CHAPTER 11

Introduction to Object-Oriented Programming and Graphics

KEY TERMS

procedural languages
object-oriented languages
classes
hybrid languages
abstract data types
objects
properties
methods
class definition
instances
instantiation
inheritance
superclass/subclass
value classes
handle classes
graphics objects
object handle
primitive objects
root object
constructor function
ordinary method
overloading
attributes
copy constructor
destructor function
events
event-driven programming
listeners
callback

CONTENTS

Most programming languages are either ***procedural*** or ***object-oriented***. Procedural programs are comprised of ***functions***, each of which performs a task. Object-oriented programs use ***classes***, which contain both data and functions to manipulate the data. ***Hybrid languages*** can utilize both of these programming paradigms. All of our programs so far have been procedural, but the MATLAB® software uses objects in its graphics, and thus, has object-oriented programming (OOP) capabilities.

In this chapter, we will first introduce some of the concepts and terminologies of OOP using graphics objects and will show how to manipulate plot properties using this system. Later, we will show how user-defined classes can be created.

11.1 OBJECT-ORIENTED PROGRAMMING

This short section is intended to introduce the very basic ideas behind OOP as well as some of the terminologies that are used. This section is very dense in terms of the terminology. It is hoped that by introducing the terms once here,

MATLAB®. https://doi.org/10.1016/B978-0-12-815479-3.00011-8

and then giving examples in the next two sections, the terms will be easier to understand.

Built-in data types have certain capabilities associated with them. For example, we can perform mathematical operations such as adding and dividing with number types such as **double**. When a variable is created that stores a value of a particular type, operations can then be performed that are suitable for that type.

Similarly, ***abstract data types*** are data types that are defined by both data and operational capabilities. In MATLAB, these are called ***classes***. Classes define both the data and the functions that manipulate the data. Once a class has been defined, ***objects*** can be created of the class.

To define a class, both the data and the functions that manipulate the data must be defined. The data are called ***properties*** and are similar to variables in that they store the values. The functions are called ***methods***. A ***class definition*** consists of the definition of the properties, and the definition of the methods.

Once a class has been defined, ***objects*** can be created from the class. The objects are called ***instances*** of the class and an object that is created is an ***instantiation*** of the class. The properties and methods of the object can be referenced using the object name and the dot operator.

Inheritance is when one class is derived from another. The initial class is called the ***base***, ***parent***, or ***superclass***, and the derived class is called the ***derived***, ***child***, or ***subclass***. A subclass is a new class that has the properties and methods of the superclass (this is what is called ***inheritance***), plus it can have its own properties and methods. The methods in a subclass can override methods from the superclass if they have the same name.

MATLAB has built-in classes and also allows for user-defined classes. There are two types of classes in MATLAB: ***value classes*** and ***handle classes***. The differences will be explained in Section 11.3. MATLAB uses handle classes to implement graphical objects used in plots.

11.2 USING OBJECTS WITH GRAPHICS AND PLOT PROPERTIES

MATLAB uses graphics in all of its figures. All figures consist of ***graphics objects***, each of which is referenced using an ***object handle***. The object handles store objects, which are derived from a superclass called **handle**.

Graphics objects include ***primitive objects***, which are basic building blocks of plots, as well as the axes used to orient the objects. The graphics primitives

include objects such as lines and text. For example, a plot that uses straight line segments uses the **line** graphics primitive. More of the graphics primitives will be discussed in Section 12.3. The objects are organized hierarchically, and there are properties associated with each object.

The computer screen is called the ***root object*** and is referenced using the function **groot** (which is short for "graphics root"). When plots are made, they appear in the Figure Window; the Figure Window itself appears on the computer screen. The hierarchy in MATLAB can be summarized as follows:

```
Root (screen)          Parent
                          |
Figure Window             |
                          |
   Axes                   |
                          ↓
Primitive objects      Children
```

In other words, the Figure Window is in the screen; it includes Axes, which are used to orient graphics primitive objects, which are the building blocks of plots.

11.2.1 Objects and Properties

A Figure Window is an object; the data in objects are stored in properties. Just calling the **figure** function will bring up a blank Figure Window and return its handle; by assigning the handle of this Figure Window to an object variable, the properties can then be referenced. For example, if no other figures have been created yet, the following will create Figure (1).

```
>> f = figure
f =
  Figure (1) with properties:

    Number: 1
      Name: ''
     Color: [0.9400 0.9400 0.9400]
  Position: [440 378 560 420]
     Units: 'pixels'

Show all properties
```

By default, only a few of the properties are listed; what is shown are the names of the properties and the values for this instance of the object; these include:

- the figure number: 1
- the name of the figure: none was given so this is an empty character vector
- the color: given as a vector storing the values of the red, green, and blue components of the color

- the position of the Figure Window within the screen, specified in the units of pixels (which is shown next); this is a vector consisting of four values: the number of pixels that the lower left corner of the Figure Window is from the left of the screen, the number of pixels that the lower left corner of the Figure Window is from the bottom of the screen, the length in pixels, and the height in pixels
- the units: pixels

For the Color property, the three numbers in the vector are real numbers in the range from 0 to 1. Zero for a color component means none of that color, whereas one is the brightest possible hue. All zeros represent black, and all ones represent the color white. The default Color property value of `[0.94 0.94 0.94]` is a very light gray.

By clicking on the "all properties" link, all of the properties can be seen. As long as the Figure Window is not closed, the handle *f* can be used to refer to the Figure Window, but when the Figure Window is closed, the handle is deleted. Another method of viewing the properties is to pass the handle to the **get** function as follows.

```
>> get(f)
              Alphamap: [1×64 double]
          BeingDeleted: 'off'
            BusyAction: 'queue'
          ButtonDownFcn: ''
              Children: [0×0 GraphicsPlaceholder]
              Clipping: 'on'
      CloseRequestFcn: 'closereq'
                 Color: [0.9400 0.9400 0.9400]
              Colormap: [64×3 double]
            CreateFcn: ''
          CurrentAxes: [0×0 GraphicsPlaceholder]
     CurrentCharacter: ''
        CurrentObject: [0×0 GraphicsPlaceholder]
         CurrentPoint: [0 0]
            DeleteFcn: ''
         DockControls: 'on'
             FileName: ''
    GraphicsSmoothing: 'on'
     HandleVisibility: 'on'
        InnerPosition: [440 378 560 420]
        IntegerHandle: 'on'
        Interruptible: 'on'
      InvertHardcopy: 'on'
         KeyPressFcn: ''
```

```
          KeyReleaseFcn: ''
                MenuBar: 'figure'
                   Name: ''
               NextPlot: 'add'
                 Number: 1
             NumberTitle: 'on'
          OuterPosition: [440 378 560 493]
       PaperOrientation: 'portrait'
          PaperPosition: [0.3611 2.5833 7.7778 5.8333]
     PaperPositionMode: 'auto'
             PaperSize: [8.5000 11]
             PaperType: 'usletter'
            PaperUnits: 'inches'
                Parent: [1×1 Root]
               Pointer: 'arrow'
     PointerShapeCData: [16×16 double]
   PointerShapeHotSpot: [1 1]
              Position: [440 378 560 420]
              Renderer: 'opengl'
          RendererMode: 'auto'
                Resize: 'on'
         SelectionType: 'normal'
       SizeChangedFcn: ''
                   Tag: ''
               ToolBar: 'auto'
                  Type: 'figure'
         UIContextMenu: [0×0 GraphicsPlaceholder]
                 Units: 'pixels'
              UserData: []
               Visible: 'on'
   WindowButtonDownFcn: ''
 WindowButtonMotionFcn: ''
     WindowButtonUpFcn: ''
     WindowKeyPressFcn: ''
   WindowKeyReleaseFcn: ''
  WindowScrollWheelFcn: ''
           WindowState: 'normal'
           WindowStyle: 'normal'
              XDisplay: 'Quartz'
```

You may not understand most of these properties; do not worry about it! Notice, however, that the Parent of this figure is the Root object, and that there are no Children since there is nothing in this Figure Window.

QUICK QUESTION!

What would the following display:

```
>> f
```

Answer: It would display the same abbreviated list of properties as was shown when the handle was first created.

PRACTICE 11.1

Call the **groot** function and store the resulting handle in an object variable. What are the dimensions of your screen?

(Note that as of R2015b, pixels are now a fixed size and do not necessarily correspond exactly to the actual number of pixels on the screen.)

The **get** function can also be used to retrieve just one particular property; e.g., the Color property as follows.

```
>> get(f,'Color')
ans =
    0.9400    0.9400    0.9400
```

The function **set** can be used to change property values. The **set** function is called in the format:

```
set(objhandle, 'PropertyName', property value)
```

For example, the position of the Figure Window could be modified as follows.

```
>> set(f,'Position',[400  450  600  550])
```

Another method for referencing or changing a property of an object is to use the dot notation. The format for this is:

```
objecthandle.PropertyName
```

This method is new as of R2014b and is an alternate to using **get** and **set**. Using the dot notation is preferable to using **get** and **set** since the handles are now objects and this is the standard syntax for referencing object properties.

For example, the following modifies the Color property to a darker gray.

```
>> f.Color = [0.5 0.5 0.5]
```

For the figure object stored in *f*, its built-in class is matlab.ui.Figure; "ui" is the abbreviation for "user interface" and is used in many graphics names. This can be seen using the **class** function.

Note
This is the same as the notation to refer to a field in a structure, but it is not a structure; this is directly referencing a property within an object.

```
>> class(f)
ans =
matlab.ui.Figure
```

Recall that a class definition consists of the data (properties) and functions to manipulate the data (methods). There are built-in functions, **properties** and **methods,** which will display the properties and methods for a particular class. For example, for the figure referenced by the handle *f*, we can find the properties; note that they are not listed in alphabetical order as with **get** and that only the names of the properties are returned (not the values).

```
>> properties(f)
Properties for class matlab.ui.Figure:
    Position
    OuterPosition
    InnerPosition
    Units
    Renderer
    RendererMode
    Visible
    Color
          etc.
```

The methods for the figure *f* are as follows.

```
>> methods(f)

Methods for class matlab.ui.Figure:

Figure          addprop     clo         eq
get             isprop      ne          set
addlistener     cat         double      findobj
horzcat         java        reset       vertcat

Static methods:

loadobj

Methods of matlab.ui.Figure inherited from handle.
```

Again, much of this will not make sense, but notice that the methods are derived from the superclass **handle**. The methods, or functions, that can be used with the object *f* include **get** and **set**. Also, the methods **eq** and **ne** are defined; these are equivalent to the equality (==) and inequality (~=) operators. That means that the equality and inequality operators can be used to determine whether two figure handles are equal to each other or not.

The various plot functions return a handle for the plot object, which can then be stored in a variable. In the following, the **plot** function plots a **sin** function in

QUICK QUESTION!

Given two figure objects

```
>> f = figure(1);
>> g = figure(2);
```

Assuming that both Figure Windows are still open, what would be the value of the following expression?

```
>> f == g
```

Answer: This would be false. However, consider the following, which demonstrates that all of the color components of the two figures are the same, and that if one figure handle variable is assigned to another, they are equivalent.

```
>> f.Color == g.Color
ans =
     1     1     1
>> h = f;
>> f == h
ans =
     1
```

a Figure Window and returns the object handle. This handle will remain valid as long as the object exists.

```
>> x = -2*pi: 1/5 : 2*pi;
>> y = sin(x);
>> hl = plot(x,y)
hl =
   Line with properties:

              Color: [0 0.4470 0.7410]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1x63 double]
              YData: [1x63 double]
              ZData: [1x0 double]

   Show all properties
>> axhan = hl.Parent
   ans =
     Axes with properties:

              XLim: [-8 8]
              YLim: [-1 1]
                etc.
```

Note

The plot is generated using the **line** primitive object. As with the Figure Window, the properties can be viewed and modified using the dot notation. For example, we can find that the parent of the plot is an Axes object.

QUICK QUESTION!

How could you change the x-axis limit to `[-10 10]`?

Answer: There are two methods:

```
>> axhan.XLim = [-10 10]
>> set(axhan, 'XLim',[-10 10])
```

Note that both modify the axes in the Figure Window. Using the dot notation (the second example) in an assignment statement will show the new value of *axhan*, whereas the **set** method does not. Again, the dot notation is preferred.

The objects, their properties, what the properties mean, and valid values can be found in the MATLAB Help Documentation. Search for Graphics Object Properties to see a list of the property names and a brief explanation of each.

For example, the Color property is a vector that stores the color of the line as three separate values for the Red, Green, and Blue intensities, in that order. Each value is in the range from 0 (which means none of that color) to 1 (which is the brightest possible hue of that color). In the previous plot example, the Color was [0 0.4470 0.7410], which means no red, some green, and a lot of blue; in other words, the line drawn for the **sin** function was a royal blue hue. This is the default color for line plots.

More examples of possible values for the Color vector include:

[1 0 0] is red
[0 1 0] is green
[0 0 1] is blue
[1 1 1] is white
[0 0 0] is black
[0.5 0.5 0.5] is a shade of grey

Note
In earlier versions of MATLAB the default color was [0 0 1], or full blue.

Changing the line width in the figure makes it easier to see the line and its color, as shown in Figure 11.1. Also, tab completion is available for class properties and methods; e.g., if you are not sure of the exact property name for the line width, typing "hl.Li" and then hitting the tab key would display the options.

```
>> hl.LineWidth = 4;
```

PRACTICE 11.2

Create *x* and *y* vectors and use the **plot** function to plot the data points represented by these vectors. Store the handle in a variable and do not close the Figure Window! Inspect the properties and then change the line width and color. Next, put markers for the points and change the marker size and edge color.

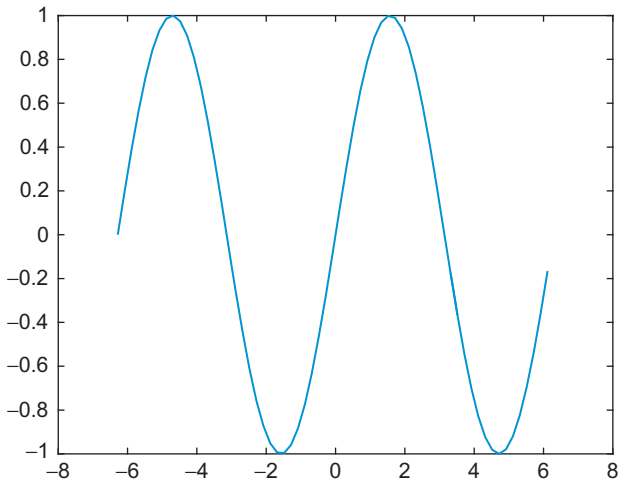


FIGURE 11.1
Line plot of sin with width of 3.

In addition to handles for objects, the built-in functions **gca** and **gcf** return the handles for the current axes and figure, respectively (the function names stand for "get current axes" and "get current figure"). In the following example, two Figure Windows are opened. The current figure and the current axes are the most recently created.

```
>> x = -pi: 0.1: pi;
>> f1 = figure(1);
>> p1 = plot(x,sin(x));
>> f2 = figure(2);
>> p2 = plot(x,cos(x),'Color',[0 1 1]);
>> curfig = gcf
curfig =
  Figure (2) with properties:

        Number: 2
          Name: ''
         Color: [0.9400 0.9400 0.9400]
      Position: [440 378 560 420]
         Units: 'pixels'

    Show all properties
```

The Children property of the current figure stores the axes that orient the plot; these axes are also returned by the **gca** function.

```
>> curfig.Children
ans =
  Axes with properties:

              XLim: [-4 4]
              YLim: [-1 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.1100 0.7750 0.8150]
             Units: 'normalized'

  Show all properties
>> curfig.Children == gca
ans =
     1
```

Within the axes, the Line primitive was used to create the plot. This can be referenced using the dot operator twice. The variable *curfig* stores the handle of the current figure; its Children property stores the current axes, and the Children property of the axes is the Line primitive.

```
>> curfig.Children.Children
ans =
  Line with properties:

              Color: [0 1 1]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'none'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [1x63 double]
              YData: [1x63 double]
              ZData: [1x0 double]

  Show all properties
```

Thus, the hierarchy is: Figure Window -> Axes -> Line.

11.3 USER-DEFINED CLASSES AND OBJECTS

There are many examples of built-in classes in MATLAB, including the **handle** class used by plot functions. It is also possible for users to define classes, and then create or instantiate objects of those classes.

11.3.1 Class Definitions

Classes are defined using the keyword **classdef**. The class definition is organized by blocks and, typically at a minimum, contains properties (the data) using the keyword **properties** and methods (the functions that act on the data) using the keyword **methods**. One special case of a method is a ***constructor function*** that initializes the properties. Class definitions are stored in code files with the same name as the class; the constructor function also is given the same name.

Every block begins with the keyword and ends with **end**. The organization of a basic class definition, in which there are two properties and one method, which is a constructor function, follows.

MyClassName.m

```
classdef MyClassName

   properties
        prop1
        prop2
   end

   methods

      % Constructor function
      function obj = MyClassName(val1, val2)
                obj.prop1 = val1;
                obj.prop2 = val2;
   end

      % Other methods that operate on properties

   end
end
```

The class definition is stored in a code file with the same name as the class. Within the **classdef** block, there are blocks for **properties** and **methods**. In the **properties** block, the names of all properties are given. It is also possible to assign default values for the properties using the assignment operator; if this is not done, MATLAB initializes each to the empty vector.

The constructor function must have the same name as the class. It only returns one output argument, which is the initialized object. If no constructor function is defined, MATLAB creates one that uses the default values given in the **properties** definition, if any, or empty vectors if not. It is best to write the constructor function to allow for the case in which no input arguments are passed, using **nargin** to check to determine how many arguments were passed to the function.

The following is a simple class definition in which there are two properties: x, which is not initialized so the default value is the empty vector, and y, which is initialized to 33. The class has one constructor function; if two arguments are passed to it, they are stored in the two properties. If not, the default values are used.

SimpleClass.m

```
classdef SimpleClass

    properties
        x
        y = 33;
    end

    methods
        function obj = SimpleClass(val1, val2)
            if nargin == 2
               obj.x = val1;
               obj.y = val2;
            end
        end
    end

end
```

Once the class has been defined, objects can be created or instantiated by assigning the name of the class. For example, the following instantiates an object named *myobject*; since the output is not suppressed, the property names and their values are shown:

```
>> myobject = SimpleClass
myobject =
   SimpleClass with properties:

      x: []
      y: 33
```

Instantiating this object automatically calls the constructor function; since no arguments were passed, the default values were used for the properties.

In the following example, input arguments are passed to the constructor.

```
>> newobject = SimpleClass(4, 22)
newobject =
   SimpleClass with properties:

      x: 4
      y: 22
```

The properties and methods can be seen using the **properties** and **methods** functions:

```
>> properties(myobject)
Properties for class SimpleClass:
  x
  y
>> methods(myobject)
Methods for class SimpleClass:
SimpleClass
```

The properties can be accessed using the dot operator to either display or modify their values.

```
>> myobject.x = 11
myobject =
 SimpleClass with properties:
  x: 11
  y: 33
```

We will now modify the class definition, making the constructor function more general and adding a new method.

SimpleClassii.m

```
classdef SimpleClassii

    properties
        x
        y = 33;
    end

    methods
        function obj = SimpleClassii(varargin)
            if nargin == 0
               obj.x = 0;
            elseif nargin == 1
               obj.x = varargin{1};
            else
               obj.x = varargin{1};
               obj.y = varargin{2};
            end
        end

        function outarg = ordMethod(obj, arg1)
            outarg = obj.x + obj.y + arg1;
    end
 end
end
```

In the class *SimpleClassii*, there are two methods. The constructor, which has the same name as the class, is general in that it accepts any number of input arguments. If no arguments are passed, the property x is initialized to 0 (y is not

initialized since a default value was already assigned to it in the **properties** block). If only one input argument is passed, it is assumed to be the value of the property *x* and is assigned to *obj.x*. If two or more input arguments are passed, the first is the value of *x* and the second is the value to be stored in *y*. Although it is best to use **nargin** and **varargin** to allow for any number of input arguments, future examples will assume the correct number of input arguments for simplicity. If the properties are to be a certain type, the constructor function should also check and ensure that the input arguments are the correct type and either typecast them or change them if not.

The following examples demonstrate instantiating two objects of the class *SimpleClassii*.

```
>> objA = SimpleClassii
objA =
  SimpleClassii with properties:

    x: 0
    y: 33

>> objB = SimpleClassii(4, 9)
objB =
  SimpleClassii with properties:

    x: 4
    y: 9
```

QUICK QUESTION!

What would the value of the properties be for the following:

```
>> ob = SimpleClassii(1, -6, 7, 200)
```

Answer:

```
x: 1
y: -6
```

The last two arguments to the constructor were ignored.

Every time an object is instantiated, the constructor function is automatically called. So, there are two ways of initializing properties: in the property definition block and by passing values to the constructor method.

The second method in SimpleClassii is an example of an ***ordinary method***. The method *ordMethod* adds the values of the input argument, the *x* property, and the *y* property together and returns the result. When calling this method, the object to be used must always be passed to the method, which is why there are two input arguments in the function header: the object and the value to be summed with the properties.

There are two ways in which the method *ordMethod* can be called. One way is by explicitly passing the object to be used:

```
>> resultA = ordMethod(objA, 5)
resultA =
     38
```

The other way is by using the dot operator with the object, as follows:

```
>> resultB = objB.ordMethod(11)
resultB =
     24
```

Both of these methods are identical in their effect; they both pass the object to be used and the value to be added. Notice that regardless of how the method is called, there are still two input arguments in the function header: one for the object (whether it is passed through the argument list or by using the dot operator) and one for the value to be added. Although it is common for the object to be the first input argument, it is not necessary to do so.

11.3.2 Overloading Functions and Operators

By default, MATLAB creates an assignment operator for classes, which allows one object of the class to be assigned to another. This performs *memberwise* assignment, which means it assigns each property individually. Thus, one object can be assigned to another using the assignment operator. Other operators, however, are not defined automatically for classes. Instead, the programmer has the power to define operators. For example, what would it mean for one object to be less than another? The programmer has the power to define "<" any way he or she wants! Of course, it makes sense for the operator to be defined in a way that is consistent with the definition for MATLAB classes. For example, it makes sense to define the equality operator to determine whether two objects are equal to each other or not (and that would typically be memberwise). An error message will be thrown if an operator is used that has not been defined.

Recall that all operators have a functional form. For example, the expression `a+b` can also be written as **plus(a,b)**. When defining an operator for a user-defined class, a member function is defined with the function name for the operator, e.g., **plus**. This is called ***overloading***, as it gives another definition for an existing function. Which function is used (the built-in or user-defined) depends on the context, which means the types of the arguments that are used in the expression.

In addition to the operator functions, it is also possible to overload other functions for a class. For example, one function that is frequently overloaded is the function **disp**. By creating a class member function **disp**, one can customize the way in which object properties are displayed. One aspect of overloading the **disp** function is that when the assignment operator is used to assign a value to an object or an object property, and the semicolon is not used to suppress the output, the **disp** function is automatically called to display the

properties—so, the format of the output that is created in the overloaded **disp** function will be seen with every unsuppressed assignment.

To illustrate some of these concepts, a class to represent a rectangle, *Rectangle*, will be developed. There are two properties, for the length and width of the rectangle. There are four methods:

- a constructor function, or method, *Rectangle*
- an ordinary method *rectarea* that calculates the area of a Rectangle object
- two overloaded functions:
  - *disp*, which displays the properties in a formatted sentence
  - *lt*, which is the function for the less than operator

What does it mean for one Rectangle object to be less than another? In the following definition, the *lt* function is defined as **true** if the area of one Rectangle object is less than another. However, this is our choice. Depending on the application, it may make more sense to define it using just the length, just the width, or perhaps based on the perimeters of the Rectangle objects. This is a cool thing about classes; the programmer can define these operator functions in any way desired.

Rectangle.m

```
classdef Rectangle

    properties
        len = 0;
        width = 0;
    end

    methods

        function obj = Rectangle(l, w)
            if nargin == 2
                obj.len = l;
                obj.width = w;
            end
        end

        function outarg = rectarea(obj)
            outarg = obj.len * obj.width;
        end

        function disp(obj)
            fprintf('The rectangle has length %.2f', obj.len)
            fprintf(' and width %.2f\n', obj.width)
        end

        function out = lt(obja, objb)
            out = rectarea(obja) < rectarea(objb);
        end
    end
end
```

For simplicity, the constructor only checks for two input arguments; it does not check for a variable number of arguments, nor does it verify the types of the input arguments. If **nargin** is not 2, the default values from the properties block are used.

Here are examples of instantiating *Rectangle* objects, both using the constructor function and using the assignment operator that MATLAB provides for classes:

```
>> rect1 = Rectangle(3,5)
rect1 =
The rectangle has length 3.00 and width 5.00
>> rect2 = rect1;
>> rect2.width = 11
rect2 =
The rectangle has length 3.00 and width 11.00
```

Notice that the overloaded *disp* function in the class definition is used for displaying the objects when the output is not suppressed. It can also be called explicitly.

```
>> rect1.disp
The rectangle has length 3.00 and width 5.00
```

As the *lt* operator was overloaded, it can be used to compare *Rectangle* objects.

```
>> rect1 < rect2
ans =
     1
```

Other operators, e.g., **gt** (greater than), however, have not been defined within the class, so they cannot be used, and MATLAB will throw an error message.

```
>> rect1 > rect2
Undefined operator '>' for input arguments of type 'Rectangle'.
```

Care must be taken when overloading operator functions. The function in the class definition takes precedence over the built-in function, when objects of the class are used in the expression.

QUICK QUESTION!

Could we mix types in the expression? For example, what if we wanted to know whether the area of *rect1* was less than 20, could we use the expression

```
rect1 < 20  ?
```

Answer: No, not with the overloaded *lt* function as written, which assumes that both arguments are *Rectangle* objects. The following error message would be generated:

```
>> rect1 < 20
Undefined function 'rectarea' for input argu-
ments of type 'double'.
Error in < (line 30)
    out = rectarea(obja) < rectarea(objb);
```

However, it is possible to rewrite the function to handle this case. In the following modified version, the type of each of the input arguments is checked. If the argument is not a *Rectangle* object, the type is checked to see whether it is the type **double**. If it is, then the input argument is modified to be a rectangle with the number specified as the length and a width of 1 (so the area will be calculated correctly). Otherwise, the argument is simply typecast to be a *Rectangle* object so that no error is thrown (another option would be to print an error message).

```
function out = lt(inp1, inp2)
    if ~isa(inp1,'Rectangle')
       if isa(inp1, 'double')
          inp1 = Rectangle(inp1,1);
       else
          inp1 = Rectangle;
     end
  end
  if ~isa(inp2,'Rectangle')
       if isa(inp2, 'double')
         inp2 = Rectangle(inp2,1);
       else
         inp2 = Rectangle;
       end
  end
  out = rectarea(inp1) < rectarea(inp2);
end
```

With the modified function, expressions mixing Rectangle objects and double values can now be used:

```
>> rect1 < 20
ans =
     1
```

11.3.3 Inheritance and the Handle Class

Inheritance is when one class is derived from another. The initial class is called the ***superclass*** and the derived class is called the ***subclass***. A subclass is a new class that has the properties and methods of the superclass, plus it can have its own properties and methods. The methods in a subclass can override methods from the superclass if they have the same name.

11.3.3.1 Subclasses

The syntax for the subclass definition includes the "<" operator followed by the name of the superclass in the first line of the code file. (Note: this is not the less than operator!) The subclass inherits all of the properties and methods of the superclass, and then its own properties and methods can be added. For example, a subclass might inherit two properties from the superclass, and then also define one of its own. The constructor function would initialize all three properties, as seen in the example that follows.

MySubclass.m

```
classdef MySubclass < Superclass

   properties
       prop3
   end

   methods

     % Constructor function
     function obj = MySubclass(val1, val2,val3)
               obj@Superclass(val1,val2)
               obj.prop3 = val3;
     end

       % Other methods that operate on properties

   end
end
```

The first line in the constructor uses the syntax obj@Superclass in order to call the constructor method of the super class to initialize the two properties defined in the super class.

For example, our class *Rectangle* can be a superclass for a subclass *Box*. The subclass *Box* inherits the *len* and *width* properties and has its own property *height*. In the following class definition for *Box*, there is also a constructor function named *Box* and an ordinary method to calculate the volume of a *Box* object.

```
classdef Box < Rectangle
     properties
          height = 0;
     end
     methods
        function obj = Box(l,w,h)
             if nargin < 3
                 l = 0;
                 w = 0;
                 h = 0;
             end
             obj@Rectangle(l,w)
             obj.height = h;
        end

        function out = calcvol(obj)
             out = obj.len * obj.width * obj.height;
        end
     end
end
```

The values of all three properties should be passed to the constructor function. If not, the input arguments are all assigned default values. Next, the *Box* constructor calls the *Rectangle* constructor to initialize the *len* and *width* properties. The syntax for the call to the *Rectangle* constructor is:

```
obj@Rectangle(l,w)
```

Note that this call to the Rectangle constructor must be executed first, before other references to the object properties. Finally, the constructor initializes the *height* property.

The following is an example of instantiating a *Box* object. Notice that since the result of the assignment is not suppressed, the *disp* function from the *Rectangle* class is called. All three of the properties were initialized, but only the length and width were displayed.

```
>> mybox = Box(2,5,8)
mybox =
The rectangle has length 2.00 and width 5.00
>> mybox.height
ans =
     8
```

To remedy this, we would have to overload the *disp* function again within the *Box* class.

```
function disp(obj)
   fprintf('The box has a length of %.2f,',obj.len)
   fprintf(' width %.2f\nand height %.2f\n',...
                obj.width,obj.height)
end

>> mybox = Box(2,5,8)
mybox =
The box has a length of 2.00, width 5.00
and height 8.00
```

11.3.3.2 Value and Handle Classes

There are two types of classes in MATLAB: ***value classes*** and ***handle classes***. Value classes are the default; so far, the classes that have been demonstrated have all been value classes. Handle classes are subclasses that are derived from the abstract class **handle**, which is a built-in class. The class definition for a **handle** class begins with:

```
classdef MyHandclass < handle
```

There is a very fundamental difference between value classes and handle classes. With value classes, if one object is copied to another, they are completely independent; changing one does not affect the other. With handle classes, on the

other hand, if one handle object is copied to another, it does not copy the data; instead, it creates a reference to the same data. All objects refer to the same data (the properties).

User-defined classes can be either value classes or handle classes. Built-in classes are also either value classes or handle classes. For example, built-in numeric types such as **double** are value classes, whereas plot objects are handle objects.

Since **double** is a value class, we can assign one **double** variable to another—and then changing the value of one does not affect the other.

```
>> num = 33;
>> value = num;
>> value = value + 4
value =
    37
>> num
num =
   33
```

On the other hand, plot object handles are handle objects. When assigning one plot handle variable to another, they both refer to the same plot.

```
>> x = 0: 0.1 : pi;
>> plothan = plot(x,sin(x));
>> handleb = plothan;
```

Both variables *plothan* and *handleb* refer to the same plot; they are not different plots. As a result, a property such as the line width could be changed by either

```
>> plothan.LineWidth = 3;
```

or

```
>> handleb.LineWidth = 3;
```

Either of these would accomplish the same thing, changing the line width in the one plot to 3.

As an example of a user-defined **handle** class, let us modify the value class *Rectangle* to be a handle class called *HandleRect* (and simplify a bit by not overloading the **lt** function).

```
classdef HandleRect < handle
   properties
      len = 0;
      width = 0;
   end
   methods
       function obj = HandleRect(l, w)
           if nargin == 2
```

```
            obj.len = l;
            obj.width = w;
        end
    end
    function outarg = rectarea(obj)
        outarg = obj.len * obj.width;
    end
    function disp(obj)
        fprintf('The rectangle has length %.2f', obj.len)
        fprintf(' and width %.2f\n', obj.width)
    end
  end
end
```

By instantiating an object, we can find the properties and methods as follows.

```
>> HRectangle = HandleRect(3,5)
HRectangle =
The rectangle has length 3.00 and width 5.00
>> properties(HRectangle)
Properties for class HandleRect:
    len
    width
>> methods(HRectangle)

Methods for class HandleRect:

HandleRect disp    rectarea

Methods of HandleRect inherited from handle.
```

By clicking on the underlined Methods link, the inherited methods can be seen.

```
Methods for class handle:

addlistener   findobj     gt         lt        listener
delete        findprop    isvalid    ne
eq            ge          le         notify
```

Notice that these inherited methods include overloaded operator functions for the operators `>`, `<`, `>=`, `<=`, `==`, and `~=`. Since the assignment operator is defined automatically for all classes, and the equality operator is overloaded for handle classes, we can assign one *HandleRect* object to another and then verify that they are identical.

```
>> HRecA = HandleRect(2,7.5)
HRecA =
The rectangle has length 2.00 and width 7.50
```

```
>> HRecB = HRecA
HRecB =
The rectangle has length 2.00 and width 7.50
>> HRecA == HRecB
ans =
     1
```

QUICK QUESTION!

What would happen if the value of *HRecA.len* was changed to 6?

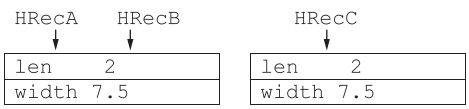
Answer:

```
>> HRecA.len = 6;
>> HRecA
HRecA =
The rectangle has length 6.00 and width 7.50
>> HRecB
HRecB =
The rectangle has length 6.00 and width 7.50
```

However, if we then create another object HRecC with the same properties as HRecA and HRecB, HRecC is not equal to either HRecA or HRecB.

```
>> HRecC = HandleRect(2,7.5)
HRecC =
The rectangle has length 2.00 and width 7.50
>> HRecA == HRecC
ans =
     0
```

This illustrates one of the important concepts about handle classes: assigning one object to another does not make a new copy; instead, it creates another reference to the same object. However, instantiating an object by calling the constructor function does create a new object, even if it happens to have the same properties as other object(s). The following is an illustration of the three *HandleRect* objects that have been created:



Since both *HRecA* and *HRecB* refer to the same object, changing a property using one of the instances will change that property for both *HRecA* and *HRecB*, but will not affect *HRecC*.

```
>> HRecA.len = 11;
>> HRecB.len
ans =
     11
```

```
>> HRecC.len
ans =
     2
```

PRACTICE 11.3

The **gt**, **lt**, **le**, **ge** functions are overloaded in **handle** classes for the operators >, <, <=, and >=, respectively. Create a **handle** class with multiple properties, instantiate at least two objects of this class, and design experiments with the objects to determine whether these overloaded operators are implemented memberwise or not.

Because handle class objects are references to the locations in which the objects are stored, there are differences between handle and value classes in the ways that objects are passed to functions and in the manner in which functions can change objects. We will create two simple classes, a value class *valClass* and a handle class *hanClass*, to illustrate the differences. Both will have just one double property *x*. There will be four methods; for simplicity, none of them error-check:

- a simple constructor
- a function *add* that receives two objects, adds the x properties together, and returns an object in which the x property is the sum of the two inputs
- a function *timestwo* that receives one object, and returns an object in which the property is the property of the input argument multiplied by two
- a function *timesthree* that receives one object, multiplies its property by three, but does not return anything

The constructor and *add* functions behave similarly in the value and handle classes. In both classes, the *add* function receives two input arguments which are objects and returns an object which is distinct from the input objects. The other two functions, however, behave differently in *valClass* and *hanClass*. We will first examine the value class.

```
classdef valClass
 properties
    x = 0;
 end
 methods
    function obj = valClass(in)
          if nargin == 1
             obj.x = in;
          end
    end
```

```
        function out = add(obja, objb)
            out = valClass(obja.x + objb.x);
        end

        function outobj = timestwo(inobj)
            outobj = valClass(inobj.x * 2);
        end

        function timesthree(obj)
            % Note: this function does not
            % accomplish anything; MATLAB
            % flags the following line
            obj.x = obj.x*3;
        end
    end
end
```

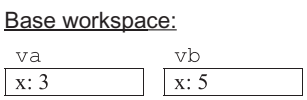
Note that the *add* and *timestwo* functions call the *valClass* constructor in order to make the output a *valClass* object.

Instantiating two objects, and calling the *timestwo* function, creates the following result.

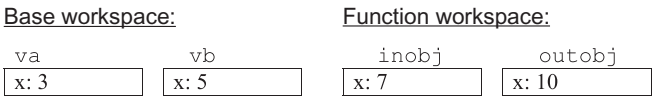
```
>> va = valClass(3);
>> vb = valClass(5);
>> vmult2 = timestwo(vb)
vmult2 =
  valClass with properties:

    x: 10
>> vb
vb =
  valClass with properties:

    x: 5
```

Note that in the base workspace, initially there are two objects *va* and *vb*. While the function is executing, the function's workspace has the input argument *inobj* and output argument *outobj*. These are all separate objects. The value of the object *va* was passed to the input object *inobj*, and a separate output argument is created in the function, which is then returned to the object *vmult2* in the assignment statement. At this point, the function's workspace would no longer exist, but the base workspace would now have *va*, *vb*, and *vmult*.

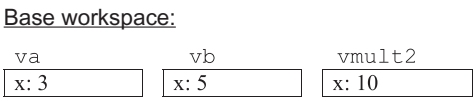
Before the function call to *timestwo*, we have:



While the *timestwo* function is executing, we have:



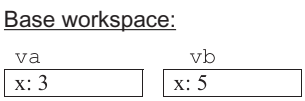
After *timestwo* has stopped executing and has returned the object, we have:



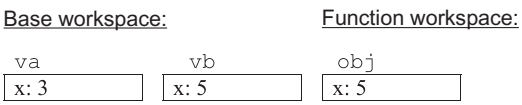
Now let us examine the behavior of the *timesthree* function, which does not return any output argument, so the call cannot occur in an assignment statement.

```
>> clear
>> va = valClass(3);
>> vb = valClass(5);
>> timesthree(vb)
>> va
va =
 valClass with properties:

     x: 3
>> vb
vb =
 valClass with properties:
     x: 5
```

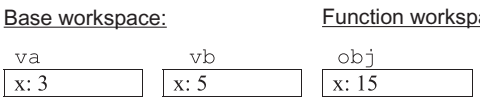
Before the function call to *timesthree,* we have:



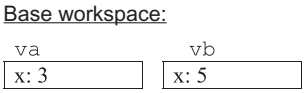
Initially in the *timesthree* function, we have:



Once the assignment statement in the *timesthree* function has executed, we have:



After *timesthree* has stopped executing, we have:



Within the body of the function, the value of the input argument was modified. However, that value was not returned. It was also a separate object from the two objects in the base workspace. Therefore, the function accomplished nothing, which is why MATLAB flags it. It appears to behave as though it is a handle class method, as we will see next.

The next example is similar, but is a **handle** class rather than a value class. Let us examine the *timestwo* and *timesthree* functions in the following **handle** class.

```
classdef hanClass < handle
    properties
       x = 0;
    end

    methods
       function obj = hanClass(in)
          if nargin == 1
            obj.x = in;
          end
       end

       function out = add(obja, objb)
          out = hanClass(obja.x + objb.x);
       end

    function outobj = timestwo(inobj)
       outobj = hanClass(inobj.x * 2);
    end

     function timesthree(obj)
         obj.x = obj.x * 3;
     end
    end
end
```

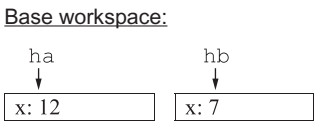
Instantiating two objects, and calling the *timestwo* function, creates the following result.

```
>> ha = hanClass(12);
>> hb = hanClass(7);
>> hmult2 = timestwo(hb)
hmult2 =
```

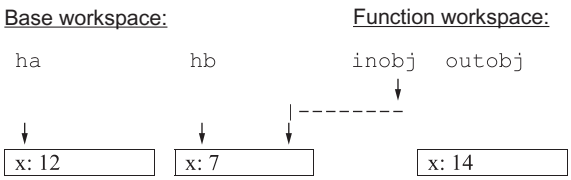
```
  hanClass with properties:

     x: 14
>> hb
hb =
  hanClass with properties:

     x: 7
>> ha
ha =
  hanClass with properties:

     x: 12
```

Note that in the base workspace, there are two objects, *ha* and *hb*. While the function is executing, there are also the input argument *inobj* and output argument *outobj*. The value of the object *hb*, which is a reference to the object, was passed to the input object *inobj*, which means that *inobj* refers to the same object. Within the body of the function, a separate object *outobj* is created.

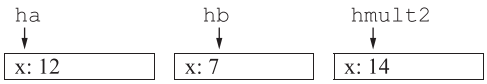
Before the function call to *timestwo*, we have:



While the *timestwo* function is executing, we have:



After *timestwo* has stopped executing and has returned the object, we have:

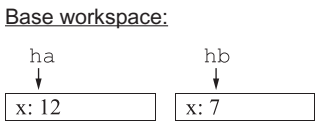


Now let us examine the behavior of the timesthree function. As with the value class, this function does not return any output argument, so calls to it cannot occur in an assignment statement. However, this function does accomplish something; it modifies the object passed as an input argument.

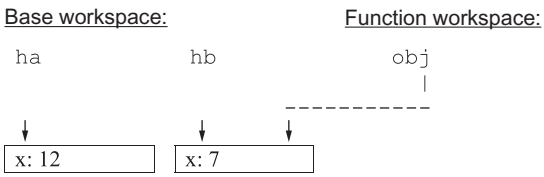
```
>> clear
>> ha = hanClass(12);
```

```
>> hb = hanClass(7);
>> timesthree(hb)
>> ha
ha =
    hanClass with properties:

     x: 12
>> hb
hb =
    hanClass with properties:
     x: 21
```

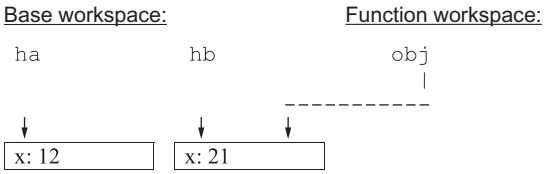
Before the function call to *timesthree*, we have:



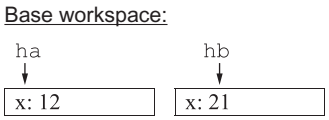
While the *timesthree* function begins, we have:



Once the *timesthree* function executes its assignment statement, we have:



After *timesthree* has stopped executing and has returned the object, we have:



Thus, because handle class objects store references, passing a handle class object to a function can allow the function to modify its properties (without returning anything). This cannot happen with value class objects.

Notice that errors will occur for both the value and handle classes if the *timesthree* functions are called in assignment statements, since these functions do not return any values.

```
>> vmult3 = timesthree(vb)
Error using valClass/timesthree
Too many output arguments.

>> hmult3 = timesthree(hb)
Error using hanClass/timesthree
Too many output arguments.
```

11.3.4 Property Definition Blocks

The behaviors of, and access to, classes, properties, and methods can be controlled by what are called ***attributes***. The attributes are specified in parentheses in the first line of any block.

There are several attributes that relate to properties. One of the most important attributes is access to the properties. There are three types of access to properties:

- **public**: access is possible from anywhere; this is the default
- **private**: access is possible only by methods in this class
- **protected**: access is possible only by methods in this class or any of its subclasses

The attributes that control the access to properties are:

- **GetAccess**: read access, which means the ability to determine the values of properties
- **SetAccess**: write access, which means the ability to initialize or modify values of properties
- **Access**: both read and write access

In the case in which different properties are to have different attributes, multiple property definition blocks can be defined. So far, since we have not specified, both the read and write access to all properties in our class definitions have been public. This means that we have been able to see, and to modify, the values of the properties, for example, from the Command Window.

The following is an example of a value class in which one property, *num*, is public and the other property, *word*, has public GetAccess, but protected SetAccess.

```
classdef valAttributes
    properties
        num = 0;
    end

    properties (SetAccess = protected)
        word = 'hello';
    end
    methods
        function obj = valAttributes(n,w)
            if nargin == 2
               obj.num = n;
               obj.word = w;
            end
        end
    end
end
```

Once an object has been instantiated using the constructor, the property *num* can be queried and modified (because Access is public by default).

```
>> valA = valAttributes(3, 'hi')
valA =
  valAttributes with properties:

     num: 3
    word: 'hi'
>> valA.num
ans =
    3
>> valA.num = 14
valA =
  valAttributes with properties:

     num: 14
    word: 'hi'
```

The property *word*, however, can be queried (because GetAccess is public by default), but it cannot be changed (because SetAccess was set to protected). Only methods within the class (or subclasses if defined) can change the word property.

```
>> valA.word
ans =
'hi'
```

```
>> valA.word = 'ciao'
You cannot set the read-only property 'word' of valAttributes.
```

This is a very useful aspect of objects. Protecting objects by only allowing class methods to modify them is a very powerful tool. One example of this is to error-check property values.

Although the access attributes are the most commonly used, there are other attributes that can be set (e.g., Constant for values that are to be constant for all objects). A table of all attributes can be found by searching the documentation for Property Attributes.

11.3.5 Method Types

There are different types of methods that can be defined in class definitions; we have already seen constructor functions, ordinary methods, and overloading functions. As with properties, there are also attributes that control the behavior of methods.

11.3.5.1 Constructor Functions

As we have seen, all classes should have constructor functions, which initialize the properties in an object. If a class definition does not include a constructor, MATLAB will initialize the properties to either default values provided in the property definition block, or to empty vectors. Constructor functions always have the same name as the class and return only the initialized object; this is true for both value and handle classes. Constructor functions should be general and should allow for the case in which no input arguments are passed by creating default property values. To be truly general, the types of the input arguments should also be checked to make sure that they are the correct types for the properties. Overloading the set.PropertyName functions for all properties allows for even more control over all functions that set the properties, including the constructor function.

If the class being defined is a subclass, the constructor function of the superclass must be called to initialize its properties. MATLAB automatically makes implicit calls to the superclass constructors using no arguments when the class is defined as a subclass in the class definition line. It is also possible to explicitly call the constructors and pass arguments to be the property values; this is necessary if the superclass constructors require that input arguments be passed.

Some languages have what are called ***copy constructor*** functions, which allow an object to be constructed by copying all properties of one object into another. MATLAB does not have a copy constructor, but it would be possible to write a constructor function so that it checks to see whether the input argument

is an object of the class type, and if so, copy the properties. The beginning of a simplified version of such a constructor for a class *MyClass* might be:

```
function obj = MyClass(varargin)
if nargin == 1
    val = varargin{1};
    if isa(val, 'MyClass')
        % Copy all properties
        obj.Prop = val.Prop;
    else
        % etc.
    end
end
end
```

11.3.5.2 Access Methods

MATLAB has special access methods that allow you to query a property and to assign a value to a property. These methods have special names that include the name of the property:

```
get.PropertyName
set.PropertyName
```

These methods cannot be called directly, and they do not show in the list of functions returned by the **methods** function. Instead, they are automatically called whenever a property is queried or an attempt is made to assign a value to a property. They can, however, be overloaded.

One reason to overload the set.PropertyName method is to be able to error-check to make sure that only correct values are being assigned to a property. For example, the following is a simple value class in which the property is a grade that should be in the inclusive range from 0 to 100; the *set.grade* function ensures this.

```
classdef valSet
    properties
        grade = 0;
    end

    methods

        function obj = valSet(in)
            if nargin == 1
                obj.grade = in;
            end
        end
```

```
    function obj = set.grade(obj,val)
        if val >= 0 && val <= 100
            obj.grade = val;
        else
            error('Grade not in range')
        end
    end
  end
end
```

The *set.grade* function restricts values for the *grade* property to be in the correct range, both when instantiating objects and attempting to modify an object. Note that the input and output argument names for the object must be the same. The **error** function throws an error message.

```
>> valobj = valSet(98)
valobj =
    valSet with properties:

     grade: 98
>> badobj = valSet(105)
Error using valSet/set.grade (line 18)
Grade not in range
Error in valSet (line 10)
                obj.grade = in;
>> valobj.grade = 99
valobj =
    valSet with properties:

     grade: 99
>> valobj.grade = -5
Error using valSet/set.grade (line 18)
Grade not in range
```

The *set.grade* function would be slightly different in a handle class, since the function can modify properties of an object without returning the object. An equivalent example for a handle class follows; note that the function does not return any output argument.

```
classdef hanSet < handle
    properties
        grade = 0;
    end

  methods
      function obj = hanSet(in)
```

```
            if nargin == 1
                obj.grade = in;
            end
        end

        function set.grade(obj,val)
            if val >= 0 && val <= 100
                obj.grade = val;
            else
                error('Grade not in range')
            end
        end
    end
end
```

Note that the **get** and **set** method blocks cannot have any attributes.

11.3.5.3 Method Attributes

As with property attributes, method attributes are defined in parentheses in the first line of the method block. In the case in which different methods are to have different attributes, multiple method definition blocks can be defined.

There are several attributes that relate to methods. These include **Access**, which controls from where the method can be called. There are three types of access to methods:

- **public**: access is possible from anywhere; this is the default
- **private**: access is possible only by methods in this class
- **protected**: access is possible only by methods in this class or any of its subclasses

All of the methods in the examples shown thus far have been public. This means, for example, that we have been able to call our class methods from the Command Window. It is very common, however, to restrict access so that only methods within the class itself (or, any subclasses) can call other methods.

Besides **Access**, other method attributes include the following, all of which are the type **logical** and have a default value of **false**.

- **Abstract**: If true, there is no implementation of the method
- **Hidden**: If true, the method is not seen in lists of methods
- **Sealed**: If true, the method cannot be redefined in a subclass
- **Static**: If true, the method is a static method which means that it is not called by any particular object; static methods are the same for all objects in the class

Static methods are not associated with any object. One reason to have a static method is to perform calculations that are typical for the class, e.g., conversion of units. The following is a simple example.

```
classdef StatClass
    methods (Static)
        function out = statex(in)
            out = in * 10;
        end
    end
end
```

As static methods are not associated with any instantiation of the class, they are therefore called with the name of the class, not by any object.

```
>> StatClass.statex(4)
ans =
    40
```

11.3.5.4 Destructor Functions

Just as constructor functions create objects, destructor functions destroy objects. In MATLAB, destructor functions are optional. If a destructor function is defined within a class, this is accomplished by overloading the **delete** function. There are specific rules that make an overloaded **delete** function a destructor function: the function must have just one input argument, which is an object of the class, and it must not have any output arguments. Also, it cannot have the value **true** for the attributes **Sealed**, **Static**, or **Abstract**. The reason for having a class destructor function is to be able to "clean up". For example, if the class opens a file, the destructor function might make sure that the file is closed properly.

11.3.6 Events and Listeners

In addition to the properties and methods blocks that are normally in a class definition, handle classes (but not value classes) can also have an **events** block. ***Events*** are some type of action that takes place; ***event-driven programming*** is when an event triggers an action. In handle classes in MATLAB, there are three main concepts:

- ***event***: an action, such as changing the value of a property or the user clicking the mouse
- ***listener***: something that detects an event and initiates an action based on it
- ***callback***: a function that is called by the listener as a result of the event

There are two methods defined in the handle class that are used with events:

- **notify**: notifies listener(s) when an event has occurred
- **addlistener**: adds a listener to an object, so that it will know when an event has occurred

Examples of these concepts will be provided later in the section on Graphical User Interfaces (GUI's). GUI's are graphical objects that the user manipulates, e.g., a push button. When the user pushes a button, for example, that is an event that can cause a callback function to be called to perform a specified operation.

11.3.7 Advantages of Classes

There are many advantages to using OOP and instantiating objects versus using procedural programming and data structures such as structs. Creating one's own classes offers more control over behaviors. Since operators and other functions can be overloaded, programmers can define them precisely as needed. Also, by redefining the set.PropertyName methods, the range of values that can be assigned is strictly controlled.

Another advantage of objects is that when using struct.field, if a fieldname is not spelled correctly in an assignment statement, e.g.,

```
struct.feild = value;
```

this would just create a new field with the incorrect fieldname, and add it to the structure! If this was attempted with a class object, however, it would throw an error.

■ Explore Other Interesting Features

Investigate creating a directory for a class so that not all methods have to be in the same file as the class definition.

Investigate the Constant attribute.

Investigate the built-in class Map and the Map containers which are data structures that utilize key/value pairs and allow for indexing using the keys. ■

SUMMARY

COMMON PITFALLS

- Confusing value and handle classes
- Not realizing that constructor functions are called automatically when objects are instantiated

PROGRAMMING STYLE GUIDELINES

- Use the dot notation to reference properties instead of **get** and **set**
- Use **nargin** to check the number of input arguments to a constructor function
- Write constructor functions to allow for no input arguments
- Call an ordinary method by using the dot operator with the object rather than explicitly passing the object

MATLAB Keyword
classdef

MATLAB Functions and Commands		
line	set	gca
groot	properties	gcf
get	methods	

MATLAB Operators
dot operator for object properties and methods.

Exercises

1. Create a **double** variable. Use the functions **methods** and **properties** to see what are available for the class **double**.
2. Create a simple **plot** and store the handle in a variable. Use the two different methods (dot notation and **get**) to view the Color property.
3. Create a simple plot, e.g., using:

```
>> y = [3 7 2 9 4 6 2 3];
>> plothan = plot(y)
```

   Create the following character vectors, and then use them to change the Color and Marker properties to a random color and a random marker. Also make the LineWidth 3.

```
>> somecolors = 'bgrcmyk';
>> somemarkers = '.ox+*sd';
```

4. Create a **bar** chart and store the handle in a variable. Experiment with the BarWidth property.
5. Create the following bar chart:

```
>> y = [2 8 3 33];
>> bhan = bar(y)
```

Make the FaceColor yellow and the EdgeColor red.

6. Assume that the following plot is created, and that the Figure Window is still open.

```
>> plothand = plot(y,'r*')
plothand =
  Line with properties:
etc.
```

What would be the value of the following expressions? Assume that they are evaluated sequentially.

```
plothand.Color
plothand.Parent == gca
gr = groot;
gr.Children == gcf
gr.Children.Children == gca
gr.Children.Children.Children == plothand
```

Test your knowledge of the hierarchy, and then try these expressions in MATLAB.

7. Create a class *circleClass* that has a property for the radius of the circle and a constructor function. Make sure that there is a default value for the radius, either in the properties block or in the constructor. Instantiate an object of your class and use the **methods** and **properties** functions.
8. Add ordinary methods to *circleClass* to calculate the area and circumference of the circle.
9. Create a class that will store the price of an item in a store, as well as the sales tax rate. Write an ordinary method to calculate the total price of the item, including the tax.
10. Create a class designed to store and view information on software packages for a particular software superstore. For every software package, the information needed includes the item number, the cost to the store, the price passed on to the customer, and a code indicating what kind of software package it is (e.g., 'c' for a compiler, 'g' for a game, etc.). Include an ordinary method *profit* that calculates the profit on a particular software product.
11. Create the *Rectangle* class from this chapter. Add a function to overload the **gt** (greater than) operator. Instantiate at least two objects and make sure that your function works.
12. Create a class *MyCourse* that has properties for a course number, number of credits, and grade. Overload the **disp** function to display this information.

13. For the following class definition:

```
classdef Oneclass
    properties
         a = 0;
    end
    methods
        function outobj = Oneclass(varargin)
            if nargin >= 1
                outobj.a = varargin{1};
            end
        end
        function out = calcit(obj, x)
            out = obj.a * x;
        end
    end
end
```

Write a **disp** function that will overload the **disp** function to display an object of the class Oneclass in the following format:

```
>> disp(object)
The object's property is 0.
```

14. Construct a class named *Money* that has 5 properties for dollars, quarters, dimes, nickels, and pennies. Include an ordinary function *equivtotal* that will calculate and return the equivalent total of the properties in an object (e.g., 5 dollars, 7 quarters, 3 dimes, 0 nickels, and 6 pennies is equivalent to $7.11). Overload the **disp** function to display the properties.
15. Write a Brownie class. It should have two properties: an integer *yield*, which is how many brownies a batch produces, and a logical *chocchips*, which is **true** if the recipe uses chocolate chips and **false** if not. Both properties should be given default values. Write two methods: a constructor function which uses the values passed to it (if any) or the default values if not; also, overload the **disp** function so that the output is as shown in the following examples.

```
>> mochab = Brownies(16, true)
mochab =
This recipe yields 16 brownies and has chocolate chips
>> bettyc = Brownies(20,false);
>> disp(bettyc)
This recipe yields 20 brownies and does not have chocolate chips
>>
```

16. Write a program that creates a class for complex numbers. A complex number is a number of the form a + bi, where a is the real part, b is the imaginary part, and

$i = \sqrt{-1}$. The class *Complex* should have properties for the real and imaginary parts. Overload the **disp** function to print a complex number.

17. A Student class is being written. There are two properties: an integer student ID number, and a string final grade for a course. So far, for methods, there is a constructor function. You are to add two more methods: an overloaded **disp** function that displays the student ID and grade, and an overloaded **mean** function that changes the grade (whatever it is) to an F (yes, truly mean!). The format should be as shown here:

```
>> studa = Student(456,'B+')
studa =
Student 456 has earned a B+
>> studa = mean(studa)
studa =
Student 456 has earned a F
>> disp(studa)
Student 456 has earned a F
>>
```

```
classdef Student
    properties
        studid = 123;
        fingrade = 'A';
 end

 methods

     function obj = Student(id,grade)
         if nargin == 2
             obj.studid = id;
             obj.fingrade = grade;
         end
     end

     % Insert the two methods here
```

18. A Studentii class is being written. There are two properties: a string *studname* and a logical *didcody*. So far there is a constructor function. You are to add an overloaded **disp** function that displays the student information. The format should be as shown here:

```
>> bob = Studentii('Bob',false)
bob =
Bob did not do the Cody problems
>> mary = Studentii('Mary')
mary =
Mary did do the Cody problems
```

```
classdef Studentii
    properties
        studname
        didcody = true;
    end

  methods
      function obj = Studentii(name,cody)
          if nargin == 1
              obj.studname = name;
          elseif nargin == 2
              obj.studname = name;
              obj.didcody = cody;
          end
      end

  end
end
```

19. Create a base class *Square* and then a derived class *Cube*, similar to the Rectangle/Box example from the chapter. Include methods to calculate the area of a square and volume of a cube.
20. Create a base class named *Point* that has properties for x and y coordinates. From this class, derive a class named *Circle* having an additional property named *radius*. For this derived class, the *x* and *y* properties represent the center coordinates of a circle. The methods of the base class should consist of a constructor, an *area* function that returns 0, and a distance function that returns the distance between two points (sqrt($(x2-x1)^2 + (y2-y1)^2$)). The derived class should have a constructor and an override function named *area* that returns the area of a circle. Write a script that has 2 objects of each class and calls all of the methods.
21. Take any value class (e.g., *MyCourse* or *Square*) and make it into a handle class. What are the differences?
22. Create a class that stores information on a company's employees. The class will have properties to store the employee's name, a 10-digit ID, their department, and a rating from 0 to 5. Overwrite the *set.propertyname* function to check that each property is the correct class and that:
    - The employee ID has 10 digits
    - The department is one of the following codes: HR (Human Resources), IT (Information Technology), MK (Marketing), AC (Accounting), or RD (research and Development)
    - The rating is a number from 0 to 5.

    The rating should not be accessible to anyone without a password. Overwrite the *set.rating* and *get.rating* functions to prompt the user for a password. Then, write a function that returns the rating.
23. Create a handle class that logs the times a company's employees arrive and leave at work. The class must have the following characteristics. As the

employer, you do not want your employees to access the information stored. The class will store date, hour, minute, second, and total time as properties. The constructor function will input the data from the *clock* function, which returns a vector with format [year month day hour minute second]. Each time an employee arrives or leaves, they must call a method *LogTime* that will store the new times with the old times. For example, property hour will be

Hour 1
Hour 2
Hour 3

Include a method *calcPay* that calculates the money owed if it is assumed that the employees are paid $15/hour. In order to do this, call a separate method that calculates the time elapsed between the last two time entries. Use the function **etime**. This method should only be available to call by *calcPay*, and the existence of *calcPay* should only be known to the coder.

24. You head a team developing a small satellite in competition for a NASA contract. Your design calls for a central satellite that will deploy sensor nodes. These nodes must remain within 30 km of the satellite to allow for data transmission. If they pass out of range, they will use an impulse of thrust propulsion to move back towards the satellite. Make a *Satellite* class with the following properties:
    - *location*: An [X Y Z] vector of coordinates, with the satellite as the origin.
    - *magnetData*: A vector storing magnetic readings.
    - *nodeAlerts*: An empty string to begin with, stores alerts when nodes go out of range.

    Satellite also has the following methods:

    - *Satellite*: The constructor, which sets location to [0 0 0] and magnetData to 0.
    - *retrieveData*: Takes data from a node, extends the magnetData vector.

    Then, make the *sensorNode* class as a subclass of Satellite. It will have the following properties:

    - *distance*: The magnitude of the distance from the satellite. Presume that a node's location comes from onboard, real-time updating GPS (i.e., do not worry about updating node.location).
    - *fuel*: Sensor nodes begin with 100 kg of fuel.

    sensorNode also has the following methods:

    - *sensorNode*: The constructor.
    - *useThrust*: Assume this propels node towards satellite. Each usage consumes 2 kg of fuel. If the fuel is below 5 kg, send an alert message to the satellite.
    - *checkDistance*: Check the magnitude of the distance between
    - useMagnetometer: Write this as a stub. Have the "magnetometer reading" be a randomized number in the range 0 to 100.
    - *sendAlert*: set the "nodeAlerts" Satellite property to the string 'Low fuel'.

    First, treat both classes as value classes. Then, adjust your code so that both are handle classes. Which code is simpler?