoria Selna',...
            XYZStock,...
```
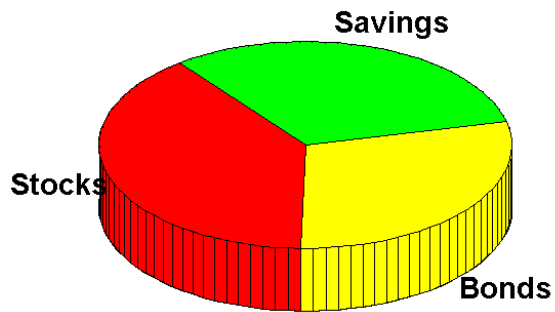
```
            SaveAccount,...
            USTBonds);
```

you can use the class's `pie3` method to display the relative mix of assets as a pie chart.

```
    pie3(VictoriaSelna)
```

Portfolio Composition for Victoria Selna
Total Value of Assets: $6296.57

Index