# Object-Oriented Programming

# **Chapter Objectives**

This chapter discusses the object-oriented paradigm, including:

- Object-oriented programming (OOP) in general
- How MATLAB implements OOP

This chapter is not intended as a full treatment of the art and science of object-oriented design and object-oriented programming (OOP). Rather, it presents some basic definitions of terms used in OOP and the implementation in MATLAB of some simple constructs. Subsequent chapters will extend these ideas to illustrate how dynamic data structures may be constructed and manipulated using OOP.

18.1	Object-Oriented Programming			
l	18.1.1 OO Background			

18.1.2 Definitions

18.1.3 Concepts

18.1.4 MATLAB
Observations

**18.2** Categories of Classes

18.2.1 Modeling Physical Things

18.2.2 Modeling Collections

18.2.3 Objects within Collections

**18.3** MATLAB Implementation

18.3.1 MATLAB Classes

18.3.2 MATLAB Objects

18.3.3 MATLAB Attributes

18.3.4 MATLAB Methods

18.3.5 Encapsulation in MATLAB Classes

18.3.6 Inheritance in MATLAB Classes

18.3.7 MATLAB Parent Classes

18.3.8 MATLAB Child Classes

18.3.9 Polymorphism in MATLAB

**18.4** Example—Modeling Bank Accounts

18.4.1 The Base Class

18.4.2 Inheritance by Extension

18.4.3 Inheritance by Redefinition

18.5 Practical Example— Vehicle Modeling

18.5.1 A Vehicle Hierarchy

18.5.2 The Containment Relationship



# 18.1 Object-Oriented Programming

In Chapters 1–17 we saw MATLAB as a means for manipulating arrays sometimes in the guise of vectors or matrices, character strings, or cell arrays. Furthermore, the manipulation performed was conducted by writing scripts that sometimes call functions—either the functions built into MATLAB or functions we create ourselves. This type of programming is referred to as being in the procedural paradigm—functions and scripts as a form of procedure.

This chapter considers a different paradigm altogether—the objectoriented (OO) paradigm. In this programming style, we still begin with a script, but the scripts that we write will usually create and interact with **objects** rather than **arrays**.

## 18.1.1 OO Background

Languages that express the essential elements of the OO paradigm have been around since the 1960s when Simula was first developed. However, in the 1980s and 1990s, as massive software projects and especially graphical user interfaces (GUIs) became common, OO emerged as the paradigm of choice for designing and developing large software systems. Major software systems (like the various releases of Microsoft Windows) faced enormous design and integration challenges that could not be met by conventional programming practices. They needed language-imposed management of the interaction between large and small collections of programs and data.

A secondary requirement in efficiently developing large software systems is the ability to reuse core software modules without rewriting their entire contents. OO principles allow core modules to be reused in three ways:

- Reused intact, because the definitions of how to use them are precisely recorded
- Reused and extended, adding specific custom capabilities not found in the original module, but using all of the original capabilities
- Reused and redefined, replacing a few attributes of the general module by more specific definitions, while retaining all of the original characteristics

The OO paradigm emerged as the framework that made large, reliable software systems possible. Note that OO encompasses far more than the syntax of any particular language, and far more than its core concepts, which are touched briefly in this section of the book. OO is primarily a design issue. OO design takes advantage of the tenets and concepts of OO languages to produce good software system designs, from which good software systems can be built.

Many books are available to students wishing to pursue this subject further. However, a book on computer science concepts would be incomplete without a serious treatment of, and some practical exposure to, OO concepts. As we study OO concepts and their MATLAB implementation, it should be in the light of realizing the power of these concepts, and their place in the development of significant software systems.

#### 18.1.2 Definitions

In general, we will use the following definitions:

- A class is the generic description of something. For example, a Toyota Prius indicates the nature of the car and the fact that one must specify a color and body style, and defines the functional relationships between, for example, the speed and fuel consumption.
- An object is an instance of a class in the same way that one specific Toyota Prius is an instance or example of the Toyota Prius design, having a specific identification number, color and body style, and more concretely, its own values for its current location, speed, fuel contents, and so on.
- An attribute is a data component of the class definition that will have a specific value in an object of that class at a given time. For example, all cars have a speed, but you must examine a particular car to determine its current speed.
- A method, much like a MATLAB function, is a procedural abstraction attached to a class that enables manipulation of the attributes of a particular object. Unlike a MATLAB function, methods have access not only to their own workspace, but also to the attributes of the class of which they are a part.
- Encapsulation is actually the core of good OO design. It is the process of packaging attributes and methods in such a way that you define and control the interfaces through which outside users (other objects) can access the attributes of your objects.
- Inheritance is the characteristic that enables the reuse and/or extension of core facilities. It is the facility by which a general, core class can be extended to add more specific attributes and methods, perhaps redefining the behavior of a few existing methods, but retaining access to the original, unmodified behavior.
- A parent class is the class from which other classes inherit characteristics.

<sup>&</sup>lt;sup>1</sup> The MATLAB implementations in this text were designed to remain as close as possible to a style that permits ready translation to more conventional OO language implementations (Java, C++, or C#). Where possible, we will note typical Java implementations of the MATLAB software artifacts.

- Child classes are classes derived from parent classes by inheritance.<sup>2</sup> Of course, grandchild classes can also inherit from child classes, and so on.
- **Polymorphism**<sup>3</sup> is the ability to treat all the children of a common parent as if they were all instances of that parent. It has two important aspects, as follows:
  - 1. All objects that are children of a parent class must be treated collectively as if they were all instances of the parent
  - 2. Individually, the system must reach the specific child methods when called for

## **18.1.3 Concepts**

**Behavioral abstraction** is the central concept in OO programming. In Section 3.2.1 we discussed **data abstraction** as the ability to group disparate data items as arrays or structures that permit us to discuss "the temperature readings for July" or "the information about that CD" without having to enumerate all the details. In Section 5.1 we also referred to functions as implementing **procedural abstraction** whereby we could collect a number of recurring instructions, group them, and invoke them without reiterating all the details.

Behavioral abstraction combines these two abstractions, allowing us to encapsulate not only related data items, but also the operations that are legal to perform on those data items. In the terms defined above, we can visualize a class as the encapsulation of a number of attributes with the methods that operate on those attributes to describe the behavior of an object.

An **abstract data type (ADT)** is a technique for describing the general character of a class without descending into specifics. The format of an ADT does not have to be graphical—there are good textual techniques for accomplishing the same objective. The specific ADT format we will use in this text is shown in Figure 18.1, where the data defined for the class are enclosed in the box and the methods for operating on the data are identified by the rectangles. We will use this form to define the overall behavior of classes before diving into the detailed implementation.

#### 18.1.4 MATLAB Observations

The basic MATLAB functionality we have already used exhibits some interesting behaviors. Consider, for example, the plus operator, A + B.

<sup>&</sup>lt;sup>2</sup> Some OO languages permit child classes to inherit from more than one set of parents—a practice that can cause significant logical challenges. Other languages (Java, for one) enforce single inheritance chains, but provide alternate mechanisms (interfaces) for enforcing other common behaviors. We will see interface-like characteristics in our MATLAB code examples.

<sup>&</sup>lt;sup>3</sup> From a Greek word meaning "taking many forms."

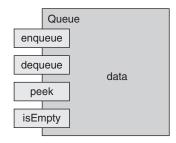


Figure 18.1 ADT illustration

Depending on the actual nature of A and B, this might have very different effects. For example, if A is an array, its actual type (referred to in the MATLAB workspace window as its class) is double. MATLAB actually interprets A + B as plus(A, B); in other words, it applies the plus method to the object A with B as the second parameter. Depending on the nature of B, this operation may have very different results:

- If A is  $1 \times 1$  (really, a scalar value), the operation will succeed when B is just about anything
- If A is any sized array, the operation will succeed only if B is either a scalar or the same size as A
- Otherwise, the operation will fail with an error message

These seem to be obvious statements, but they are actually quite profound and fundamental to our understanding of OO. This means that what we have previously considered to be an "open collection" of numbers is actually a collection "protected from the world" by a set of methods that "understand" what can and cannot be done with the data (see Figure 18.2).

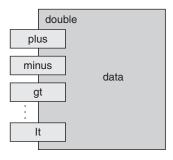


Figure 18.2 ADT for the class double



# 18.2 Categories of Classes

In general, classes fall into two categories: classes that model individual objects (physical or abstract), such as vehicles, bank accounts, or GUI widgets;

and classes that model collections of things, such as arrays or queues. It is important to keep these model categories separate, but to provide for the fact that collection classes need to be able to hold objects of many different types—individual objects and other collection objects.

For example, if we intended to model traffic flow in a city, we might begin by considering the city map as a collection of streets and intersections. The role of a street object is to contain and organize the vehicles moving on that street; the role of an intersection object (which might temporarily contain vehicles) is primarily to move vehicles from one street to another. Since one of the attributes of each vehicle should be the route it is currently following, the intersection should query a vehicle to discover which direction it should turn.

## 18.2.1 Modeling Physical Things

We talk generically about the vehicles on the streets. In general, a vehicle would have a size, heading, location, speed, and route plan. It would have a method for moving along a street, and perhaps a generic method for drawing the vehicle when necessary. Specific vehicles would have specific constraints—motorcycles might have a higher maximum speed, for example. If we were drawing the vehicles on the streets, each specific vehicle would also have its own specific drawing method. So a vehicle class would encapsulate the basic data about a specific vehicle with methods like draw(...) and move(...).

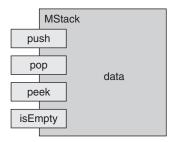
# **18.2.2 Modeling Collections**

When we look closer at the behavior of vehicles on a street (excluding passing, accidents, and so on), they behave as if they are in a queue. They join at the back of the queue, and arrive in order at the intersection at the other end of the street. So the streets really are containers of collections of vehicles with a specific behavior that we might describe as a queue. Therefore, a queue class would encapsulate objects it currently contains with methods like enqueue(...) and dequeue(...), as shown in Figure 18.1.

In a totally different application, if we were, for example, emulating a Polish notation calculator, we would need a stack to hold the intermediate values. We might visualize a stack<sup>4</sup> class Mstack, as shown in Figure 18.3. By convention, we refer to the process of adding data to a queue as enqueuing and the process of adding data to a stack as pushing. We refer to removing data from queues and stacks as dequeuing and popping, respectively.

Finally, there are advanced processing applications that require a special kind of queue called a priority queue. Imagine, for example, a process for organizing print jobs where one does not have to wait for small print jobs to

<sup>&</sup>lt;sup>4</sup> Actually, since MATLAB has a predefined stack class, we will name our stack class MStack.



**Figure 18.3** *ADT for a stack* 

print while long print jobs are being processed. Jobs would be enqueued for printing, and small jobs would be put in the queue ahead of larger jobs. This is an example of a priority queue.

## 18.2.3 Objects within Collections

A street will process each vehicle generically by traversing its queue contents, but the vehicle-specific characteristics will govern their actual behavior. In upcoming chapters we will discuss the MATLAB implementation of such a model that integrates objects with object collections; but first, we will separately consider modeling objects and collections with simple examples.



# **18.3 MATLAB Implementation**

MATLAB does a credible job implementing the OO language characteristics detailed in Section 18.1. Consider the very simple pair of classes shown in Figure 18.4. Both the parent class, <code>Fred</code>, and the child class, <code>FredChild</code>, contain one data item and host one method named <code>add</code>. The parent <code>add</code> method adds the value provided to its local data storage. The child <code>add</code> method adds to its own data and to the parent's data.

The following paragraphs indicate in this context how MATLAB implements the OO characteristics enumerated in Section 18.1.

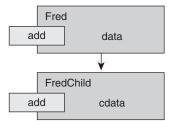


Figure 18.4 Example classes

#### 18.3.1 MATLAB Classes

In MATLAB a class is a collection of functions (the methods of that class) stored as M-files in a specific subdirectory of the current workspace. For example, if we were modeling the class named <code>Fred</code> illustrated in the figure, all of the public methods (those you want to be accessible from outside the class) must be stored in a directory named <code>@Fred</code>. There are two ways to withhold public access to methods. If you need a private utility for use by only one of <code>Fred</code>'s methods, it can be included as a local function inside that method definition file. Utilities to be shared by more than one method but not from outside the class can be stored in a subdirectory named <code>private</code>. MATLAB requires the following methods to be present:

- A constructor named the same as the class (Fred in our example) that may contain parameters used to initialize the object and that returns the completed object. However, it must make provisions for two features: calling the constructor with no parameters and calling the constructor to copy another object of the same class—accomplished by calling the constructor with the object to copy as the only parameter. The Fred constructor is shown in Listing 18.1.
- A display(...) method that is used by MATLAB whenever an assignment is made to a variable of this class without the semicolon to suppress the display.
- A char(...) method<sup>6</sup> that returns a string describing the content of this object. Typically, the display(...) method merely contains disp(char(...)), but this method also allows functions like fprintf(...) to display the object as a string.
- set and get methods for any attributes accessible by child classes.
- Other methods required by the class specification.

# 18.3.2 MATLAB Objects

An **object** in MATLAB is created in a test script or a method of another class by assigning to a variable a call to the constructor of the required class with or without initializing parameters. For example, myFred = Fred would create an object myFred with default values of its attributes. Having created the object, its methods are called by invoking them by name with myFred as the first parameter.<sup>7</sup> For example, to display the object, one might use the following:

fprintf('the object is %s\n ', char(myFred) );

<sup>&</sup>lt;sup>5</sup> Java constructors normally do not return the object—they just initialize its attributes.

<sup>&</sup>lt;sup>6</sup> Java enthusiasts might recognize this as toString() except that MATLAB does not automatically invoke char(...) if it is expecting a string but sees an object.

 $<sup>^7</sup>$  Most OO implementation languages use the "structure access" style for invoking methods on an object like: myFred.char().

#### 18.3.3 MATLAB Attributes

Attributes in MATLAB are stored in a structure we will refer to as the **attribute structure** that actually becomes typed as the object. The structure is made into an object by casting its type in the class constructor. This is shown in Listing 18.1.

## In Listing 18.1:

Line 1: The constructor for any class looks exactly like a function returning an object of that class.

Line 2: To indicate the purpose of this particular function, we include <code>@Fred\</code> in the first comment line to indicate that this constructor should be stored in that directory.

Lines 3–5: Show the remainder of the documentation.

Line 6: If no arguments were provided, this is the default constructor that must populate the attributes of the class with default values. Since MATLAB uses default constructors internally, this logic must always be there.

Line 7: Sets the default contents of the attribute named value. Notice that at this point we have created a structure named frd.

Line 8: We change the data type of the structure from struct to Fred, thereby restricting external access to the attributes of any object.

Line 9: This could also be a copy constructor designed to make a copy of another object of type Fred. We detect this by checking the class of the parameter.

Line 10: Since MATLAB always passes by value, the data item is already a copy of the original object, so just return that.

# **Listing 18.1** Constructor for the Fred class

Lines 12 and 13: This is the real constructor, which is very similar to the default constructor except the attribute is set to the data provided.

#### 18.3.4 MATLAB Methods

A method in MATLAB is written like a function, is stored in the class folder, and obeys all the normal rules of functions. The object attributes are presented to the method as its first parameter (which is actually an attribute structure). If the method changes any attributes of the object, it is necessary to return the updated attribute structure to the calling program. This is an ugly dilemma. Most conventional OOP languages implement method calls by treating the methods of a class like attributes of a structure, and allow the user to write calling code as follows:

```
myObject = Fred(4);
myObject.add(3);
```

As a result, we would optimistically imagine that the value stored in myobject would be 7. However, MATLAB cannot invoke methods in this form because it has no mechanism for treating functions as attributes. Therefore, it must provide the object as a parameter (usually the first one) to a regular function call. Consequently, the above code has to be implemented as follows:

```
myObject = Fred(4);
add(myObject, 3);
```

So we ask ourselves whether myobject now contains the value 7. Of course not, because the function add(...) received a copy of myobject and has no access to the original. We would be tempted to concede defeat and write the example for a third time as follows, where the function has to return an updated copy of the original object:

```
myObject = Fred(4);
myObject = add(myObject, 3);
```

Although this last approach works, at least from the user's perspective, it is far removed from the original concept of an object as one with persistent data. Furthermore, it takes away the ability to easily return a result from an object method. Fortunately, there is a construct in MATLAB that allows the call add(myobject,3) to tunnel back to the caller's workspace and modify the original object. In this text we have chosen the tunneling back approach shown in Listing 18.2, which illustrates the code to add something to the data stored in an object of type Fred. After performing the addition, the following line:

```
assignin('caller', inputname(1), frd)
```

is the command that tunnels back. First it obtains the name of the variable provided to this method using inputname(1), and then it assigns the updated attribute structure frd to that name in the caller's workspace.

#### Listing 18.2 Fred's add method

```
1. function add(frd, data)
2  % @Fred\add to add data to its value.
3. % add(frd, data)
4.  frd.value = frd.value + data;
5.  assignin('caller', inputname(1), frd);
```

#### In Listing 18.2:

Line 1: Shows the function header with an object of class Fred as the first parameter.

Lines 2 and 3: Show the documentation.

Line 4: Modifies the copy of the original object.

Line 5: Tunnels back to the caller's workspace, finds the variable provided as the first parameter name, and copies our new object to that variable.

## 18.3.5 Encapsulation in MATLAB Classes

**Encapsulation** is accomplished in MATLAB by storing all the methods for a class in the folder named for the class (@fred in the example).

#### 18.3.6 Inheritance in MATLAB Classes

Inheritance is sometimes referred to as an "IS-A" relationship because it defines a child class that exhibits all the behavior of the parent class as well as its own unique characteristics. It is accomplished in the MATLAB constructor for the child class. A local variable is first initialized as an instance of the parent class. The class cast method that creates the new child object includes this parent instance as a third parameter. The end result of this manipulation is an additional attribute in the child class that has the same name as the parent class and contains a reference to the parent object. When the child class needs access to the methods of the parent, it refers to these methods by way of this reference. Listing 18.3 shows the code for the child constructor.

#### **Listing 18.3** Fred child constructor

```
    function fc = FredChild(pd, cd)
    % @FredChild class constructor.
    fc = FredChild (pd, cd) creates a FredChild
    whose parent value is pd, and local
    attribute is cd
```

continued on next page

<sup>&</sup>lt;sup>8</sup> MATLAB child classes cannot directly access the attributes of the parent. They are accessed and modified only by way of the parent's set and get methods.

```
6. % This could also be a copy constructor:
7. % fc = FredChild ( ofc ) copies the object ofc
8. if nargin == 0
9.    super = Fred;
10.    fc.cdata = 0;
11.    fc = class(fc, 'FredChild', super);
12. elseif isa(pd, 'FredChild')
13.    fc = pd;
14. else
15.    super = Fred(pd);
16.    fc.cdata = cd;
17.    fc = class(fc, 'FredChild', super);
18. end
```

#### In Listing 18.3:

Lines 1–8: Show a typical header block for this constructor.

Line 9: When constructing a default child, we first construct a local parent object named super with default contents.

Line 10: We set the child data items to default values.

Line 11: We include the parent object in the class cast to establish the parent-child relationship. The actual result of including super is to establish another field in the FredChild structure named Fred whose content is a Fred object.

Lines 12 and 13: Show the copy constructor.

Line 15: The real constructor passes the pd parameter to create the parent object.

Line 16: Sets the child data field.

Line 17: As with the default constructor, we establish the parentchild link by passing the parent object to the class cast.

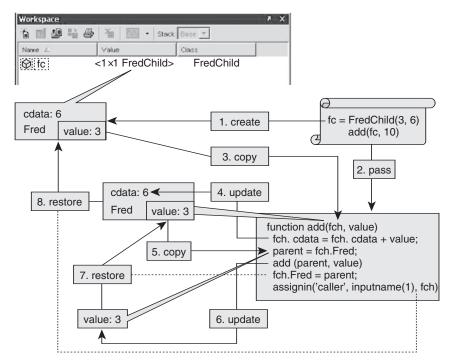
#### 18.3.7 MATLAB Parent Classes

A parent class is the class from which other classes inherit characteristics.

#### 18.3.8 MATLAB Child Classes

Child classes are derived from parent classes by inheritance. Of course, grandchild classes can also inherit from child classes. A child class cannot directly access the attribute structure of its parent. However, it can access the methods of the parent class by way of an attribute with the name of the parent class. This logic needs careful study. Examine the diagram shown in Figure 18.5, the explanatory notes, and the code for the add method of the child class

<sup>&</sup>lt;sup>9</sup> Some OO languages permit child classes to inherit from more than one set of parents—a practice that can cause significant logical challenges. Other languages (Java, for one) enforce single inheritance chains, but provide alternate mechanisms (interfaces) for enforcing other common behaviors. We will see interface-like characteristics in our MATLAB code examples.



**Figure 18.5** *Functionality of a child class* 

shown in Listing 18.4. Note that for illustrative purposes, this method is intended to add the provided data to the attributes of both parent and child. This method might be invoked in the following code:

```
fc = FredChild(3, 6);
add(fc, 10);
```

Figure 18.5 illustrates the sequence of creation and access for these two "simple" operations. The notes below follow the figure callouts.

- When a script creates an object, a structure whose class name is the name of the child class appears in the workspace. Its attributes are the local data values for the child object and a special attribute with the name of the parent class. That parent class attribute contains the parent data items defined for this object.
- 2. When a child method is called, since the parameters are passed by value, the numerical parameters are copied in directly.
- 3. The original object is copied into the workspace of the child method, so the child method is working on a copy of the original object.
- 4. The child method can directly modify those parts of the attribute structure that belong to that child.

#### Listing 18.4 child class add method

```
1. function add(fc, value)
2. % @FredChild\add
3. % add this value to both the child's and
4. % parent's stored data
5. fc.cdata = fc.cdata + value;
6. parent = fc.Fred;
7. add(parent, value)
8. fc.Fred = parent;
9. assignin('caller', inputname(1), fc);
```

- 5. The child reaches the methods of the parent class via the attribute named for the parent class—in this case, fch.Fred. This attribute contains the attributes of the parent object that are defined for this child class. Because this parent structure is an attribute structure, it cannot be directly passed as a parameter to the parent's methods. Rather, we must first extract a copy of the parent's attribute structure.
- 6. We call the parent method with that copy, which modifies our local copy of the parent attributes.
- 7. Then, we put the copy of the parent attributes back into our local attribute structure.
- 8. Finally, since this method is also working with a copy of this object's attributes, these updated attributes must be copied back to the caller.

## In Listing 18.4:

Lines 1–4: Show a suitable header block for this method.

Line 5: Adds the value to the child's data attribute.

Line 6: This line needs explanation. We really want to say the following:

```
add(fc.Fred, value)
```

However, this will tunnel back to change the value of fc.Fred, and the addressing capability of assignin(...) is restricted to elementary variables; it cannot reach structure fields. So we have to extract the parent, add to it, and copy the result back to the parent.

Line 7: Adds to the parent's attribute.

Line 8: Restores the local parent structure.

Line 9: Copies the result back to the caller.

# **18.3.9 Polymorphism in MATLAB**

There are two crucial aspects of polymorphism:

1. All objects that are children of a parent class must be treated collectively as if they were all instances of the parent

2. Individually, the system must reach the specific child methods when called for

MATLAB achieves the first very naturally because it ignores the type of all data until forced to operate on that data. The power of this polymorphic approach is this: Throughout MATLAB, all data objects are self-aware—of their data type and the methods they can implement. Therefore, the second objective is achieved because when MATLAB calls a method on a particular object, it goes to the definition of that object for the method implementation.

A simple example might suffice. In the Fred class, in addition to the add and constructor methods discussed, we implemented a display(...) method and a char(...) method to show the contents of objects of type Fred. In the FredChild class there is no display(...) method, but there is a char(...) method to return the contents of this child and its parent class. Listing 18.5 shows a simple test script for a child class.

## In Listing 18.5:

Lines 1 and 2: Show the typical beginning of a script.

Line 3: Shows an additional initializer to remove previous class definitions. If this is not present, for some reason MATLAB balks when you define a class you have previously defined.

Line 4: Creates an instance of the parent and because the semicolon is missing, shows it by calling display(...) on the parent class. As shown in Listing 18.6, the display(...) method calls char(...) on this object (a parent class) that acts exactly like a char cast of the object, producing a character string.

Line 5: Creates and displays a child object. Note that this uses the display method on the parent because the child does not have one, but calls the <code>char(...)</code> method on the child. Note also that the child <code>char(...)</code> method invokes the parent's <code>char(...)</code> method.

Line 6: Changes the data in both parent and child by adding 10 to them. Line 7: Displays the child again.

Listings 18.6, 18.7, and 18.8 show the parent display method and the char methods for the parent and child.

## **Listing 18.5** Fred child test program

```
1. clear
2. clc
3. clear classes
4. f = Fred(20)
5. fc = FredChild(3, 6)
6. add(fc, 10)
7. fc
```

#### **Listing 18.6** Parent's display(...) method

```
1. function display(frd)
```

- 2. % @Fred\display method
- 3. disp(char(frd))

## In Listing 18.6:

Lines 1 and 2: Show the typical header.

Line 3: Calls the char(...) method for whichever object is provided as a parameter, and passes the resulting string to the standard MATLAB disp(...) method.

## In Listing 18.7:

Lines 1 and 2: Show the typical header.

Line 3: Creates a string using sprintf(...), which incorporates the parent data attribute.

## In Listing 18.8:

Lines 1 and 2: Show the typical header.

Line 3: Calls the char(...) method of the parent to retrieve that string, and then creates a string using sprintf(...), which incorporates both child and parent data.

Listing 18.9 shows the results of the script in Figure 18.5.

#### **Listing 18.7** Parent's char(...) method

```
1. function str = char(frd)
```

- 2. % @Fred\char method
- 3. str = sprintf('Fred with %d', frd.value );

#### **Listing 18.8** Child's char(...) method

```
1. function str = char(frdc)
```

- 2. % @FredChild\char method
- 3. str = ...

```
sprintf('FredChild with %d containing %s', ...
frdc.cdata, char(frdc.Fred) );
```

#### **Listing 18.9** Results in the Command window

- 1. Fred with 20
- 2. FredChild with 6 containing Fred with 3
- 3. FredChild with 16 containing Fred with 13



# **₹** 18.4 Example—Modeling Bank Accounts

The overall goal of this section is to show a slightly more practical model than the previous illustrations without having to write too much code, and then illustrate inheritance by extension, and by redefining a small part of the code. In particular, the emphasis will be on designing and gathering functionality in the parent class in order to minimize the amount of content in the child classes.

We will first see the ADT and MATLAB model for a simple bank account class, and then consider two techniques for extending that class—using extension to model a savings account, and then using redefinition to model an overdraft-protected account.

#### 18.4.1 The Base Class

Figure 18.6 illustrates the ADT for a basic BankAccount class. While there are a number of attributes of a real account associated with the identity and ownership of that account, we will keep things as simple as possible by considering just the balance on an account as the attribute of interest. The deposit and withdraw methods provide normal access to the account balance for external users. The getBalance and setBalance methods provide balance access for the derived classes.

All of the following files will be stored in the folder @BankAccount except the test program, which should be in the work directory above the object directories. Listing 18.10 shows the constructor setting up the single attribute, the balance in the account. Listings 18.11, 18.12, 18.13, 18.14, 18.15, and 18.16 show the code for the methods identified in Figure 18.6 together with the standard display(...) and char(...) methods.

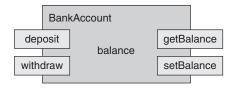


Figure 18.6 ADT for the BankAccount class

## Listing 18.10 BankAccount Constructor

- 1. function acct = BankAccount(data)
- 2. % @BankAccount\BankAccount class constructor.
- 3. % ba = BankAccount(amt) creates a bank account
- with balance amt
- 5. % could also be a copy constructor:
- 6. % ba = BankAccount( oba ) copies the account oba

```
7. if nargin == 0
8.    acct.balance = 0;
9.    acct = class(acct, 'BankAccount');
10. elseif isa(data, 'BankAccount')
11.    acct = data;
12. else
13.    acct.balance = data;
14.    acct = class(acct, 'BankAccount');
15. end
```

#### In Listing 18.10:

Lines 1–6: Show the constructor header.

Lines 7–9: Show the default constructor.

Lines 12–15: Show the real constructor.

#### In Listing 18.11:

Lines 1 and 2: Show the method header.

Line 3: Invokes the setBalance(...) method (Listing 18.14) to add the deposit to the current balance.

Line 4: Returns the new object with the updated balance.

Access to important data like the balance of an account should go through access methods to allow for validation, writing audit trails, and so on to be added in a central location.

## In Listing 18.12:

Lines 1 and 2: Show the method header.

Line 3: In a typical account, withdrawals are limited to the amount in

#### Listing 18.11 The BankAccount deposit method

```
    function deposit(acct, amount)
    % @BankAccount\deposit to the account
    setBalance(acct, acct.balance + amount);
    assignin('caller', inputname(1), acct);
```

## Listing 18.12 The BankAccount withdraw method

```
1. function gets = withdraw(acct, amount)
2. % @BankAccount\withdraw from the account
3.    gets = amount;
4.    if gets > acct.balance
5.        gets = acct.balance;
6.    end
7.    setBalance(acct, acct.balance - gets);
8.    assignin('caller', inputname(1), acct);
```

#### Listing 18.13 BankAccount getBalance method

- 1. function ans = getBalance(ba)
- 2. % @BankAccount\getbalance get the balance
- 3. ans = ba.balance;

#### Listing 18.14 BankAccount setBalance method

- function setBalance(acct, amount)
- 2.% @BankAccount\setBalance set the balance
- 3. acct.balance = amount;
- 4. assignin('caller', inputname(1), acct);

#### Listing 18.15 BankAccount display method

- 1. function display(ba)
- 2. % @BankAccount\disp for the BankAccount class
- 3. % displays string representation of the account
- 4. disp(char(ba));

the account. Consequently, we must limit the amount returned to the current balance. We initialize what the user gets to what he asked for.

Lines 4–6: If this is more money than the user has, give him only the current balance.

Line 7: Sets the new balance.

Line 8: Updates the object.

# In Listing 18.13:

Lines 1 and 2: Show the method header.

Line 3: Shows a simple process of extracting the balance from the object. Although we do not need this method in the parent class, child classes do not have access to the parents' attributes and therefore must use this accessor.

# In Listing 18.14:

Lines 1 and 2: Show the method header.

Line 3: Sets the balance to the given amount.

Line 4: Returns the object to the caller.

# In Listing 18.15:

Lines 1–3: Show the method header.

Line 4: Invokes the char(...) method on the object provided. Every object could operate with the same display(...) method, if there were some way to accomplish this.

#### Listing 18.16 BankAccount char method

```
    function s = char(ba)
    % @BankAccount\char for the BankAccount class
    % returns string representation of the account
    s = sprintf('Account with $%.2f\n', ba.balance);
```

#### In Listing 18.16:

Lines 1–3: Show the method header.

Line 4: Creates a string representing the attributes of this object. This string should be sufficiently generic to enable its use by children classes.

Listing 18.17 shows a typical script to test the behavior of the BankAccount class.

#### In Listing 18.17:

Line 1: Creates a bank account with a \$1,000 initial deposit. With no semicolon, the display(...) method is called to show the result.

Line 2: Shows the \$20.11 deposit. Although there is no semicolon, this is not an assignment and therefore no output is generated.

Line 3: Shows the printout directly using the char(...) method to describe the object.

Line 4: Shows the \$200 withdrawal from the account.

Line 5: Demonstrates that the amount was withdrawn and shows the remaining balance.

Listing 18.18 shows the result from running this test script.

## Listing 18.17 BankAccount test script

#### **Listing 18.18** Test results

```
Account with $1000.00 deposit 20.11 -> Account with $1020.11 withdraw 200 -> $200.00; Account with $820.11
```

## 18.4.2 Inheritance by Extension

Having invested significant effort in preparing the BankAccount class to be extended, writing a SavingsAccount class involves only the constructor and two methods. The SavingsAccount class will have all the characteristics of a BankAccount, plus the ability to calculate interest periodically. For simplicity, we will omit any time-sensitive calculations, and presume that the calcinterest is run only when appropriate. The interest will accumulate at a specified rate, and will be applied only if the account balance exceeds a given minimum balance.

The savingsaccount class and its parent are shown in Figure 18.7. The only new code needed will be for the constructor (Listing 18.19), the method that calculates the interest (Listing 18.20), and the char(...) method to display its content (Listing 18.21). Even that method will invoke the parent char(...) method for most of the work.

All the following files except the test script must be stored in the directory @SavingsAccount. Notice the following important characteristics:

- The calcInterest method does not have to specifically request the getBalance(...) method of the parent; because there is no getBalance method on this class, the parent supplies it.
- Similarly, the test program merely invokes the deposit method. Since the SavingsAccount does not have one, the parent deposit method is used.
- On the other hand, the char(...) method does need to invoke the parent char(...) method explicitly in order to avoid recursive behavior.
- Perhaps most challenging, notice that there is no display(...) method in this class, so the parent's display(...) method is called. However, since it calls a char(...) method, we have to ask: Which char(...) method is invoked? The answer, of course, is that the SavingsAccount char(...) method is used because the object provided is a SavingsAccount first and a BankAccount second.

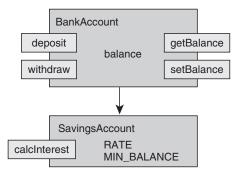


Figure 18.7 The SavingsAccount class

#### Listing 18.19 SavingsAccount Constructor

```
1. function acct = SavingsAccount(data)
 2. % @SavingsAccount\SavingsAccount constructor.
 3. % ba = SavingsAccount(amt) creates an account
                              with balance amt
 5. % could also be a copy constructor:
 6. % ba = SavingsAccount( osa ) copies account osa
 7. if nargin == 0
 8. super = BankAccount;
9.
      acct.RATE = 5;
    acct.MIN BALANCE = 1000;
10.
11. acct = class(acct, 'SavingsAccount', super);
12. elseif isa(data, 'SavingsAccount')
13. acct = data;
14. else
15.
      super = BankAccount(data);
     acct.RATE = 5;
    acct.MIN BALANCE = 1000;
18. acct = class(acct, 'SavingsAccount', super);
19. end
```

#### In Listing 18.19:

Lines 1-6: Show the constructor header.

Line 7: Detects the need for the default constructor logic.

Line 8: Creates a default parent object.

# Hint

You can make a child class containing no local attributes by creating the parent object and then passing the empty vector to the class cast in line 18 as follows:

Lines 9 and 10: Set the default local attributes (constants in this example).

Line 11: Sets the class of the default object with the parent object included.

Lines 12 and 13: Show the copy constructor.

Lines 15–19: Show the real constructor, which differs only in passing the initial balance to the parent constructor.

# In Listing 18.20:

Lines 1–3: Show the method header.

# Listing 18.20 SavingsAccount calcInterest method

- Line 4: You can earn interest only if your balance is above the specified minimum balance.
- Line 5: Calculates the interest earned during this time period.
- Line 7: No interest is earned on small balances.

## In Listing 18.21:

Lines 1–3: Show the method header.

Line 4: Invokes the parent char(...) method for the balance information; this function adds only the account type. We could, of course, extend this to include the local attributes.

The code shown in Listing 18.22 tests the savings account logic.

## In Listing 18.22:

- Line 1: Creates a savings account with an initial deposit of \$2,000.
- This calls the parent display(...) method.
- Line 2: Invokes the parent's deposit method to add another \$3,000.
- Line 3: Displays this account information.
- Line 4: Invokes our calcInterest(...) method.
- Line 5: Deposits this interest back into the account.
- Line 6: Shows the updated information.

The output from this test is shown in Listing 18.23.

# Listing 18.21 SavingsAccount char(...) method

```
    function s = char(sa)
    % @SavingsAccount\char for the SavingsAccount
    % returns string representation of the account
    s=sprintf( 'Savings %s', char(sa.BankAccount) );
```

## Listing 18.22 SavingsAccount tests

# Listing 18.23 SavingsAccount test results

```
Savings Account with $2000.00
deposit 3000 -> Savings Account with $5000.00
deposit interest 250.00
-> Savings Account with $5250.00
```

## 18.4.3 Inheritance by Redefinition

We now consider a further extension of the Bankaccount family—a savingsaccount with a guaranteed overdraft, as shown in Figure 18.8. Once overdraft privileges have been authorized, users of this account can withdraw as much as they want. Of course, if the resulting balance is negative, the bank applies an overdraft charge, thereby taking away even more money than is available.

The coding for this account will follow the general guidelines used above—a new constructor (Listing 18.24), a method for permitting overdrafts to occur (Listing 18.25), and a new char(...) method (Listing 18.26). The data for this class will include a Boolean value indicating that overdraft has been approved for this particular account object. However, this account will also need its own withdraw method to implement the new withdrawal rules (Listing 18.27).

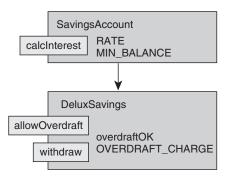


Figure 18.8 The DeluxSavingsAccount class

#### Listing 18.24 DeluxSavingsAccount Constructor

```
1. function acct = DeluxSavingsAccount(data)
 2. % DeluxSavingsAccount class constructor.
 3. % ba = DeluxSavingsAccount(amt) creates an
           account with balance amt
 5. % could also be a copy constructor:
 6. % ba = DeluxSavingsAccount( oda ) copies oda
 7. if nargin == 0
      super = SavingsAccount;
      acct.overdraftOK = false:
10.
      acct.OVERDRAFT CHARGE = 20;
      acct=class(acct,'DeluxSavingsAccount', super);
12. elseif isa(data, 'DeluxSavingsAccount')
13.
      acct = data;
      super = SavingsAccount(data);
     acct.overdraftOK = false;
17.
      acct.OVERDRAFT CHARGE = 20;
18.
      acct=class(acct,'DeluxSavingsAccount', super);
19. end
```

#### In Listing 18.24:

Lines 1-6: Show the constructor header.

Lines 7–11: Show the default constructor defining the local attributes.

Lines 12 and 13: Show the copy constructor.

Lines 15–18: Show the real constructor.

## In Listing 18.25:

Lines 1-3: Show the method header.

Line 4: Sets the overdraft permission attribute to the value provided.

Line 5: Returns the updated object.

## In Listing 18.26:

Lines 1–3: Show the method header.

Lines 4 and 5: Set the output string to 'ok' or 'off'.

Line 6: Creates the string.

## In Listing 18.27:

Lines 3 and 4: Check to see if there is enough money, or if overdraft is allowed.

Lines 5–7: Either way, we can use the parent's withdraw method, but the parent object must be extracted and then replaced.

Line 9: Assumes that the penalty is \$0.

# Listing 18.25 DeluxSavingsAccount allowOverdraft method

```
1. function allowOverdraft(acct, value)
```

- 2. % @DeluxSavingsAccount\allowOverdraft
- 3. % approve overdraft on DeluxSavingsAccount
- 4. acct.overdraftOK = value;
- 5. assignin('caller', inputname(1), acct);

## **Listing 18.26** DeluxSavingsAccount char(...) method

```
    function s = char(sa)
    % @DeluxSavingsAccount\char method
    % returns string representation of the account
```

4. if sa.overdraftOK, strng = 'OK';

5. else, strng = 'off'; end

#### Listing 18.27 Redefined withdraw method

```
1. function gets = withdraw(acct, amount)
 2. % @DeluxSavingsAccount\withdraw method
      if (amount < getBalance(acct))</pre>
 Δ
                      | | ~acct.overdraftOK
 5.
          parent = acct.SavingsAccount;
 6.
          gets = withdraw(parent, amount);
 7.
          acct.SavingsAccount = parent;
 8.
    else
9.
          penalty = 0;
10.
          if amount > getBalance(acct)
              penalty = acct.OVERDRAFT CHARGE;
11.
12.
13.
          setBalance(acct,
14.
                getBalance(acct) - amount - penalty);
15. end
16. assignin('caller', inputname(1), acct);
```

Lines 10–12: However, if the user is asking for more than the balance, the penalty amount applies.

Lines 13 and 14: Update the balance using the parent's getBalance(...) and setBalance(...) methods.

Line 16: Returns the updated object as usual.

Notice the following observations: Like its predecessor, the char(...) method of the DeluxSavingSaccount class lets its parent classes generate much of the string, adding locally only the additional information. Also, the redefined withdraw(...) method invokes the parent's withdraw(...) method if it can, but changes the balance by way of its get and set methods when it has to. In particular, observe that when it can use the parent withdraw(...) method, it only reaches back one level to the SavingSaccount parent. The fact that this parent does not implement withdraw(...) and passes the call to its parent is not a concern for this child class.

Listing 18.28 shows a script to test the DeluxSavingsAccount class.

#### **Listing 18.28** Testing the DeluxSavingsAccount

#### In Listing 18.28:

Line 1: Creates a DeluxSavingsAccount object.

Line 2: Uses the parent's deposit and calcinterest methods.

Line 3: Shows the result.

Line 4: Attempts to withdraw \$3,000 without overdraft enabled.

Line 5: Shows the result—getting only the balance and an empty account.

Line 6 and 7: Deposit another \$500 and show the account.

Line 8: Enables overdrafts.

Line 9: Tries again for the \$3,000.

Line 10: Displays the final result.

Listing 18.29 shows the result of testing the DeluxSavingsAccount class.

#### Listing 18.29 Result of testing the DeluxSavingsAccount

```
Delux Savings Account with $2000.00
with overdraft off
deposit interest -> Delux Savings Account with $2100.00
with overdraft off
try to get 3000 -> $ 2100.00; Delux Savings Account with $0.00
with overdraft off
deposit 500 -> Delux Savings Account with $500.00
with overdraft off
enable overdraft and try again -> $ 3000.00; Delux Savings
Account with $-2520.00
with overdraft OK
```

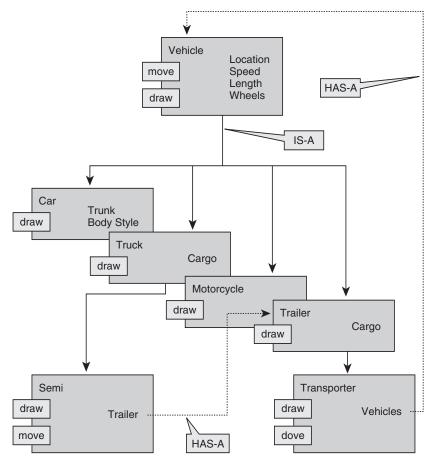


# 18.5 Practical Example—Vehicle Modeling

It would be wrong to leave this discussion without an example of a more practical application of these techniques. Figure 18.9 illustrates the inheritance hierarchy for a collection of vehicles that might be used, for example, in a traffic simulation.

# 18.5.1 A Vehicle Hierarchy

To simulate traffic dynamically, every vehicle must be able to be drawn, and to move in accordance with the rules that govern its behavior. Notice that many of the "simpler" actual vehicles do not contain their own move methods. Given a speed and heading for one of these vehicles, the generic vehicle move method can be applied. However, they all contain their own draw methods, because in general, drawing specific objects requires specific behavior in that draw method.



**Figure 18.9** A vehicle hierarchy

The more complex vehicles (a semi pulling a trailer, for example) do have their own draw methods, however, because in addition to moving the semi, the trailer must be moved in such a way that it remains attached to the semi. Similarly, a transporter trailer containing other vehicles must move all the other vehicles in a manner consistent with their remaining on the trailer.

# 18.5.2 The Containment Relationship

There is one other consideration to note. Any practical hierarchy actually reflects two different relationships between classes. We discussed the inheritance (IS-A) relationship in the previous paragraphs—it is all about inheriting data and methods from a parent class, and is implemented in the child constructor. Now we illustrate the other relationship—that of containment (HAS-A), used to indicate a relationship of possession between objects. Two examples follow:

- 1. A semi class has a trailer attached. There is no inheritance of data or methods between the semi and trailer classes. Rather, one of the attributes peculiar to the semi is a trailer object—that attribute can be accessed by set and get methods, permitting the semi to move and draw its trailer.
- A transporter contains a collection of vehicles. Its move and draw methods must be able to traverse that collection, calling the move or draw methods for the children as appropriate.



# **Chapter Summary**

This chapter presented the object-oriented paradigm, including class categories and MATLAB implementations:

- Fundamentals of Object-Oriented Programming (OOP)
- How MATLAB implements OOP
- Examples of inheritance and polymorphism
- An example of vehicle modeling