Chapter 5

Messages, Instances, and Initialization

In Chapter 4 we briefly outlined some of the compile-time features of object-oriented programming languages. That is, we described how to create new types, new classes, and new methods. In this chapter, we continue our exploration of the mechanics of object-oriented programming by examining the *dynamic* features. These include how values are instantiated (or created), how they are initialized, and how they communicate with each other by means of message passing.

In the first section, we will explore the mechanics of message passing. Then we will investigate creation and initialization. By *creation* we mean the allocation of memory space for a new object and the binding of that space to a name. By *initialization* we mean not only the setting of initial values in the data area for the object, similar to the initialization of fields in a record, but also the more general process of establishing the initial conditions necessary for the manipulation of an object. The degree to which this latter task can be hidden from clients who use an object in most object-oriented languages is an important aspect of *encapsulation*, which we identified as one of the principle advantages of object-oriented techniques over other programming styles.

5.1 Message-Passing Syntax

We are using the term *message passing* (sometimes also called *method lookup*) to mean the dynamic process of asking an object to perform a specific action. In Chapter 1 we informally described message passing and noted how a message differs from an ordinary procedure call. In particular:

- A *message* is always given *to* some object, called the *receiver*.
- The action performed in response to the message is not fixed, but may differ depending upon the class of the receiver. That is, different objects

C++, C#, Java, Python, Ruby	`aCard.flip ();`
	`aCard.setFaceUp(true);`
	`aGame.displayCard(aCard, 45, 56);`

Pascal, Delphi, Eiffel, Oberon	`aCard.flip;`
	`aCard.setFaceUp(true);`
	`aGame.displayCard(aCard, 45, 56);`

Smalltalk	`aCard flip.`
	`aCard setFaceUp: true.`
	`aGame display: aCard atLocation: 45 and: 56.`

Objective-C	`[ aCard flip ].`
	`[ aCard setFaceUp: true ].`
	`[ aGame display: aCard atLocation: 45 and: 56 ]`

CLOS	`(flip aCard)`
	`(setFaceUp aCard true)`
	`(displayCard aGame 45 56)`

Figure 5.1: Message Passing Syntax in various Languages

may accept the same message, and yet perform different actions.

There are three identifiable parts to any message-passing expression. These are the *receiver* (the object to which the message is being sent), the *message selector* (the text that indicates the particular message being sent), and the *arguments* used in responding to the message.

$$\underbrace{\mathsf{aGame}}_{receiver}.\underbrace{\mathsf{displayCard}}_{selector}\underbrace{(\mathsf{aCard}, 42, 27)}_{arguments}$$

As Figure 5.1 indicates, the most common syntax for message passing uses a period to separate the receiver from the message selector. Minor variations include features such as whether an empty pair of parenthesis are required when a method has no arguments (they can be omitted in Pascal and some other languages).

Smalltalk and Objective-C use a slightly different syntax. In these languages a space is used as a separator. Unary messages (messages that take no argument) are simply written following the receiver. Messages that take arguments are written using *keyword notation*. The message selector is split into parts, one part before each argument. A colon follows each part of the key.

```
aGame display: aCard atLocation: 45 and: 56.
```

In Smalltalk even binary operations, such as addition, are interpreted as a message sent to the left value with the right value as argument:

```
z <- x + y.  " message to x to add y to itself and return sum "
```

It is possible to define binary operators in C++ to have similar meanings. In Objective-C a Smalltalk-like message is enclosed in a pair of square braces, termed a *message passing expression*. The brackets only surround the message itself. They do not, for example, surround an assignment the places the result of a message into a variable:

```
int cardrank = [ aCard getRank ];
```

The syntax used in CLOS follows the traditional Lisp syntax. All expressions in Lisp are written as parenthesis-bounded lists. The operation is the first element of the list, followed by the arguments. The receiver is simply the first argument.

5.2 Statically and Dynamically Typed Languages

Languages can be divided into two groups depending upon whether they are statically or dynamically typed. Fundamentally, a statically typed language associates types with variables (usually the binding is established by means of declaration statements), while a dynamically typed language treats variables simply as names, and associates types with values. Java, C++, C#, and Pascal are statically typed languages, while Smalltalk, CLOS and Python are dynamically typed.

Objective-C holds a curious middle ground between the two camps. In Objective-C a variable can be declared with a fixed type, and if so the variable is statically typed. On the other hand, a variable can also be declared using the object type id. A variable declared in this fashion can hold any object value, and hence is dynamically typed.

```
PlayingCard aCard;  /* a statically typed variable */
id anotherCard; /* a dynamically typed variable */
```

The difference between statically typed languages and dynamically typed languages is important in regards to message passing because a statically typed language will use the type of the receiver to check, at compile time, that a receiver will understand the message it is being presented. A dynamically typed language, on the other hand, has no way to verify this information at compile time. Thus in a dynamically typed language a message can generate a run-time

error if the receiver does not understand the message selector. Such a run-time error can never occur in a statically typed language.

5.3 Accessing the Receiver from Within a Method

As we indicated at the beginning of this chapter, a message is always passed to a receiver. In most object-oriented languages, however, the receiver does not appear in the argument list for the method. Instead, the receiver is only implicitly involved in the method definition. In those rare situations when it is necessary to access the receiver value from within a method body a *pseudo-variable* is used. A pseudo-variable is like an ordinary variable, only it need not be declared and cannot be modified. (The term *pseudo constant* might therefore seem more appropriate, but this term does not seem to be used in any language definitions).

The pseudo-variable that designates the receiver is named this in Java and C++, Current in Eiffel, and self in Smalltalk, Objective-C, Object Pascal and many other languages. The pseudo-variable can be used as if it refers to an instance of the class. For example, the method color could be written in Pascal as follows:

```
function PlayingCard.color : colors;
begin
    if (self.suit = Heart) or (self.suit = Diamond) then
        color := Red
    else
        color := Black;
end
```

In most languages the majority of uses of the receiver pseudo-variable can be omitted. If a data field is accessed or a method is invoked without reference to a receiver it is implicitly assumed that the receiver pseudo-variable is the intended basis for the message. We saw this earlier in the method flip, which acted by invoking the method setFaceUp:

```
class PlayingCard {
    ...
    public void flip () { setFaceUp( ! faceUp ); }
    ...
}
```

The method could be rewritten to make the receivers explicit as follows:

```
class PlayingCard {
    ...
```

```
    public void flip () { this.setFaceUp( ! this.faceUp); }
    ...
}
```

One place where the use of the variable often cannot be avoided is when a method wishes to pass itself as an argument to another function, as in the following bit of Java:

```
class QuitButton extends Button implements ActionListener {
    public QuitButton () {
        ...
            // install ourselves as a listener for button events
        addActionListener(this);
    }
    ...
};
```

Some style guidelines for Java suggest the use of this when arguments in a constructor are used to initialize a data member. The same name can then be used for the argument and the data member, with the explicit this being used to distinguish the two names:

```
class PlayingCard {
    public PlayingCard (int suit, int rank) {
        this.rank = rank; // this.rank is the data member
        this.suit  = suit;  // rank is the argument value
        this.faceUp = true;
    }
    ...
    private int suit;
    private int rank;
    private boolean faceUp;
}
```

A few object-oriented languages, such as Python, CLOS, or Oberon, buck the trend and require that the receiver be declared explicitly in a method body. In Python, for example, a message might appear to have two arguments, as follows:

```
    aCard.moveTo(27, 3)
```

but the corresponding method would declare three parameter values:

```
class PlayingCard:
    def moveTo (self, x, y):
        ...
```

While the first argument could in principle be named anything, it is common to name it self or this, so as to indicate the association with the receiver pseudo-variables in other languages. Examples in the previous chapter illustrated the syntax used by CLOS and Oberon, which also must name the receiver as an argument.

5.4 Object Creation

In most conventional programming languages, variables are created by means of a declaration statement, as in the following Pascal example:

```
var
    sum : integer;
begin
    sum := 0.0;
    ...
end;
```

Some programming languages allow the user to combine declaration with initialization, as in the following Java example:

```
int sum = 0.0; // declare and initialize variable with zero
...
```

A variable declared within the bounds of a function or procedure generally exists only as long as the procedure is executing. The same is true for some object-oriented languages. The following declaration statement, for example, can be used to create a variable in C++:

```
PlayingCard aCard(Diamond, 4);  // create 4 of diamonds
```

Most object oriented languages, however, separate the process of variable naming from the process of object creation. The declaration of a variable only creates the name by which the variable will be known. To create an object value the programmer must perform a separate operation. Often this operation is denoted by the operator new, as in this Smalltalk example:

```
| aCard |  " name a new variable named aCard "

aCard <- PlayingCard new.  " allocate memory space to variable "
```

C++	`PlayingCard * aCard = new PlayingCard(Diamond, 3);`
Java, C#	`PlayingCard aCard = new PlayingCard(Diamond, 3);`

Object Pascal	`var`
	`  aCard : ^ PlayingCard;`
	`begin`
	`  new (aCard);`
	`  ...`
	`end`

Objective-C	`aCard = [ PlayingCard new ];`
Python	`aCard = PlayingCard(2, 3)`
Ruby	`aCard = PlayingCard.new`
Smalltalk	`aCard <- PlayingCard new.`

Figure 5.2: Syntax Used for Object Creation

The syntax used in object creation for various different languages is shown in Figure 5.2. Python does not use the `new` operator explicitly, instead, in Python creation occurs when a class name is used in the fashion of a function.

5.4.1 Creation of Arrays of Objects

The creation of an array of objects presents two levels of complication. There is the allocation and creation of the array itself, and then the allocation and creation of the objects that the array will hold.

In C++ these features are combined, and an array will consist of objects that are each initialized using the default (that is, no-argument) constructor (see Section 5.6):

```
    // create an array of 52 cards, all the same
PlayingCard cardArray [52];
```

In Java, on the other hand, a superficially similar statement has a very different effect. The `new` operator used to create an array creates only the array. The values held by the array must be created separately, typically in a loop:

```
PlayingCard cardArray[ ] = new PlayingCard[13];
for (int i = 0; i < 13; i++)
```

```
cardArray[i] = new PlayingCard(Spade, i+1);
```

A frequent source of error for C or C++ programmers moving to Java is to forget that in Java the allocation of an array occurs separately from the allocation of the elements the array will contain.

5.5 Pointers and Memory Allocation

All object-oriented languages use pointers in their underlying representation. Not all languages expose this representation to the programmer. It is sometimes said that "Java has no pointers" as a point of contrast to C++. A more accurate statement would be that Java has no pointers that the programmer can see, since all object references are in fact pointers in the internal representation.

The issue is important for three reasons. Pointers normally reference memory that is *heap allocated*, and thus does not obey the normal rules associated with variables in conventional imperative languages. In an imperative language a value created inside a procedure will exist as long as the procedure is active, and will disappear when the procedure returns. A heap allocated value, on the other hand, will continue to exist as long as there are references to it, which often will be much longer than the lifetime of the procedure in which it is created.

The second reason is that heap based memory must be recovered in one fashion or another, a topic we will address in the next section.

A third reason is that some languages, notably C++, distinguish between conventional values and pointer values. In C++ a variable that is declared in the normal fashion, a so-called *automatic* variable, has a lifetime tied to the function in which it is created. When the procedure exits, the memory for the variable is recovered:

```
void exampleProcedure
{
    PlayingCard ace(Diamond, 1);
    ...
    // memory is recovered for ace
    // at end of execution of the procedure
}
```

Values that are assigned to pointers (or as *references*, which are another form of pointers) are not tied to procedure entry. Such values differ from automatic variables in a number of important respects. As we will note in the next section, memory for such values must be explicitly recovered by the programmer. When we introduce inheritance in Chapter 8 we will see that such values also differ in the way they use that feature.

5.5.1 Memory Recovery

Memory created using the new operator is known as *heap-based* memory, or simply *heap* memory. Unlike ordinary variables, heap-based memory is not tied to procedure entry and exit. Nevertheless, memory is always a finite commodity, and hence some mechanism must be provided to recover memory values. Memory that has been allocated to object values is then recycled and used to satisfy subsequent memory requests.

There are two general approaches to the task of memory recovery. Some languages (such as C++ and Delphi Pascal) insist the programmer indicate when an object value is no longer being used by a program, and hence can be recovered and recycled. The keywords used for this purpose vary from one language to another. In Object Pascal the keyword is free, as in the following example:

```
free aCard;
```

Objective-C uses the same keyword, but written as a message, with the receiver first:

```
[ aCard free ];
```

In C++ the keyword is delete:

```
delete aCard;
```

When an array is deleted a pair of square braces must be placed after the keyword:

```
delete [ ] cardArray;
```

The alternative to having the programmer explicitly manage memory is an idea termed *garbage collection*. A language that uses garbage collection (such as Java, C#, Smalltalk or CLOS) monitors the manipulation of object values, and will automatically recover memory from objects that are no longer being used. Generally garbage collection systems wait until memory is nearly exhausted, then will suspend execution of the running program while they recover the unused space, before finally resuming execution. Garbage collection uses a certain amount of execution time, which may make it more costly than the alternative of insisting that programmers free their own memory. But garbage collection prevents a number of common programming errors:

- It is not possible for a program to run out of memory because the programmer forgot to free up unused memory. (Programs can still run out of memory if the total memory required at any one time exceeds the available memory, of course).

- It is not possible for a programmer to try to use memory after it has been freed. Freed memory can be reused, and hence the contents of the memory values may be overwritten. Using a value after it has been freed can therefore cause unpredictable results.

```
PlayingCard * aCard = new PlayingCard(Spade, 1);
    ...
delete aCard;
    ...
cout << aCard.rank(); // attempt to use after deletion
```

- It is not possible for a programmer to try and free the same memory value more than once. Doing this can also cause unpredictable results.

```
Playingcard * aCard = new PlayingCard(Space, 1);
    ...
delete aCard;
    ...
delete aCard; // deleting already deleted value
```

When a garbage collection system is not available, to avoid these problems it is often necessary to ensure that every dynamically allocated memory object has a designated *owner*. The owner of the memory is responsible for ensuring that the memory location is used properly and is freed when it is no longer required. In large programs, as in real life, disputes over the ownership of shared resources can be a source of difficulty.

When a single object cannot be designated as the owner of a shared resource, another common technique is to use *reference counts*. A reference count is a count of the number of pointers that reference the shared object. Care is needed to ensure that the count is accurate; whenever a new pointer is added the count is incremented, and whenever a pointer is removed the count is decremented. When the count reaches zero it indicates that no pointers refer to the object, and its memory can be recovered.

As with the arguments for and against dynamic typing, the arguments for and against garbage collection tend to pit efficiency against flexibility. Automatic garbage collection can be expensive, as it necessitates a run-time system to manage memory. On the other hand, the cost of memory errors can be equally expensive.

5.6 Constructors

As we indicated in Chapter 4, a constructor is a method that is used to initialize a newly created object. Linking creation and initialization together has many beneficial consequences. Most importantly, it guarantees that an object can never

be used before it has been properly initialized. When creation and initialization are separated (as they must be in languages that do not have constructors), a programmer can easily forget to call an initialization routine after creating a new value, often with unfortunate consequences. A less common problem, although often just as unfortunate, is to invoke an initialization procedure twice on the same value. This problem, too, is avoided by the use of constructors.

In Java and C++ a constructor can be identified by the fact that it has the same name as the class in which it appears. Another small difference is that constructors do not declare a return type:

```
class PlayingCard {     // a Java constructor
    public PlayingCard (int s, int r) {
        suit = s;
        rank = r;
        faceUp = true;
    }
    ...
}
```

When memory is allocated using the new operator, any arguments required by the constructor appear following the class name:

```
    aCard = new PlayingCard(PlayingCard.Diamond, 3);
```

Data fields in Java (as well as in C#) that are initialized with a simple value, independent of any constructor argument, can be assigned a value at the point they are declared, even if they are subsequently reassigned:

```
class Complex { // complex numbers
    public Complex (double rv) { realPart = rv; }

    public double realPart = 0.0;  // initialize data areas
    public double imagPart = 0.0; // to zero
}
```

A similar syntax can be used in C++ if the data members are are declared to be static and/or const.

In C++ and Java there can be more than one function definition that uses the same name, as long as the number, type and order of arguments are sufficient to distinguish which function is intended in any invocation. This facility is frequently used with constructors, allowing the creation of one constructor to be used when no arguments are provided, and another to be used with arguments:

```
class PlayingCard {
public:
```

```
    PlayingCard ( )  // default constructor,
            // used when no arguments are given
        { suit = Diamond; rank = 1; faceUp = true; }

    PlayingCard (Suit is) // constructor with one argument
        { suit = is; rank = 1; faceUp = true; }

    PlayingCard (Suit is, int ir) // constructor with two arguments
        { suit = is; rank = ir; faceUp = true; }
};
```

The combination of number, type and order of arguments is termed a function *type signature*. We say that the meaning of an overloaded constructor (or any other overloaded function, for that matter) is resolved by examining the type signature of the invocation.

```
    PlayingCard cardOne; // invokes default
    PlayingCard * cardTwo = new PlayingCard;
    PlayingCard cardThree(PlayingCard.Heart);
    PlayingCard * cardFour = new PlayingCard(PlayingCard.Spade, 6);
```

In C++ one must be careful to omit the parenthesis from an invocation of the default constructor. Using a parenthesis in this situation is legal, but has an entirely different meaning:

```
    PlayingCard cardFive; // creates a new card
    PlayingCard cardSix(); // forward definition for function
                // named cardSix that returns a PlayingCard
```

On the other hand, when using the `new` operator and no arguments, parenthesis are omitted in C++ but not in Java or C#:

```
    PlayingCard cardSeven = new PlayingCard(); // Java
    PlayingCard * cardEight = new PlayingCard; // C++
```

Constructors in C++ can also use a slightly different syntax to specify the initial value for data members. A colon, followed by a named value in parenthesis, is termed an *initializer*. Our constructor written using initializer syntax would look as follows:

```
Class PlayingCard {
public:
    PlayingCard (Suits is, int ir)
        : suit(is), rank(ir), faceUp(true) { }
    ...
```

```
};
```

For simple values such as integers there is no difference between the use of an initializer and the use of an assignment statement within the body of the constructor. We will subsequently encounter different forms of initialization that can only be performed in C++ by means of an initializer.

Constructors in Objective-C need not have the same name as the class, and are signified by the use of a plus sign, rather than a minus sign, in the first column of their definition. Such a function is termed a *factory method*. The factory method uses the new operator to perform the actual memory allocation, then performs whatever actions are necessary to initialize the object.

```
@ implementation PlayingCard

+  suit: (int) s rank: (int) r {
    self = [ Card new ];
    suit = s;
    rank = r;
    return self;
}

@end
```

Factory methods are invoked using the class as the receiver, rather than an instance object:

```
PlayingCard aCard = [ PlayingCard suit: Diamond rank: 3 ];
```

Constructors in Python all have the unusual name `__init__`. When an object is created, the init function is implicitly invoked, passing as argument the newly created object, and any other arguments used in the creation expression:

```
aCard = PlayingCard(2, 3)
    # invokes PlayingCard.__init__(aCard, 2 3)
```

In Apple Object Pascal there are no constructors. New objects are created using the operator new, and often programmers define their own initialization routines that should be invoked using the newly created object as receiver. The Delphi version of Object Pascal is much closer to C++. In Delphi programmers can define a constructor, although unlike C++ this function need not have the same name as the class. It is typical (although not required) to use the name Create as a constructor name:

```
interface
    type
```

```
        TPlayingCard = class (TObject)
            constructor Create (is : Suits, ir : integer);
            ...
        end;
implementation
    constructor TPlayingCard.Create (is : Suits, ir : integer);
    begin
        suit = is;
        rank = ir;
        faceUp = true;
    end;
```

New objects are then created using the constructor method with the class as receiver:

```
    aCard := TPlayingCard.Create (Spade, 4);
```

5.6.1 The Orthodox Canonical Class Form*

Several authors of style guides for C++ have suggested that almost all classes should define four important functions. This has come to be termed the *orthodox canonical class* form. The four important functions are:

- A default constructor. This is used internally to initialize objects and data members when no other value is available.
- A copy constructor. This is used, among other places, in the implementation of call-by-value parameters.
- An assignment operator. This is used to assign one value to another.
- A destructor. This is invoked when an object is deleted. (We will shortly give an example to illustrate the use of destructors).

A default constructor we have seen already. This is simply a constructor that takes no arguments. A copy constructor takes a reference to an instance of the class as argument, and initializes itself as a copy of the argument:

```
class PlayingCard {
public:
    ...
    PlayingCard (PlayingCard & aCard)
    {
            // initialize ourself as copy of argument
```

[0] Section headings followed by an asterisk indicate optional material.

```
        rank = aCard.getRank();
        suit = aCard.getSuit();
        faceUp = aCard.isFaceUp();
    }
    ...
};
```

The system will implicitly create default versions of each of these if the user does not provide an alternative. However, in many situations (particuarly those involving the management of dynanically allocated memory) the default versions are not what the programmer might wish. Even if empty bodies are supplied for these functions, writing the class body will at least suggest that the program designer has *thought* about the issues involved in each of these. Furthermore, appropriate use of visibility modifiers give the programmer great power is allowing or disallowing different operations used with the class.

5.6.2 Constant Values

In Chapter 4 we pointed out that some languages, such as C++ and Java, permit the creation of data fields that can be assigned once and thereafter are not allowed to change. Having introduced constructors, we can now complete that discussion by showing how such values can be initialized.

In Java an immutable data field is simply declared as final and can be initialized directly:

```
class ListofImportantPeople {
public:
    final int max = 100; // maximum number of people
    ...
}
```

Alternatively, a final value can be assigned in the constructor. If there is more than one constructor, each constructor must initialize the data field:

```
class PlayingCard {
    public PlayingCard ( )
        { suit = Diamond; rank = 1; faceUp = true; }
    public PlayingCard ( int is, int ir)
        { suit = is; rank = ir; faceUp = true; }
    ...
    public final int suit;  // suit and rank are
    public final int rank; // immutable
    private boolean faceUp; // faceUp is not
}
```

Immutable values in C++ are designated using the keyword const. They can only be given a value using an initializer clause in a constructor:

```
class PlayingCard {
public:
    PlayingCard () : suit(Diamond), rank(1) { faceUp = true; }
    PlayingCard (Suits is, int ir) : suit(is), rank(ir)
        { faceUp = true; }
    ...
    const Suits suit;
    const int rank;
private:
    boolean faceUp;
};
```

There is one subtle but nevertheless important difference between const and final values. The const modifier in C++ says that the associated value is truly constant, and is not allowed to change. The final modifier in Java only asserts that the associated variable will not be assigned a new value. Nothing prevents the value itself from changing its own internal state, for example in response to messages. To illustrate this, consider the following definition for a data type named Box.

```
class Box {
    public void setValue (int v);
    public int getValue () { return v; }
    private int v = 0;
}
```

Declaring a variable using the final modifier simply means it will not be reassigned, it does not mean it will not change:

```
final aBox = new Box(); // can be assigned only once
aBox.setValue(8); // but can change
aBox.setValue(12); // as often as you like
```

A variable declared using the const modifier in C++, on the other hand, is not allowed to change in any way, not even in its internal state. (Individual fields can be named as mutable, in which case they are allowed to change even within a const object. However, use of this facility is rare.)

5.7 Destructors and Finalizers

A constructor allows the programmer to perform certain actions when an object value is created, when it is being born (so to speak). Occasionally it is useful to also be able to specify actions that should be performed at the other end of a values lifetime, when the variable is about to die and have its memory recovered.

This can be performed in C++ using a method termed a *destructor*. The destructor is invoked automatically whenever memory space for an object is released. For automatic variables, space is released when the function containing the declaration for the variable is existed. For dynamically allocated variable space is released with the operator delete. The destructor function is written as the name of the class preceded by a tilde (~). It does not take any arguments and is never directly invoked by the user.

A simple but clever function will illustrate the use of constructors and destructors. The class Trace defines a simple class that can be used to trace the flow of execution. The constructor class takes as argument a descriptive string and prints a message when space for the associated variable is allocated (which is when the procedure containing the declaration is entered). A second message is printed by the destructor when space for the variable is released, which occurs when the procedure is exited.

```
class Trace {
public:
    // constructor and destructor
    Trace    (string);
    ~Trace    ();
private:
    string text;
};

Trace::Trace (string t) : text(t)
{     cout << "entering " << text << endl; }

Trace::~Trace ()
{     cout << "exiting " << text << endl; }
```

To trace the flow of execution, the programmer simply creates a declaration for a dummy variable of type Trace in each procedure to be traced. Consider the following pair of routines:

```
void procedureA ()
{
    Trace dummy ("procedure A");
    procedureB (7);
}
```

```
void procedureB (int x)
{
    Trace dummy ("procedure B");
    if (x < 5) {
        Trace aaa("true case in Procedure B");
        ...
        }
    else {
        Trace bbb("false case in Procedure B");
        ...
        }
}
```

By their output, the values of type Trace will trace out the flow of execution. A typical output might be:

```
entering procedure A
entering procedure B
entering false case in Procedure B
...
exiting false case in Procedure B
exiting procedure B
exiting procedure A
```

Delphi Pascal also supports a form of destructor. A destructor function (usually called Destroy) is declared by the keyword destructor. When a dynamically allocated object is freed, the memory management system will call the destructor function.

```
type
    TPlayingCard = class (TObject)
        ...
        destructor Destroy;
    end;

destructor PlayingCard.Destroy;
begin
    (* whatever housekeeping is necessary *)
    ...
end;
```

Java and Eiffel have similar facilities, although since both languages use garbage collection their utilization is different. A method named finalize in Java will be invoked just before the point where a variable is recovered by the garbage

collection system. Since this can occur at any time, or may never occur, the use of this facility is much less common than the use of destructors in C++.

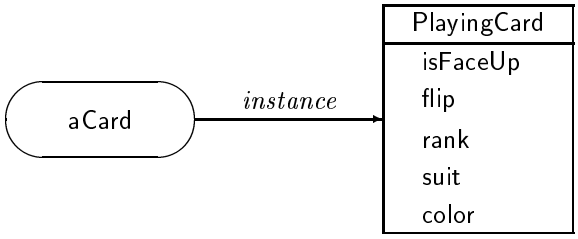
```
class FinalizeExample {
    public void finalize () {
        System.out.println("finally doing finalization");
        System.exit(0);
    }
}

    ...

        // first create an instance
    Object x = new FinalizeExample();
        // redefining x releases memory
    x = new Integer(3);
        // now do lots of memory allocations
        // at some indeterminent point garbage collection
        // will occur and final method will be called
    for (int i = 0; i < 1000; i++) {
        System.out.println("i is " + i);
        for (int j = 0; j < 1000; j++)
            x = new Integer(j);
    }
```

In Eiffel the same effect is achieved by inheriting from the class Memory and overriding the method dispose. (We will discuss inheritance and overriding later in Chapter 8.)

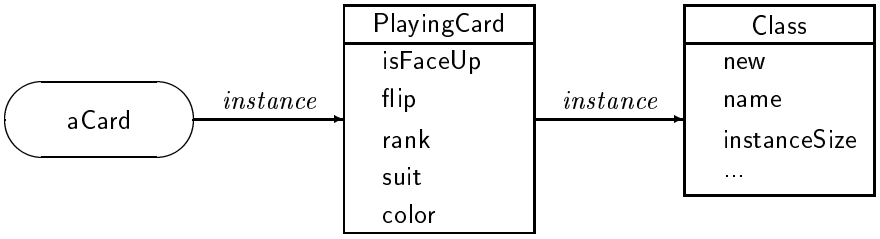
5.8 Metaclasses in Smalltalk*

The discussion of object creation provides an excuse to introduce a curious concept found in Smalltalk and a few similar languages, termed *metaclasses*. To understand metaclasses, note first that methods are associated not with objects, but with classes. That is, if we create a playing card, the methods associated with the card are found not in the object itself, but in the class PlayingCard.

[0] Section headings followed by an asterisk indicate optional material.



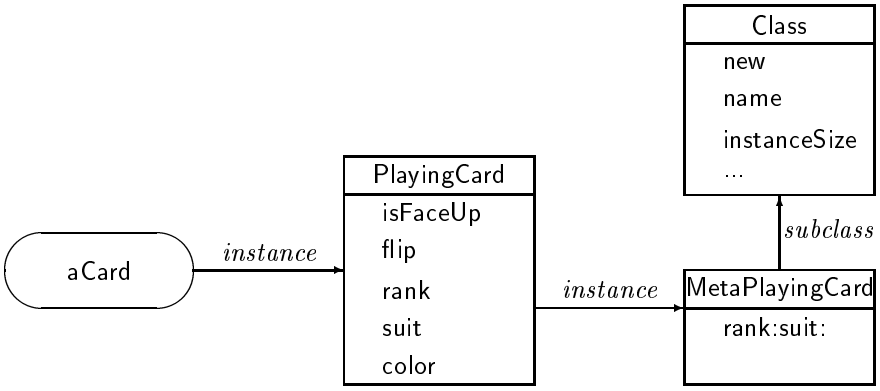
But in Smalltalk classes *are* objects. We explored this briefly in the previous chapter. Thus classes themselves respond to certain messages, such as the object creation message new.



Given this situation, let us now imagine that we want to create a method that can be used in the fashion of a constructor. That is, we want a method, let us call it rank:suit:, that can be given to a specific class object, say PlayingCard, and when executed it will both create a new instance and ensure it is properly initialized. Where in the picture just given can this method be placed? It cannot be part of the methods held by PlayingCard, as those are methods that are to be executed by *instances* of the class, and at the time of creation we do not yet have an instance. Nor can it be part of the methods held by Class, since those represent behavior common to all classes, and our initialization is something we want to do only for this one class.

The solution is to create a new "hidden" class, termed a metaclass. The object named PlayingCard is not actually an instance of Class, but is in reality an instance of MetaPlayingCard, which is formed from inheritance from Class.[1] Initialization specific behavior can then be placed in this new class.

[1] We are being slightly premature in presenting the discussion here, since inheritance will not be discussed in detail until Chapter 8. But the intuitive description of inheritance given in Chapter 1 is sufficient to understand the concept of metaclasses.



The behavior placed in the class MetaPlayingCard is understood by the object PlayingCard, and by no other object. This object is the only instance of the class.

Smalltalk browsers generally hide the existence of metaobjects from programmers, calling such methods by the term *class methods*, and acting as if they were associated with the same class as other methods. But behind the browser, class methods are simply ordinary methods associated with metaclasses.

Chapter Summary

In this chapter we have examined the syntax and techniques used on object creation and initialization for each of the different languages we are considering.

- We have examined the syntax used for message passing.
- We introduced the two major categories of programming languages, statically and dynamically typed languages. In a statically typed language types are associated with variables, while in a dynamically typed language a variable is simply a name, and types are associated with values.
- In many languages the receiver of a message can be accessed from within the body of the method used to respond to the message. The receiver is represented by a pseudo-variable. This variable can be named this, self, or current (depending upon the language being used).
- Automatic memory allociates the lifetime of an object with the procedure in which it is declared. Heap-based memory is explicitly allocated (in most languages using an operator named new) and is either explicitly deallocated or recovered by a garbage collection system.
- A constructor ties together the two tasks of memory allocation and initialization. This ensures that all objects which are allocated are properly initialized.
- A destructor is executed with a value is deleted.

- Finally, we have examined how metaclasses in Smalltalk address the problem of creation and initialization in that language.

Further Reading

The works cited at the end of the previous chapter should be consulted for more detailed information on any of the languages we are considering in this book.

Cohen [Cohen 1981] provides a good overview of garbage collection techniques. An interesting comparison between garbage collection and automatic memory allocation is given by Appel [Appel 1987]. Techniques for Reference counting in C++ are described in [Budd 1999].

Metaclasses in Smalltalk will be examined in detail later in Chapter 25.

Self Study Questions

1. In what ways is a message passing operation different from a procedure call?

2. What are the three parts of a message passing expression?

3. How does Smalltalk style keyword notation differ from Java or C++ style notation?

4. What is the difference between a statically typed language and a dynamically typed language?

5. Why are run-time errors of the form "receiver does not understand message" not common in statically typed languages? Why are they more common in dynamically typed languages?

6. What does the pseudo-variable this (or self in Smalltalk) refer to?

7. What is the difference between stack allocated and heap allocated memory?

8. What are the two general approaches to recovery of heap allocated memory?

9. What common programming errors does the use of a garbage collection system eliminate?

10. What two tasks are brought together by a constructor?

11. When is a destructor method executed?

12. What is a metaclass? What problem is solved through the use of metaclasses?

Exercises

1. Write a method copy for the class Card of Chapter 4. This method should return a new instance of the class Card with the suit and rank fields initialized to be the same as the receiver.

2. In a language that does not provide direct support for immutable instance variables, how might you design a software tool that would help to detect violations of access? (Hint: The programmer can provide directives in the form of comments that tell the tool which variables should be considered immutable.)

3. We have seen two styles for invoking methods. The approach used in C++ is similar to a conventional function call. The Smalltalk and Objective-C approaches separate arguments with keyword identifiers. Which do you think is more readable? Which is more descriptive? Which is more error-prone? Present short arguments to support your opinions.

4. How might you design a tool to detect the different types of memory allocation and free problems described in Section 5.5.1?

5. Andrew Appel [Appel 1987] argues that under certain circumstances heap-based memory allocation can be more efficient than stack-based memory allocation. Read this article and summarize the points of Appel's argument. Are the situations in which this is true likely to be encountered in practice?

6. Write a short (two- or three-paragraph) essay arguing for or against automatic memory-management (garbage-collection) systems.