Chapter 5

Object-Oriented Programming

In the imperative programming paradigm that has dominated the way programmers think about solutions to problems for the past twenty years or so, a program consists of one or more procedures that transfer control among themselves and manipulate one or more data items to solve a problem. Object-oriented programming (OOP) is a different paradigm based on Simula's classes. Many people like it because it allows code to be reused in an organized fashion.

Object-oriented programming is an area of current research. There is an annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).

1 ◆ DEFINITIONS

An **object-oriented** program consists of one or more **objects** that interact with one another to solve a problem. An object contains state information (data, represented by other objects) and operations (code). Objects interact by sending **messages** to each other. These messages are like procedure calls; the procedures are called **methods**. Every object is an instance of a **class**, which determines what data the object keeps as state information and what messages the object understands. The **protocol** of the class is the set of messages that its instances understand.

Objects in object-oriented programming correspond to variables and constants in structured programming. Classes in object-oriented programming correspond to types: Every object of a particular class has the same structure as every other object of that class.

Objects are a form of abstract data type, in that if two objects respond to the same messages in the same way, there is no way to distinguish them. Such objects may be freely interchanged. For example, I might have two `Stack` objects that respond to `push` and `pop` messages. One object might inter-

nally use an array, the other a linked list. The two `stack` objects are indistinguishable to their clients. I might even have an array of stacks, some of whose components are implemented one way, while others are implemented the other way.

The term object-oriented has started to appear prominently in many advertisements, but people disagree about what object-oriented programming is and is not. The consensus seems to be that a programming language must support data encapsulation, inheritance, and overloading to be called an object-oriented programming language.

Data encapsulation dictates that an object `A` that wishes to examine or modify another object `B` may do so only in ways defined by `B`'s protocol. In other words, the data associated with an object is hidden from public view. Only the operations an object supports are known to its clients. Data encapsulation makes it unlikely that changes in the implementation of an object or extensions to its protocol will cause failures in the code for unrelated objects. As long as the object's new protocol is a superset of its old one, code that relies on the old protocol will continue to work correctly.

Inheritance allows one class to share the properties of another. For example, Smalltalk includes the predefined class `Magnitude`, which defines several operations, including `max` (maximum). Any class that inherits from `Magnitude`, such as `Integer`, inherits this operation. The `max` operation for all subclasses of `Magnitude` is thus defined in one place, so any enhancements or corrections to the `max` operation become available automatically to all such classes. Inheritance is used in practice for two purposes: (1) to indicate that the new class specializes the old class, and (2) to allow the new class to use code from the old class. Inheritance makes the job of enhancement and maintenance much easier.

Overloading dictates that the code invoked to perform an operation must depend not only on the operation but on what sort of objects the operation is to manipulate. For example, the `max` operation provided by the `Magnitude` class is defined in terms of the `>` (greater than) operation. The `>` operation performed to obtain the larger of two integers and the `>` operation performed to obtain the larger of two real numbers are two different operations. Overloading ensures that the appropriate `>` operation is performed in each case. Overloading makes it possible to define an operation such as `max` in an abstract sense. So long as the parameters to the operation exhibit the appropriate behavior (in this case, they define `>`), the operation will succeed.

2 ◆ A SHORT EXAMPLE

The principal advantage claimed for object-oriented programming is that it promotes reuse of valuable code. If an abstract data type has been implemented as a class, then a related data type can be implemented as a subclass, automatically reusing the code that still applies (by inheriting it) and redefining those operations that differ (by overloading the old names with new implementations).

For example, consider the abstract type `Collection`, values of which are unordered groups of integers, where individual integers may appear more than once in a collection. Such an abstract data type would have several operations, such as the following:

```
insert(C : reference Collection;
     what : value integer)
present(C : reference Collection;
     what : value integer) : Boolean;
remove(C : reference Collection;
     what : value integer)
write(C : reference Collection)
```

The implementation of collections could use a linked list or an array. Let's not worry about what to do if there is an error, such as inserting when there is no more space, or removing an integer that is not in the collection; perhaps an exception mechanism (discussed in Chapter 2) could be used.

Collections sometimes have special requirements. I might want, for example, the related data type Set, which makes sure that an item is not inserted multiple times. Object-oriented programming lets me declare a class Set as a subclass of Collection, inheriting all the operations, but letting me reimplement the insert routine.

A different related data type is Queue, which is different in two ways. First, it must retain the order of inserted values. Second, the remove operation has a different form:

```
remove(Q : reference Queue) : integer;
```

I would define a class Queue as a subclass of Collection. Depending on the implementation of Collection, I may be able to reuse most of its code or very little. If I end up rewriting major amounts of code, I might decide to use the Queue-friendly code in Collection in order to save duplication of effort.

Finally, I may wish to introduce the type InstrumentedQueue, which has one additional operation:

```
report(I : reference  InstrumentedQueue)
```

This operation writes the number of insertions and deletions that have been performed on the given queue. In order to reuse the statistics-gathering facility in other programs, I might implement it as a new class Statistic with operations increment and report (not to be confused with the report provided by InstrumentedQueue). Objects of class InstrumentedQueue would contain extra fields of type Statistic to hold the number of insertions and deletions.

The classes I have introduced form a tree, as shown in Figure 5.1.

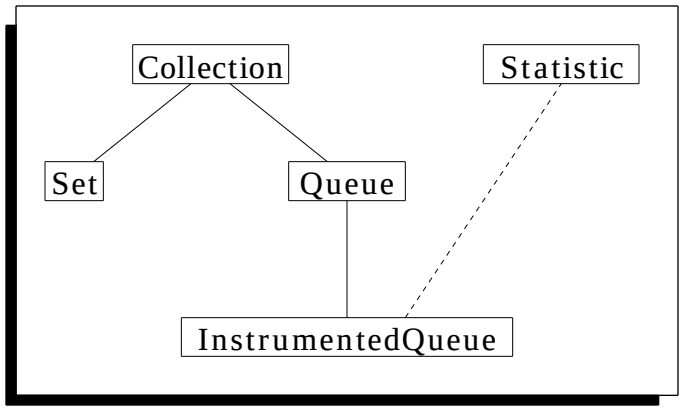

Figure 5.1 Class hierarchy

The solid lines indicate which classes are subclasses of others. The dashed line indicates that `InstrumentedQueue` has local fields of class `Statistic`.

A value of class `Collection` has only the operations from that class. It would not be appropriate to invoke a `report` operation on a `Collection` value. If the compiler can tell the class of any variable, then it can determine which operations are valid and which code to invoke. However, as you will see later, there is good reason to postpone binding the actual value with variables. I might want to invoke `report` on a variable and let it be decided at runtime which version of `report` is to be invoked, if any. The compiler might therefore need to generate code that decides at runtime which operations are valid and which code to invoke. We will see that object-oriented programming languages differ in the extent to which they allow such **deferred binding**.

I might want to generalize these classes to allow elements to be not just integers, but of any type, such as reals, records, and even other classes. In other words, I might want to build polymorphic classes.

This chapter starts with a brief look at Simula, the ancestor of all object-oriented programming languages, to introduce the concepts and the issues surrounding object-oriented programming. I then turn to Smalltalk, a good example of object-oriented programming, in which most binding is performed at runtime. Smalltalk uses dynamic typing, deferred binding of operations, and even deferred declaration of classes. Smalltalk is a "pure" language, in the sense that everything in the language follows the object-oriented paradigm. I also discuss C++, which is a hybrid language that adds support for object-oriented programming to C. It uses static typing, static binding of operations (by default), and static declaration of classes.

3 ◆ SIMULA

Object-oriented programming began when Simula introduced a novel concept: A record may contain a procedure field. Such records are called **classes**.[1] As an example, consider Figure 5.2.

Figure 5.2

```
class Stack;                                                    1
     Size : 0..MaxStackSize := 0; -- initialized                2
     Data : array 0..MaxStackSize-1 of integer;                 3

     procedure Push(readonly What : integer);                   4
     begin                                                      5
          Data[Size] := What;                                   6
          Size := Size+1;                                       7
     end; -- Push;                                              8

     procedure Pop() : integer;                                 9
     begin                                                     10
          Size := Size-1;                                      11
          return Data[Size];                                   12
     end -- Pop;                                               13

     procedure Empty() : Boolean;                              14
     begin                                                     15
          return Size = 0;                                     16
     end; -- Empty                                             17
end; -- Stack                                                  18

variable                                                       19
     S1, S2 : Stack;                                           20

begin                                                          21
     S2 := S1;                                                 22
     S1.Push(34);                                              23
     if not S2.Empty() then S2.Pop() end;                      24
end;                                                           25
```

Classes are like types; variables may be declared of a class type, as in line 20. Each such declaration introduces a new instance of the class, that is, a new object. The object contains fields that are variables (that is, instance variables) and fields that are procedures (that is, methods). Object variables can be assigned (line 22); objects can be manipulated by their methods, which are named just like fields (lines 23 and 24). The three methods of Stack all have an implicit parameter: the stack object itself. Therefore, the call in line 23 implicitly acts on stack S1.

A problem with classes is that a binary operation, such as testing two stacks for equality, must be performed by either the first or the second object, taking the other object as a parameter, as shown in Figure 5.3.

[1] Simula's classes are the ancestors of Pascal's records and coroutines (Chapter 2), in addition to object-oriented programming.

Figure 5.3

```
class Stack;                                                    1
     Size : 0..MaxStackSize := 0; -- initialized                2
     Data : array 0..MaxStackSize-1 of integer;                 3

     procedure Equal(readonly Other : Stack) : Boolean;         4
     begin                                                      5
          return Other.Size = Size and                          6
               Other.Data[0..Other.Size-1] =                    7
                    Data[0..Size-1] -- equality of slices       8
     end; -- Equal                                              9

     ... -- other procedures as before                          10
end; -- Stack                                                   11

variable                                                        12
     S1, S2 : Stack;                                            13

begin                                                           14
     if S1.Equal(S2) then ...                                   15
end;                                                            16
```

In lines 6–7, fields `Size` and `Data` of the implicitly passed `Stack` have simple names, but the variables of **`Other`**, which is explicitly passed, must be **qualified** by the object intended. (If needed, the pseudovariable `Self` may be used to name the implicit object explicitly.) Invoking the `Equal` method in line 15 shows how asymmetric the binary operation has become. The same problem appears in Smalltalk, but is solved in C++, as you will see below.

In this example, I have allowed the `Equal` method of one object to access the instance variables of another object of the same class. Object-oriented languages differ in how much access they permit to instance variables and how much the programmer can control that access. I will return to this issue when I discuss C++.

Simula allows new classes to inherit instance variables and methods of old classes. Subclasses raise the issues of assignment compatibility, overloading of procedures, and dynamic binding of procedures, all of which are discussed in detail below.

4 ◆ SMALLTALK

Smalltalk is the name of a family of programming languages developed at Xerox PARC (Palo Alto Research Center) as part of the Dynabook project. Dynabook was envisioned as the ultimate personal computer — small, portable, with excellent graphics and virtually unlimited memory and computing power. Smalltalk was designed as Dynabook's programming language.

Smalltalk has gone through a long evolution, including Smalltalk-72, Smalltalk-76, Smalltalk-78, and Smalltalk-80. Many individuals have contributed to the development of its variants, most notably Alan Kay, Daniel Ingalls, and Peter Deutsch. I will consider only Smalltalk-80, and whenever I say "Smalltalk," I mean Smalltalk-80. A standard Smalltalk reference is known as the Blue Book [Goldberg 83].

Smalltalk is remarkable in many ways. It has a very elaborate program development environment, with bit-mapped graphics and a variety of specially designed utilities for entering, browsing, saving, and debugging code. Even though syntactic forms exist for entering entire programs, they are seldom used. I will not discuss the program development environment at all. Instead, my examples will be in the "file-in" syntax used for bringing programs in from text files.

My primary interest is in the object-oriented programming model that Smalltalk presents. In Smalltalk, both data and program are represented as objects. Integers are objects, complex data structures are objects, all Smalltalk programs are encapsulated into objects. Objects interact through messages, which are requests for an object to perform some operation.

Messages are philosophically different from conventional procedure and function calls in that they request an operation rather than demanding it. An object may act on a message, it may pass the message to another object, or it may even ignore the message. Objects cannot directly access the contents of other objects. An object can send a message requesting information about another object's internal state, but it cannot force the information to be provided. Objects thus represent a very tightly controlled encapsulation of data and function.

Objects differ in their properties. Each object is an instance of some class. A class specifies the local data (called **instance variables**) and routines (called **methods**). Together, I will refer to instance variables and methods as **members**. Smalltalk classes are direct descendants of the classes of Simula.

4.1 Assignment and Messages

Assignment binds an object to an identifier, as in Figure 5.4,

Figure 5.4

```
count := 10
```

which binds the integer object 10 (that is, the particular instance of the class `Integer` that represents the number 10) to the identifier `count`. `Count` temporarily acquires type integer. Smalltalk has no type declarations for variables, but objects are typed by their class. Assignment statements are also expressions; they return the value of the right-hand side.

Literals are provided for some objects. These include numbers (integer or real), single characters (for example, `$M` or `$a`, where `$` quotes a single character), strings (for example, `'hi there'`), symbols (for example, `#red`, `#left`, where `#` quotes symbols), and heterogeneous arrays (for example, `#(1 $a 'xyz')`). Literals actually refer to unnamed objects of an appropriate class that are initialized to the appropriate values. Literals are no different from other objects — the protocol of their class defines the messages they will respond to.

Smalltalk predefines several objects, including `nil` (the only instance of class `UndefinedObject`), `true` (the only instance of class `True`), and `false` (the only instance of class `False`).

Expressions illustrate the way Smalltalk uses messages to define interobject communication. The expression 1+2 does not pass the values 1 and 2 to a

+ operator to produce a result. Instead, the message selector + and the parameter 2 are sent as a message to the integer object represented by the literal 1. An integer object responds to a + message by adding the parameter to its own value and returning a new object representing the correct sum. The message selector + can be used in composing messages to other object classes (for example, strings), so overloading of message selectors is trivially provided. Ordinary usage is inverted — data aren't sent to operators; rather operators (and parameters) are sent to data. This inversion emphasizes Smalltalk's view that an object is an active entity, interacting with other objects via messages.

Smalltalk recognizes three classes of message selector: unary, binary, and keyword. I will show unary and keyword selectors in **bold font** to make examples easier to read. A **unary** selector takes no parameters and appears after the object to which it is directed, as in `x` **`sqrt`** or `theta` **`sin`**.

Binary selectors look like ordinary operators; they are composed of one or two non-alphanumeric characters. A message with a binary selector takes a single parameter that follows the selector.

Keyword selectors allow one or more parameters to be included in a message. A keyword selector is an identifier suffixed with `:`. For example, the expression in Figure 5.5

Figure 5.5

```
anArray at: 3 put: 'xyz'
```

sends a message with two parameters to `anArray` (which happens to be an array, as indicated by its name). The **`at:put:`** message specifies an array update and is the standard way to access arrays (and symbol tables) in Smalltalk. To read an array value, the program sends an **`at:`** message, such as `anArray` **`at:`** `5`. Unless parentheses are used, all keyword parameters are gathered into a single message.

In the absence of parentheses, if unary, binary, and keyword selectors are intermixed, unary selectors have the highest precedence, then binary selectors, and finally keyword selectors. Parsing proceeds strictly from left to right; there is no operator precedence. Figure 5.6, for instance,

Figure 5.6

```
anArray at: 2 + a * b abs squared
```

is interpreted as shown in Figure 5.7.

Figure 5.7

```
anArray at: ((2 + a) * ((b abs) squared))
```

An object that is sent a message is called the **receiver** of the message. The response to a message is an object. A receiver often returns itself (possibly after modifying its instance variables), but it may return a different object entirely.

4.2 Blocks

Smalltalk blocks represent a sequence of actions that are encapsulated into a single object. Blocks are used to implement control structures as well as functions. A block is a sequence of expressions, separated by periods and delimited by square brackets, as shown in Figure 5.8.

Figure 5.8

```
[index := index + 1.  anArray at: index put: 0]
```

When a block expression is encountered, the statements in the block aren't immediately executed. For example, the code in Figure 5.9

Figure 5.9

```
incr := [index := index + 1.  anArray at: index put: 0]
```

assigns the block to variable `incr`, but doesn't perform the addition or array update. The unary message selector **`value`** causes a block to be executed. Thus `incr` **`value`** will increment `index` and zero an element of `anArray`. In particular, `[statement list]` **`value`** directly executes the anonymous block. The value returned by a block when it is evaluated is the value returned by the last statement in the block.

Blocks are used in conditional and iterative constructs in an interesting manner. Consider an **`if`** statement, which is coded in Smalltalk, by sending two blocks (one for each branch) to a Boolean value, which selects the appropriate block and then executes it, as in Figure 5.10.

Figure 5.10

```
a < 0                                                    1
     ifTrue: [b := 0]                                    2
     ifFalse: [b := a sqrt]                              3
```

Our usual model of conditional statements has been inverted. A Boolean value isn't passed to an **`if`** statement; rather the **`if`** statement is passed to the Boolean!

Repetitive execution is obtained by passing a loop body to an integer or to a Boolean block, as in Figure 5.11.

Figure 5.11

```
4 timesRepeat: [x := x sin]                              1
[a < b] whileTrue: [b := b sqrt]                         2
```

The Boolean value `a < b` is enclosed in a block in line 2. The **`whileTrue:`** message is only understood by blocks, not by Booleans. The reason for this design is that the block can be reevaluated after each iteration, eventually resulting in `False` and terminating the loop. A Boolean value is immutable, so it is worthless for loop control.

Blocks can also take parameters and be used as functions. Block parameters are prefixed with `:` and are separated from the block body by `|`, as in Figure 5.12.

Figure 5.12

```
[:elem | elem sqrt]
```

Parameters can be supplied by using one or more **value:** keyword selectors, as in Figure 5.13.

Figure 5.13

```
[:elem | elem sqrt] value: 10 -- 10 sqrt
```

Anonymous blocks with parameters are handy for applying a function to an array of elements. The keyword selector **collect:** creates an array by applying a block to each element of an array, as in Figure 5.14.

Figure 5.14

```
#(1 2 3 4) collect: [:elem | elem sqrt]                    1
     -- (1 1.414 1.732 2)                                   2
```

4.3 Classes and Methods

Since all objects are instances of classes, the properties of an object are defined in its class definition. A class contains instance variables (each instance of the class contains an instance of each of these variables) and instance methods (each instance of the class responds to messages that invoke these methods). Instance variables have values that are private to a single object. Syntax rules require that instance variables begin with a lowercase letter. (Uppercase letters are only used for **shared variables**, visible globally, such as `Object`.)

Classes are themselves objects, and therefore are members (instances) of some other class. For example, `Integer` belongs to `Integer class`. `Integer class` is called a **metaclass**. All metaclasses belong to class `Metaclass`, which is the only instance of `Metaclass class`, which is itself an instance of `Metaclass`.

A new instance of a class is created by sending the message **new** to the corresponding class. Thus the code in Figure 5.15

Figure 5.15

```
anArray := Array new: 4
```

would create a new array object with 4 cells (each initialized to `nil`) and assign it to `anArray`.

Programming languages that support abstract data types (and a class is a glorified abstract data type) often allow the programmer to separate the description of a type (here, class) into a specification and implementation part. Smalltalk does not let the programmer separate specification from implementation cleanly, but it does provide much of what you would expect from abstract data types, particularly information hiding.

Figure 5.16 shows how to declare the abstract data type `Stack`.

Figure 5.16

```
Object subclass: #Stack                                                    1
     instanceVariableNames: 'count elements'                               2
     classVariableNames: 'MaxDepth'                                        3
     poolDictionaries: ''                                                  4
     category: 'Example'                                                   5
!                                                                          6

!Stack class methodsFor: 'creation'!                                       7
initialize -- sets default depth                                           8
     MaxDepth := 100                                                       9
!                                                                          10
new -- builds a new stack of default depth                                 11
     ^ super new init: MaxDepth                                            12
!                                                                          13
new: desiredDepth -- builds new stack of given depth                       14
     ^ super new init: desiredDepth                                        15
! !                                                                        16

!Stack methodsFor: 'initialization'!                                       17
init: depth                                                                18
     count := 0.                                                           19
     elements := Array new: depth                                          20
! !                                                                        21

!Stack methodsFor: 'access'!                                               22
empty                                                                      23
     ^ count = 0                                                           24
!                                                                          25
push: elem                                                                 26
     count >= elements size                                                27
     ifTrue: [self error: 'Stack overflow']                                28
     ifFalse: [                                                            29
          count := count + 1. elements at: count put: elem]                30
!                                                                          31
pop |top|                                                                  32
     self empty                                                            33
     ifTrue: [self error: 'Stack is empty']                                34
     ifFalse: [                                                            35
          top := elements at: count.                                       36
          count := count - 1.                                              37
          ^ top                                                            38
     ]                                                                     39
! !                                                                        40
```

The definition of a class first specifies its name (line 1), the names of the instance variables (line 2), and some other things (lines 3– 6). For the stack example, the instance variables are `count` and `elements`, the first an integer counting how many elements are in the stack, and the second an array holding the stack items. Since Smalltalk is dynamically typed, these declarations do not indicate any type.

`Stack` is a subclass of the class `Object`. You will see the implications of subclasses later; for now, it suffices that class `Object` will respond to a message of type **`subclass:instanceVariableNames:...category:`** and build a

new class. The ! symbol in line 6 tells the Smalltalk interpreter to accept the previous expression(s) and evaluate them. In this case, evaluating the expression leads to the definition of a new class.

Methods are categorized for documentation sake; I have separated methods for creation (lines 7–16), initialization (lines 17–21), and access (lines 22–40). These names are arbitrary and have no meaning to Smalltalk itself.

Consider first the **init:** method (lines 18–20), which initializes the instance variables count and elements. This method takes a formal parameter called depth. To create the elements array (line 20), it sends a message to class Array using a keyword selector **new:** to define its extent. The **init:** method will (by default) return the stack object to which the message is sent.

The **empty** method (lines 23–24) tests count to determine if the stack is empty. The ^ symbol explicitly names the object to be returned by the message, superseding the default, which is to return the stack object itself.

The **push:** method (lines 26–30) first tests if elements is full. It does so by sending the **size** message to the elements array and comparing the result with the current count of elements. If the stack has overflowed, the method generates a diagnostic by directing a message with the **error:** keyword selector and a string parameter to itself. The destination of this message is specified by the pseudovariable self. (**Pseudovariables** are readonly variables with a Smalltalk-specific meaning.) As I will explain in detail later, all objects inherit the capability to respond to certain messages, including **error:**. The **push:** message to a stack object can therefore elicit an **error:** message to that same object.

Finally, **pop:** (lines 32–39) tests if the stack is empty by invoking the object's own **empty** method. If so, **pop:** issues an error message; otherwise, it modifies count and returns the popped element. This method shows how to declare variables such as top local to a method invocation.

It is the responsibility of each method that modifies instance variables to make sure that it leaves the object in a consistent state. For example, the **push:** method must adjust count as well as placing the new element in elements. In Chapter 7, you will see that the possibility of many simultaneous actions on the same object makes it harder to keep the internal state consistent.

You have probably noticed that some of these methods are declared as Stack methods, and others are Stack class methods. In general, most methods will be **instance methods** (here, Stack methods). Messages for these methods are sent to individual instances of the class. However, building a new instance pertains to the class, not to instances. Messages for such methods are sent to the class object itself and invoke **class methods**. For this reason, lines 7–16 are Stack class methods.

The **new** method not only creates a new instance but also sends it an **init:** message to cause it to initialize itself. Creation is accomplished by super **new**. The pseudovariable super has the same meaning as self, except that it ignores local redefinitions of inherited methods. I need super here because I am redefining **new**, and I want to make sure that I get the original meaning of **new** in lines 12 and 15.

Just as there are class methods, there are also **class variables**, which are shared by all instances of the class. Syntax rules require that class variables begin with an upper-case letter. In line 3, I declare a single class variable,

MaxDepth. I use it only in initializing new stacks (line 12) to build an array of the required length. Although I only need a "class constant," Smalltalk does not provide either constants or readonly variables.

Class variables are conventionally initialized by a class method called **initialize** (lines 8– 9). **Initialize** is called only once, just after the single object corresponding to the class (such as Stack) is created.

A class can provide several alternative instance constructors. The example shows both **new** (lines 11–12), which creates a stack with a default maximum depth, and **new:** (lines 14–15), which takes a parameter specifying the maximum depth of the new instance. Overloading two selectors with the same name causes no problems so long as one is unary and the other is keyword.

4.4 Superclasses and Subclasses

There are two hierarchies of objects in Smalltalk. You have already seen one: the hierarchy of instances and classes. Every object is an instance of some class. Climbing up the hierarchy quickly leads to a cycle of Metaclass and Metaclass class. The other, richer, hierarchy is built from the subclass and superclass relations. Climbing up this hierarchy leads to Object, which has no superclass. The Stack class of the previous example is a direct subclass of Object.

Each new class is defined as a subclass of some existing class. A **subclass** inherits all the members of its immediate **superclass** as well as those of its indirect superclasses. You have already seen that instances of class Stack inherit the methods **error:** and **new** from Stack's superclass Object. A subclass may declare its own members and may introduce methods that override those inherited from its superclass. When a reference to a message or a variable appears in an object, it is resolved (if possible) in that object. Failing this, the object's superclass is considered, then the superclass of the superclass, up to class Object. If no definition is found, a runtime error occurs.

Subclasses are used to extend or refine the protocol of an existing class. In Figure 5.17, I define a subclass of Stack called IntegerStack. IntegerStacks will limit stack elements to integers and will provide a new operation + , which adds corresponding elements of two stacks, yielding a new stack, similar to vector addition.

I will add two additional methods, **pos:**, which returns the stack element at a particular position, and **currentDepth**, which returns the current depth of the stack. I need **pos:** and **currentDepth** because + can directly access only its own stack, not the stack passed to it as a parameter. (The same asymmetry of access plagues Simula, as discussed earlier.) I want these new methods to be private to the class; they are to be used only by + , not by the clients of the class. Unfortunately, Smalltalk does not provide a way to prevent such misuse. Still, I have placed comments on lines 35 and 38 to indicate my intent.

Figure 5.17

```
Stack subclass: #IntegerStack                                          1
     instanceVariableNames: ''                                         2
     classVariableNames: ''                                            3
     poolDictionaries: ''                                              4
     category: 'Example'                                               5
!                                                                      6

!IntegerStack methodsFor: 'access'!                                    7

push: elem                                                             8
     count >= elements size                                            9
     ifTrue: [self error: 'Stack overflow']                            10
     ifFalse: [                                                        11
          elem class = Integer                                         12
          ifTrue: [                                                    13
               count := count + 1.                                     14
               elements at: count put: elem                            15
          ]                                                            16
          ifFalse: [self error: 'Can only push integers.']             17
     ]                                                                 18
!                                                                      19

+ aStack |answer i|                                                    20
     (self currentDepth) = (aStack currentDepth)                       21
     ifFalse: [self error:                                             22
          'Incompatible stacks for addition']                          23
     ifTrue:  [                                                        24
          answer := IntegerStack init: (elements size).                25
          i := 1.                                                      26
          self currentDepth timesRepeat: [                             27
               answer push:                                            28
                    (elements at: i) + (aStack pos: i).                29
               i := i + 1                                              30
          ].                                                           31
     ^ answer                                                          32
     ]                                                                 33
! !                                                                    34

pos: i -- a private method                                             35
     ^ elements at: i                                                  36
!                                                                      37

currentDepth -- a private method                                       38
     ^ count                                                           39
!                                                                      40
```

In line 12, the **push:** method checks that the class of the element about to be pushed is, in fact, `Integer`. All objects answer the message **class** with the class of which they are an instance. They inherit this ability from `Object`.

The class hierarchy based on the subclass relation is quite extensive. Figure 5.18 shows a part of the hierarchy.

Figure 5.18

```
Object                                                  1
|    BlockContext                                       2
|    Boolean                                            3
|    |    True                                          4
|    |    False                                         5
|    Collection                                         6
|    |    Set                                           7
|    |    |    Dictionary                               8
|    |    |    |    SystemDictionary                    9
|    |    |    |    IdentityDictionary                  10
|    |    MappedCollection                              11
|    |    Bag                                           12
|    |    SequenceableCollection                        13
|    |    |    OrderedCollection                        14
|    |    |    |    SortedCollection                    15
|    |    |    Interval                                 16
|    |    |    ArrayedCollection                        17
|    |    |    |    CompiledMethod                      18
|    |    |    |    ByteArray                           19
|    |    |    |    String                              20
|    |    |    |    |    Symbol                         21
|    |    |    |    Array                               22
|    |    |    LinkedList                               23
|    |    |    |    Semaphore                           24
|    Magnitude                                          25
|    |    LookupKey                                     26
|    |    Number                                        27
|    |    |    Integer                                  28
|    |    |    Float                                    29
|    |    Date                                          30
|    |    Time                                          31
|    |    Character                                     32
|    UndefinedObject                                    33
```

This hierarchy shows how different data types are related. For example, an `Integer` is a kind of `Number`, but it has some extra methods (such as **even**, `+`, and **printOn:base:**) and some overriding methods (such as `=`). A `Number` is a kind of `Magnitude`, but it has its own extra methods (such as **squared** and **abs**, which are actually defined in terms of methods of the subclasses). A `Magnitude` is a sort of `Object`, but it has some extra methods (such as `<=`). Finally, an `Object` has no superclass, and provides methods for all other classes (such as **new** and **error:**).

4.5 Implementation of Smalltalk

Smalltalk is designed to be portable. Ironically, Smalltalk has only recently become widely available because of proprietary restrictions. Over 97 percent of the Smalltalk package, including editors, compilers, and debuggers, is written in Smalltalk. Smalltalk executes under a virtual machine that requires 6–12 KB of code. Creating a Smalltalk virtual machine for a new target machine takes about one staff-year.

The Smalltalk virtual machine consists of a storage manager, an interpreter, and a collection of primitive methods. The storage manager creates

and frees all objects (it uses garbage collection), and provides access to fields of objects. The manager also makes it possible to determine the class of any object. Methods are compiled into an intermediate form called "bytecode" (since each operation is represented in a single byte). The interpreter executes bytecode to evaluate expressions and methods. Primitive methods, such as I/O, arithmetic operations, and array indexing, are implemented directly in machine language for fast execution.

To understand how Smalltalk is implemented, you must understand how objects, classes and messages/methods are implemented. Objects are represented uniformly; they contain a header field (indicating the size of the object), a class (realized as a pointer to the corresponding class), and the instance variables of the object. If an object is of a primitive type, the object contains bit patterns defining the value of the object to the interpreter. Instance variables in a programmer-defined object are represented by pointers to the objects that the instance variables represent. The only exception to this rule is the primitive class `SmallInteger`, which is limited to the range from –16384 to 16383. Smalltalk provides other integer classes, admitting values as large as 2^{524288}. All objects are required to have an even address. An odd address is an immediate representation of a `SmallInteger`, encoded as the integer value concatenated with a low-order 1.

This representation of objects influences how operations are implemented. In particular, consider assignment (that is, copying) of objects. Since most objects are accessed through a pointer, does `a := b` mean "copy `b`" or does it mean "copy a pointer to `b`"? Smalltalk understands `:=` to mean pointer copying; it is very fast. However, the class `Object` includes two copy methods: **`shallowCopy`** and **`deepCopy`**. **`shallowCopy`** creates a new object, but pointers in the new object reference the same objects as the pointers in the old object. If `b` is assigned a shallow copy of variable `a`, and `b` contains an instance variable `s` that is a stack, then both `a` and `b` will share the same stack. A message to the stack that causes it to be changed (for example, **`pop`**) will be reflected in both `a` and `b`. One the other hand, if a message to `b` causes its `s` to be assigned a different stack, this assignment won't affect `a`'s instance variable `s`.

In contrast, **`deepCopy`** creates new copies of all instance variables in an object. If `b` is assigned to variable `a` by deep copy, then a change to `b`'s instance variables never affects `a`'s instance variables. Deep copying an object causes all its instance variables to be deep copied, which can lead to infinite loops in cyclic structures.

These distinctions also apply to equality testing. Smalltalk uses pointer equality; it is possible to program shallow and deep equality operations as well. Franz LISP provides all these operations, which can lead to confusion.

Classes are themselves objects, so they fit the same format as all other objects. For example, the class `Stack` is an object (of class `Stack class`). The instance variables of a class are the variables that define the properties of a class. Classes contain pointers to the superclass, class variables, and strings representing the name of the class and its variables (for display purposes). They also include "method dictionaries," which are hash tables that allow rapid access to methods. All messages are given a unique message code by the Smalltalk compiler. This message code is searched for in the method dictionaries of first the instance and then the class to determine if a corresponding method exists. If it does, the dictionary contains a pointer to the bytecode

(or primitive method) used to implement the method.

Messages are implemented in much the same way as ordinary procedure calls. The main difference is that the method to which a message is addressed is determined by the receiver of the message rather than by the sender. To transmit a message to an object, the sender pushes the object and the message's parameters (if any) onto the central stack. It then saves its own state (such as the program counter) and passes control to the appropriate bytecode, determined by the search described above. Space is allocated to accommodate the parameters as well as temporary variables for the method. A link to the caller (to allow return) is saved. A link is also created to the object containing the method that handles the message; this link allows access to instance (and indirectly) class variables, as well as class variables of superclasses. This device is very similar to the static chain used to implement imperative languages.

A given method always knows at compile time in which class or superclass a given variable is defined. As a result, all variables can be addressed at a known offset relative to some object (either the object handling the message or some superclass). Method execution is comparatively fast, since variable names don't need to be resolved at runtime. Once a method is finished, it uses a saved link to return to the caller and returns a pointer to the object computed by the method.

Bytecode is quite compact and reasonably efficient to execute. The main cost is that all computation is done with messages, and all messages must be resolved at runtime to determine the method that will handle the message. Consider Figure 5.19.

Figure 5.19

```
a := b + c
```

This program translates to the following bytecode sequence:

1. Push the address of `b` (the receiver of the message) onto the central stack.
2. Push the address of `c` (the parameter) onto the stack.
3. Construct a message with the message code for `+`. Search for that code in `b`'s instance dictionary, then its class dictionary, then superclass dictionaries in turn. Send the message to the method that is found.
4. Pop the result off the stack and store it as `a`.

In an ordinary compiler, this program translates to two or three instructions if `b` and `c` are simple integers. Smalltalk doesn't know until runtime what `b` and `c` are. In the case of `SmallIntegers`, things aren't too bad. The addresses of `b` and `c` encode their class membership, and a primitive method can be invoked. Nonetheless, substantially more than two or three instructions have been executed.

For all classes other than `SmallInteger`, a dictionary must be consulted to determine the method that will handle the message. For example, `+` might be used to concatenate strings or add stacks. The advantage of a using a primitive method is that the overhead of creating local space and linking to the method's object are avoided; the operation is performed directly on the objects on the central stack, and the resulting object replaces them.

4.6 Subtle Features

Blocks are implemented by closures. Even though a block may be executed in an environment quite different from its defining environment, it can still access its nonlocal variables correctly. Thus Smalltalk uses deep binding. Smalltalk manages to avoid the pitfall of accessing deallocated regions of a stack by using a garbage collector instead of a stack (with its explicit release) to manage object store. The anonymous method in Figure 5.20 prints 4.

Figure 5.20

```
|innerBlock outerBlock aVariable|                                    1
     outerBlock := [ :aVariable |                                    2
          innerBlock := [                                            3
               aVariable write -- write aVariable                    4
          ]                                                          5
     ] .                                                             6
     outerBlock value: 4 .                                           7
     aVariable := 6 . -- Try to confuse the issue                    8
     innerBlock value -- writes 4                                    9
!                                                                    10
```

Smalltalk methods are perfectly capable of coercing their parameters; only by looking at the documentation (or the implementation) can a programmer be sure what types are accepted by a method and whether the method will coerce types.

Even more surprising is the **become:** method provided by Object (and therefore available in all classes unless overridden). It is used as in Figure 5.21.

Figure 5.21

```
anObject become: anotherObject
```

After this message, all references to anObject and anotherObject are interchanged. Conventionally, anotherObject had no references before, so it now acquires references. This facility can be used to build abstract data types that change their implementation at some point. That is, in response to some message, an object may execute the code in Figure 5.22.

Figure 5.22

```
self become: newObject
```

From that point on, all references to the old object are rerouted to the new object, which could be of a different class entirely.

In many ways tree-structured inheritance rules are too restrictive. For example, I might have a class DisplayItem that represents items that can be graphically displayed on a screen. Objects of this class would respond to messages like **rotate:** or **highlight**. Another useful class might be InventoryItem, which represents items that I might inventory. Objects of this class would respond to messages like **reportBacklog** or **nameSupplier**. It would be nice to allow some objects to be both DisplayItems and InventoryItems (for example, a bumper or aircraft wing). This can only be done in Smalltalk 1.0 by making DisplayItem a subclass of InventoryItem or vice versa. Neither alternative is attractive, because not all objects of one class necessarily belong

to the other. (For example, I might be able to display a Saturn V rocket, but I probably won't have it in my inventory.)

A means of achieving **multiple inheritance**, in which a class is a direct subclass of more than one superclass, was introduced in Smalltalk 2.0. It is complicated to use (none of the built-in Smalltalk classes uses it), because there can be name conflicts among the multiple ancestral lines. Smalltalk 2.0 notices such conflicts at runtime and declares a conflict error. Since all classes are subclasses of `Object`, a class that inherits multiply sees `Object` along two different ancestry lines. The programmer needs to indicate whether such multiply defined ancestors are to be treated as a single ancestor or whether both are wanted. In the latter case, every invocation of a multiply defined method or access to a multiply defined instance variable must be qualified to indicate which ancestor is meant. In Eiffel, the programmer may rename inherited identifiers to avoid such name conflicts. Circular inheritance is always disallowed.

5 ◆ C++

C++ was developed at AT&T by Bjarne Stroustrup, who wanted to write event-driven simulations for which Simula would have been ideal but would also have been too inefficient. The original version of the language was developed in 1980; at that time it was known as "C with Classes" and lacked a number of its present features. The name C++ was coined by Rick Mascitti in 1983 as a pun on the C operator `++`, which increments its operand. C++ is explicitly intended as the successor to C. (The same pun has been used to name [incr Tcl], an object-oriented enhancement to Tcl.) C++ was implemented for some time as a preprocessor that generated C. Full compilers are now available. C++ has an ANSI standard and a standard reference, the ARM [Ellis 90], which also includes some of the design rationale. Of particular interest is Meyers' book [Meyers 92], which explains how to use some of the language features and also why C++ does things the way it does.

5.1 The Consequences of Static Binding

Most of the differences between C++ and Smalltalk can be explained by the fact that C++ is designed to be an efficient, compiled language. It performs as much binding as possible statically, not dynamically. Unlike Smalltalk, C++ is a statically typed programming language. Every identifier in a C++ program has a type associated with it by a declaration. That type can be either an ordinary C type or a class.

One consequence of static typing is that C++ does not allow classes to be introduced at runtime, unlike Smalltalk, in which introducing a class is a runtime operation accomplished by an appropriate invocation to the superclass. For this reason, the class hierarchy based on the subclass relation is less extensive than in Smalltalk. It is common for C++ programs to build many top-level classes, whereas in Smalltalk, all classes are subclasses, directly or indirectly, of the class `Object`.

Another consequence of static typing is that classes are not themselves objects. C++ programs have no hierarchy of instances and classes. In this regard, C++ displays less uniformity (in the sense introduced in Chapter 1)

than Smalltalk. On the other hand, the result is perhaps easier to comprehend. C++ has no need to introduce metaclasses.

A third consequence of static typing is that polymorphism in C++ is much more limited than in Smalltalk. It is not possible to build heterogeneous stacks, for example, except by the awkward trick of declaring the elements to be members of a statically declared choice type (in C++, called a "union") or by circumventing type checking by casting of pointers. However, C++ follows Ada's lead in providing generic classes. (Ada's generic modules are discussed in Chapter 3). I will show later how to implement a generic `Stack` class.

In order to generate efficient code, C++ tries as far as possible to bind method invocations to methods at compile time. Every variable has a known type (that is, its class), so the compiler can determine exactly which method is intended by any invocation. If a subclass introduces an overriding method or instance-variable declaration, then variables of that subclass use the new method or instance variable. The programmer may still access the hidden identifiers by qualifying accesses by the name of the superclass.

C++ must deal with variables declared to be of one class and assigned a value of a subclass. In particular, any pointer variable may be assigned a pointer to an object of its declared class `C` or any direct or indirect subclass `S`.[2] The compiler cannot tell whether this variable will be pointing to an object of its declared class `C`; it may be dynamically assigned an object of subclass `S`. Therefore, the compiler cannot tell for certain which method to use if `S` overrides a method of its superclass `C`. C++ solves this problem by distinguishing static and dynamic binding of methods.

By default, all binding is static. In order to force the compiler to generate the more expensive code necessary to defer binding until runtime, the programmer must declare that the method in the superclass `S` is **virtual**. Objects of class `S` (and its subclasses) contain not only fields for the instance variables, but also pointers to the code for all virtual methods.

It is also possible for the programmer to specify dynamic binding for a particular method and not have `C` implement that method at all. Such a method is called **pure virtual**. In this case, subclasses of `C` are expected to provide the method; it is erroneous to invoke a method that is not provided by an object or one of its superclasses. For example, class `Complex` could be a subclass of `Magnitude`, which could define a `max` operation. In Smalltalk, if you send a `max` message to a `Complex` object, the inherited version of `max` will be automatically invoked; all binding is dynamic. This method might in turn invoke the `>` method, which is also provided by `Magnitude`. However, `Magnitude`'s version of `>` is not meant to be invoked; it is meant to be overridden by a method introduced in subclasses like `Complex`. `Magnitude`'s `>` method just generates an error message. In C++, the `>` method would be declared as a pure virtual method of `Magnitude`, and `Complex` would be obliged to provide it.

If the programmer knows that a particular pointer (declared to be pointing to a value of class `C`) in fact references a value of subclass `S`, a method specific to `S` may be invoked by fully qualifying it with the subclass name. This

[2] Simula has exactly the same problem and uses the same solution. However, in Simula, all variables of object type are actually pointers to objects; in C++, a variable may either have a stack-based value or point to a heap-based value.

qualification leads to a runtime check to make sure the value is in fact of class S.

5.2 Sample Classes

I will rely on examples to describe C++. The syntax of C++ is compatible with the syntax of C; the examples use correct C++ syntax. The first example shows how to introduce complex numbers in C++, even though the standard C++ library already includes an implementation of complex numbers.

I first declare a new class Complex and make it a subclass of Magnitude, which I will not show here. A Complex object contains two floating-point numbers, one to hold the real part of the number and one to hold the imaginary part. In Smalltalk, a program creates a new class dynamically by sending a message to its intended superclass, in this case Magnitude. In C++, the programmer creates a new class statically by declaring it, as in Figure 5.23.

Figure 5.23

```
class Complex : Magnitude {                                    1
     double realPart;                                          2
     double imaginaryPart;                                     3
};                                                             4
```

The braces { and } take the role of **begin** and **end**. The class Complex is declared in line 1 to be a subclass of Magnitude. Top-level classes omit the colon and the name of the superclass. Complex contains two instance variables, realPart and imaginaryPart, both declared to be of type double. Instance variables are called "data members" in C++, and the methods are called "member functions." I will continue to follow Smalltalk nomenclature for consistency.

The first operation I will declare is to create and initialize a complex number. The Smalltalk class inherits a method called **new** from its superclass for this purpose. The C++ compiler provides a default new function that is passed a hidden parameter that specifies the amount of space to be allocated from the heap; an explicit allocator function can be provided if the programmer desires. Complex variables can also be allocated from the central stack in the normal manner without recourse to the new function, as in Figure 5.24.

Figure 5.24

```
Complex *pz = new Complex; // allocated from the heap          1
Complex z; // allocated from the central stack                2
```

The comment delimiter in C++ is // . The * in line 1 declares pz as a pointer type, pointing to objects of type Complex. The proper version of new is specified by adding the class name.

I will rely on the defaults provided to allocate Complex objects; however, I must provide a way to initialize such objects. In Smalltalk, I would establish a **real:imaginary:** method to set the values of a Complex object. C++ allows the program to provide an initializer (also called a "constructor") and a finalizer (also called a "destructor") for each class. The **initializer** is called each time an object of the class is created (either explicitly or implicitly, as when a parameter is passed by value), and the **finalizer** is called whenever an instance of the class goes out of scope or is explicitly freed. Initializers can be used to establish values of instance variables; finalizers can be used to free

storage pointed to by instance variables. Both are good for gathering statistics.

A reasonable declaration of Complex, including some procedure headers that I will need later, is given in Figure 5.25.

Figure 5.25

```
class Complex {                                                    1
private: // the following are generally hidden                     2
     double realPart;                                              3
     double imaginaryPart;                                         4
public: // the following are generally visible                     5
     Complex(); // initializer                                     6
     Complex(double,double); // another initializer                7
     ~Complex(); // finalizer                                      8
     Complex operator << (ostream); // write                       9
     int operator > (Complex); // compare                          10
};                                                                 11
```

Lines 6–10 introduce methods. C++ does not use a keyword **procedure**; the presence of the parentheses for the parameter lists indicates that procedures are being declared. The fact that the procedures in lines 6–7 have the same name as the class is understood to mean that they are initializers. The compiler resolves the overloaded initializer identifier by noting the number of parameters and their types. The name of the finalizer is the name of the class preceded by a tilde ~, as in line 8. Initializers and finalizers do not explicitly produce a result, so they are not given types. The operators <<, used for output, and >, used for numeric comparison, are overloaded as well (lines 9–10). The ostream type in line 9 is a class used for output and declared in a standard library. Comparison returns an integer, because C++ does not distinguish Booleans from integers. The operator procedures do not require two parameters, because a Complex value is understood to be presented as the left-hand operand.

So far, the example has only included the method headers, that is, the specification of the methods. The implementation of each procedure (declarations of local variables and the body) may be separated from the specification to promote modularity. C++ also allows the implementation to be presented immediately after the method specification (within the scope of the **class** declaration). Immediate placement informs the compiler that the programmer intends calls on such methods to be compiled with inline code. It is usually better programming practice to separate the implementations from their specifications, perhaps in a separate file. Figure 5.26 presents separate implementations of the initializers specified in Figure 5.25:

Figure 5.26

```
Complex::Complex()                                                 1
{                                                                  2
     realPart = 0;                                                 3
     imaginaryPart = 0;                                            4
};                                                                 5
```

```
Complex::Complex(double real, double imaginary)                    6
{                                                                  7
     realPart = real;                                              8
     imaginaryPart = imaginary;                                    9
};                                                                 10

// sample usage in a declaration                                   11
Complex z1 = Complex(5.0, 7.0);                                    12
Complex z2; // will be initialized by Complex::Complex()           13
```

In lines 1 and 6, the procedure names are qualified by the class name. C++ uses :: instead of . to indicate qualification. In lines 3, 4, 8, and 9, instance variables are named without any qualification; they refer to the variables in the instance for which the procedure is invoked.

The next operation (Figure 5.27) prints complex values; the specification is in line 9 of Figure 5.25.

Figure 5.27

```
Complex Complex::operator << (ostream output)                      1
{                                                                  2
     output << realPart;                                           3
     if (imaginaryPart >= 0) {                                     4
          output << '+';                                           5
          (output << imaginaryPart) << "i";                        6
     }                                                             7
     return *this;                                                 8
};                                                                 9

main() { // sample usage                                           10
     z1 << cout; // cout is the standard output stream             11
}                                                                  12
```

The way I have defined the Complex operator << (line 1) requires that it output complex values as shown in line 11, instead of the more stylistic cout << z. There is a way to define the operator that avoids this reversal, but it is more complicated; I will show it later. Line 6 shows that the << operator for doubles returns the stream; the stream is then given another << message with a string parameter. Clearly, << is highly overloaded.

A Smalltalk > method for Complex must have access to the instance variables of both operands. That is, the object receiving the > message must be able to inspect the instance variables of the parameter. But Smalltalk objects never export instance variables. The IntegerStack example in Figure 5.17 (page 152) shows how to wrestle with this problem; I needed to define private selector methods. Alternatively, I could have introduced a **magnitude** unary operator. Neither solution is particularly elegant.

C++ addresses this concern by allowing the programmer to relax the walls of separation between objects. Members can be declared public, protected, or private. By default, instance variables are private. The procedures shown in Figure 5.25 (page 160) are declared public. The following chart shows what accesses are permitted for each level of security.

	Same	Friend	Subclass	Client
Smalltalk instance variables	n	—	y	n
Smalltalk methods	y	—	y	y
C++ public members	y	y	y	y
C++ protected members	y	y	y	n
C++ private members	y	y	n	n

Each entry shows whether a member of an object O of class C is exported to various contexts. "Same" means other objects of the same class C. C++ allows such contexts access to instance variables; Smalltalk does not. Therefore, the > operator in C++ has permission to access the instance variables of its parameter. "Friends" are procedures or classes that class C declares to be its friends. Friends are permitted to refer to all members of instances of C. More restrictively, a method M may declare procedures and classes to be its friends; those friends are permitted to invoke M even though M may be hidden to others. "Subclass" refers to code within subclasses of C. Subclasses have access to all members except for private ones in C++. "Clients" are instances of classes unrelated to C.

The design of C++ makes it easier to deal with binary operations that take two instances of the same class than in Smalltalk. All instance variables of one instance are visible to the other. For example, the implementation of > (the specification is in line 10 of Figure 5.25 on page 160) can inspect the instance variables of its formal parameter right (see Figure 5.28).

Figure 5.28

```
int Complex::operator > (Complex right)                          1
{                                                                2
     double leftmag, rightmag;                                   3
     leftmag = (realPart * realPart) +                           4
          (imaginaryPart * imaginaryPart);                       5
     leftmag = (right.realPart * right.realPart) +               6
          (right.imaginaryPart * right.imaginaryPart);           7
     return leftmag > rightmag;                                  8
}                                                                9
```

C++ lets subclasses further restrict the visibility of identifiers by explicitly redeclaring all inherited public and protected identifiers protected or private. (Subclasses do not inherit private identifiers.)

Other security arrangements are possible. In Oberon-2, instance variables may be exported read-write, exported readonly, or not exported at all [Reiser 92]. Oberon-2 does not distinguish between exports to other classes and inheritance by subclasses.

Polymorphism in C++ is achieved by generic classes. A generic Stack class that can be instantiated (at compile time) to become a class of any given type, including a stack of stacks, can be written as in Figure 5.29.

Figure 5.29

```
#define MAXSIZE 10                                                    1

template <class BaseType> class Stack {                               2
private:                                                              3
     BaseType elements[MAXSIZE];                                      4
     int count;                                                       5
public:                                                               6
     Stack() { // initializer                                         7
          count = 0;                                                  8
     }                                                                9
     void Push(BaseType element) {                                    10
          elements[count] = element;                                  11
          count = count + 1;                                          12
     }                                                                13
     BaseType Pop() {                                                 14
          count = count - 1;                                          15
          return(elements[count]);                                    16
     }                                                                17
     int Empty() {                                                    18
          return(count == 0);                                         19
     }                                                                20
     friend ostream                                                   21
          operator << (ostream output, Stack<BaseType> S) {           22
          int index;                                                  23
          output << "[";                                             24
          for (index = 0; index < S.count; index++) {                 25
               output << S.elements[index];                           26
               if (index+1 == S.count) break;                         27
               output << ",";                                         28
          }                                                           29
          output << "]";                                             30
          return(output);                                             31
     } // <<                                                          32
}; // Stack                                                          33

main(){ // sample usage                                               34
     Stack<int> myIntStack;                                           35
     Stack<float> myFloatStack;                                       36
     Stack<Stack<int> > myRecursiveStack;                             37
     myIntStack.Push(4);                                              38
     myIntStack.Push(8);                                              39
     cout << myIntStack; // [4,8]                                     40
     myFloatStack.Push(4.2);                                          41
     cout << myFloatStack; // [4.2]                                   42
     myRecursiveStack.Push(myIntStack);                               43
     cout << myRecursiveStack; // [[4,8]]                             44
}                                                                     45
```

The definition in line 1 effectively declares MAXSIZE a constant with value 10. The **template** declaration in line 2 indicates that Stack is parameterized by a type (literally, by a class). I have ignored all error conditions in the methods for simplicity's sake. The declarations in lines 35–37 show how to supply parameters to the generic class; a generic class must have all parameters bound in order to declare a variable. The compiler compiles a specific class sepa-

rately for each instantiation of the generic class. In this example, three specific classes are compiled. Line 37 shows that it is possible to declare stacks of stacks of integer. (The extra space between the > characters is needed to prevent the parser from misinterpreting them as the single >> operator.) The header for the overloaded operator << (lines 21–32) is unusual; it is declared to be a friend of `ostream`, so that the stream output routines have access to the contents of the stack. The stack itself is passed as a parameter, so that the output statements of lines 40, 42, and 44 can be written with `cout` on the left, as is proper style in C++, not the right, as I have been doing previously.

6 ◆ FINAL COMMENTS

At first glance, object-oriented languages are just a fancy way of presenting abstract data types. You could argue that they don't present a new paradigm of programming, but rather a structuring principle that languages of any sort might employ. However, I would counter that Smalltalk and C++ have developed the concept of abstract data type into a new form of programming.

First, object-oriented programming provides a new view of types. The type of an object is the protocol it accepts. Two objects are type-compatible if they respond to the same set of messages. This view is highly abstract, because it doesn't say the objects have the same form, only that they are functionally interchangeable. There is no straightforward way to check type compatibility.

Second, the nature of instantiation distinguishes a class from an abstract data type exported from a module. Each instance is independent, with its own data and procedures, although all instances may share common class variables. Data types exported from a module may be instantiated, but the exporting module itself cannot be. A module is much more of a static, compile-time, passive entity. A class is more dynamic, with more runtime and active qualities.

Third, the hierarchy of subclasses leads to a style of programming known as **programming by classification**. The abstract data types are organized into a tree, with the most abstract at the root and the most specified at the leaves. Incremental modifications to a program are accomplished by introducing new subclasses of existing classes. Each new class automatically acquires much of what it needs by reference to its superclass. Methods are automatically overloaded. Programming by classification is an important tool for achieving reuse of valuable code, and this tool goes well beyond the reuse that comes from modularization into abstract data types.

Smalltalk is attractive in many ways. It provides a highly interactive and integrated programming environment that uses the latest in computer technology (in short, a graphical user interface). Its object-oriented style provides an interesting inversion of our usual view of programming. Objects are preeminent, uniform, and autonomous. They cooperate by passing messages, but no object controls any other. Objects are fairly robust, since the worst thing a program can do is send one a message it can't handle. In this case, the object doesn't fail, but rather ignores the message and issues an error.

Smalltalk is not without its problems, both real and perceived. It has only recently become generally available for small machines. Smalltalk may have

been too quick to abandon concepts common in imperative programming languages. For example, it makes no provision for named constants; you must create a variable and then be careful not to change it. This shortcoming could easily be overcome without injustice to the design of the language. Similarly, it would be nice to introduce some type checking (even if dynamically checked) by allowing variables to have a type (that is, a class) specifier. Programs would be easier to read and less error-prone. It might also make manipulation of variables more efficient, particularly for primitive types. As a small step in this direction, Smalltalk suggests that identifiers like `anInteger` or `aStack` be used conventionally for variables whose class is meant to be fixed. Of course, an identifier named `anInteger` need not actually map to an integer under Smalltalk's rules. Finally, Smalltalk provides no control over whether identifiers are exported (instance variables aren't, methods are) or inherited (all are). A finer level of control, such as that provided by C++, could improve the safety of programs.

C++ fixes some of these problems. First, types and methods are bound at compile time. Numeric values and code blocks are not objects at all. There is no need to understand expressions as evaluated by messages sent to objects. The result is that C++ programs execute quite efficiently, because they usually avoid Smalltalk's dynamic method binding, use ordinary procedure invocation, and use inline code to accomplish arithmetic. Second, members can be individually controlled with regard to export and inheritability. The concept of friends allows identifiers to be exported only to instances of particular classes.

However, C++ suffers from its ancestry; C is notorious for being error-prone (the operators for equality and assignment are easily confused, for example), syntactically obscure (complicated types are hard for humans to parse), and unsafe (loopholes allow all type checking to be circumvented).

Other object-oriented languages have been designed, of course. Best known perhaps is Eiffel [Meyer 92]. Nor is C the only language that has been extended to give it an object orientation. Other good examples include CLOS (the Common LISP Object System) and [incr Tcl]. Some people believe that Ada 95 will be the most widespread object-oriented programming language in a few years. Even object-oriented COBOL has been considered [Clement 92].

EXERCISES

Review Exercises

5.1 What are the consequences in C++ of static typing?

5.2 Does an object in Smalltalk require its own private stack? In C++?

5.3 Write a class in Smalltalk and/or in C for rational numbers, that is, numbers that can be represented by an integer numerator and denominator. Instance variables should include both the numerator and the denominator. Your implementation should always reduce fractions to their lowest terms. You must overload all arithmetic and conditional operators.

5.4 Consider an array of stacks, some of whose components are implemented one way, while others are implemented the other way. Is this a homogeneous array?

5.5 How would you implement the array of stacks mentioned in Problem 5.4 in C++?

Challenge Exercises

5.6 Simula classes contain procedures. Unlike ordinary procedures, procedures inside classes may not declare **own** variables, that is, variables whose values are retained from one invocation to the next. If you want to add such a feature, what would you like it to mean?

5.7 In Smalltalk, not everything is an object. Name three programming-language entities that are not objects. Could Smalltalk be modified so that they are objects?

5.8 Show how the **timesRepeat:** method in Figure 5.11 (page 147) could be coded.

5.9 In line 12 of Figure 5.17 (page 152), show how an element of a different class could masquerade as an `Integer` and bypass the type check.

5.10 `True` and `False` are subclasses of `Boolean`. Each has only one instance (`true` and `false`, respectively). First, how can class `True` prevent other instances from being created? Second, why not use the simpler organization in which `Boolean` has two instances? Hint: Consider the code for **ifTrue:ifFalse:**.

5.11 Build a method for `Block` that accepts a **for:from:to:** message to implement **for** loops. Don't use **whileTrue:**.

5.12 Build a subclass of `Block` that accepts a **for:in:** message to implement CLU-style iterators.

5.13 Defend Smalltalk's design decision that error messages are to be generated by objects via messages to themselves, and that the **error:** method is to be inherited from `Object`.

5.14 Why should `Magnitude` define methods like > but give them error-generating code? In other words, what is the point of introducing pure virtual methods?

5.15 In Smalltalk, a new class is constructed at runtime by sending a message to its superclass. In C++, classes are constructed at compile time by declaration. Show how the Smalltalk method is more powerful.

5.16 Enumerate what is missing in Smalltalk and in C++ for building abstract data types.

5.17 What is the effect of a C++ class declaring that it is its own friend?

5.18 C is not block-structured. In particular, one cannot introduce a type within a name scope. What complexities would be introduced if C++ were based on a block-structured language, and classes could be introduced in a name scope?

5.19 Is a class a first-class value, a second-class value, or neither, in Smalltalk and in C++?

5.20 In C++, say there is a class `A` with a protected instance variable `varA`. Subclasses `B` and `C` inherit this variable. May instances of `B` and `C` access each other's copy of `varA`?

5.21 In Figure 5.29 (page 163), I went to considerable trouble to allow output statements to place `cout` on the left of the `<<` operator. Why was this so important for this example?