Chapter 4

Classes and Methods

Although the terms they use may be different, all object-oriented languages have in common the concepts introduced in Chapter 1: *classes*, *instances*, *message passing*, *methods*, and *inheritance*. As noted already, the use of different terms for similar concepts is rampant in object-oriented programming languages. We will use a consistent and, we hope, clear terminology for all languages, and we will note in language-specific sections the various synonyms for our terms. Readers can refer to the glossary at the end of the book for explanations of unfamiliar terms.

This chapter will describe the definition or creation of classes, and Chapter 5 will outline their dynamic use. Here we will illustrate the mechanics of declaring a class and defining methods associated with instances of the class. In Chapter 5 we will examine how instances of classes are created and how messages are passed to those instances. For the most part we will defer an explanation of the mechanics of inheritance until Chapter 8.

4.1 Encapsulation

In Chapter 1, we noted that object-oriented programming, and objects in particular, can be viewed from many perspectives. In Chapter 2 we described the many levels of abstraction from which one could examine a program. In this chapter, we wish to view objects as examples of *abstract data types*.

Programming that makes use of data abstractions is a methodological approach to problem solving where information is consciously hidden in a small part of a program. In particular, the programmer develops a series of abstract data types, each of which can be viewed as having two faces. This is similar to the dichotomy in Parnas's principles, discussed in Chapter 3. From the outside, a client (user) of an abstract data type sees only a collection of operations that define the behavior of the abstraction. On the other side of the interface, the programmer defining the abstraction sees the data variables that are used to maintain the internal state of the object.

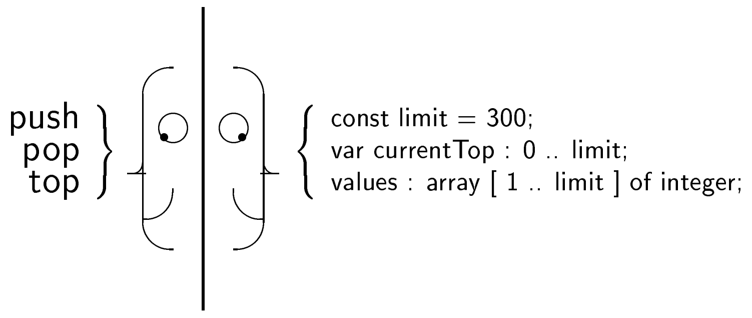

push
pop
top
const limit = 300;
var currentTop : 0 .. limit;
values : array [1 .. limit] of integer;

Figure 4.1: – The interface and implementation faces of a stack.

For example, in an abstraction of a stack data type, the user would see only the description of the legal operations–say, push, pop, and top. The implementor, on the other hand, needs to know the actual concrete data structures used to implement the abstraction (Figure 4.1). The concrete details are encapsulated within a more abstract framework.

We have been using the term *instance* to mean a representative, or example, of a class. We will accordingly use the term *instance variable* to mean an internal variable maintained by an instance. Other terms we will occasionally use are data field, or data members. Each instance has its own collection of instance variables. These values should not be changed directly by clients, but rather should be changed only by methods associated with the class.

A simple view of an object is, then, a combination of *state* and *behavior*. The state is described by the instance variables, whereas the behavior is characterized by the methods. From the outside, clients can see only the behavior of objects; from the inside, the methods provide the appropriate behavior through modifications of the state as well as by interacting with other objects.

4.2 Class Definitions

Throughout this chapter and the next we will use as an example the development of a playing card abstraction, such as would be used in a card game application. We will develop this abstraction through a sequence of refinements, each refinement incorporating a small number of new features.

We start by imagining that a playing card can be abstracted as a container for two data values; the card rank and card suit. We can use a number between 1 and 13 to represent the rank (1 is ace, 11, 12 and 13 are Jack, Queen and King). To represent the suit we can use an enumerated data type, if our language provides such facilities. In languages that do not have enumerated data types we can use symbolic constants and integer values from 1 to 4. (The advantage of the enumerated data type is that type errors are avoided, as we can guaranteed the suit is one of the four specified values. If we use integers for this purpose than nothing prevents a programmer from assigning an invalid integer number, for example 42, to the suit variable.)

4.2.1 C++, Java and C#

We begin by looking at class definitions in three very similar languages; C++, Java and C#. The syntax used by these three languages in shown in Figure 4.2. There are some superficial differences, for example a class definition is terminated by a semicolon in C++, and not in the other two. Visibility modifiers (that is, public) mark an entire block of declarations in C++, and are placed on each declaration independently in the other two languages. In C++ and C# a programmer can define an enumerated data type for representing the playing card suits. By placing the definition inside the class in C++ the programmer makes clear the link between the two data types. (This is not possible in C#). Outside of the class definition the symbolic constants that represent the suits must be prefixed by the class name, as in C++:

```
if (aCard.suit() == PlayingCard::Diamond) ...
```

or by the type name as in C#:

```
if (aCard.suit() == Suits.Diamond) ...
```

Here aCard is the name of an instance of Playing Card, and we are invoking the method named suit in order to test the suit of the card. The data fields suitValue and rankValue represent the instance data for this abstraction. Each instance of the class PlayingCard will have their own separate fields, maintaining their own suit and rank values. Notice that the value of the suit is obtained by invoking a method named suit, which simply returns the data field named suitValue.

The first letter of the class name has here been capitalized. This is a convention that is followed by many langauges, although not universally (in particular, many C++ programmers prefer to use names that are all small letters). Normally instance variables are given names that are not capitalized, so as to make it easier to distinguish between the two categories. Some languages, such as Delphi Pascal, have other conventions.

Enumerated data types are not provided by the language Java, and so the programmer typically resorts to defining a series of symbolic constants. A symbolic constant is characterized by the two modifiers final and static. In Java the modifier final means that the assignment of the name cannot subsequently be changed. The modifier static means that there exists only one instance of a variable, regardless of how many instances of the class are created. Taken together, the two define a unique variable that cannot change; that is, a constant.

C++

```
class PlayingCard {
public:
    enum Suits {Spade, Diamond, Club, Heart};

    Suits suit () { return suitValue; }
    int   rank () { return rankValue; }

private:
    Suits suitValue;
    int   rankValue;
};
```

Java

```
class PlayingCard {
    public  int suit () { return suitValue; }
    public  int rank () { return rankValue; }

    private int suitValue;
    private int rankValue;

    public static final int Spade = 1;
    public static final int Diamond = 2;
    public static final int Club = 3;
    public static final int Heart = 4;
}
```

C#

```
enum Suits {Spade, Diamond, Club, Heart};

class PlayingCard {
    public Suits suit () { return suitValue; }
    public int   rank () { return rankValue; }

    private Suits suitValue;
    private int   rankValue;
}
```

Figure 4.2: A Simple Class Definition in C++, Java and C#

Note the essential difference between the data fields suitValue and rankValue and the constants Heart, Spade, Diamond and Club in the Java definition. Because the latter are declared as static they exist outside of any one instance of the class, and are shared by all instances. The suit and rank fields, on the other hand, are not static and hence each instance of the class will have their own copy of these values.

Visibility Modifiers

Note the use of the terms public and private in several places in these examples. These are *visibility* modifiers. All three languages, as well as a number of other object-oriented programming languages, provide a way of describing features that are known and can be used *outside* the class definition, and distinguishing those from features that can only be used *within* a class definition. The later are indicated by the keyword private.

4.2.2 Apple Object Pascal and Delphi Pascal

The next two languages we will consider are also very similar. Both Apples Object Pascal language and Borlands Delphi Pascal (called Kylix on Linux platforms) were based on an earlier language named simply Pascal. Thus, many features derived from the original language are the same. However, the two vendors have extended the language in slightly different ways.

Both languages permit the creation of enumerated data types, similar to C++. The Apple version of the language uses the keyword object to declare a new class (and hence classes are sometimes termed *object types* in that language). The Delphi language uses the keyword class, and furthermore requires that every class inherit from some existing class. We have here used the class TObject for this purpose. It is conventional in Delphi that all classes must have names that begin with the letter T. Delphi uses visibility modifiers, the Apple language does not. Finally the Delphi language requires the creation of a constructor, a topic we will return to shortly.

4.2.3 Smalltalk

Smalltalk does not actually have a textual representation of a class. Instead, classes are described using an interactive interface called the browser. A screen shot of the browser is shown in Figure 4.4. Using the browser, the programmer can define a new class using a message sent to the parent class Object. As in Delphi Pascal, all classes in Smalltalk must name a specific parent class from which they will inherit. Figure 4.4 illustrates the creation of the class PlayingCard with two instance data fields.

Object Pascal

```
type
    Suits = (Heart, Club, Diamond, Spade);

    PlayingCard = object
        suit : Suits;
        rand : integer;
    end;
```

Delphi Pascal

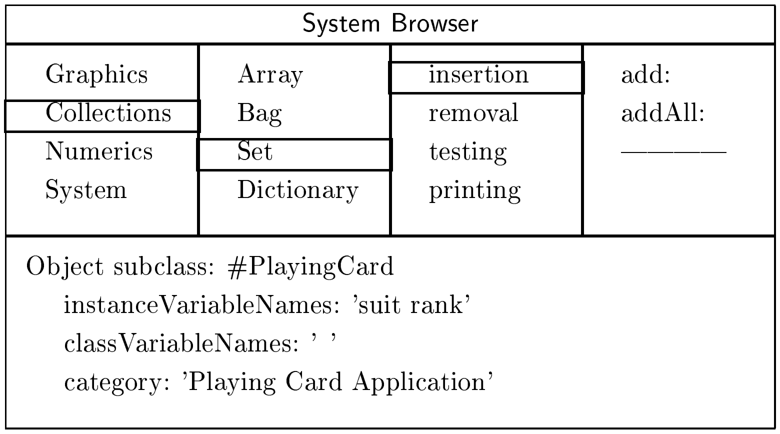
```
type
    Suits = (Heart, Club, Diamond, Spade);

    TPlayingCard = class (TObject)
        public
            constructor Create (r : integer; s : Suits);

            function suit : Suits;
            function rank : int;

        private
            suitValue : Suits;
            rankValue : integer;
    end;
```

Figure 4.3: Class Definitions in Object Pascal and Delphi Pascal



System Browser

Graphics	Array	insertion	add:
Collections	Bag	removal	addAll:
Numerics	Set	testing	————
System	Dictionary	printing	

```
Object subclass: #PlayingCard
    instanceVariableNames: 'suit rank'
    classVariableNames: ' '
    category: 'Playing Card Application'
```

Figure 4.4: – A view of the Smalltalk browser.

CLOS	`(defclass PlayingCard ( ) (rank suit) )`

Eiffel	`class PlayingCard`
	`feature`
	`    Spade, Diamond, Heart, Club : Integer is Unique;`
	
	`    suit : integer;`
	`    rank : integer;`
	`end`

Objective-C	`enum suits {Heart, Club, Diamond, Spade};`
	
	`@ interface PlayingCard : Object`
	`{`
	`    suits suit;`
	`    int rank;`
	`}`
	`@end`

Python	`class PlayingCard:`
	`    "A playing card class"`
	`    def __init__ (self, s, r):`
	`        self.suit = s`
	`        self.rank = r`

Figure 4.5: Class Definitions in Other Object-Oriented Languages

4.2.4 Other Languages

We will periodically throughout the book refer to a number of languages, particularly when they include features that are unique or not widely found in alternative languages. Some of these include Objective-C, CLOS, Eiffel, Dylan and Python. Class definitions for some of these are shown in Figure 4.5. Python is interesting in that indentation levels, rather than beginning and ending tokens, are used to indicate class, function and statement nesting.

4.3 Methods

In the next revision of our playing card abstraction we make the following changes:

- We add a method that will return the face color of the card, either red or black.

```
class PlayingCard {
        // constructor, initialize new playing card
    public PlayingCard (Suits is, int ir)
        { suit = is; rank = ir; faceUp = true; }

        // operations on a playing card
    public boolean isFaceUp  ()           { return faceUp; }
    public int     rank      ()           { return rankValue; }
    public Suits   suit      ()           { return suitValue; }
    public void    setFaceUp (boolean up) { faceUp = up; }
    public void    flip      ()           { setFaceUp( !faceUp);}
    public Color   color     ()           {
        if ((suit() == Suits.Diamond) || (suit() == Suits.Heart))
            return Color.Red;
        return Color.Black;
    }
        // private data values
    private Suits suitValue;
    private int rankValue;
    private boolean faceUp;
}
```

Figure 4.6: The Revised PlayingCard class in C#

- We add a data field to maintain whether the card is face up or face down, and methods both to test the state of this value and to flip the card.

A typical class that illustrates these changes is the C# definition shown in Figure 4.6. Some features to note are that we have added a second enumerated data type to represent the colors, and the data fields (including the third data field representing the face up state of the card) are declared private. By declaring the data fields private it means that access outside the class definition is not permitted. This guarantees that the only way the data fields will be modified is by methods associated with the class. Most object-oriented style guidelines will instruct that data fields should never be declared public, and should always be private or protected, the latter a third level of protection we will discuss after we introduce inheritance in Chapter 8.

The constructor is a special method that has the same name as the class, and is used to initialize the data fields in an object. As we noted earlier, we will discuss constructors in more detail in the next chapter.

Where access to data fields must be provided, good object-oriented style says that access should be mediated by methods defined in the class. A method that does nothing more than return the value of a data field is termed an *accessor*, or sometimes a *getter*. An example is the method isFaceUp, which returns the value of the data field faceUp. Another example is the method rank, which return the value of the rankValue data field.

Why is it better to use a method for this simple action, rather than permitting access to the data field directly? One reason is that the method makes the data field *read-only*. A function can only be called, whereas a data field can be both read and written. By the combination of a private data field and a public accessor we ensure the rank of the playing card cannot change once it has been created.

The naming conventions for the methods shown here are typical. It is good practice to name a method that returns a boolean value with a term that begins with is and indicates the meaning when a true value is returned. Following this convention makes it easy to understand the use of the method in a conditional statement, such as the following:

```
    if (aCard.isFaceUp()) ...
```

Here aCard is once again an instance of class PlayingCard. Many style guidelines suggest that all other accessor methods should begin with the word get, so as to to most clearly indicate that the most important purpose of the method is to simply get the value of a data field. Again this convention makes it easy to understand statements that use this method:

```
    int cardRank = aCard.getRank();
```

However, this convention is not universally advocated. In particular, we will continue to use the simpler names rank and suit for our methods.

Methods whose major purpose is simply to set a value are termed *mutator methods* or *setters*. As the name suggests, a setter most generally is named beginning with the word set. An example setter is the method setFaceUp which sets the value for the faceUp accessor:

```
class PlayingCard {
    ...
    void setFaceUp (boolean up) { faceUp = up; }
    ...
}
```

The method flip is neither a getter nor a setter, since it neither gets nor sets a data field. It is simply a method. The method color is not technically a getter, since it is not getting a data field held by the class. Nevertheless, because it is returning an attribute of the object, some style guidelines would suggest that a better name would be getColor.

Visibility modifiers are not found in the language Smalltalk. By default all data fields are private, that is accessible only within the class definition itself. To allow access to a data field an accessor method must be provided:

```
    rank
    " return the face value of a card "
    ↑ rank
```

The convention of using get names is not widely followed in Smalltalk. Instead, it is conventional for accessor methods to have the same name as the data field they are returning. No confusion arises in the Smalltalk system; we say nothing of confusion that can arise in the programmers mind.

4.3.1 Order of Methods in a Class Declaration

For the most part, programming languages do not specify the order that methods are declared within a class definition. However, the order can have a significant impact on readability, an issue that is often of critical importance to programmers. Many style guidelines offer differing advice on this issue, often conflicting with each other. The following are some of the most significant considerations:

- Important features should be listed earlier in the class definition, less important features listed later.
- Constructors are one of the most important aspects of an object definition, and hence should appear very near the top of a class definition.
- The declaration of methods should be grouped so as to facilitate rapidly finding the body associated with a given message selector. Ways of doing this include listing methods in alphabetical order, or grouping methods by their purpose.
- Private data fields are mostly important only to the class developer. They should be listed near the end of a class definition.

4.3.2 Constant or Immutable Data Fields

As an alternative to accessor methods, some programming languages provide a way to specify that a data field is constant, or *immutable*. This means that once set the value of the data field can not subsequently be changed. With this restriction there is less need to hide access to a data value behind a method.

Two different ways of describing constant data fields are shown in Figure 4.7. Such a field is declared as final in Java. The modifier const is used in C++ for much the same purpose.

4.3.3 Separating Definition and Implementation

Some languages, such as Java and C#, place the body of a method directly in the class definition, as shown in Figure 4.6. Other languages, such as C++ and Object Pascal, separate these two aspects. In C++ the programmer has a choice. Small methods can be defined in the class, while larger methods are

C++	`class PlayingCard {`
	`public:`
	`    ....`
	`    const int rank; // since immutable, can allow`
	`    const Suits suit; // public access to data field`
	`};`

Java	`class PlayingCard {`
	`    ....`
	`    public final int rank;`
	`    public final int suit;`
	`}`

Figure 4.7: Syntax for Defining Immutable Data Values

defined outside. A C++ class definition for our playing card abstraction might look something like the following:

```
class PlayingCard {
public:
        // enumerated types
    enum Suits {Spade, Diamond, Club, Heart};
    enum Colors {Red, Black};

        // constructor, initialize new playing card
    PlayingCard (Suits is, int ir)
        { suit = is; rank = ir; faceUp = true; }

        // operations on a playing card
    boolean isFaceUp  ()        { return faceUp; }
    void    setFaceUp (bool up) { faceUp = up; }
    void    flip      ()        { setFaceUp( ! faceUp); }
    int     rank      ()        { return rankValue; }
    Suits   suit      ()        { return suitValue; }
    Colors  color     () ;
private:        // private data values
    Suits suitValue;
    int rankValue;
    boolean faceUp;
};
```

Notice the body of the method color has been omitted, as it is longer than the other methods defined in this class. A subsequent method definition (sometimes termed a *function member*) provides the body of the function:

```
PlayingCard::Colors PlayingCard::color ( )
{
        // return the face color of a playing card
    if ((suit == Diamond) || (suit == Heart))
        return Red;
    return Black;
}
```

The method heading is very similar to a normal C style function definition, except that the name has been expanded into a *fully-qualified* name. The qualified name provides both the class name and the method name for the method being defined. This is analogous to identifying a person by both their given and family names (for example, "Chris Smith").

C++ programmers have the choice between defining methods in-line as part of the class definition, or defining them in a separate section of the program. Typically only methods that are one or two statements long are placed in-line, and anything more complex than one or two lines is defined outside the class.

There are two reasons for playing a method body outside the class definition. Method bodies that are longer than one statement can obscure other features of the class definition, and thus removing long method bodies can improve readability. (Readability, however, is in the eye of the beholder. Not all programmers think that this separation improves readability, since the programmer must now look in two different places to find a method body). A second reason involves semantics. When method bodies are declared within a class definition a C++ compiler is permitted (although not obligated) to expand invocations of the method directly in-line, without creating a function call. An inline definition can be executed much faster than the combination of function call and method body.

Often the class definition and the larger method bodies in a C++ program will not even be found in the same file. The class heading will be given in an *interface file* (by convention a file with the extension .h on Unix systems, or .hpp on Windows systems) while the function bodies will be found in an implementation file (by convention a file with the extension .cpp or .C).

Objective-C also separates a class definition from a class implementation. The definition includes a description of methods for the class. These are indicated by a + or − sign followed by the return type in parenthesis followed by a description of the method:

```
@ interface PlayingCard : Object
{
    int suit;
    int rank;
    int faceUp;
}

+ suit: (int) s rank: (int) i
- (int) color;
- (int) rank;
- (int) suit;
- (int) isFaceUp;
- (void) flip;
@ end
```

The implementation section then provides the body of the methods:

```
@ implementation PlayingCard

- (int) color
{
    if ((suit == Diamond) || (suit == Heart))
        return red;
    return black;
}

- (int) rank
{
    return rank;
}
    ... ./* other method bodies */
@ end
```

Object Pascal and Delphi similarly separate the class definition from the method function bodies, however the two parts remain in the same file. The class definitions are described in a section labeled with the name interface, while the implementations are found in a section labeled, clearly enough, implementation. The following is a Delphi example:

```
interface

type
    Suits = (Heart, Club, Diamond, Spade);

    Colors = (Red, Black);

    TPlayingCard = class (TObject)
        public
            constructor Create (r : integer; s : Suits);
            function color : Colors;
            function isFaceUp : boolean;
            procedure flip;
            function rank : integer;
            function suit : Suits;
        private
            suit : Suits;
            rank : integer;
            faceUp : boolean;
    end;

implementation
    function TPlayingCard.color : Colors;
    begin
        case suit of
            Diamond: color := Red;
            Heart: color := Red;
            Spade: color := Black;
            Club: color := Black;
    end

    ... (* other methods similarly defined *)
end.
```

Note that fully qualified names in Pascal are formed using a period between the class name and the method name, instead of the double colon used by C++.

In CLOS accessor functions can be automatically created when a class is defined, using the :accessor keyword followed by the name of the accessor function:

```
    (defclass PlayingCard ()
        ((rank :accessor getRank) (suit :accessor getSuit) ))
```

Other methods are defined using the function defmethod. Unlike Java or C++, the receiver for the method is named as an explicit parameter:

```
    (defmethod color ((card PlayingCard))
        (cond
            ((eq (getSuit card) 'Diamond) 'Red)
            ((eq (getSuit card) 'Heart) 'Red)
            (t 'Black)))
```

The receiver must also be named as an explicit parameter in Python:

```
class PlayingCard:
    "A playing card class"
    def __init__ (self, s, r):
        self.suit = s
        self.rank = r
    def rank (self)
        return self.rank
    def color (self)
        if self.suit == 1 or self.suit == 2
            return 1
        return 0
```

4.4 Variations on Class Themes*

While the concept of a class is fundamental to object-oriented programming, some languages go further in providing variations on this basic idea. In the following sections we will describe some of the more notable among these variations.

4.4.1 Methods without Classes in Oberon

The language Oberon does not have classes in the sense of other object oriented languages, but only the more traditional concept of data records. Nevertheless, it does support message passing, including many of the dynamic method binding features found in object-oriented langauges.

A method in Oberon is not defined inside a record, but is instead declared using a special syntax where the receiver is described in an argument list separately from the other arguments. Often the receiver is required to be a pointer type, rather than the data record type:

```
TYPE
    PlayingCard = POINTER TO PlayingCardDesc;

    PlayingCardDesc = RECORD
        suit : INTEGER;
        rank : INTEGER;
        faceUp: BOOLEAN;
    END

PROCEDURE (aCard: PlayingCard) setFaceUp (b : BOOLEAN);
BEGIN
    aCard.faceUp = b;
END
```

[0]Section headings followed by an asterisk indicate optional material. Instructors should feel free to select subsections of this section that seem most appropriate for their own situation.

The record PlayingCardDesc contains the data fields, which can be modified by the procedure setFaceUp, which must take a pointer to a playing card as a receiver.

4.4.2 Interfaces

Some object-oriented languages, such as Java, support a concept called an *interface*. An interface defines the protocol for certain behavior but does not provide an implementation. The following is an example interface, describing objects that can read from and write to an input/output stream.

```
public interface Storing {
    void writeOut (Stream s);
    void readFrom (Stream s);
};
```

Like a class, an interface defines a new type. This means that variables can be declared simply by the interface name.

```
    Storing storableValue;
```

A class can indicate that it implements the protocol defined by an interface. Instances of the class can be assigned to variables declared as the interface type.

```
public class BitImage implements Storing {
    void writeOut (Stream s) {
        // ...
    }
    void readFrom (Stream s) {
        // ...
    }
};

    storableValue = new BitImage();
```

The use of interfaces is very similar to the concept of inheritance, and thus we will return to a more detailed consideration of interfaces in Chapter 8.

4.4.3 Properties

Delphi, Visual Basic, C# and other programming languages (both object-oriented and not) incorporate an idea called a property. A property is manipulated syntactically in the fashion of a data field, but operates internally like a method. That is, a property can be read as an expression, or assigned to as a value:

```
    writeln ('rank is ', aCard.rank); (* rank is property of card *)
    aCard.rank = 5; (* changing the rank property *)
```

However, in both cases the value assigned or set will be mediated by a function, rather than a simple data value. In Delphi a property is declared using the keyword property and the modifiers read and write. The values following the read and write keyword can be either a data field or a method name. The read attribute will be invoked when a property is used in the fashion of an expression, and the write attribute when the property is the target of an assignment. Having a read attribute and no write makes a property read only. We could recast our rank and suit values as properties as follows:

```
type
    TPlayingcard = class (TObject)
        public
            ...
            property rank : Integer read rankValue;
            property suit : Suits read suitValue write suitValue;
        private
            rankValue : Integer;
            suitValue : Suits;
    end;
```

Here we have made rank read only, but allowed suit to be both read and written. It is also possible to make a property write-only, although this is not very common.

In C# a property is defined by writing a method without an argument list, and including either a get or a set section.

```
public class PlayingCard {
    public int rank {
        get
        {
            return rankValue;
        }
        set
        {
            rankValue = value;
        }
    }
    ...
    private int rankValue;
}
```

A get section must return a value. A set section can use the pseudo-variable value to set the property. If no set section is provided the property is read-only. If no get section is given the property is write only. Properties are commonly used in C# programs for functions that take no arguments and return a value.

4.4.4 Forward Definitions

A program can sometimes require that two or more classes each have references to the other. This situation is termed *mutual recursion*. We might need to represent the horse and buggy trade, for example, where every horse is associated with their own buggy, and every buggy with one horse. Some languages will have little trouble with this. Java, for example, scans an entire file before it starts to generate code, and so classes that are referenced later in a file can be used earlier in a file with no conflict.

Other languages, such as C++, deal with classes and methods one by one as they are encountered. A name must have at least a partial definition before it can be used. In C++ this often results in the need for a *forward definition*. A definition that serves no other purpose than to place a name in circulation, leaving the completion of the definition until later. Our horse and buggy example, for instance, might require something like the following:

```
class Horse; // forward definition

class Buggy {
    ...
    Horse * myHorse;
};

class Horse {
    ...
    Buggy * myBuggy;
};
```

The first line simply indicates that Horse is the name of a class, and that the definition will be forthcoming shortly. Knowing only this little bit of information, however, is sufficient for the C++ compiler to permit the creation of a pointer to the unknown class.

Of course, nothing can be done with this object until the class definition has been read. Solving this problem requires a careful ordering of the class definitions and the the implementations of their associated methods. First one class definition, then the second class definition, then methods from the first class, finally methods from the second.

4.4.5 Inner or Nested Classes

Both Java and C++ allow the programmer to write one class definition inside of another. Such a definition is termed an *inner class* in Java, and a *nested class* in C++. Despite the similar appearances, there is a major semantic difference between the two concepts. An inner class in Java is linked to a specific instance of the surrounding class (the instance in which it was created), and is permitted access to data fields and methods in this object. A nested class in C++ is simply a naming device, it restricts the visibility of features associated with the inner class, but otherwise the two are not related.

To illustrate the use of nested classes, let us imagine that a programmer wants to write a doubly linked list abstraction in Java. The programmer might decide to place the Link class inside the List abstraction:

```
    // Java List class
class List {
    private Link firstElement = null;

    public void push_front(Object val)
    {
        if (firstElement == null)
            firstElement = new Link(val, null, null);
        else
            firstElement.addBefore (val);
    }

    ... // other methods omitted

    private class Link { // inner class definition
        public Object value;
        public Link forwardLink;
        public Link backwardLink;

        public Link (Object v, Link f, Link b)
            { value = v; forwardLink = f; backwardLink = b; }

        public void addBefore (Object val)
        {
            Link newLink = new Link(val, this, backwardLink);
            if (backwardLink == null)
                firstElement = newLink;
            else {
                backwardLink.forwardLink = newLink;
                backwardLink = newLink;
            }
        }
        ... // other methods omitted
    }
}
```

Note that the method addBefore references the data field firstElement, in order to handle the special case where an element is being inserted into the front of a list. A direct translation of this code into C++ will produce the following:

```
    // C++ List class
class List {
private:
    class Link;    // forward definition
    Link * firstElement;

    class Link { // nested class definition
    public:
        int value;
        Link * forwardLink;
        Link * backwardLink;

        Link (int v, Link * f, Link * b)
            { value = v; forwardLink = f; backwardLink = b; }

        void addBefore (int val)
        {
            Link * newLink = new Link(val, this, backwardLink);
            if (backwardLink == 0)
                firstElement = newLink; // ERROR !
            else {
                backwardLink->forwardLink = newLink;
                backwardLink = newLink;
            }
        }

        ... // other methods omitted
        };


public:
    void push_front(int val)
    {
        if (firstElement == 0)
            firstElement = new Link(val, 0, 0);
        else
```

```
            firstElement->addBefore (val);
    }
    ... // other methods omitted
};
```

It has been necessary to introduce a forward reference for the Link class, so that the pointer firstElement could be declared before the class was defined. Also C++ uses the value zero for a null element, rather than the pseudo-constant null. Finally links are pointers, rather than values, and so the pointer access operator is necessary. But the feature to note occurs on the line marked as an error. The class Link is not permitted to access the variable firstElement, because the scope for the class is not actually nested in the scope for the surrounding class. In order to access the List object, it would have to be explicitly available through a variable. In this case, the most reasonable solution would probably be to have the List method pass itself as argument, using the pseudo-variable this, to the inner Link method addBefore. (An alternative solution, having each Link maintain a reference to its creating List, is probably too memory intensive).

```
class List {
    Link * firstElement;

    class Link {
        void addBefore (int val, List * theList)
        {
            ...
            if (backwardLink == 0)
                theList->firstElement = newLink;
            ...
        }
    };
public:
    void push_front(int val)
    {
            ...
                // pass self as argument
            firstElement->addBefore (val, this);
    }
    ... // other methods omitted
};
```

When nested class methods are defined outside the class body, the name may require multiple levels of qualification. The following, for example, would be how the method addBefore would be written in this fashion:

```
    void List::Link::addBefore (int val, List * theList)
    {
        Link * newLink = new Link(val, this, backwardLink);
        if (backwardLink == 0)
            theList->firstElement = newLink;
        else {
            backwardLink->forwardLink = newLink;
            backwardLink = newLink;
        }
    }
```

The name of the function indicates that this is the method addBefore that is part of the class Link, which is in turn defined as part of the class List.

4.4.6 Class Data Fields

In many problems it is useful to have a common data field that is shared by all instances of a class. However, the manipulation of such an object creates a curious paradox for the object oriented language designer. To understand this problem, consider that the reason for the invention of the concept of a class was to reduce the amount of work necessary to create similar objects; every instance of a class has exactly the same behavior as every other instance. Now imagine that we have somehow defined a common data area shared by all instances of a class, and think about the task of initializing this common area. There seem to be two choices, neither satisfactory. Either everybody performs the initialization task (and the field is initialized and reinitialized over and over again), or nobody does (leaving the data area uninitialized).

Resolving this paradox requires moving outside of the simple class/method/instance paradigm. Another mechanism, not the objects themselves, must take responsibility for the initialization of shared data. If objects are automatically initialized to a special value (such as zero) by the memory manager, then every instance can test for this special value, and perform initialization if they are the first. However, there are other (and better) techniques.

In both C++ and Java shared data fields are created using the static modifier. We have seen a use of this already in the creation of symbolic constants in Java. In Java the intialization of a static data field is accomplished by a *static block*, which is executed when the class is loaded. For example, suppose we wanted to keep track of how many instances of a class have been created:

```
class CountingClass {

    CountingClass () {
        count = count + 1; // increment count
        ...
    }
    ...

    private static int count; // shared by all

    static {     // static block
        count = 0;
    }
}
```

In C++ there are two different mechanisms. Data fields that are static (or const) and represented by primitive data types can be initialized in the class body, as we have seen already. Alternatively, a global initialization can be defined that is separate from the class:

```
class CountingClass {
public:
    CountingClass () { count++; ... }

private:
    static int count;
};

// global initialization is separate from class
int CountingClass::count = 0;
```

In C# static data fields can be initialized by a static constructor, a constructor method that is declared static. This constructor is not permitted to have any arguments.

In Python class data fields are simply named at the level of methods, while instance variables are named inside of methods (typically inside the constructor method):

```
class CountingClass:
    count = 0
    def __init__ (self)
        self.otherField = 3
```

4.4.7 Classes as Objects

In a number of languages (Smalltalk, Java, many others) a class is itself an object. Of course, one must then ask what class represents the category to which this object belongs, that is, what class is the class? In most cases there is a special class, typically Class, that is the class for classes.

Since objects are classes, they have behavior. What can you do with a class? Frequently the creation of an instance of the class is simply a message given to a class object. This occurs in the following example from Smalltalk, for instance:

```
aCard <- PlayingCard new. "message new given to object PlayingCard"
```

Other common behaviors include returning the name of the class, the size of instances of the class, or a list of messages that instances of the class will recognize. The following bit of Java code illustrates one use:[1]

```
    Object obj = new PlayingCard();
    Class c = obj.getClass();
    System.out.println("class is " + c.getName());
PlayingCard
```

We wil return to an exploration of classes as objects when we investigate the concept of *reflection* in Chapter 25.

Chapter Summary

In this chapter we have started our exploration of the concept of *class* in object-oriented languages. We have described the syntax for class and method definitions in various languages, including Java, C++, C#, Object Pascal, Objective-C, and Eiffel. Throughout the text we will occasional refer to other example languages as well.

Some of the features of classes that we have seen in this chapter include the following:

- Visibility modifiers. The keywords public and private that are used to control the visibility, and hence the manipulation, of class features.
- Getter and Setter functions. Sometimes termed accessors and mutators, these are methods that provide access to data fields. By using methods rather than providing direct access, programmers have greater control over the way data is modified, and where it can used.
- Constant, or immutable data fields. Data fields that are guaranteed to not change during the course of execution.
- Interfaces. Class like entities that describe behavior, but do not provide an implementation.

[1]In non interactive languages it is sometimes difficult to show the relationship between program statements and their output. Throughout the rest of the book we will use the convention illustrated by this example, indenting a sequence of statement and then showing the resulting output without indentation. The reader will hopefully be able to distinguish the executable statements from the non-executable output.

- Nested classes. Class definitions that appear inside of other class definitions.

- Class data fields. The particular paradox that arises over the initialization data fields that are shared in common among all instances of a class.

Further Reading

The Apple Object Pascal language was original defined by Larry Tesler of Apple Computer [Tesler 1985]. The Borland language was originally known as Turbo Pascal [Turbo 1988]. More recent descriptions of the Delphi version of this language can be found in [Lischner 2000, Kerman 2002].

The classic definition of the language Smalltalk is [Goldberg 1983]. More recent treatments of the language include [LaLonde 1990b, Smith 1995]. A popular public-domain version of Smalltalk is Squeak [Guzdial 2001].

The Java language is described in [Arnold 2000]. A Good tutorial on Java can be found in [Campione 1998]. In Java and C++ the concept of *interfaces* is closely related to the concept of a class. We will discuss interfaces when we examine inheritance in Chapter 8. A good style guidebook for Java programmers is [Vermeulen 2000].

Since C# is a relatively recent language there are still only a few published references. Two recent sources are [Gunnerson 2000, Albahari 2001].

Objective-C was created as an extension to C at about the same time that C++ was developed. A good introduction Objective-C is the book written by its creator, Brad Cox [Cox 1986]. Python is described in [Beazley 2000].

An interesting question is whether classes are necessary for object-oriented programming. It turns out that you can achieve much of the desirable characteristics of object-oriented languages without using classes, by means of an idea termed *delegation* [Lieberman 1986]. However, in the period between when delegation languages were first proposed and the present there has not been a ground-swell of support for this idea, so most people seem to prefer classes.

Self Study Questions

1. What is the difference between a class declaration and an object declaration (the latter also known as an instantiation)?

2. What is an instance variable?

3. What are the two most basic aspects of a class?

4. What is the meaning of the modifiers `final` and `static` in Java? How do these two features combine to form a symbolic constant?

5. What does the term `public` mean? What does the term `private` mean?

6. What is a constructor?

7. What is an accessor method? What is the advantage of using accessor methods instead of providing direct access to a data field?

8. What is a mutator, or setter method?

9. What are some guidelines for selecting the order of features in a class definition?

10. What is an immutable data field?

11. What is a fully qualified name?

12. How is an interface different from a class? How is it similar?

13. What is an inner or nested class?

14. Explain the paradox arising the the initialization of common data fields or class data fields.

Exercises

1. Suppose you were required to program in a non-object-oriented language, such as Pascal or C. How would you simulate the notion of classes and methods?

2. In Smalltalk and Objective-C, methods that take multiple arguments are described using a keyword to separate each argument; in C++ the argument list follows a single method name. Describe some of the advantages and disadvantages of each approach; in particular, explain the effect on readability and understandability.

3. A digital counter is a bounded counter that turns over when its integer value reaches a certain maximum. Examples include the numbers in a digital clock and the odometer in a car. Define a class description for a bounded counter. Provide the ability to set maximum and minimum values, to increment the counter, and to return the current counter value.

4. Write a class description for complex numbers. Write methods for addition, subtraction, and multiplication of complex numbers.

5. Write a class description for a fraction, a rational number composed of two integer values. Write methods for addition, subtraction, multiplication, and division of fractions. How do you handle the reduction of fractions to lowest-common-demoninator form?

6. Consider the following two combinations of class and function in C++. Explain the difference in using the function addi as the user would see it.

```
class example1 {
public:
    int i;
};

int addi(example1 & x, int j)
{
    x.i = x.i + j;
    return x.i;
}


class example2 {
public:
    int i;
    int addi(int j)
        { i = i + j; return i; }
};
```

7. In both the C++ and Objective-C versions of the playing card abstraction, the modular division instruction is used to determine the color of a card based on the suit value. Is this a good practice? Discuss a few of the advantages and disadvantages. Rewrite the methods to remove the dependency on the particular values associated with the suits.

8. Do you think it is better to have the access modifiers private and public associated with every individual object, as in Java, or used to create separate areas in the declaration, as in C++, Objective-C, and Delphi Pascal? Give reasons to support your view.

9. Contrast the encapsulation provided by the class mechanism with the encapsulation provided by the module facility. How are they different? How are they the same?