CHAPTER 10

Subclasses and Subtypes

There is a paradox that lies at the heart of the way inheritance and substitution are used in statically typed object-oriented languages. This paradox derives from the twin concepts of *subclass* and *subtype*. In this chapter we will explore these concepts and this paradox.

To say that one class is a *subclass* of another is to simply assert that it has been built using inheritance. The new class is declared using an existing class as a basis, as in the following Java class declaration.

```
class Child extends Parent {
    ... // class definition
}
```

The point is that the subclass relationship is asserting a statement about definition, about how the new class was constructed. It says nothing about the meaning or purpose of the child class.

10.1 ▫ Substitutability

One of the more interesting features of statically typed object-oriented languages is that the type associated with a *value* held by a variable may not exactly match the type associated with the *declaration* for that variable. We saw this near the end of the billiard simulation program in Chapter 7, where a variable declared as a GraphicalObject in fact held a value of type Ball, Wall, or Hole. To appreciate how unusual this is, note that variables in conventional typed programming languages never have this property. A variable declared as a Integer, for example, can never hold a value of type String.

In Chapter 8 we termed this property of object-oriented languages the principle of *substitution* (sometimes called *subsumption*). A variable declared as a parent class can hold (that is, be filled with) a value that is an instance of a child class. The variable declared as pointing to a GraphicalObject in Chapter 7 could, in fact, be pointing to a Ball. Barbara Liskov described substitution this way.

> What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a *subtype* of T. [Liskov 1988]

10.2 ▫ Subtypes

To understand the concept of a *subtype* and how it relates to a subclass, we must first explore how types are used in programming languages. For an idea that, at least with experience, seems so intuitive, the exact nature of types may seem surprisingly complicated.

When we say that a value, such as the number 17, has an integer type, what are we asserting? One answer is that we are asserting that the value is a member of a set of values. The name int is the handle by which we describe this set. In Java, for example, the set of integers represents the whole numbers that range from −2147483648 (which is -2^{31}) to 2147483647 (which is $2^{31} - 1$). Other primitive types denote similar sets of values.

But there is more that we know by virtue of the assertion that a value has type int. We know, for example, that we can perform a variety of *operations* on the value, such as addition, subtraction, multiplication, and division. We are not permitted to perform these operations on a value of type boolean, as the definition of the Boolean data type does not include these tasks.

So in one sense, a type is simply a set of values and a set of operations. But we know yet more about integers. We expect integers to satisfy a set of *properties*. If we add the values 3 and 5, we expect to produce the value 8. This we always expect to be true. On the other hand, if we divide the integer 8 by the integer 5, we get 1 and not the fractional value 1.6. We are not surprised at this because a property of division for integers in most languages is to truncate any fractional remainders. There are further features of type we could explore, but these three—values, operations, and properties—are sufficient for our discussion of subtypes.

Now consider how the term *type* is used when we describe an *abstract data type*, or ADT. Let us employ for our example the concept of a stack. In Java, for example, the starting place for the description of an ADT is the definition of an interface. This might be described, for example, using the following declaration.

```
interface Stack {
  public void push (Object value);
  public Object top ();
  public void pop ();
}
```

But the interface is just the beginning. Beyond the interface is the meaning, the properties that we want to associate with this data type. A Stack, for example, is characterized by the LIFO (Last-In, First-Out) property. The item retrieved by a top operation must be the most recent value pushed onto the stack. If that value is removed (by executing pop), then the next item at the top of the stack should be the item inserted prior to the removed value, and so on. A mental image, such as a stack of plates, provides a good intuition regarding the utilization of this data type.

Syntax versus Semantics

Although not an exact correspondence, the issue of subclass versus subtype is in some ways similar to the more familiar language distinction between syntax and semantics. Syntax deals with how a statement can be written, just as subclasses deal with how a class is declared. Semantics deals with what a statement means, just as subtypes deal with how a child class preserves the meaning of a parent class.

The following class definition satisfies the Stack interface but does *not* satisfy the properties we expect for a stack, since it violates the LIFO property for all but the most recent item placed into the stack.

```
class NonStack implements Stack {
  public void push (Object value) { v = value; }
  public Object top () { return v; }
  public void pop () { v = null; }

  private Object v = null;
}
```

From this example we see that the properties that are key to the meaning of the stack are not specified by the interface definition. And it is not that we were lazy; Java (like most other languages) gives us no way to specify the properties that an interface should satisfy. And thus most programmers (and most programming

languages) resort to something much less formal, such as an English language description of the class requirements.

Having worked through this explanation, we are now in a position to understand how a *subtype* differs from a *subclass*. To say that a new class is a subtype of an existing class is to assert that the new class provides all of the operations of the existing class and *furthermore* satisfies the properties associated with the class. For example, any method in the new class that matches in type signature a method in the old class also satisfies the properties associated with that method. (The new class is free to add new operations, of course.)

The subtype relationship is described purely in terms of behavior; it says nothing about how the new class is defined or constructed. As the class NotStack demonstrates, it is easy for a child class to be a subclass and not a subtype. It is also easy to construct examples that represent a subtype that is not a subclass. For

Types Are a Poor Man's Specifications

The most important reason for having types in programming languages is to eliminate typing errors. Andrew Black calls types "a poor man's specification." By this he means that what we really would like to ensure, when an argument is passed to a procedure or a value is assigned to a variable, is that the argument values satisfy the requirements of the procedure and that those requirements should really be expressed in terms of specifications. Any value that satisfies the specifications (that is, any subtype) should be permitted, even if it is not formed as a subclass.

But formal specifications are both difficult to write and difficult to check or verify. Types, on the other hand, are by comparison almost trivial for a compiler to check. Therefore, even though types are not exactly what a programmer or a language designer might like, they do have the following advantages.

- They are a useful approximation.
- They are simple to explain.
- They are easy to check.

Thus, computer languages continue to treat types as a shorthand for specifications, despite their obvious shortcomings.

instance, the Smalltalk type Array defines a data type with roughly the following interface.

```
at: index put: value
at: index
size
```

The properties we associate with this data type include the fact that if a value is placed into the collection using a given index, then when the array is subsequently accessed using the same index the value will be returned. The Dictionary data type supports the same interface but does not limit the index values to being integer. Even though the class Dictionary has no inheritance relationship to the class Array because it supports the same interface, we can say that Dictionary is a subtype of Array.

Another way to look at the subclass/subtype distinction is by extending the parent/child metaphor. Parent classes provide the genetic makeup for a class. As any parent knows, genes have an influence on behavior, but they do not dictate it. The concept of subtype, on the other hand, describes behavior without reference to pedigree.

10.3 ▫ The Substitutability Paradox

Recall the argument in Chapter 8 that justified the principle of substitutability for statically typed object-oriented languages. It went something like this.

1. A child class inherits all the data fields defined in the parent class.
2. A child class must recognize all the behaviors associated with the parent class, either inheriting them directly or overriding them but preserving the type signature.
3. Therefore, an instance of a child class can be used in any situation where an instance of the parent class is expected, with no observable difference.

With the benefit of our knowledge of the subclass/subtype distinction, we see that the weak link in this argument is Step 2. Simply asserting that a child class satisfies the interface common to the parent does not ensure that it will satisfy any or all of the properties of the parent. (Remember the class NonStack.)

So why don't programming languages use the subtype relation rather than the subclass relation? Here we run into the fundamental bedrock of computer science: the *halting problem*. The halting problem asserts that there is no procedure that can, in general, tell if an arbitrary program will ever terminate. One of the classic corollaries of this principle is that there is no procedure that can determine, in general, if two programs have equivalent behavior. From this it is only a small

step to showing that it is impossible to tell (again, in general) if a method in a child class matches in behavior a method in the parent class.

And so we arrive at the paradox of substitutability. In statically typed object-oriented programming languages, the veracity of substitutability for assignment is determined by ensuring that the type of a value is a *subclass* of the type of the variable being assigned. And yet such an assignment only makes logical sense if the type of the value is a *subtype* to that of the variable.

This is not to say that languages do not make some attempt to ensure that subclasses remain subtypes. Programmers are not generally permitted to delete an inherited method or hide a public data field, for example. Nor is it permitted to change the type signature of a method inherited from the parent (although this issue is subtle, as we will see in Chapter 15). But in the end all of these restrictions, while useful, are insufficient. There is simply no way that a compiler can ensure that a subclass created by a programmer is indeed a subtype.

In fact, great havoc can ensue if values are assigned through substitution that do not satisfy the subtype relationship. Imagine, for example, using an instance of NonStack in a situation where a Stack data type is expected.

10.3.1 Is this a problem?

Notice what it takes to create a subclass that is not also a subtype. To do so it is necessary to redefine an operation that is inherited from a parent class but to do so in a way that compromises some property that the parent class expects and without violating the type signature of the method. One can legitimately ask how common this problem can be. The answer is, probably not very common at all. But nevertheless when it does occur, it is almost certainly a source of annoyance, if not error. And the fact that such a fundamental problem would lie at the heart of something so basic as substitutability is a frustration to those who would aspire to place programming on a more secure theoretical foundation.

10.4 ▫ Subclassing for Construction

A common situation where subclasses are created that are not subtypes is when inheritance is employed purely for the reuse of code. This is termed subclassing for construction, and we have discussed this idea already in Chapter 8. For example, suppose a C++ programmer needs to create a dictionary abstraction that will hold key/value pairs composed of strings and integers. The programmer decides to use the standard STL class list for the container (we will discuss the STL library in detail later in Chapter 20).

The first step is to define a small helper class for each element.

```
class association { // a single key/value pair
public:
  association (string s, int i)
    { key = s; value = i; }
  string key;
  int value;
};
```

The next step is the critical juncture. In defining the dictionary class the programmer decides to use inheritance and declare the new abstraction as a subclass of the existing list class.

```
class dictionary : public list<association> {
public:
  void add (string key, int value) {
    association ele(key, value);
    push_front(ele); // add to list
  }

  int at (string key) { // find value stored at given key
    iterator start, stop;
    stop = end();
    for (start = begin(); start != stop; ++start) {
      association ele = *start;
      if (ele.key == key)
        return ele.value;
    }
    return -1;
  }
}
```

Because the new class inherits from list, all the behaviors defined for the existing class become available to users of the new abstraction. For example, you can determine how many elements are being held by a dictionary using the list method size.

```
dictionary d;
cout << "started" << '\n';
d.add("abc", 42);
d.add("pdq", 12);
cout << "value of abc is " << d.at("abc") << '\n';
cout << "number of elements in dictionary is "
  << d.size() << '\n';
```

But for all the advantages of this arrangement, there are disadvantages as well. All of the list methods can now be used with this new class, and some of them may not be appropriate. For example, the method pop_front removes the first element from the collection. But what exactly is the first element of a dictionary?

```
d.pop_front();
cout << "number of elements in dictionary is "
  << d.size() << '\n';
```

*10.4.1 Private inheritance in C++

The problem with the dictionary abstraction is that the mechanism of inheritance is bringing with it too much semantic baggage. In this case, the programmer wants to make use of inheritance within their own class, but does not want those operations to pass through the class and be available to the world at large. In short, the programmer wants the advantages of subclassing (access to both the data and behavior defined by the parent) but does not want to assert that their new class is a subtype.

One way to do this in C++ is to use *private inheritance*. Private inheritance is indicated by using the keyword private in the class header instead of the keyword public.

```
class dictionary : private list<association> {
public:
  void add (string key, int value) { ... }

  int at (string key) { ... }

  int size () { return list<association>::size(); }
};
```

As with normal inheritance, a private inheritance makes the facilities of the parent class accessible to the new class. But unlike a public inheritance, the abilities of the parent are not accessible outside the class definition. Thus, the programmer must explicitly define whatever behavior they want associated with the new class. For example, the programmer has defined the method size so that it returns the number of elements in the list, making use of the similarly named method from the parent class. In this fashion the programmer has created a subclass of list that is not a subtype of list inasmuch as it cannot be assigned to a variable declared using the parent class.

10.5 ▫ Dynamically Typed Languages

The contrast between subclasses and subtypes takes on a slightly different form in dynamically typed languages, such as Smalltalk, because types themselves are used differently in such languages. The distinction between subclasses and subtypes is perhaps easier to appreciate if we consider argument values. For example, consider the following Smalltalk method from class Magnitude.

```
" class Magnitude "

  between: low and: high
    ↑ (low <= self) and: [ self <= high ]
```

Although the formal arguments low and high are not typed, there are nevertheless implicit restrictions placed on them. In particular, low must recognize the <= message (operators are simply messages in Smalltalk), and high must be usable as an argument to such a message. It is assumed, but not stated, that these messages define an ordering on their values. Thus, there is an implicit typing for these variables. (The set of operations that a variable must recognize is sometimes termed its *protocol*.) Any value passed as an argument to this method must understand these messages. This assertion says nothing about the class of the arguments. While class Magnitude in Smalltalk defines these operations, nothing prevents the Smalltalk programmer from implementing the appropriate operations in an unrelated class and thereafter using instances of the class with this method. Thus, we say that arguments for this method are expected to be of a certain subtype (the type implicitly defined by the messages it must understand) but not necessarily a specific subclass.

The disadvantage of dynamic typing is that type errors may not be caught until run time. Using an incorrect value as an argument may result in a "receiver does not understand" error, something that cannot occur in statically typed languages.

Although template or generic arguments in C++ are not dynamically typed, they have similar properties. The following is a simple but typical template function in C++.

```
template <class T>
T max (T left, T right)
{
  if (left < right)
    return right;
  return left;
}
```

It is only by examining the body of the function that the programmer can determine that the argument values must understand the < operator. One advantage of templates over dynamic typing is that errors resulting from values that do not support the required operations will be caught at compile time, although frequently the error messages that result from these situations are difficult to decipher.

*10.6 ▫ Pre- and Postconditions

One traditional way to specify the behavior of methods in an abstract data type is through pre- and postconditions. A precondition describes features of the state that must be true before a method can be executed, whereas a postcondition specifies the resulting state produced by executing a method. While most programmers use pre- and postconditions informally as comments in their code, some programming languages include formal notation for specifying these features.

The language Eiffel takes this idea one step further. It is possible to specify pre- and postconditions for methods, and these are inherited by child classes. While a child class can override the body of a method, they are not allowed to override the pre- and postconditions, and furthermore, these conditions will be tested when the method is executed in order to assure that proper state is maintained. By ensuring a minimal level of functionality, preconditions and postconditions make it harder to create a subclass that is not also a subtype.

Figure 10.1 illustrates these features in an Eiffel class that defines, but does not implement, the Stack abstraction. Preconditions are indicated by a require clause. For example, the class requires that a stack is not empty before a pop operation can be performed. Postconditions are indicated by ensure clauses. For example, when an item is pushed on the stack, the class ensures that the stack is not empty. Ensure clauses are also allowed to access the value of data fields before the method was executed, using the old notation. Notice, for example, that this class ensures the count is properly updated by checking that the count is one larger than the previous count.

The problem with formal pre- and postconditions is that they seldom can capture all the features of interest. Note that these assertions ensure that the stack does not exhibit underflow or overflow but do not guarantee that an implementation will satisfy the fundamental LIFO property. Hence, the NonStack described earlier would satisfy both the preconditions and the postconditions and yet still not be a true stack. It is for reasons such as this that formal pre- and postconditions are not widely found in programming languages.

```
class Stack [ EleType ] feature
  count : INTEGER -- number of stack elements
feature
  empty : boolean is deferred -- is stack empty
  full : boolean is deferred -- is stack full?

  push (x : EleType) is
    require
      not full
    do deferred
    ensure
      not empty
      count = old count + 1
    end

  pop () is
    require
      not empty
    do deferred
    ensure
      count = old count - 1
      not full
    end
end
```

Figure 10.1 — An Eiffel program showing the use of preconditions

*10.7 ▫ Refinement Semantics

It has been noted that one cause of the problem of subclasses that are not subtypes is the fact that a child class can override a method from a parent class and completely replace it. Thus, whatever the parent method was doing is lost entirely.

One school of language design, the so-called Scandinavian school, asserts that this problem can be avoided by *always* executing the code from the parent class. The child class can add to this behavior but cannot completely replace it. Such an overriding is termed a *refinement* instead of a *replacement*. Notable object-oriented languages based on this principle include Simula (the first object-oriented language) and Beta.

Opponents of this view assert that while it is true that refinement semantics make it more difficult to create subclasses that are not subtypes, they still cannot make it impossible, and furthermore, they also make many other useful tasks more difficult—tasks that are not dangerous and that are in fact very desirable. So-called American-style replacement semantics, while in theory more dangerous, are for this reason also more useful.

Summary

- To say that one class is a subclass of another simply asserts that it was built using inheritance.
- By itself, the subclass relationship says nothing about the behavior of the child in relation to the behavior of the parent.
- The term *subtype*, on the other hand, describes the behavior of one class in relation to the behavior of another, regardless of how they were constructed.
- Most of the time the two terms intersect, although it is possible to build subclasses that are not subtypes and conversely to build subtypes that are not subclasses.

In strongly typed object-oriented languages a paradox arises around the principle of substitution. Because verifying the subclass relationship is easy and verifying the subtype relationship is impossible, languages permit substitution based on subclasses rather than subtypes. If a programmer is not careful that the subclasses are also subtypes, subtle errors that are difficult to detect can result.

Other topics we have discussed in this chapter include private inheritance in C++ (a technique for using inheritance to create child classes that are explicitly not subtypes), preconditions and postconditions in Eiffel, and refinement semantics.

Further Reading

Earlier in this chapter I argued that types are not specifications but are used as the next best thing because they are sufficiently close to being specifications and are inexpensive to implement. There is a considerable body of work that has as its goal coming up with a better typing system, thereby improving the specification power of types. See, for example, [Cardelli 1985, Raj 1991, Bruce 1994].

The Liskov quote is from [Liskov 1988]. Another frequently cited paper on subtypes is Cook [Cook 1990]. The terms "American semantics" and "Scandinavian semantics" are due to a different Cook [Cook 1988].

CHAPTER 11

Static and Dynamic Behavior

Much of the power of object-oriented languages derives from the ability of objects to change their behavior dynamically at run time. Understanding object-oriented programming mechanisms therefore requires an appreciation of the differences between static and dynamic behavior and the implications of this difference.

In programming languages, the term *static* almost always refers to a property or feature that is bound at compile time and cannot thereafter be modified. A statically typed variable, for example, means that the type associated with the variable is set at compile time and cannot change during the course of execution. The term *dynamic*, on the other hand, almost always refers to a property or feature that cannot be bound until run time. Thus, a dynamically typed variable will have its type determined by the value it is currently holding, which may change during the course of execution.

In this chapter we explore this static versus dynamic distinction in the context of three related concepts. First, we examine in detail static versus dynamic typing. Next, we explore the concepts of static and dynamic classes in statically typed languages. And finally, we investigate static and dynamic binding of message and methods.

11.1 ⊡ Static versus Dynamic Typing

The most obvious dichotomy in programming languages occurs between those languages that are statically typed and those that are dynamically typed. Table 11.1 describes a few of these, both object-oriented and non-object-oriented

	Object-Oriented	Non-Object-Oriented
Statically	C++, Delphi Pascal, Eiffel, Java, Objective-C (sometimes)	Ada, Algol, C, Fortran, Haskell, ML, Modula
Dynamically	Objective-C, Smalltalk, Dylan, Python	APL, Forth, Lisp, Prolog, Snobol

▫ Table 11.1 — Statically typed and dynamically typed languages

languages. All programming languages incorporate the concept of *type*. The difference between static and dynamically typed languages is the question of whether types are a property of variables or of values.

In a statically typed language, a type is attached to a variable at compile time. Most often this happens through the use of declaration statements.

```
int a; // variable a will maintain integer values
double m; // while variable m holds floating point quantities
```

Some statically typed programming languages use a more subtle process of *type inference*. The (non-object-oriented) language ML is the best known example of a language that uses this technique. Here the type associated with variables is inferred from the program, thereby avoiding the requirement that the programmer provide explicit declaration statements.

```
// b must be integer, since it is being added
// to the integer constant 2.
// therefore n must also be integer,
// as must be a
val a = (b + 2) / n;
```

In contrast, there are languages that are dynamically typed. In a dynamically typed language (sometimes called an *untyped* language) types are associated with values but not with variables. A variable is simply a name. Not only may the value that a variable holds change during the course of execution, but the type of value the variable is maintaining can also change, as in the following sequences of assignments in Smalltalk.

```
a <- 2. " value is currently integer "
a <- true. " now it is boolean "
a <- 'true'  " now it is string "
```

The dichotomy between statically typed and dynamically typed languages has ed since the earliest days of programming languages. Fortran and Lisp, both

created in the 1950s, were two early examples of each type. Arguments concerning the benefits of the two approaches have raged almost as long. Generally the arguments pit the competing goals of efficiency versus flexibility.

It is true that statically binding a language feature means that more can be done at compile time, and hence less work is generally required at run time. For instance, the memory layout for statically declared variables can be determined once at compile time, and this is generally more efficient than if memory requirements must be reevaluated each time the value of a variable changes. Similarly static compile time typing does allow many programming errors to be caught early, before a program begins execution.

The advocates of dynamic languages, on the other hand, assert that flexibility is more important than efficiency. Consider, for example, a simple procedure such as the following.

```
function max (left, right) {
  if  (left < right)
    then return right;
  return left;
}
```

The only requirements we need to impose on the arguments are that they understand the comparison operator. The proponents of dynamically typed languages argue that we should be able to write functions such as max and use them with a wide variety of types: integers, floats, strings, or anything else.

Both arguments have their valid points, and we are not going to take sides. Both statically typed and dynamically typed object-oriented languages exist, and both will probably continue to be used for the foreseeable future.

11.2 ▫ Static and Dynamic Classes

The introduction of object-oriented features to a statically typed programming language requires a relaxation of some of the principles of static typing. Recall the principle of substitution described in Chapter 8. This principle asserted that a variable declared as a parent class type could in fact be maintaining a value from a child class. In order to distinguish these two types we introduce a pair of terms. The *static class* of a variable is the class that was used in the declaration of the variable. The static class (as the name suggests) is fixed at compile time and does not ever change. The *dynamic class* of a variable is the class associated with the value it currently holds. Again, as the name suggests, the dynamic class can change during the course of execution as a variable is assigned new values.

```
var
  obj : GrapicalObject; (* GraphicalObject is the static class *)
begin
  obj = new Ball(); (* Ball is the current dynamic class *)
  .
  .
  .
  obj = new Wall(); (* Wall is now the dynamic class *)
end
```

The most important aspect of the distinction between static and dynamic types is the following.

> In a statically typed object-oriented programming language, the legality of a message-passing expression is determined at compile time based on the static class of the receiver and not on its current dynamic value.

To see what we mean by this, consider a class hierarchy such as the set of Java classes shown in Figure 11.1. All Animals know how to speak, but only dogs and seals know how to bark. Imagine that we then have the following statements.

```
Dog fido;
fido = new Dog();
fido.speak(); // will bark
```

```
class Animal {
  public void speak () { System.out.println("Animal Speak !"); }
}

class Dog extends Animal {
  public void speak () { bark(); }
  public void bark () { System.out.println("Woof !"); }
}

class Seal extends Animal {
  public void bark () { System.out.println("Arf !"); }
}

class Bird extends Animal {
  public void speak () { System.out.println("Tweet !");
}
```

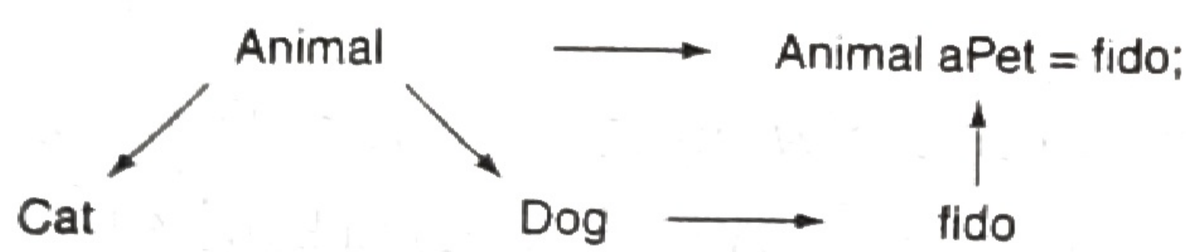
▫ Figure 11.1 — A Java animal class hierarchy

```
Woof !
  fido.bark(); // will bark
Woof !

  Animal pet;
  pet = fido; // legal to assign Dog to Animal

  pet.speak(); // will work
Woof !
  pet.bark(); // error, pets do not know how to bark
```

The variable fido is declared as Dog and thus can both speak and bark. When assigned to a variable of type Animal, however, the static class changes to Animal, although the dynamic class remains Dog. Because of the change in the declared class, the variable pet is not guaranteed to understand the message bark, and thus a compile time error will be given. This is true despite the fact that the value currently being held by the variable certainly is able to understand the message.

11.2.1 Run-time type determination

The principle of substitution can be viewed as moving a value up the inheritance hierarchy. Assigning a value of type Dog to a variable of type Animal moves the presumed type from the child class to the parent.



Occasionally it is necessary to do the reverse: to determine if a value currently being held by a variable declared with one class type is, in fact, derived from a class that is lower in the class hierarchy—for example, to determine if a value held by a variable of type Animal is, in fact, a Dog.

Every object-oriented language has the ability to perform such a test, but the syntax used by the various languages differs greatly. Figure 11.2 illustrates some of these facilities. These types of tests are useful even in dynamically typed object-oriented languages, and thus we have included features from those languages as well.

The functions shown in Figure 11.2 test a value to determine if it is a descendant of a given class—that is, whether it was created as an instance of the class or a child class. Some languages also provide a more precise test to determine if a value was created precisely from a given class and not from a child class. These facilities are shown in Figure 11.3.

C++

```
Animal * aPet = ...; // a pointer to an animal
Dog * d = dynamic_cast<Dog *>(aPet);
if (d != 0) { // null if not legal, nonnull if ok
    .
    .
    .
}
```

CLOS

```
(if (typep aPet 'Dog) ... )
```

Delphi Pascal

```
if (aPet is Dog) then
    .
    .
    .
```

Eiffel

```
aPet : Animal
aDog : Dog

aDog ?= aPet -- assignment attempt
if (aDog /= Void) ... -- will succeed if pet was a dog
```

Java

```
if (aPet instanceof Dog)
    .
    .
    .
```

Object Pascal

```
if Member (aPet, Dog) then
    .
    .
    .
```

Python

```
if isinstance(aPet, Dog):
    .
    .
    .
```

Ruby

```
if aPet kind_of? Dog
```

Smalltalk

```
(aPet isKindOf: Dog) ifTrue: [ ... ]
```

■ Figure 11.2 — Mechanisms for testing class

Delphi Pascal

```
if aPet.classInfo = Dog then ...
```

Ruby

```
if aPet.instance_of? Dog
```

Smalltalk

```
(aPet isMemberOf: Dog) ifTrue: [ ... ]
or
(aPet class == Dog) ifTrue: [ ... ]
```

Figure 11.3 — Facilities to test for a specific class

11.2.2 Down casting (reverse polymorphism)

Once it has been determined that a value is from a given class, often the next step is to convert the value from the parent class to the child class. This process is termed *down casting*, or *reverse polymorphism* (since it reverses the effect of a polymorphic assignment). The syntax used for down casting operations in various languages is shown in Figure 11.4. Note that several languages combine down casting with type testing by having a function that returns a valid result if the conversion is proper and an empty (or null) result if the conversion cannot legally be performed.

*11.2.3 Run-time testing without language support

Before the introduction of the RTTI system (the dynamic_cast operator) to C++ it was common for programmers to simulate run-time typing using nothing more than method overriding. Although this is no longer required in C++, similar tricks may still be useful in languages that lack built-in primitives for testing the dynamic class of variables.

Imagine that we have modified our class hierarchy as follows.[1]

```
class Animal {
public:
  virtual Dog * isDog () { return 0; }
  virtual Bird * isBird () { return 0; }
  .
  .
  .
```

1. The meaning of the virtual keyword will be explain in the next section.

```
};

class Dog : public Animal {
public:
  virtual Dog * isDog () { return this; }
  .
  .
};
```

C++

```
Animal * aPet = ... ;
Dog * d = dynamic_cast<Dog *>(aPet);
if (d != 0) { // null if not legal, nonnull if ok
    .
    .
}
```

Delphi Pascal

```
var
  aPet : Animal;
  aDog : Dog;

aDog := aPet as Child
```

Eiffel

```
aPet : Animal
aDog : Dog

aDog ?= aPet -- assignment attempt
if (aDog |= Void) ... -- will succeed if pet was a dog
```

Java

```
Animal aPet;
Dog d;
d = (Dog) aPet;
```

Object Pascal

```
aDog = Dog(aPet)
```

▫ Figure 11.4 —— Mechanisms for downcasting

```
class Bird : public Animal {
public:
  virtual Bird * isBird () { return this; }
  :
  :
};
```

Imagine now that we have a variable declared as a pointer to type Animal, and we want to know if it really is maintaining a value of type Dog. If we pass it the message isDog, we will get back either a null value, if it is not a dog, or a valid pointer to a Dog. This is because the method isDog in the root class is returning a null value but has been overridden in the class Dog, to return a pointer to itself (using the pseudo-variable this to reference itself). If the value is truly a Dog the overridden method will be executed; otherwise the method in the root class will be. Thus, the message isDog is simulating the behavior of the dynamic_cast used by the RTTI system, albeit at the cost of introducing a new method.

```
Animal * aPet = ...;
Dog * aDog;
aDog = aPet.isDog(); // try the assignment
if (aDog)   ... // true if it was a dog, false otherwise
```

11.2.4 Testing message understanding

In dynamically typed object-oriented languages it is generally true that the legality of a message-passing expression cannot be known until run-time. If the receiver does not know how to respond to a message, then a run-time exception is thrown. Occasionally it is useful for the programmer to be able to test whether a receiver will understand a particular message before actually attempting to pass the message to the receiver. Mechanisms to do this are shown in Figure 11.5.

Smalltalk

```
(aPet respondsTo: #bark) ifTrue: [ ... ]
```

Ruby

```
if aPet.respond_to?("bark") ...
```

Objective-C

```
if ( [ aPet respondsTo: @selector(bark) ] ) { ... }
```

Figure 11.5 — Mechanism to determine if an object understands a message

11.3 ▫ Static versus Dynamic Method Binding

In almost all object-oriented programming languages, the binding of a method to execute in response to a message is determined by the dynamic value currently being held by the receiver. Thus, in the following Java statements the first invocation of speak will produce "woof" and in the second "tweet," as the dynamic type changes from Dog to Bird.

```
Animal pet;
pet = new Dog();
pet.speak();
woof !
pet = new Bird();
pet.speak();
tweet !
```

In a few languages the rules for overriding are more difficult. Most notable among these is C++, although C#, Delphi Pascal, and Oberon-2 also have similar restrictions. The binding of methods to messages in C++ is complex, involving a number of choices of keywords and mechanisms for declaration. Some of these variations are shown in Figure 11.6.

```
class Animal {
public:
  virtual void speak () { cout << "Animal Speak !\n"; }
  void reply () { cout << "Animal Reply !\n"; }
};

class Dog : public Animal {
public:
  virtual void speak () { cout << "woof !\n"; }
  void reply () { cout << "woof again!\n"; }
};

class Bird : public Animal {
public:
  virtual void speak () { cout << "tweet !\n"; }
};
```

Figure 11.6 — Virtual and nonvirtual overriding in C++

Let us say that a variable is *polymorphic* if the binding of message to a method is determined by the type associated with the value most recently assigned to the variable. (All variables in Smalltalk, Java, and most other object-oriented languages are, in this sense, polymorphic.) Variables in C++ that are declared with a simple type are not polymorphic, in this sense. Consider the following declarations of values and their executions in response to different messages.

```
Animal a;
  Dog b;
  b.speak();
woof !
  a  = b;
  a.speak();
Animal speak !
  Bird c;
  c.speak();
tweet !
  a = c;
  a.speak();
Animal speak !
```

Note that in each instance, the method executed in response to the message speak when the receiver is type Animal is that found in the static class Animal. This is true even if the value held by the variable came from an assignment of a different type.

Now contrast this with what happens when pointers are used instead of simple values.

```
  Animal * d;
  d = &b; // point to the dog from earlier example
  (*d).speak();
woof !
  d = & c;
  d->speak(); // arrow is shorthand for (*).
tweet !
```

Thus, object values referenced by pointers are polymorphic in the sense that we defined the term earlier. The language C++ includes the notion of a reference, which is simply a pointer that is guaranteed to point to a valid object. References, too, are polymorphic.

```
  Animal & e = b;
  e.speak();
woof !
```

```
  Animal & f = c;
  f.speak();
tweet !
```

The careful reader will have noted the presence of the keyword virtual in the declaration of the methods shown in Figure 11.6. This keyword is important. If it is left off, then even object references made by means of pointers are no longer polymorphic. This is shown by the message reply, which has not been declared as virtual.

```
  Animal * g = &b;
  g.reply();
Animal reply!
  g = &c;
  g.reply();
Animal reply!
```

A nonvirtual method that is modified in the child class will be executed if the static class of the receiver is the child class and not the parent class.

```
  b.reply()
woof again !
```

So polymorphic message sending is found in C++ only when using pointers or references, and even there only when the associated methods have been declared as virtual. To understand why the rules for C++ are so much more complex than the rules used by other programming languages, we need to explore the link between object-oriented polymorphic variables and memory management. We will investigate this further in the next chapter.

C# and Delphi Pascal also require the keyword virtual, although since they lack references, their rules for determining the meaning of a message expression are not as complex as those of C++. We will consider message overriding in these languages later in Chapter 16. In Oberon-2, like in C++, the meaning of a message given to a pointer variable differs from the meaning of the same message given to an ordinary variable.

Summary ▫

Static features are aspects of a program that are fixed at compile time. Dynamic features can change at run-time. The dichotomy between static and dynamic is manifested in object-oriented languages in a number of different ways. In this chapter we have considered static and dynamic typing, static and dynamic classes, and static and dynamic method binding.

CHAPTER 12

Implications of Substitution

The introduction of inheritance and, in particular, the principle of substitution, has a subtle but pervasive impact on almost all aspects of a programming language. In this chapter, we will examine some of these effects, considering in detail the type system, the meaning of assignment, testing for equivalence, the creation of copies, and storage allocation.

We have described the is-a relationship as a fundamental property of inheritance. One way to view the is-a relationship is as a means of associating a *type*, as in a type of a variable, with a set of *values*—namely, the values the variable can legally hold. If a variable win is declared as an instance of a specific class—say, Window—certainly it should be legal for win to hold values of type Window. If we have a subclass of Window—say, TextWindow, since a TextWindow is-a Window—it should certainly make sense that win can hold a value of type TextWindow. This is the principle of substitution that we encountered in previous chapters.

While this principle makes intuitive sense, from a practical point of view there are difficulties associated with implementing object-oriented languages in a way that this intuitive behavior can be realized. These difficulties are not insurmountable, but the way various language designers have chosen to address them differs from language to language. An examination of these problems and of how they affect the language illuminates the reasons for some of the obscure features of languages over which the unwary programmer is likely to stumble.

12.1 ▫ Memory Layout

Let us start by considering a seemingly simple question, the various answers to which will lead us in different directions: How much storage should we allocate

to an object instantiated from a specific class? To take a concrete example, how much storage should we allocate to the variable win that we earlier described as being an instance of class Window?

It is commonly believed that variables allocated on the stack as part of the procedure-activation process are more efficient than are variables allocated on the heap.[1] Accordingly, language designers and implementors go to great lengths to make it possible for variables to be stack-allocated. But there is a major problem with stack allocation: The storage requirements must be determined statically at compile time or, at the latest, at procedure entry time. These times are well before the values the variable will hold are known.

The difficulty is that subclasses can introduce data not present in a superclass. The class TextWindow, for example, probably brings with it data areas for character buffers, locations of the current edit point, and so on. The following might be a typical declaration.

```
class Window {
  public:
    virtual void oops();
    .
    .
    .
  private:
    int height;
    int width;
};

class TextWindow : public Window {
  public:
    virtual void oops();
    .
    .
    .
  private:
    char * contents;
    int cursorLocation;
};

Window win;   // declare a variable of type window
```

Should the additional data values (contents and cursorLocation here) be taken into consideration when space for win is allocated? There are at least three plausible answers,

1. But see [Appel 1987] for a dissenting opinion.

1. Allocate the amount of space necessary for the base class only. That is, allocate to win only those data areas declared as part of the class Window, ignoring the space requirements of subclasses.
2. Allocate the maximum amount of space necessary for any legal value, whether from the base class or from any subclass.
3. Allocate only the amount of space necessary to hold a single pointer. Allocate the space necessary for the value at run time on the heap, and set the pointer value appropriately.

All three solutions are possible, and two of them are found in the languages we are considering. In the following sections, we will investigate some of the implications of this design decision.

12.1.1 Minimum static space allocation

The language C was designed to be run-time efficient. Thus, given the widespread belief that stack-based allocation of memory locations results in faster execution times than are possible with dynamic variables, it is not surprising that its successor, C++, retains the concepts of both nondynamic and dynamic (run-time-allocated) variables.

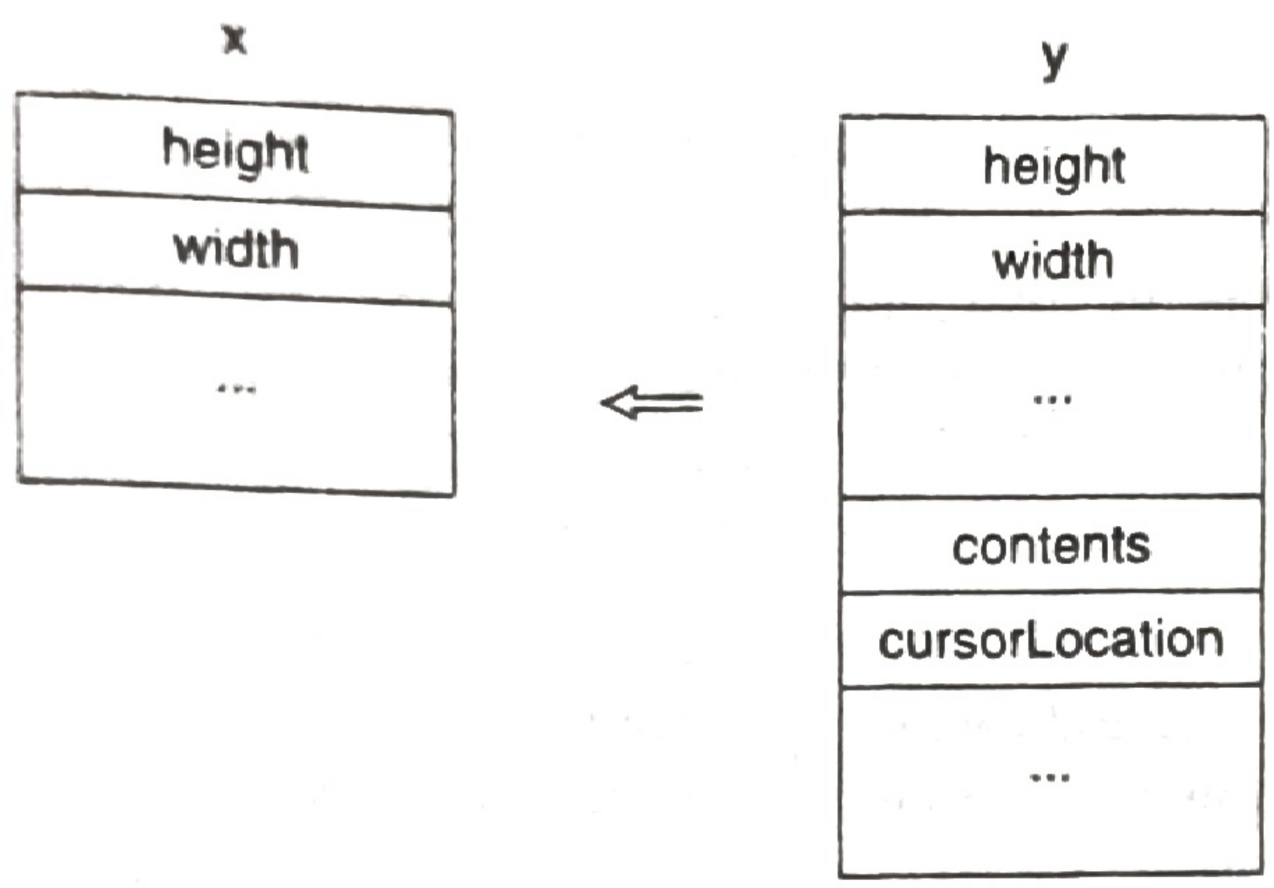
In C++, the distinction is made in how a variable is declared and, accordingly, whether pointers are used to access the values of the variable. In the following, for example, the variable win is allocated on the stack. Space for it will be set aside on the stack when the procedure containing the declaration is entered. The size of this area will be the size of the base class alone. The variable tWinPtr, on the other hand, contains only a pointer. Space for the value pointed to by tWinPtr will be allocated dynamically when a new statement is executed. Since by this time the size of a TextWindow is known, there are no problems associated with allocating an amount of storage on the heap sufficient to hold a TextWindow.

```
Window win;
Window *tWinPtr;
.
.
.
tWinPtr = new TextWindow;
```

What happens when the value pointed to by tWinPtr is assigned to win? In other words, what happens when the user executes this statement?

```
win = *tWinPtr;
```

The space allocated to win is only large enough to accommodate a Window, whereas the value pointed to by tWinPtr is larger (Figure 12.1). Clearly, not all of the values pointed to by tWinPtr can be copied. The default behavior is to copy



▫ Figure 12.1 —— Assigning a larger value to a smaller box

only the corresponding fields. (In C++, the user can override the meaning of the assignment operator and provide any semantics desired. Thus, we refer here only to the default behavior observed in the absence of any alternative provided by the user.) Clearly, then, some information is lost (the information contained in the extra fields of tWinPtr). Some authors use the term *slicing* for this process, since the fields in the right side that are not found in the left side are sliced off during assignment.

Is it important that this information is lost? A cynic will answer, "Only if the user can tell the difference." The question is therefore how the user might be able to notice.

The semantics of the C++ language ensure that only methods defined in the class Window can be invoked with win and no methods defined in TextWindow. Methods defined in Window and implemented in that class cannot access or modify data defined in subclasses, so no access is possible there. But what about methods defined in class Window but *overridden* in the subclass?

Consider, for example, the two procedures oops() shown here. If the user executed win.oops() and the method from class TextWindow was selected, an attempt would be made to display the data value win.cursorLocation, which does not exist in the storage assigned to win. This would either cause a memory violation or (more likely) produce garbage.

```
void Window::oops()
{
  cout << "Window oops" << endl;
}

void TextWindow::oops()
```

```
{
  cout << "TextWindow oops " << cursorLocation << endl;
}
```

The solution to this dilemma selected by the designer of C++ was to change the rules that are used to bind a procedure to the invocation of a virtual method. The new rules can be summarized as follows.

- For pointers (and references): When a message invokes a member function that could potentially have been overridden, the member function selected is determined by the dynamic value of the receiver.
- For other variables: The binding on a call of a virtual member function is determined by the static class (the class of the declaration) and not by the dynamic class (the class of the actual value).

More accurately, during the process of assignment the value is *changed* from the type representing the subclass to a value of the type represented by the parent class. This is analogous to the way an integer variable might be changed during assignment to a floating-point variable. With this interpretation, it is possible to ensure that, for stack-based variables, the dynamic class is *always* the same as the static class. Given this rule, it is not possible for a procedure to access fields that are not physically present in the object. The method selected in the call win.oops() would be that found in class Window, and the user would not notice the fact that memory was lost during the assignment.

Nevertheless, this solution is achieved only at the expense of introducing a subtle inconsistency. Expressions involving pointers bind virtual methods in the manner we described in earlier chapters. Thus, these values will perform differently from expressions using nondynamic values. Consider the following.

```
Window win;
TextWindow *tWinPtr, *tWin;
.
.
.
tWinPtr = new TextWindow;
win = * tWinPtr;
tWin = tWinPtr;

win.oops( ) ;
tWin->oops( ) ;
```

Although the user is likely to think that win and the value pointed to by tWin are the same, it is important to remember that the assignment to win has transformed the type of the value. Because of this change, the first call on oops() will invoke the method in class Window, whereas the second will invoke that in class TextWindow.

Oberon-2 is another language that uses slicing on assignment, with a similar impact on semantics.

12.1.2 Maximum static space allocation

A different solution to the problem of deciding how much space to allocate to a declaration for an object is to assign the maximum amount of space used by any value the variable might hold, whether from the class named in the declaration or from any subclass. This approach is similar to the one used to lay out overlaid types in conventional languages, such as variant records in Pascal or union structures in C. On assignment, it is not possible to assign a value larger than what fits in the target destination, so the picture shown in Figure 12.1 cannot occur, and the subsequent problems described in the last section do not arise.

This would seem to be an ideal solution were it not for one small problem: The size of any object cannot be known until an entire program has been seen. Not simply a module (unit in Object Pascal, file in C++) but the entire program must be scanned before the size of any object can be determined. Because this requirement is so restrictive, no major object-oriented language uses this approach.

12.1.3 Dynamic memory allocation

The third approach does not store the *value* of objects on the stack at all. When space for an identifier is allocated on the stack at the beginning of a procedure, it is simply large enough for a pointer. The values are maintained in a separate data area, the heap, that is not subject to the first-in last-out allocation protocol of the stack. Since all pointers have a constant fixed size, no problem arises when a value from a subclass is assigned to a variable declared to be from a superclass.

This is the approach used in Object Pascal, Smalltalk, Java, and Objective-C. The reader might already have guessed by the close similarity of objects and pointers in Object Pascal. For both pointers and objects, it is necessary to invoke the standard procedure new to allocate space before the object can be manipulated. Similarly, it is necessary for the programmer to call free explicitly to release space allocated for the object.

Besides the requirement for explicit user memory allocation, another problem with this technique is that it is often tied to the use of *pointer semantics* for assignment. When pointer semantics are used, the value transferred in an assignment statement is simply the pointer value rather than the value indicated by the pointer. Consider the following program, which implements a one-integer buffer that can be set and retrieved by the user.

```
type
  intBuffer = object
    value : integer;
  end;
var
  x, y : intBuffer;

begin
  new(x);     { create a buffer }
  x.value := 5;
  writeln(x.value);
  y := x;     { y is same buffer as x }
  y.value := 7;
  writeln(x.value);
end;
```

Notice the two variables x and y declared to be instances of this class. In executing the program, the user might be surprised when the last statement prints out the value 7 rather than the value 5. The reason for this surprising result is that x and y do not just have the same value; they point to the same object. This situation is shown in Figure 12.2. The use of pointer semantics for objects in Object Pascal is particularly confusing because the alternative, *copy semantics*, is used for all other data types. If x and y were structures, the assignment of y to x would result in the copying of information from y to x. Since this would create two separate copies, changes to y would not be reflected in changes in x.

Garbage collection

Because dynamically allocated objects are stored on the heap, they will not be automatically recovered when a procedure exits. As we noted in an earlier chapter, some languages force the programmer to explicitly indicate when the memory associated with an object can be recovered. Other languages use a technique termed *garbage collection* to automatically monitor the utilization of object values

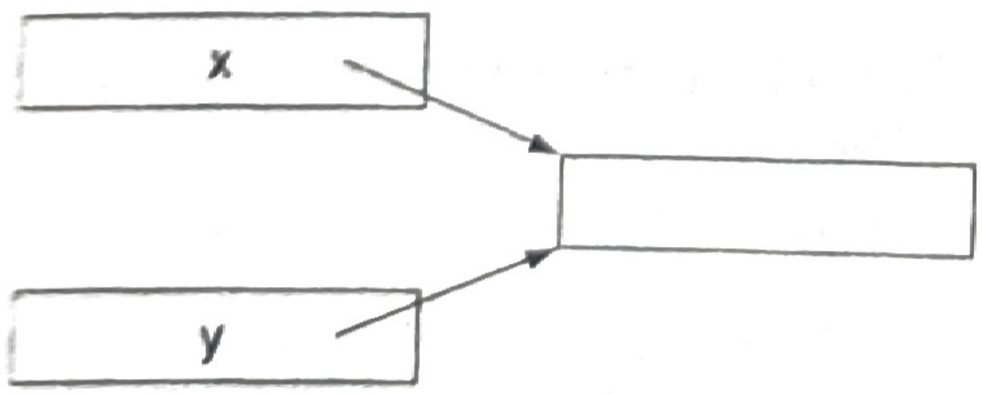


Figure 12.2 — Two object variables pointing to the same value

and reclaim them when they can no longer have any impact on the execution of a program.

Generally, the use of garbage collection systems results in fewer programming errors due to the mismanagement of memory, but it imposes a greater cost in execution time.

12.2 ▫ Assignment

The memory-allocation strategies used in a programming language have an effect on the meaning of assignment, so we will summarize the exact meaning of this operator in the various languages we are considering. As noted in the last section, there are two interpretations we can give to assignment.

Copy semantics. Assignment copies the entire value of the right side, assigning it to the left side. Thereafter, the two values are independent, and changes in one are not reflected in changes in the other. Copy semantics are sometimes used in C++ and sometimes not (we will explain shortly).

Pointer semantics. Assignment changes the reference of the left side to be the right side. (This approach is sometimes referred to as *pointer assignment*.) Thus, the two variables not only have the same value but also refer to the same memory location. Changes in one will alter the value held in common, which will be reflected in references obtained under either name. Pointer semantics are used in Java, CLOS, Object Pascal, and many other object-oriented languages.

(A compromise position between copy semantics and pointer semantics is found in some programming languages, although not in any of the languages we are considering in our case studies. The idea is to use pointer semantics for assignment but to convert a value into a new structure if it is ever modified. In this manner assignments are very efficient, but a value cannot be inadvertently modified by a change to an aliasing variable. This technique is often termed *copy on write*.)

Generally, if pointer semantics are used, languages provide some means for producing a true copy. Also, pointer semantics are generally more often used when all objects are allocated on a heap (dynamically) rather than on the stack (automatically). When pointer semantics are used, it is common for a value to outlive the context in which it is created.

Object-oriented languages differ in which of the two semantics they use, providing one, the other, or combinations of both.

*12.2.1 Assignment in C++

The default algorithm used in C++ to assign a class value to a variable is to copy corresponding data fields recursively. However, it is possible to overload

the assignment operator to produce any behavior desired. This technique is so common that some C++ compilers issue a warning if the default assignment rule is used.

In assignment overloading the interpretation is that assignment is a message given to the expression to the left of the assignment operator, passing the expression to the right of the operator as argument. The result can be void if embedded assignments are not possible, although more typically the result is a reference to the receiver. The following example shows assignment of a string data type, which redefines assignment so that two copies of the same string share characters.

```
String & String::operator = (String& right)
{
  len = right.len;   // copy the length
  buffer = right.buffer;   // copy the pointer to values
  return (*this);
}
```

A common source of confusion for new C++ programmers is the use of the same symbol for assignment and for initialization. In conventional C, an assignment used in a declaration statement is simply a syntactic shorthand. That is, the effect of

```
int limit = 300;
```

is the same as

```
int limit;
limit = 300;
```

In C++ an assignment used in a declaration may select the constructors invoked and may not use the assignment operator at all. That is, a statement such as

```
Complex x = 4;
```

is interpreted to mean the same as the declaration

```
Complex x(4);
```

Reference variables.

Initialization is often used with reference variables and yields a situation very similar to pointer semantics. If s is a valid String, for example, the following makes t an alias for the value of s, so any change in one will be reflected in the other. As we have noted, declarations in C++ can be inserted into a program at any point.

```
... use of variable s
String & t = s;
...// t and s now refer to the same value
```

Reference variables are most often used to implement call-by-reference parameter passing. This use can be considered a form of pointer assignment, where the parameter is being assigned the argument value. Of course, pointer semantics in C++ can also be achieved through pointer variables.

Parameter passing is in part a form of assignment: the assignment of parameter values to the arguments, and so it is not surprising that the same issues occur here as in assignment. For example, consider the definitions below:

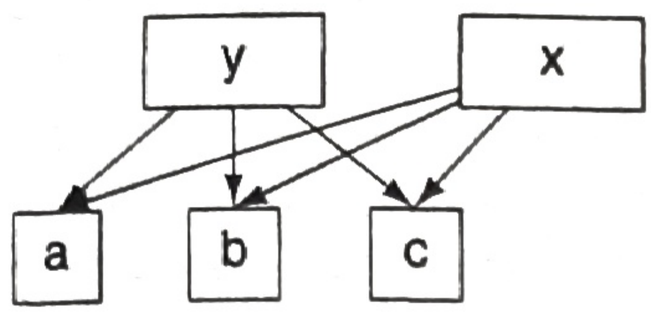
```
class Parent {
public:
  virtual void see() { cout << "I parent I" ;}
};

class Child : public Parent {
public:
  virtual void see() { cout << "I child I"; }
};

void f (Parent x) { x.see(); }
void g (Parent & x) { x.see(); }

Child aChild;
f(aChild);
g(aChild);
```

Both the functions f and g take as argument a value declared as the parent type, but g declares the value as a reference type. If f is called with a value of the child type, the value is converted (*sliced*) to create a value of the parent type as part of the assignment of the arguments. Thus, if see is invoked from within f, the virtual function from the base class will be used. On the other hand, this conversion, or slicing, does not occur as part of the parameter passing to g. So if see is invoked from within g, the procedure from the child class will be used. This difference in interpretation, which depends only on the one character in the function header, is sometimes known as the *slicing problem*.

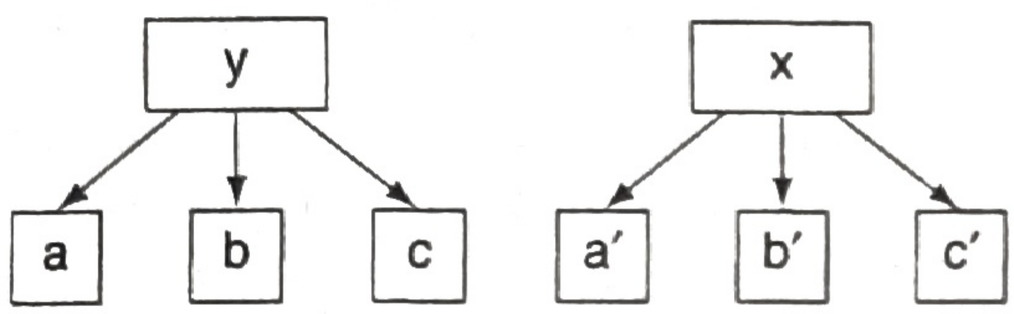
The overloading of the assignment symbol and the choice of parameter-passing mechanisms provided by value and reference assignment in C++ are powerful features, but they can also be quite subtle. For example, the assignment symbol used in initialization, although it is the same as the assignment symbol used in statements, is not altered by a redefinition of the assignment operator. A good explanation of the uses and power of assignment in C++ is given by Koenig [Koenig 1989a, Koenig 1989b].

12.3 ▫ Copies and Clones

When languages use pointer semantics for assignment, they almost always provide a means to create a copy, or clone, from a value. But even in this simple task there are subtle issues that can trap the unwary. In particular, what actions should be taken when creating a copy of a value that itself points to other objects? There are two possible interpretations. A *shallow copy* shares instance variables with the original. That is, the original and the copy reference the exact same values.



On the other hand, a deepCopy creates new copies of the instance variables, yielding a picture like this.



And, of course, the questions can continue recursively by asking what techniques are used to create the copies of a, b, and c.

In the following sections we will describe the facilities for creating copies in some of the languages we are considering.

12.3.1 Copies in Smalltalk and Objective-C

In Smalltalk the fundamental root class Object defines three methods: copy, shallowCopy, and deepCopy. A deep copy recursively creates a deep copy of any instance data it may hold. Classes are themselves free to override any of these methods to exhibit different behavior. Objective-C uses methods that are analogous to those used in Smalltalk.

12.3.2 Copy constructors in C++

Copies of values are frequently produced by the C++ run-time system as temporaries or arguments to procedures. The user can control this task by defining a *copy constructor*. The argument to a copy constructor is a reference parameter of the same type as the class itself. It is considered good practice to always create a copy constructor.

```
class Complex {
public:
  .
  .
  .
  Complex (const Complex & source)
    {
      // simply duplicate fields from source
      rl = source.rl;
      im = source.im;
    }
  .
  .
  .
private:
  double rl;
  double im;
}
```

12.3.3 Cloning in Java

The class Object in Java defines a method named clone, but the method is declared as protected. To make an object that can be copied using this method the programmer must do two things. First, the programmer must declare that their class supports the Cloneable interface. Second, the method clone must be overridden and made public. This method potentially throws the error CloneNotSupportedException, which must also be part of the declaration. The method need not actually do anything other than invoke the method inherited from the parent class.

```
class PlayingCard implements Cloneable {
  .
  .
  .
  public Object clone ( ) throws CloneNotSupportedException {
    Object newCard = super.clone();
    return newCard;
  }
}
```

The default behavior creates a shallow copy. If a deep copy, or any other domain-specific behavior, is desired, the programmer can add additional code subsequent to invoking the parent method.

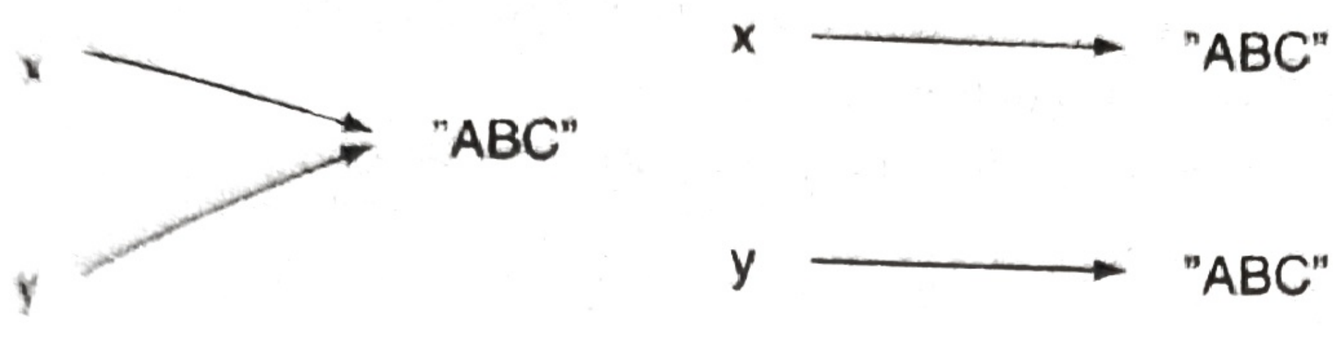


Figure 12.3 — Identity and equality for character strings

12.4 ▫ Equality

Like assignment, deciding whether one object is equivalent to another is more subtle than it might at first appear. There are two issues to examine in relation to this problem. The first is the difference between equality and identity (Figure 12.3). The second is the paradoxes that can arise in the context of substitution if classes are allowed to redefine the meaning of equality.

12.4.1 Equality and identity

There is an old joke that goes something like this: A man walks into a pizza parlor and sits down. A waiter comes to the table and asks the man what he would like to order. The man looks around the room, then points to the woman sitting at the next table and says, "I'll have what she is eating." The waiter thereupon walks to the woman's table, picks up the half-eaten pizza from in front of her, and places it before the startled customer.

The reason this story is funny is because it confuses two related but distinct concepts: identity and equality. When we ask questions of identity, we are seeking to determine if two objects are precisely the same entity. This is what is being asked if somebody inquires, "Is the morning star the same as the evening star?" (Morning star and evening star are both common names for the planet Venus.)

On the other hand, when the man said he wanted what the woman was eating, he did not mean he wanted literally the same object she was eating but a different object that was in all important respects equivalent to what she had before her.

The question of identity versus equality did not originate with object-oriented languages, although they do seem to be more common in such languages. But strings, for example, have always needed special treatment for just this reason (Figure 12.3). Many beginning C programmers have had a program fail because they used the equality operator on two character pointers instead of the special purpose string comparison function.

```
char * a = "abc";
char * b = "abc";

if (a == b)   // will surprisingly not be true...
```

This fails because the two pointers reference different memory locations, even though the values stored at those two different locations are the same.

The double equal sign used to test identity in C and C++ is also used for the identity operator in Java, C#, and Objective-C.

12.4.2 The paradoxes of equality testing

While identity is easily defined, the meaning of equality is inherently domain specific. Should two strings, for example, be considered equal if they have the same character representation? What about two triangles? Should they be considered equal if they have the same side lengths even if they are different objects? There is no single correct answer, and every problem (and every programmer) must decide as the situation requires.

In order to accommodate this flexibility, many object-oriented languages permit the programmer to define the meaning of an equality testing operator. Equality testing is, then, nothing more than a message that is given to one value, passing the other as argument. Because message passing is not symmetric, there can be no guarantees that if a is equal to b, it follows that b will be equal to a.

In languages in which all classes descend from a common root, the equality testing operator is often defined in the root class and can be overridden in subclasses. This presents a quandary for strongly typed languages: What type should be associated with the argument value? Almost always this problem is solved by declaring the argument as the root type. Child classes are then obligated to use the same type signature, and hence the argument must be tested and cast before a meaningful comparison can be performed. For example, in the following Java class definition, the argument to equals (a method inherited from class Object) must be declared as Object. But the programmer is only concerned with testing playing cards to playing cards. Hence, a conversion is necessary before the actual test can be performed.

```
class PlayingCard extends Object {
  ⋮

  public boolean equals (Object right) {
    if (right instanceof PlayingCard) {
      PlayingCard rightCard = (PlayingCard) right;
        // do card-card comparison
      return (rank = rightCard.rank) &&
        (suit == rightCard.suit);
    }
    return false; // false on all other comparisons
  }
}
```

Because overriding of the equality operator can be performed at various levels of a class hierarchy, a number of curious paradoxes can arise. Assume, for example, that we have two classes, Parent and Child, that both provide a meaning for the equality testing message. Because each class is free to provide whatever meaning it wishes for any method, we can in general say nothing concerning the relationship between these two methods. So each of the following situations could easily occur.

- Let p be an instance of the Parent class, and c an instance of the Child class. Since testing whether p equals c is performed by the method in the parent class, while testing if c equals p is performed by the method in the child class, it can happen that one is true and the other is false.
- Similarly assume there are two instances of the child class, c1 and c2. It can happen that the method in the parent class determines that p is equal to both c1 and c2, but the comparison of c1 to c2, which uses the method in the child class, could return false.
- Conversely, assume that there is another instance of the parent class, named p2. It can happen that p is equal to c, and c is equal to p2, but p is not equal to p2.

We will return to investigate further some of the issues involved in equality testing when we take up the topic of overriding in Chapter 16.

The Eiffel Type LIKE current.
In Eiffel the problem of equality is solved in part by the ability to *anchor* a type to that of another value. Most often the anchor is the receiver pseudo-variable, which is known as Current in Eiffel. Thus, an equality method could be written in this language as follows.

```
Class PlayingCard
feature
  .
  .
  .
  equal (other : LIKE Current) is
  do
    -- compare two playing cards
    .
    .
    .
  end
end
```

The LIKE anchor continues after inheritance. Thus, if we were to form a new subclass of PlayingCard—say, PokerCard—then when an argument to PokerCard was used as the receiver, the argument would have to be of type PokerCard.

This mechanism is not without cost. In general, type correctness can only be assured by a run-time check for argument compatibility. Thus, in practice the code is no more efficient than the Java example presented earlier, where the programmer was forced to write the run-time check.

Summary ▫

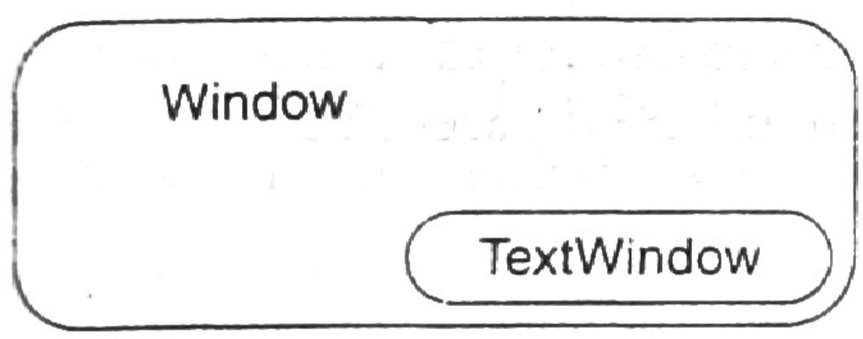
In this chapter we have followed some of the language design implications that flow from the decision to support the object-oriented concept of substitution. We have examined the implications for memory management, assignment, object copies, and equality testing.

- Because a child class can introduce fields not found in the parent, the support of substitution is difficult.
- One approach is to essentially disallow substitution. This is the approach taken by C++. An assignment of a child class to a parent value slices away any data fields found only in the child.
- Most other object-oriented languages avoid the slicing problem by storing objects as pointers to heap-based memory.
- Since objects are most naturally heap based, a heap memory management system is important in object-oriented languages.
- Some languages force the programmer to manage the heap, freeing memory when it is no longer needed. Other languages include a garbage collection system that automatically detects and recovers unused memory.
- Because it is natural to implement objects as pointers to heap-based memory, most object-oriented langauges also use pointer assignment semantics.
- When a true copy of an object is needed, it can be generated either as a shallow copy (where pointer assignment is used to copy the inner state) or as a deep copy (where the inner state is copied recursively).
- Two objects can be compared either for identity or equality.
- Identity asks if two objects represent the same memory location.
- Equality asks if two objects represent the same state, even if they are different memory locations. The meaning of equality is inherently domain specific.
- Because equality can be redefined by programmers, it is difficult to ensure that basic properties are preserved, such as commutivity.
- There can be subtle interactions between equality testing and inheritance.

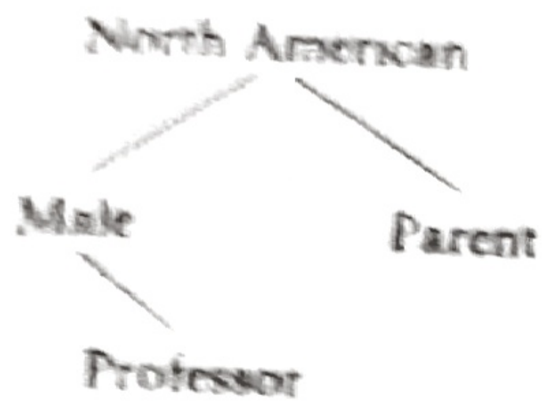
CHAPTER 13

Multiple Inheritance

At the heart of the concept of inheritance is the is-a relationship. Viewed in one way, the is-a relationship is a form of classification. For example, a *TextWindow* can be classified as a type of *Window*, and therefore it makes sense that the class TextWindow can be created by inheritance from class Window. Expressed another way, the category TextWindow is a subset of the more general category Window.

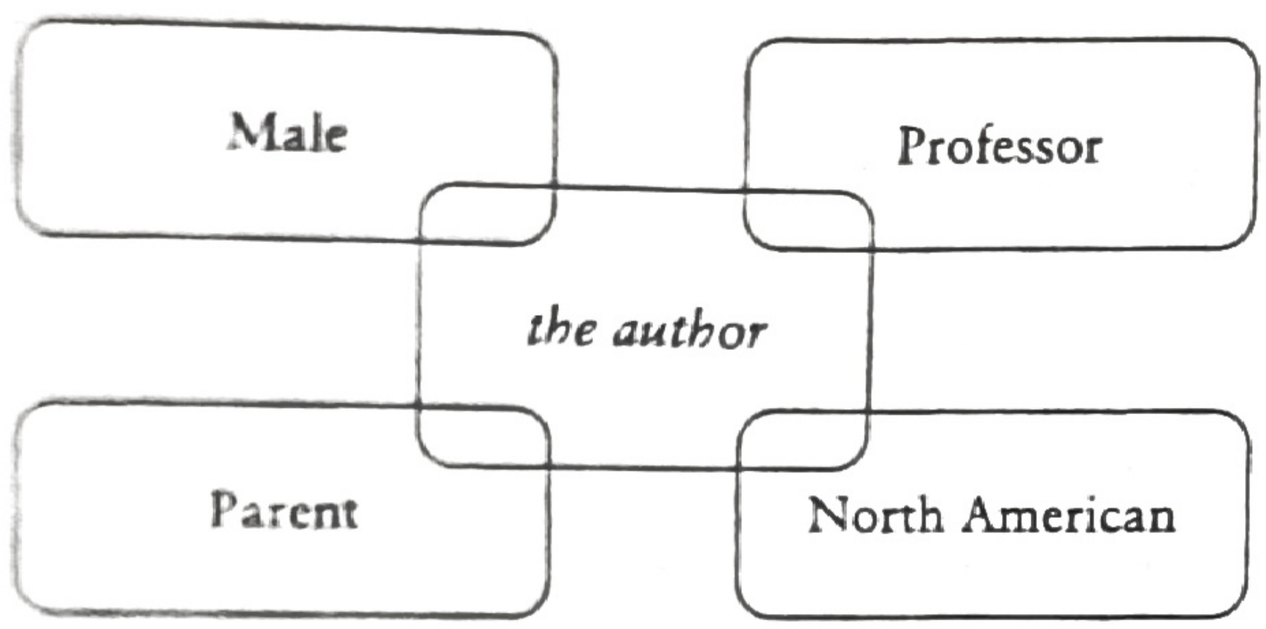


However, when we look at classifications of objects in the real world, they seldom fit the simple single parent hierarchies we have been constructing. This is because real world objects are almost always classified in multiple, mutually nonoverlapping ways. The author of this book, for example, can be described as a male, a professor of computer science, a parent, and a North American. Each of these categories reveals something about the author, and none is a proper subset of any of the others. If we try to form these categories into an inheritance hierarchy, we end up with invalid categorizations, such as asserting that all males are North American or that all professors are male or some other equally improbable classifications.

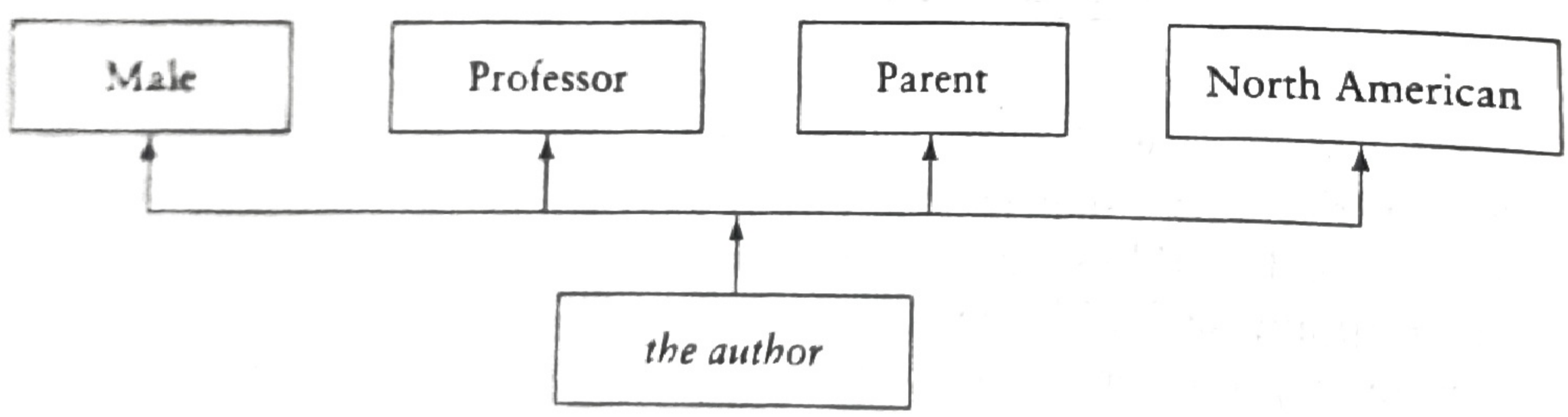


13.1 ▫ Inheritance as Categorization

The proper way to express these relationships is to assert that the author is a *combination* of many nonoverlapping categories. Each category contributes something to the result. Each relationship can legitimately be described using the is-a rule (the author is a parent; the author is a North American).



Modeling this behavior in programs seems to call for the concept of *multiple inheritance*. An object can have two or more different parent classes and inherit both data and behavior from each.



It is important to point out that we have not abandoned the is-a rule as the fundamental test for the inheritance relationship. The derived category must possess the is-a relationship to each of its parent classes, or it may have a variation in is-a: the as-a relationship.

Is-a, Has-a, and As-a

A common analogy in describing multiple inheritance is that of an actor playing many different roles, often at the same time. Likewise, multiple inheritance is not the has-a relationship, and not quite the is-a relationship (since the parent class is no longer unique) but a variation on is-a that could be described as *as-a*. The rule-of-thumb test for multiple inheritance is whether the sentence "A(n) A can be viewed as-a B" sounds right.

13.1.1 Incomparable complex numbers

Another example will illustrate the practical problems that arise from forcing concepts to fit a single inheritance hierarchy. In Smalltalk, the class Magnitude defines a protocol for objects that have measure—that is, that can be compared with one another. For example, individual characters (instances of class Char) can be compared if we use the underlying ASCII representation as a basis for measure. A more common class of objects that can be compared are numbers, which are represented as instances of the class Number in Smalltalk. In addition to being measurable, instances of class Number support arithmetic operations—addition, multiplication, and so forth. These operations do not make sense for objects of class Char. There are various number types supported by Smalltalk; examples include the classes Integer, Fraction, and Float. A portion of the class hierarchy is shown in Figure 13.1.

Now suppose we add the class Complex, representing the complex number abstraction. The arithmetic operations are certainly well defined for complex numbers, and it is preferable to make the class Complex a subclass of Number so that,

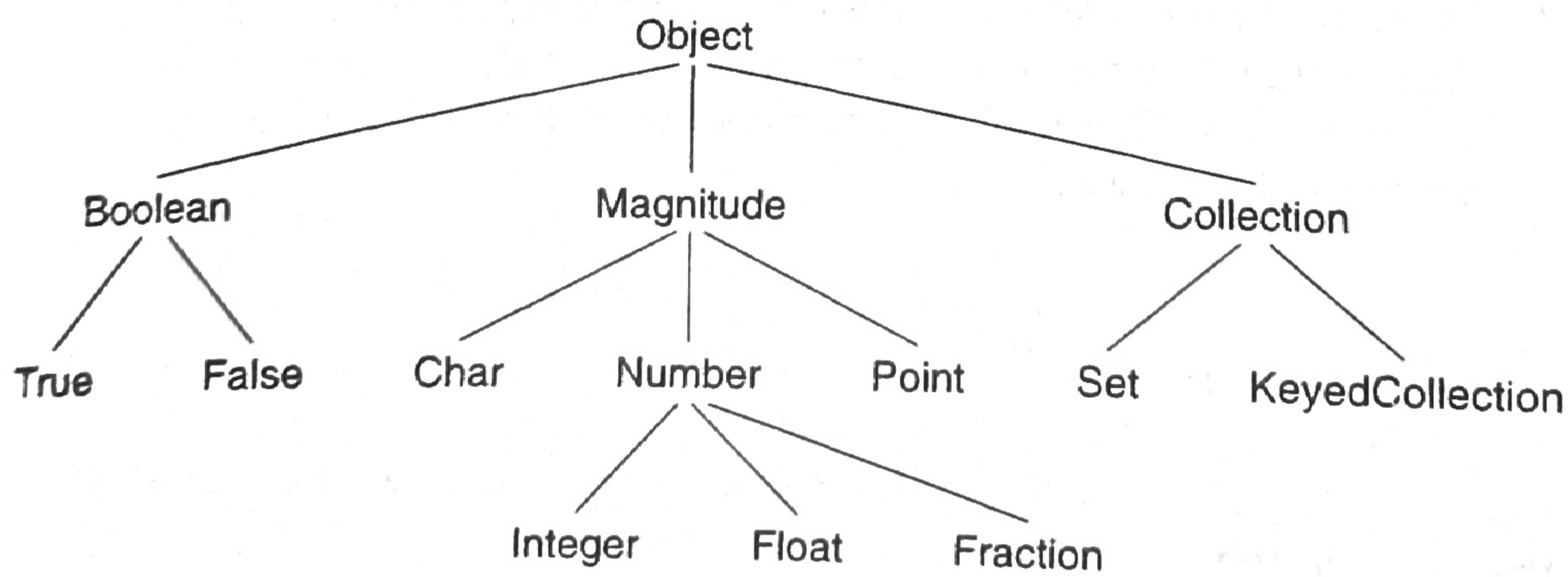


Figure 13.1 — A portion of the Smalltalk class hierarchy

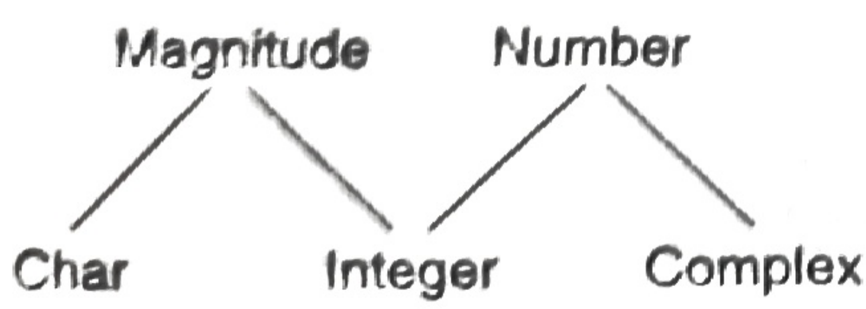
for example, mixed-mode arithmetic is provided automatically. The difficulty is that comparison between two complex numbers is ambiguous. That is, complex numbers are simply not measurable.

Thus, we have the following constraints.

- The class Char should be a subclass of Magnitude but not of Number.
- The class Integer should be a subclass of both Magnitude and Number.
- The class Complex should be a subclass of Number but not of Magnitude.

It is not possible to satisfy all of these requirements in a single inheritance hierarchy. There are some alternative solutions to this problem.

1. Make Complex a subclass of Number, which is in turn a subclass of Magnitude, and then redefine the methods relating to measure in class Complex to produce error messages if they are invoked. This is subclassing for limitation, as described in Chapter 8. Although not elegant, this solution is sometimes the most expedient if your programming language does not support multiple inheritance.
2. Avoid the use of inheritance altogether and redefine every method in each of the classes Char, Integer, Complex, and so on. This solution is sometimes called *flattening the inheritance tree*. Of course, it eliminates all the benefits of inheritance described in Chapter 8—for example, code reuse and guaranteed interfaces. Furthermore, in a statically typed language, such as C++ or Object Pascal, it also prevents the creation of polymorphic objects. Thus, for example, it is not possible to create a variable that can hold an arbitrary measurable object or an arbitrary type of number.
3. Use part of the inheritance hierarchy and simulate the rest. For example, place all numbers under class Number, but have each measurable object (whether character or number) implement the comparison operations. This also eliminates many of the benefits that can be derived from inheritance.
4. Make the two classes Magnitude and Number independent of each other and thus require the class Integer to use inheritance to derive properties from *both* of the parents (Figure 13.2). The class Float will similarly inherit from both Number and Magnitude.



▫ Figure 13.2 — A multiple inheritance hierarchy for complex numbers

Clearly the most elegant solution to this problem is the last. A class that inherits from two or more parent classes is said to exhibit *multiple inheritance*.

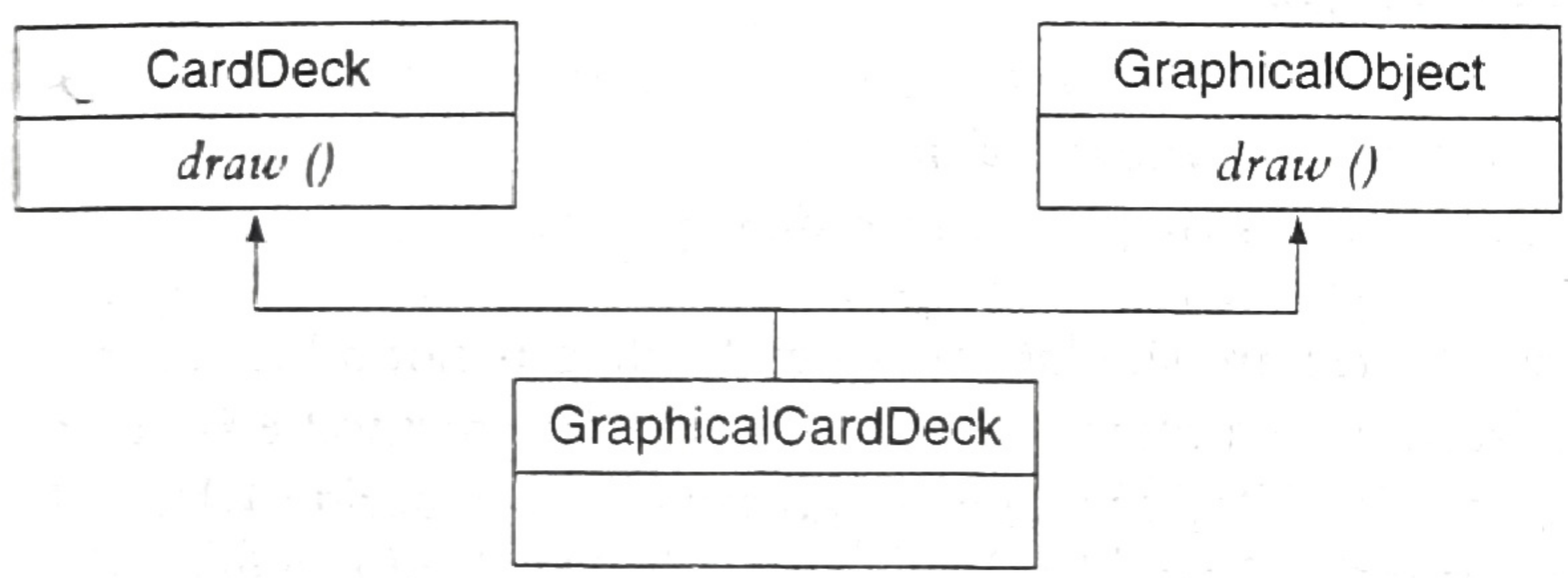
13.2 ▫ Problems Arising from Multiple Inheritance

The problems that arise from the inclusion of multiple inheritance in a language fall into two broad categories. There are problems of meaning and problems of implementation. In this chapter we will consider problems of meaning, and in Chapter 27 we will consider the impact of multiple inheritance on the implementation.

13.2.1 Name ambiguity

The most common difficulty arising from the use of multiple inheritance is that names can be used to mean more than one operation. To illustrate this, consider a programmer developing a card game simulation. Suppose there is already a data abstraction, CardDeck, that provides the functionality associated with a deck of cards (such as shuffling and being able to draw a single card from the deck) but has no graphical capabilities. Suppose further that another set of existing classes implements graphical objects. Graphical objects maintain a location on a two-dimensional display surface. In addition, graphical objects must all know how to display themselves by means of the virtual method called draw.

The programmer decides that to achieve maximum leverage from these two existing classes, he will have the class for the new abstraction, named GraphicalCardDeck, inherit from both the classes CardDeck and GraphicalObject. It is clear that conceptually the class GraphicalCardDeck is-a CardDeck and is thus logically descendant from that class and also that a GraphicalCardDeck is-a GraphicalObject. The only trouble is the clash between the two meanings of the command draw.



As Bertrand Meyer points out [Meyer 1988a] the problem is clearly with the child and not with the parent classes. The meaning of draw is unambiguous and meaningful in each of the parent classes when taken in isolation. The difficulty is

with the combination. Since the problem arises only in the child class, the solution should also be found in that class. In this case, the child class must decide how to disambiguate the overloaded term.

One solution is to simply always use fully qualified names. Rather than asking an instance of GraphicalCardDeck to draw, the programmer specifically indicates which type of drawing is intended.

```
GraphicalCardDeck gcd;
Card * aCard = gcd->CardDeck::draw();
gcd->GraphicalObject::draw();
```

However, this solution is less than ideal, since the syntax is different from other function calls, and the programmer must remember which method comes from which parent.

The more common solution usually involves a combination of *renaming* and *redefinition*. By *redefinition* we mean a change in the operation of a command, such as what happens when a virtual method is overridden in a subclass. By *renaming* we simply mean changing the name by which a method is invoked without altering the command's functionality.

If the type signatures of the parent methods are distinguishable and our language allows overloading, then the child class can simply include both methods.

```
class GraphicalCardDeck : public CardDeck, public GraphicalObject {
public:
  virtual Card * draw () { return CardDeck::draw(); }
  virtual void draw (Graphics * g) { GraphicalObject::draw(g); }
}
```

An instance of the child can then select the appropriate method by using the correct arguments.

```
GraphicalCardDeck gcd;
Graphics g;
gcd->draw(); // selects CardDeck draw
gcd->draw(g); // selects GraphicalObject draw
```

If the type signatures are similar, or if overloading is not allowed, then the programmer is faced with a more difficult decision. The method draw can mean only one thing to the child, and the programmer must decide which of the parents interpretations it should be. In our graphical deck of playing cards, the programmer might have draw mean the process of removing a card from the deck and rename the process of drawing the image as paint.

```
class GraphicalCardDeck : public CardDeck, public GraphicalObject {
public:
  virtual void draw () { return CardDeck::draw(); }
  virtual void paint () { GraphicalObject::draw(); }
}

  GraphicalCardDeck gcd;
  gcd->draw(); // selects CardDeck draw
  gcd->paint(); // selects GraphicalObject draw
```

13.2.2 Impact on substitutability

While the redefinition of the function name draw solves the problem locally when a GraphicalCardDeck is used in isolation, further problems arise when we consider the implications of the principle of substitution. Suppose our GraphicalCardDeck is maintained on a list of graphical objects, say for the purposes of rendering a window image. When the method draw is invoked on the graphical object, it will have the effect of executing the CardDeck method and not the graphical operation.

```
GraphicalObject * g = new GraphicalCardDeck();
g->draw(); // oops, wrong method
```

The typical way this problem is overcome in C++ is through the introduction of two new helper classes. Each of these classes inherits from one parent and redefines the draw operation to use a method of a different name.

```
class CardDeckParent : public CardDeck {
public:
  virtual void draw () { cardDeckDraw(); }
  virtual void cardDeckDraw() { CardDeck::draw(); }
};

class GraphicalObjectParent : public GraphicalObject {
public:
  virtual void draw () { goDraw(); }
  virtual void goDraw () { GraphicalObject::draw(); }
}
```

The child class can now inherit from these new parents and override the new methods, which no longer have similar names. When used by itself the new child class has access to both behaviors, and when assigned by substitution to an instance of either parent class, the expected behavior is produced.

```
class GraphicalCardDeck : public CardDeckParent, GraphicalObjectParent
{
public:
  virtual void cardDeckDraw () { ... }
  virtual goDraw () { ... }
};
```

Note that method named *draw* is still ambiguous in the child class but not when the object is referenced as an instance of either parent class. C++ does not care about this ambiguity as long as no attempt is made to use the name. An error will result if an attempt is made to reference the ambiguous method.

```
GraphicalCardDeck *gcd = new GraphicalCardDeck();
CardDeck *cd = gcd; // all three variables refer
GraphicalObject *go = gcd; // to same object
cd->draw(); // ok, will execute cardDeckDraw
go->draw(); // ok, will execute goDraw
gcd->draw(); // compiler error, ambiguous invocation
```

One might imagine that this ambiguity could be overcome by redefining draw in the child class GraphicalCardDeck and selecting one or the other meaning. However, because once a method is declared as virtual it continues to be virtual in all child classes, this cannot be done without hiding the other meaning and thus preventing substitution. However, it is still possible to use fully qualified names for the parent methods, as described earlier.

*13.2.3 Redefinition in Eiffel

The object-oriented language Eiffel allows multiple inheritance and overcomes ambiguity through a technique that permits inherited names to be redefined. The child class can therefore access inherited methods through the redefined names in an unambiguous fashion.

```
class GraphicalCardDeck
inherit
  CardDeck
    rename
      draw as cardDeckDraw
    end
  GraphicalObject
    rename
      draw as goDraw
    end
```

```
feature
  -- can use cardDeckDraw and goDraw without conflict
  -- can even override the redefined names
end
```

This solution is roughly equivalent to the earlier C++ program but avoids the need to introduce the two intermediary classes. Like the C++ program it requires that when the child class is used by itself, it must use the renamed methods. When an instance of the child class is accessed through a reference to either parent, they can use the method draw, and the correct method will be executed. However, again like the C++ solution, when the method draw is used with a child class reference, the meaning is ambiguous and will produce a compiler error.

*13.2.4 Resolution by class ordering in CLOS

While C++ and Eiffel resolve naming conflicts by insisting that the child class make the intent clear, the language CLOS takes a different approach. In CLOS conflicts are resolved by the order that parent classes are declared. For example, suppose we have two classes animated-character and movie-star, both possessing a name field and both having a method named name-of.

```
(defclass animated-character () ((name :initarg :name)))

(defmethod name-of ((self animated-character))
  (list "Animated Character" (slot-value self 'name)))

(defclass movie-star () ((name :initarg :name)))

(defmethod name-of ((self movie-star))
  (list "Movie Star" (slot-value self 'name)))
```

We can create a new class of animated movie stars by inheriting from both animated characters and movie stars.

```
(defclass animated-star (animated-character movie-star) () )
```

Note that there are two name slots in such an object, one from each parent class. If we define an instance of the child class and specify a value, both will be filled. If we print the value of the object, we can see that the method from class animated-character is selected.

```
  (defvar micky (make-instance 'animated-star :name "Micky"))
  (name-of micky)
("Animated Character" "Micky")
```

This is because the parent class animated-character is searched before the class movie-star. If we listed the parent classes in the reverse order, the method in class movie-star would be used.

```
(defclass star-animated (movie-star animated-character) () )
(defvar minnie (make-instance 'star-animated :name "Minnie"))
(name-of minnie)
("Movie Star" "Minnie")
```

Wrappers

Because multiple method bodies match the invocation of the name-of function, we can use the idea of *wrappers* to execute each function in turn. The method call-next-method will execute the next method that matches the given arguments. However, this can only be executed if there *is* a next method, something that can be checked using the predicate next-method-p. For example, we could rewrite both name-of methods as follows.

```
(defmethod name-of ((self animated-character))
  (list "Animated Character" (slot-value self 'name)
    (if (next-method-p) (call-next-method))))

(defmethod name-of ((self movie-star))
  (list "Movie Star" (slot-value self 'name)
    (if (next-method-p) (call-next-method))))
```

If we now execute the method, we will see that both functions have been executed and the results combined. (The if statement in the second function returns nil, since there is no next method, something we could have fixed with more programming.)

```
(name-of minnie)
("Movie Star" "Minnie" ("Animated Character" "Minnie" NIL))
```

We will examine a further use of call-next-method in Section 13.3.1.

Multiple inheritance in Python

Python is another language that resolves name conflicts that arise from multiple inheritance by performing a depth-first search of the ancestor hierarchy. Consider the following class definitions.

```
class A:
  def method1(self):
    print "method1 A"
```

```
  def method2(self)
    print "method2 A"

class B(A):
  def method2(self)
    print "method2 B"

class C:
  def method1(self):
    print "method1 A"
  def method2(self)
    print "method2 C"

class D(B,C):
  def method2(self)
    C.method2(self)
```

If we create an instance of D and execute method1, it will select the method in class A.

```
  d = D()  # create a D
  d.method1()
method1 A
  d.method2()
method2 C
```

If inheritance had been specified as D(C,B), then the call on method1 would have found the function in class C, rather than the function in class A. Programmers can override and select the desired meaning by using the class qualified name, as show in the body of the function method2 in class D. This is similar to the use of fully qualified names in C++, which we described earlier.

13.3 ▫ Multiple Inheritance of Interfaces

Neither Java nor C# allow multiple inheritance of classes, but both languages permit multiple inheritance of interfaces. While sometimes used for similar purposes, there are important differences between interfaces and classes. Both mechanisms can be used for classification (the is-a relation is a valid test for both), but interfaces do not supply code to the child class. For this reason the conflict between two inherited code bodies does not arise. Either methods in the parent interfaces have the same type signature, in which case they are merged and the child need implement only one.

```
interface CardDeck {
  public void draw ();
}

interface GraphicalObject {
  public void draw ();
}

class GraphicalCardDeck implements CardDeck, GraphicalObject {
  public void draw () { ... } // only one method
}
```

Or the parent classes have different type signatures, in which case the child class must implement both.

```
interface CardDeck {
  public Card draw ();
}

interface GraphicalObject {
  public void draw (Graphics g);
}

class GraphicalCardDeck implements CardDeck, GraphicalObject {
  public Card draw () { ... } // must implement both methods
  public void draw (Graphics g) { ... }
}
```

The only restriction is similar to that encountered in simple overloading. Namely, two methods cannot differ only in their return type or in the type of exceptions they throw, since the compiler cannot determine from a function invocation which method is being called. The compiler will detect an error and issue a diagnostic message in this situation.

```
interface CardDeck {
  public void draw () throws EmptyDeckException;
}

interface GraphicalObject {
  public void draw ();
}

class GraphicalCardDeck implements CardDeck, GraphicalObject {
  public void draw () { ... } // which one is this?
}
```

In Java, interfaces are allowed to define constants (using the final modifier), and so there is the potential for two parent interfaces to define a constant using the same name and different values. However, the compiler checks for this condition and will issue an error diagnostic should it arise.

Explicit interface implementation in C#

The language C# is much more restrictive than Java with regard to the combination of methods from different interfaces. The C# compiler will report this condition as an error, even if the type signatures match. Presumably this policy is due to the assumption that if the interfaces arise from different sources they therefore most likely specify different actions.

The programmer can get around this restriction by explicitly naming the interface associated with the method being defined. If the method is to be further exposed from the child class, a third method can be defined with the same name.

```
class GraphicalCardDeck: CardDeck, GraphicalObject {
  void CardDeck.draw () {  // method number 1
      .
      .
      .
  }

  void GraphicalObject.draw () { // method number 2
      .
      .
      .
  }

  public void draw () { // method number 3
      .
      .
      .
  }
}
```

Each of these three methods can be executed, depending on the static type of a declaration.

```
GraphicalCardDeck theDeck = new GraphicalCardDeck();
theDeck.draw();   // executes method number 3
CardDeck cd = (CardDeck) theDeck;
cd.draw(); // executes method 1
GraphicalObject go = (GraphicalObject) theDeck;
go.draw(); // executed method 2
```

*13.3.1 Mixins in CLOS

CLOS programmers frequently make use of a programming idiom called a *mixin*. The concept of a mixin is in some ways almost the exact opposite of the notion of inheritance of interfaces. While interfaces specify a protocol but do not provide code, mixins provide code without changing the interface. Mixins are snippets of code that are combined precisely because of their behavior and not their type. The child class inherits a bit of code from one parent, a bit of code from the other, and combines them to form their own new behavior.

For example, suppose we have a class Person with a slot for a name and a method named title to return the name.

```
(defclass Person () ((name :initarg :name)))

(defmethod title ((self Person))
  (list (slot-value self 'name)))

(defvar tom (make-instance 'Person :name "Tom Smith"))

(title tom)
("Tom Smith")
```

We can create two mixins, one that will add the prefix "Prof." to a title, and one that will add the prefix "Dr." and tack a "PhD" on the end.

```
(defclass Prof () () )

(defmethod title ((self Prof))
  (cons "Prof." (call-next-method)))

(defclass PhD () () )

(defmethod title ((self PhD))
  (append (cons "Dr." (call-next-method)) (list "PhD")))
```

Note that these are declared as classes but do not explicitly reference the class Parent. The use of call-next-method is what distinguishes these classes as mixins. To use these, we can create a new class that uses multiple inheritance from one or more mixin before finally associating a base class. For example, we might make a simple Professor class.

```
(defclass Professor (Prof Person) () )
```

If we make an instance of this class and execute the inherited method title, we see that first the method from Prof is executed and then the method from Person.

```
  (defvar bill (make-instance 'Professor :name "Bill Jones"))
  (title bill)
("Prof." "Bill Jones")
```

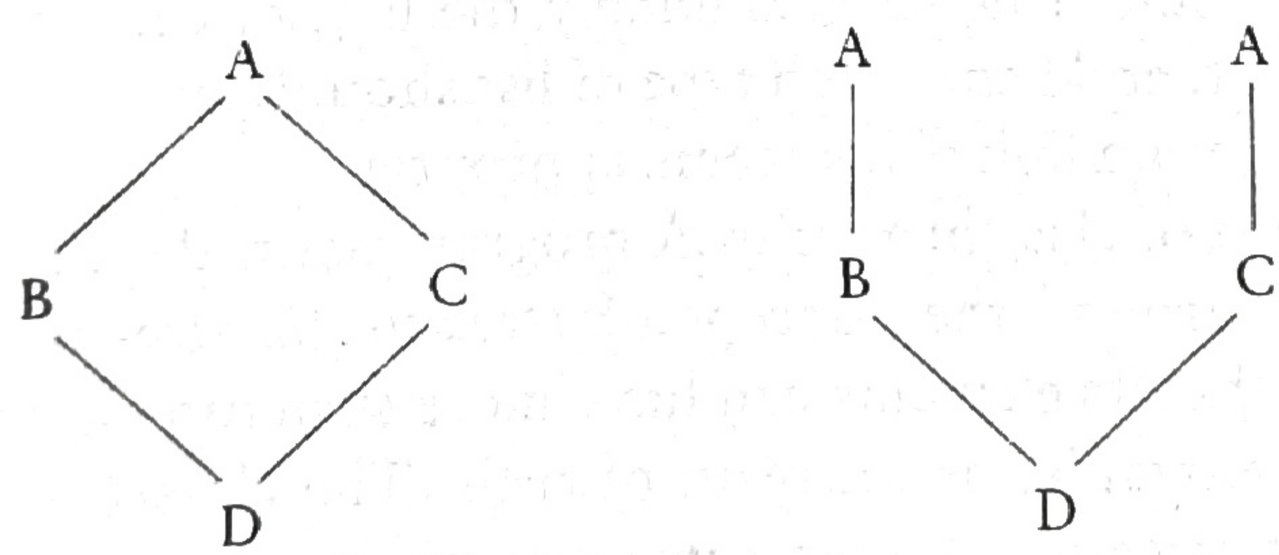
A characteristic of mixins is that they can be combined in different orders. For example, we can make both classes for Professor Doctors and for Doctor Professors.

```
(defclass ProfDr (Prof PhD Person) () )
(defclass DrProf (PhD Prof Person) () )

  (defvar alice (make-instance 'ProfDr :name "Alice Smith"))
  (title alice)
("Prof." "Dr." "Alice Smith" "PhD")
  (defvar nancy (make-instance 'DrProf :name "Nancy Jones"))
  (title nancy)
("Dr." "Prof." "Nancy Jones" "PhD")
```

The programming language Ruby provides similar mixin facilities in the context of a language that only supports single inheritance.

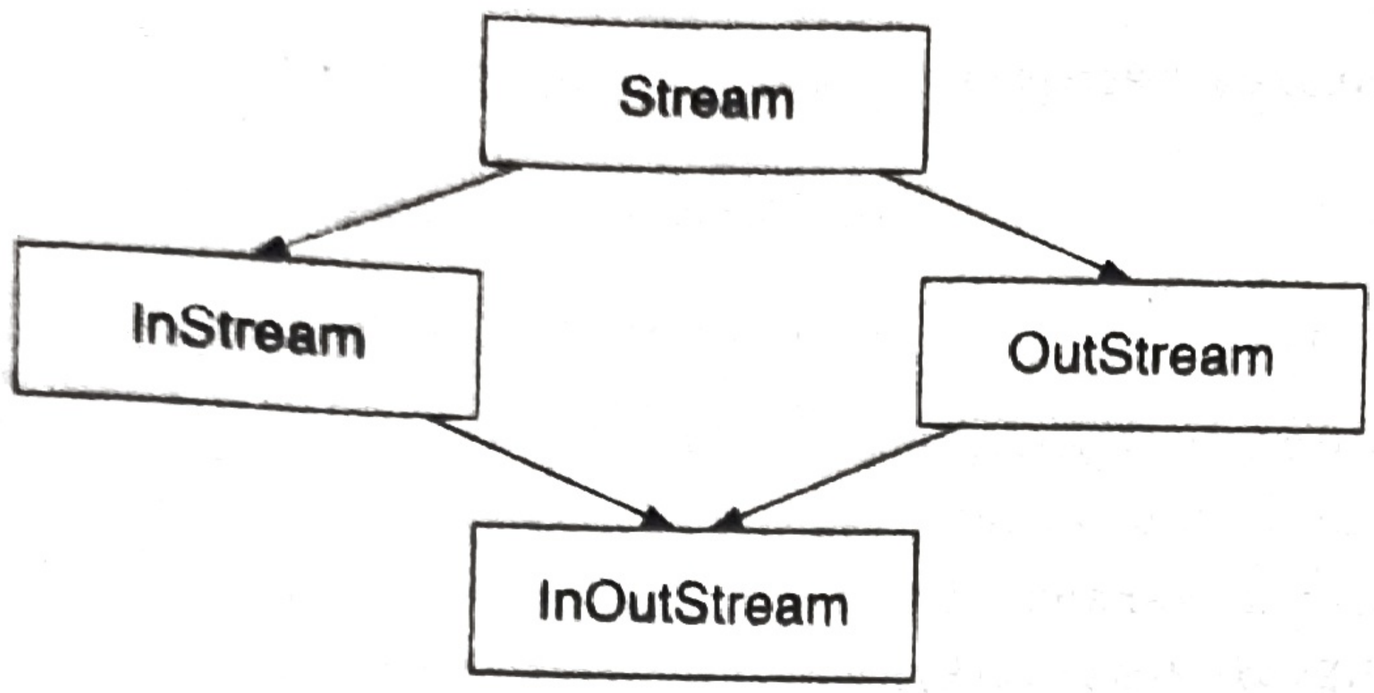
13.4 ▫ Inheritance from Common Ancestors

By far the most subtle problems involving multiple inheritance arise when two or more classes inherit from a common parent class, such as in Figure 13.3. Scott Meyers calls this, rather melodramatically, the "diamond of death" [Meyers 1998]. The picture itself explains part of the controversy. Should it be drawn so that the common ancestors are merged or to keep them distinct?



The difference between these two representations can be appreciated if we imagine that the common ancestor class defines a data member. Should the child class have one copy of this data member or two? There is no right answer to this question, since both arise in practice.

For example, suppose a programmer creates a linked list by making a parent Link class and declaring objects on the list as children of the Link class. (This is not a particularly good idea, since using standard container abstractions makes



▫ Figure 13.3 — An inheritance graph

the code more robust, but nevertheless the technique is, perhaps unfortunately, common and is easy to understand.)

```
class Link {
public:
  Link * nextLink;
};
```

Declaring CardDeck to be a child class of Link allows the programmer to maintain a list of card decks. Similarly, declaring GraphicalObject to be a child class of Link allows the programmer to keep a list of graphical objects.

```
class CardDeck : public Link { ... };
class GraphicalObject : public Link { ... };
```

When the child class is created that inherits from both CardDeck and GraphicalObject, how many nextLink fields should it hold? Presumably the lists of card decks and graphical objects are distinct, and hence each type of list should have its own links. Therefore having two separate link fields seems appropriate.

Now imagine a different type of class hierarchy. A programmer is developing an input/output library based around the concept of *streams*. A stream is a generalization of a file, except that its elements can have more structure. We can have a stream of integers, for example, or a stream of reals. The class InStream provides a protocol for input streams. A user can open an input stream by attaching it to a file, retrieve the next element in the stream, and so on. The class OutStream provides similar functionality for output streams. Both classes inherit from a single parent class, Stream. The information that points to the actual underlying file (for example, a file pointer) is maintained in the parent class.

Now suppose the user wants to create a combined input-output stream. It makes sense to claim that an input-output stream is a descendant of both an

```
class Stream {
  File *fid;
  .
  .
  .
  };

class InStream : public virtual Stream {
  .
  .
  .
  int open(File *);
  };

class OutStream : public virtual Stream {
  .
  .
  .
  int open(File *);
  };

class InOutStream: public InStream, public OutStream {
  .
  .
  .
  };
```

Figure 13.4 — An example of virtual inheritance

input stream and an output stream. But in this case, there is only one object, and hence we want the two file pointers (and any other common data) to refer to the same values. That is, we want only one copy of the common ancestor data.

In C++ this problem is overcome through the use of the virtual modifier in the parent class list. The virtual keyword indicates that the superclass may appear more than once in descendant classes of the current class but that only one copy of the superclass should be included. Figure 13.4 shows the declarations for the four classes.

An unfortunate consequence of the C++ approach is that, as we noted earlier, the name confusion is a problem only for the child class, but the solution (making the common ancestor virtual) involves changes to the parent classes. It is the intermediate parent classes that give the virtual designation, not the final combined class.

The fact that visibility keywords can be attached to parent classes independently means that it is possible for a virtual ancestor class to be inherited in different ways—for example, as both public and protected. In this case, the lesser

```
class D {
public:
  D() { ... }
  D(int i) { ... }
  D(double d) { ... }
};

class A : public virtual D {
public:

  A() : D(7)  { ... }
};

class B : public virtual D {
public:
  B() : D(3.14) { ... }
};

class C: public A, public B
{
  public:
    C() : B(),A() { ... }
  };
```

▫ Figure 13.5 — Constructors in multiple inheritance

level of protection (for example, protected) is ignored and the more general category used.

*13.4.1 Constructors and multiple inheritance

When more than one parent class defines a constructor, the order of execution of the various constructors, and hence initialization of their data fields, may be important. The user can control this by invoking the constructors for the base classes directly in the constructor for the child class. For example, in Figure 13.5, the user explicitly directs that when an instance of class C is initialized, the constructor for B is to be invoked first, before the constructor for A. Reversing the order of invocations of the constructor in class C has the effect of reversing the order of initialization.

An exception to this rule occurs with virtual base classes. A virtual base class is always initialized once, before any other initialization takes place, by the

constructor (provided by the system if not by the user), which takes no arguments. Thus, in Figure 13.5 the order of initialization when a new element of type C is constructed is first class D with the no-argument constructor, then class B, then class A. The two seeming calls on the constructor for class D that appear in the constructors for classes A and B actually have no effect, since the parent class is marked as virtual.

If it is imperative that arguments for the virtual base class be provided with the constructor, class C may legally provide these values, even though D is not an immediate ancestor for C. This is the only situation in which it is legal for a class to provide a constructor for another class that is not an immediate ancestor. That is, the constructor for class C could have been written as follows.

```
C()  :  D(12), B(),  A() { . . . }
```

Constructors for virtual base classes must be invoked first, before the constructors for nonvirtual ancestors.

Virtual methods in the common ancestor

Virtual methods defined in virtual superclasses can also cause trouble. Suppose that each of the four classes shown in Figure 13.4 defines a method named initialize(). This method is defined as virtual in the class Stream and redefined in each of the other three classes. The initialize methods in InStream and OutStream each invoke Stream::initialize and, in addition, do some subclass-specific initialization.

Now consider the method InOutStream. It cannot call both the inherited methods InStream::initialize and OutStream::initialize without invoking the method Stream::initialize twice. The repeated invocation of Stream::initialize may have unintended effects. The way to avoid this problem is to rewrite Stream::initialize so it detects whether it has been initialized or to redefine the methods in the subclasses InStream and OutStream so they avoid the invocation of the method from class Stream. In the latter case, the class InOutStream must then invoke the initialization procedures explicitly for each of the three other classes.

13.5 ▫ Inner Classes

The ability to nest classes in Java and C++ provides a mechanism that can be used in a fashion that is very nearly equivalent to multiple inheritance but that avoids many of the semantic problems we have been discussing. A nested class in Java retains access to methods in the surrounding outer class. (This is not true in C++, but it can be simulated by passing a reference to the outer class when the inner class is constructed.) To create an object that inherits from two parents, the outer class inherits from the first parent, and the inner class inherits from the second.

```
class GraphicalCardDeck extends CardDeck {
  // outer class has access to all CardDeck behavior

  public void draw () { // can override the CardDeck methods
      .
      .
      .
  }

  private drawingClass drawer = new drawingClass();

  public GraphicalObject myDrawingObject () { return drawer; }

  private class drawingClass extends GraphicalObject {
    // inner class has access to all GraphicalObject behavior
    public void draw () {
      // can override the GraphicalObject methods
      .
      .
      .
    }
  }
}
```

The outer class can override methods from one parent, the inner class from the second. The inner class has access to methods from both parents. Seemingly the only difficulty this solution introduces is the problem of substitution. An instance of GraphicalCardDeck can be assigned to a variable of type CardDeck but not to one declared as type GraphicalObject. But even here there is an easy way to overcome this problem. The outer class can return an instance of the inner class (as is done by the method myDrawingObject) that can then be assigned to the parent class variable. There is, indeed, no way that the *same variable* can be assigned to two different parent classes, but this situation does not seem to arise in practice. The case study presented in Chapter 22 makes use of an inner class in just this fashion.

Summary ▣

One way to view the mechanism of inheritance is as a process of categorization. The child class is a more specialized form of the category defined by the parent class. However, in most real world categorizations objects fit into several mutually independent categories. This observation leads to the concept of multiple inheritance, a class that can inherit from two or more parent classes.

While multiple inheritance on the surface seems desirable, there are several pitfalls that can trap the unwary. The most important of these is the ambiguity of

CHAPTER 14

Polymorphism and Software Reuse

In this chapter and in the four that follow we begin an investigation of the mechanisms that are in object-oriented programming languages collectively described using the term *polymorphism*. We have seen simple examples of most of these—for example, overriding—in earlier chapters. However, an in-depth treatment will allow us to explore details that might not be obvious from a cursory explanation. After examining the various techniques denoted by this term we then examine, starting in Chapter 19, several applications of polymorphism to a range of common problems.

The term *polymorphic* has Greek roots and means, roughly, "many forms." (*poly* = many, *morphos* = form. Morphos is related to the Greek god Morphus, who could appear to sleeping individuals in any form he wished and hence was truly polymorphic.) In biology, a polymorphic species is one, such as *Homo sapiens*, that is characterized by the occurrence of different forms or color types in individual organisms or among organisms. Sponges are another polymorphic species. In chemistry, a polymorphic compound is one that can crystallize in at least two distinct forms, such as carbon, which can crystallize as graphite, diamond, or fullerenes.

14.1 ▣ Polymorphism in Programming Languages

As is perhaps appropriate, the name polymorphism means different things to different people. In part this confusion is related to language paradigms. The term originated with work in the functional language world, and only later was it

adopted by designers of object-oriented languages. As a consequence the term has come to mean one thing in the object-oriented community and something slightly different when applied to functional languages.

At heart, the term means there is one name and many different meanings. But names are used for a variety of purposes (variable names, function names, class names), and meanings can be defined in a number of different ways. It is therefore useful to consider at least four different forms of polymorphism.

- The term *overloading* (also known as *ad hoc polymorphism*) is used to describe the situation where a single function name (or method name) has several alternative implementations. Typically overloaded function names are distinguished at compile time based on their type signatures.

```
class OverLoader {
    // three overloaded meanings for the same name
  public void example (int x) { ... }
  public void example (int x, double y) { ... }
  public void example (string x) { ... }
}
```

We will discuss overloading in Chapter 15.

- *Overriding* (or *inclusion polymorphism*) is in some sense a special case of overloading but occurs within the context of the parent class/child class relationship.

```
class Parent {
  public void example (int x) { ... }
}

class Child extends Parent {
    // same name, different method body
  public void example (int x) { ... }
}
```

Like ad-hoc polymorphism there are two definitions that use the same method name. However, unlike overloading, these two definitions must have the same type signature, and the two methods must appear in classes related by inheritance. Overriding will be described in Chapter 16.

- The *polymorphic variable* (or *assignment polymorphism*) is a variable that is declared as one type but in fact holds a value of a different type.

```
Parent p = new Child(); // declared as parent, holding child value
```

When a polymorphic variable is used as argument, the resulting function is said to exhibit *pure polymorphism*. Both the polymorphic variable and pure polymorphism are discussed in Chapter 17.

- Finally, *generics* (or *templates*) provide a way of creating general purpose tools and specializing them to specific situations.

```
template <class T> T max (T left, T right)
{
    // return largest argument
  if (left < right)
    return right;
  return left;
}
```

A generic function or class is parameterized by a type, in much the same way that a function is parameterized by values. By leaving the type unspecified, to be filled in later, a generic allows the function or class to be used in a wider range of situations. Generics will be examined in Chapter 18.

14.1.1 Many tools, one goal

It is polymorphism more than any other feature that distinguishes object-oriented languages from other language paradigms. Each of the different polymorphic techniques permits a different form of software reuse. Each is powerful in its own way, a fact we will demonstrate in the case study chapters that will follow our investigation of the basic techniques. By facilitating the reuse of standard software components, each contributes to the goals we earlier identified as the driving forces for object-oriented programming—namely, rapid development, reliability, and ease of use.

14.2 ⊡ Mechanisms for Software Reuse

Object-oriented programming has been billed as the technology that will finally permit software to be constructed from general-purpose reusable components. Writers such as Brad Cox have even gone so far as to describe object orientation as heralding the "industrial revolution" in software development. While the reality may not quite match the hyperbole of OOP pioneers, it *is* true that object-oriented programming makes possible a level of software reuse that is orders of magnitude more powerful than that permitted by previous software construction techniques. In this section, we will investigate and contrast the two most common mechanisms for software reuse: *inheritance* and *composition*.

To illustrate these two techniques, we will use the construction of a set abstraction by using an existing class, List, which maintains a list of integer values. Imagine we have already developed a class List with the following interface.

```
class List {
public:
    // constructor
  List  ();

    // methods
  void  add       (int);
  int   firstElement  ();
  int   size          ();
  int   includes      (int)
  void  remove        (int);
    .
    .
    .
};
```

That is, our list abstraction permits us to add a new element to the front of the list, to return the first element of the list, to compute the number of elements in the list, to see if a value is contained in the list, and to remove an element from the list.

We want to develop a set abstraction to perform operations such as adding a value to the set, determining the number of elements in the set, and determining whether a specific value occurs in the set.

Is-a and Has-a Revisited

It should be clear that the is-a and has-a relationships that we have been discussing since Chapter 2 correspond directly to the programming mechanisms of inheritance and composition. If class A has-a class B, then naturally a data field of class B should be part of an instance of class A. On the other hand, if class A is-a class B, then inheritance is an appropriate coding mechanism.

14.2.1 Using composition

We will first investigate how the set abstraction can be formed with composition (also sometimes called *layering*). Recall from our earlier discussion that an object is simply an encapsulation of data (data values) and behavior. When composition

is employed to reuse an existing data abstraction in the development of a new data type, a portion of the state of the new data structure is simply an instance of the existing structure. This is illustrated here, where the data type Set contains an instance field named theData, which is declared to be of type List.

```
class Set {
public:
  Set  ();  // constructor

    // operations
  void  add       (int);
  int   size    ();
  int   includes  (int)

private:  // data area for set values
  List  theData;
};
```

Because the List abstraction is stored as part of the data area for our set, it must be initialized in the constructor. As with the initialization of data fields in classes (Chapter 5), the initializer clause in the constructor provides the arguments used to initialize the data field. In this case the constructor we invoke for class List is the constructor with no arguments.

```
  // initialize list
Set::Set() : theData()
{
  // no further initialization
}
```

Operations that manipulate the new structure are implemented by use of the existing operations provided for the earlier data type. For example, the implementation of the includes operation for our set data structure simply invokes the similarly named function already defined for lists.

```
int Set::size ()
{
  return theData.size();
}

int Set::includes (int newValue)
{
  return theData.includes(newValue);
}
```

The only operation that is slightly more complex is addition, which must first check to ensure that the value is not already contained in the collection (since values can appear in a set no more than once).

```
void Set::add (int newValue)
{
    // if not already in set
  if (! includes (newValue))
      // then add
    theData.add(newValue);

    // otherwise do nothing
}
```

The important point is the fact that composition provides a way to leverage an existing software component in the creation of a new application. By using the existing List class, the majority of the difficult work in managing the data values for our new component have already been addressed.

On the other hand, composition makes no explicit or implicit claims about substitutability. When formed in this fashion, the data types Set and List are entirely distinct, and neither can be substituted in situations where the other is required.

Composition in other languages

Composition can be applied in any of the object-oriented languages we consider in this book. Indeed, it can be applied in non-object-oriented languages as well. The only significant difference in the various languages is the way in which the encapsulated data abstraction is initialized.

14.2.2 Using inheritance

An entirely different mechanism for software reuse in object-oriented programming is the concept of inheritance; with inheritance a new class can be declared a *subclass*, or *child class*, of an existing class. In this way, all data areas and functions associated with the original class are automatically associated with the new data abstraction. The new class can, in addition, define new data values or new functions. It can also *override* functions in the original class, simply by defining new functions with the same names as those of functions that appear in the parent class.

These possibilities are illustrated in the class description following, which implements a different version of the Set abstraction. By naming the class List in the class heading, we indicate that our Set abstraction is an extension, or a

refinement, of the existing class List. Thus, all operations associated with lists are immediately applicable to sets as well.

```
class Set : public List {
public:
    // constructor
  Set  ();

    // operations
  void  add  (int);
  int   size  ();
};
```

Notice that the new class does not define any new data fields. Instead, the data fields defined in the List class will be used to maintain the set elements, but they must still be initialized. This is performed by invocation of the constructor for the parent class as part of the constructor for the new class.

```
Set::Set()  : List()
{
  // no further initialization
}
```

Similarly, functions defined in the parent class can be used without any further effort, so we need not bother to define the includes method because the inherited method from List uses the same name and shares the same purpose. The addition of an element to a set, however, requires slightly more work and is handled as follows.

```
void  Set::add  (int newValue)
{
    // add only if not already in set
  if (! includes(newValue))
    List::add (newValue);
}
```

Compare this function with the earlier version. Both techniques are powerful mechanisms for code reuse, but unlike composition, inheritance carries an implicit assumption that subclasses are, in fact, subtypes. This means that instances of the new abstraction should act in a similar fashion to instances of the parent class.

14.2.3 Composition and inheritance contrasted

Having illustrated two mechanisms for software reuse, and having seen that they are both applicable to the implementation of sets, we can comment on some of the advantages and disadvantages of the two approaches.

- Composition is the simpler of the two techniques. Its advantage is that it more clearly indicates exactly what operations can be performed on a particular data structure. Looking at the declaration for the Set data abstraction, it is clear that the only operations provided for the data type are addition, the inclusion test, and size. This is true regardless of what operations are defined for lists.
- In inheritance the operations of the new data abstraction are a superset of the operations of the original data structure on which the new object is built. Thus, to know exactly what operations are legal for the new structure, the programmer must examine the declaration for the original. An examination of the Set declaration, for example, does not immediately indicate that the includes test can be legally applied to sets. It is only by examination of the declaration for the earlier List data abstraction that the entire set of legal operations can be ascertained.

  But there is a difficulty that can occur when a programmer must try to understand a class constructed using inheritance. Often the programmer must repeatedly flip back and forth between two (or more) class declarations. This has been labeled the "yo-yo" problem, due to the image of the programmer's head bobbing up and down as he (or she) moves between various class definitions.
- The brevity of data abstractions constructed with inheritance is, in another light, an advantage. Using inheritance it is not necessary to write any code to access the functionality provided by the class on which the new structure is built. For this reason, implementations using inheritance are almost always, as in the present case, considerably shorter in code than are implementations constructed with composition, and they often provide greater functionality. For example, the inheritance implementation makes available not only the includes test for sets but also the function remove.
- Inheritance does not prevent users from manipulating the new structure using methods from the parent class, even if these are not appropriate. For example, when we use inheritance to derive the class Set from the class List, nothing prevents users from retrieving the first element using the method firstElement.
- In composition the fact that the class List is used as the storage mechanism for our sets is merely an implementation detail. With this technique it would be easy to reimplement the class to use a different technique (such as a hash table) with minimal impact on the users of the Set abstraction. If users counted on

the fact that a Set is merely a specialized form of List, such changes would be more difficult to implement.

- Inheritance may allow us to use the new abstraction as an argument in an existing *polymorphic* function. We will investigate this possibility in more detail in Chapter 17. Because composition does not imply substitutability, it usually precludes polymorphism.
- Understandability and maintainability are difficult to judge. Inheritance has the advantage of brevity of code but not of protocol. Composition code, although longer, is the only code that another programmer must understand to use the abstraction. A programmer faced with understanding the inheritance version needs to ask whether any behavior inherited from the parent class was necessary for proper utilization of the new class and would thus have to understand both classes.
- Data structures implemented through inheritance tend to have a very small advantage in execution time over those constructed with composition, since one additional function call is avoided (although techniques such as in-line functions in C++ can be used to eliminate much of this overhead).

Of the two possible implementation techniques, can we say which is better in this case? One answer involves the substitution principle. Ask yourself whether in an application that expected to use a List data abstraction it is correct to substitute instead an instance of class Set. While the technical answer might be yes (the Set abstraction does implement all the List operations), the more commonsense answer is no. While the type signature has been preserved, the meaning of the add operation has changed. For this reason it appears that, *in this case*, composition is the better approach.

The bottom line is that the two techniques are very useful, and an object-oriented programmer should be familiar with both of them.

14.3 Efficiency and Polymorphism

Almost all the programming techniques we will discuss in the next four chapters extract some cost in run-time efficiency. That being the case, a few words on the relative importance of efficiency in relation to other concerns is in order.

The fundamental truth is that programming always involves compromises. In particular, programming with polymorphism involves compromises between ease of development and use, readability, and efficiency. In large part, efficiency has been already considered and dismissed; however, it would be remiss not to admit that it is an issue, however slight.

A polymorphic function that does not know the type of its arguments can seldom be as efficient as a function that has more complete information. A relational

test may correspond to only a few assembly language instructions if the arguments are integer, whereas much more extensive operations are necessary if the arguments are strings or dates and considerably more work if the determination of types is postponed until run time. Nevertheless, the advantages of rapid development and consistent application behavior and the possibilities of code reuse usually (but not always!) more than make up for any small losses in efficiency.

14.4 ⊡ Will Widespread Software Reuse Become Reality?

In the early days of object-oriented programming, it was said that computer science had finally found a way to create software from general-purpose interchangeable parts and that programming would henceforth be more like electrical or civil engineering, where new ideas were created using a collection of standard components. Polymorphism in its various guises was the tool that would make this transformation possible.

However, while certain progress has been made (there are now a large number of commercial vendors who market general-purpose object-oriented software libraries for various applications, such as user interfaces and container classes), the overall process has not yet lived up to the early expectations. There are a variety of reasons for this.

- Inheritance and composition provide the *means* for producing reusable components, but they do not by themselves provide *guidelines* for how such a task should be performed. It turns out that producing good and useful software components is, if anything, almost always more difficult than developing special-purpose software to solve the task at hand.
- Because producing reusable components is difficult, the benefits cannot usually be realized within a single project. Indeed, they may slow down project development. Rather than creating immediate benefits, the cost of such development must be amortized over many programming projects. But as each project usually has its own budget and schedule, there is often no management mechanism to support such amortization.
- Because the benefits of developing reusable components do not immediately improve a project, there is usually little incentive for programmers to strive toward reusability.
- Because each new problem typically requires a slightly different set of behaviors, it is often difficult to design a truly useful and general-purpose software component the first time. Rather, useful reusable software components evolve slowly over many projects until they finally reach a stable state.

- Many programmers and managers are leery of software that has not been developed "in-house." This wariness is called the "not-invented-here" syndrome. Because managers pride themselves on the quality of their programmer teams, they naturally believe they can do better than whatever team developed the reusable software.
- Because many programmers have little formal training or have not kept pace with recent programming innovations (such as object-oriented techniques), they may not be aware of the mechanisms available for the development of reusable software components.

In short, development of software mechanisms for reuse does not by itself guarantee the development of a technological and management *culture* that will support and encourage reuse. Human organizations tend to move much more slowly than does technological change, so it may be many years before we see the true benefits promised by the object-oriented approach. Nevertheless, even though object reuse is probably not anywhere as frequent as claimed, it does occur and has been proven many times to be useful and cost-saving when applied correctly. For this reason it is inevitable that reuse will eventually become the norm for software development.

Summary

In this chapter we have begun our in-depth investigation of polymorphism. As is perhaps appropriate (since polymorphism means roughly "many forms") there are a variety of mechanisms that are known by this term.

- Overloading
- Overriding
- The polymorphic variable
- Generics

All the various mechanisms are united by the goal of easing software reuse and thereby promoting reliability, rapid development, and ease of use.

Further Information

Brad Cox's very readable manifesto on the software industrial revolution is [Cox 1986].

The yo-yo problem was first described in [Taenzer 1989].

CHAPTER 15

Overloading

We say a term is *overloaded* if it has many different meanings.[1] A glance in any dictionary will show that many words in the English language are overloaded, and the meaning of a word in a particular situation is determined by context. In programming languages it is usually function or method names that are overloaded. As with words in English, a compiler will use contextual information to determine the exact meaning.

Overloading and overriding are both techniques that take a function or method named in a specific invocation and select for execution one out of potentially many different function bodies. But the two mechanisms differ in several regards. Most importantly, overloading is performed at compile time (early binding), whereas overriding uses a run-time selection (late binding). The compiler examines the context in which an overloaded name is being used and determines the most appropriate function body. Normally the most critical feature is the type signature of the arguments being used by the call. Because the function selection is being made at compile time, it is the static (or declared) types of the arguments that are used and not their dynamic (or run-time) values.

In one sense, overloading (also called *ad hoc polymorphism*) may seem to have little to do with object-orientation, since the ability to overload function names can be found in a number of non-object-oriented languages—for example, C and Ada. But overloading is a powerful form of polymorphism, and furthermore, the interaction between overloading and other more object-oriented forms of

1. The term *polymorphism* is, for example, overloaded.

polymorphism, such as overriding, can be both powerful and unintuitive. In this chapter we explore some of the aspects of this idea.

15.1 ▫ Type Signatures and Scopes

A key idea necessary to understand overloading is the concept of a function (or method) *type signature*. A function type signature, you will recall, is a description of the argument types associated with a function, the order of arguments, and the return type. For methods defined within a class, it is normal to omit the type of the receiver from the type signature. This permits us to say that the type signature of a method in the parent class can be the same as the signature of a method in a child class. The following function, for example, has type signature *double* × *int* → *double*.

```
double power (double base, int exponent) {
  // raise base to exponent power
  double result = 1.0;
  for (int i = 0; i < exponent; i++)
    result = result * base;
  return result;
}
```

The second key idea needed in order to understand function overloading is the notion of a name *scope*. A scope defines the portion of a program in which a name can be used or the way in which the name can be used. Local variables, such as the identifiers result and i in the previous procedure, can only be used inside the function in which they are declared. After the final closing brace for the function, the name no longer has any meaning. Outside of a class definition, a data member name only has meaning if the name was declared as public inside the class and it is qualified by an instance of a class (or, for static features, qualified by the class name). For example, if a class defines a public data field named buffer, then outside the class definition an expression such as

```
anInstance.buffer
```

would be acceptable, but the simple name buffer would not. A class that is formed using inheritance creates a new name scope that extends the name scope of the parent. Thus, name scopes are formed in a variety of different ways in different programming languages.

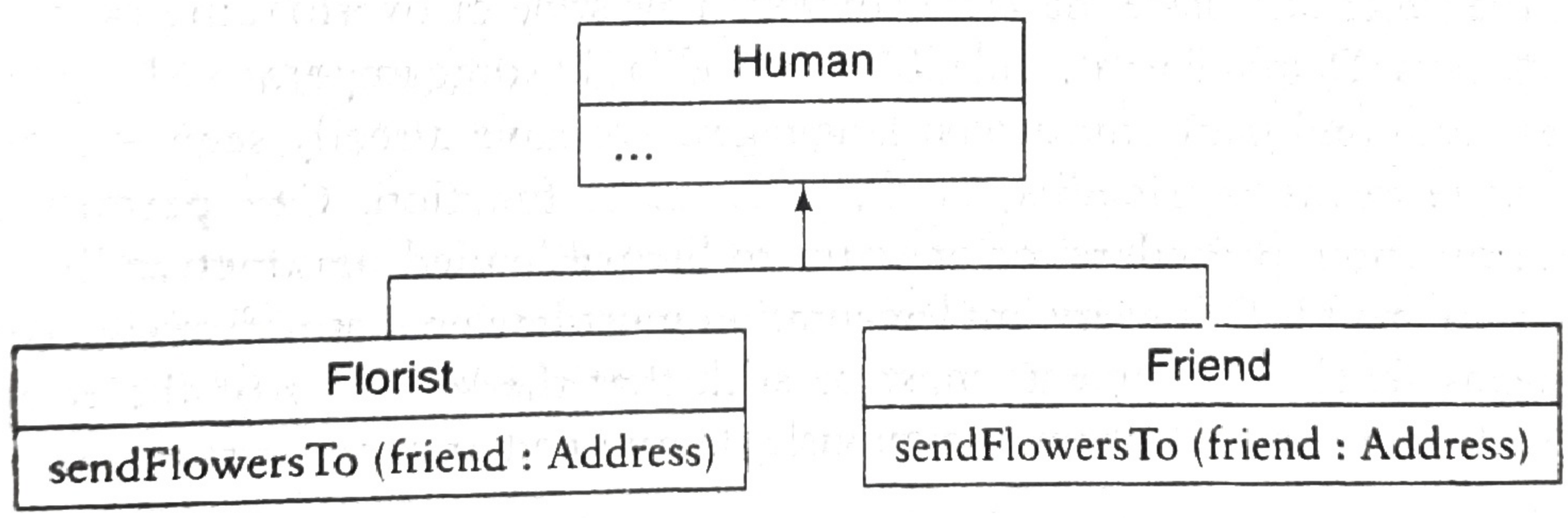
At any point in the textual representation of a program there can be multiple active scopes. A method declared inside a class will have both the class scope (all the class data fields) and a local scope (local variables declared inside the method). Inheritance, nested functions, and/

We can use these two concepts—type signatures and scope—to characterize two broad categories of overloading. Overloading can be based on methods having different scopes, regardless of their type signatures. For example, the same name might be used in two or more unrelated classes. Alternatively, overloading can be based on methods having different signatures, regardless of their scopes. For example, two or more functions defined in one class might have the same name but different arguments. The first form is examined in Section 15.2, and the second is described in Section 15.3.

Associated with the latter are a number of different mechanisms, each of which will be explored in a separate section. The process of coercion or conversion (Section 15.3.1) is used when the actual arguments of a method do not exactly match the formal parameter specifications, but can be converted into a form that will match. Redefinition (Section 15.4) occurs when a child class defines a method with the same name as a method in the parent class but with a *different* type signature. Polyadic functions (Section 15.5) are functions that can be invoked with varying number of parameters. Finally, in Section 15.6 we examine multi-methods, which combine features of both overloading and overriding.

15.2 ⊡ Overloading Based on Scopes

A useful consequence of the division of a program into independent name scopes is that the same name can appear in different scopes with no ambiguity or loss of precision. Two different functions can have local variables of the same name, and no confusion will arise because their scopes cannot overlap. Methods can also have the same name, without introducing confusion or ambiguity. We encountered an example of this in the flower story in Chapter 1. There were two possible ways for Chris to send flowers to Robin. One solution was for Chris to give the message sendFlowersTo to the local florist; another was for Chris to give the *same* message to his friend Elizabeth. Both Chris's florist and his friend (an instance of class Friend) would have understood the message, and both would have acted on it to produce a similar result.



Note carefully that there is no inheritance involved in this example. The first common ancestor superclass for Friend and Florist is the category Human. But certainly the behavior sendFlowersTo is not something that is associated with all humans.

There is nothing intrinsic to overloading that requires the functions associated with an overloaded name to have any semantic similarity. Consider a program that plays a card game, such as the solitaire game we examined in Chapter 9. The method draw was used to draw the image of a card on the screen. In another application we might also have included a draw method for the pack of cards—that is, to draw a single card from the top of the deck. This draw method is not even remotely similar in semantics to the draw method for the single card, and yet they share the same name. Nor is it required that they have the same type signature. In fact, the only similarity between the two methods is their name.

Overloading of a single name with independent and unrelated meanings should *not* necessarily be considered bad style, and generally it will not contribute to confusion. In fact, the selection of short, clear, and meaningful names such as add, draw, and so on, contributes to ease of understanding and correct use of object-oriented components. It is far simpler to remember that you can add an element to a set than to recall that to do so requires invoking the addNewElement method or, worse, that it requires calling the routine Set_Module_Addition_Method.

All the object-oriented languages we are considering permit the occurrence of methods with similar names in unrelated classes. In this case the resolution of overloaded names is determined by the class of the receiver for the message. Nevertheless, this does not mean that functions or methods can be written that take arbitrary arguments. The statically typed nature of C++ and Object Pascal still requires specific declarations of all variables.

15.3 ▫ Overloading Based on Type Signatures

Another style of overloading occurs when procedures (or functions or methods) are allowed to share a name and are disambiguated by the number, order, and, in statically typed languages, the type of arguments they require. This is allowed even if the functions share the same context. This style of overloading occurs in C++, C#, Java, Delphi Pascal, and CLOS, as well as in some imperative languages (such as Ada) and many functional languages. We have already seen examples of this style in the overloading of the constructor function. C++ permits any method, function, procedure, or operator to be overloaded parametrically. The only requirement is that every implementation must have a distinctive signature. This means that the arguments must be such that the selection of the routine intended by the user can be unambiguously determined at compile time.

```
class Example {
    // same name, three different methods
  int sum (int a) { return a; }
  int sum (int a, int b) { return a + b; }
  int sum (int a, int b, int c) { return a + b + c; }
}
```

Because a single method body must be selected based only on a specific function invocation, methods are not permitted to differ only in their return types nor in other features that are not observable by examining the invocation. (An example of a feature that cannot be determined from the invocation is the type of exceptions that might be thrown.)

In Delphi Pascal two methods in the same scope can overload a common name, but the programmer must explicitly declare the fact using the overload directive.

```
type
  Example = class
  public
    function sum (a : Integer) : Integer; overload;
    function sum (a, b : Integer) : Integer; overload;
  end;
```

It is important to note that the resolution of overloading—that is, the selection of a method body to match to a particular invocation—is performed at compile time, based on the static types of the argument values. Unlike overriding, no run-time mechanism is involved. Imagine, for example, there are two methods named Test, each taking a single argument. The first requires an instance of a parent class, and the second an instance of a child class that inherits from the parent.

```
class Parent { ... };

class Child : public Parent { ... };

void Test(Parent * p) { cout << "in parent" << endl; }
void Test(Child * c) { cout << "in child" << endl }
```

The following would, perhaps surprisingly, execute the first method and not the second.

```
Parent * value = new Child();
  // resolution based on static type,
  // not dynamic value in parent
Test(value);
```

Stream output in C++. The stream output facility in C++ is an excellent case study that illustrates how overloading can be used to solve the problem of extending a library to new user-defined data types. A problem with traditional output libraries is that they are designed to work only with the built-in types. Consider the printf facility in C. A printf statement is easy to write if the output consists of the basic data types provided as part of the language.

```
printf("average of %d and %d is %g", 3, 5, (3+5)/2.0);
```

But what happens when a programmer creates a new data type? Imagine, for example, that a programmer has defined a Fraction data abstraction, where fractions are represented by two integer data fields.

```
class Fraction {
public:
  Fraction (int top, int bottom) { t = top; b = bottom; }

  int numerator() { return t; }
  int denominator() { return b; }

private:
  int t, b;
};
```

The only way to print a fractional value is to break it into the two parts and print them each independently.

```
Fraction f(3,4);
printf("The value of f is %d/%d", f.numerator(), f.denominator());
```

The designers of C++ realized that an alternative could be found by using the ability to overload operators with several different meanings. They selected the left shift operator, <<, as the output operator. An abstraction termed an *output stream* (or ostream) represents an entity that will accept a sequence of characters. Writing to a stream thus generalizes such activities as writing to a file, a window, or a network. The stream output library provides a collection of overloaded definitions for the left shift operator.

```
ostream & operator << (ostream & destination, int source);
ostream & operator << (ostream & destination, short source);
ostream & operator << (ostream & destination, long source);
ostream & operator << (ostream & destination, char source);
ostream & operator << (ostream & destination, char *  source);
// ⋮  and so on
```

Each operator takes a stream as the left argument and another value as the right argument. When executed, each operator will, as a side effect, write the right argument to the stream, then return the stream value as the result. The latter action allows a complex stream expression to be built out of parts. For example, consider the evaluation of the statement,

```
double d = 3.14;
cout << "The answer is " << d << '\n';
```

The output statement would be evaluated in stages, as follows.

```
cout << "The answer is "  << d   << '\n'
        cout             << d   << '\n'
                cout            << '\n'
```

First, the operator overloaded to work with a stream and a character array would be executed. After printing the string, this operator returns the stream. Next, the operator overloaded to work with a stream and a double precision value would be executed. Once more, after printing the number, the stream is returned. Finally, the operator overloaded to work with a stream and a single character would be executed. The character would be printed, and the stream value returned. The final value is then discarded, since the purpose of printing has been achieved as a side effect.

Using overloaded operators makes it simple to extend the output library to new user-defined data types. All that is needed to add the ability to output fractional values, for example, is to define yet another overloaded meaning for the output stream operator.

```
ostream & operator << (ostream & destination, Fraction & source)
{
  destination << source.numerator() << "/" << source.denominator();
  return destination;
}
```

With this new ability the output of fractions can be intermixed with the output of other data types.

```
Fraction f(3, 4);
cout << "The value of f is " << f << '\n';
```

15.3.1 Coercion and conversion

Overloading based on type signatures occurs frequently in all programming languages, not simply object-oriented languages. Perhaps the most common example is the overloading of the addition operator, +. The code generated by a compiler

Coercion, conversion, and casts

The meanings of the terms *coercion* and *conversion* are easily confused, since both represent a change in type. A *coercion* is an *implicit* change in type, one that occurs without overt reference in the program. The canonical example of this is the addition of two variables, one declared as real and one declared as integer. The value of the integer variable will be converted to real before the addition.

```
double x = 2.8;
int i = 3;
x = i + x; // integer i will be converted to real
```

In contrast, the term *conversion* usually represents a change in type that is explicitly requested by the programmer. In many languages the operator used to perform this change is termed a *cast*. A cast in C++ or Java would be written as follows.

```
x = ( (double) i ) + x;
```

Parentheses surround the expected type following the conversion, in this case double.

Casts, and conversions in general, can either change the underlying representation (such as when an integer is converted into a real) or simply change the type without changing the representation (such as when a pointer to a child class is changed into a pointer to the parent class).

for an integer addition is often radically different from the code generated for a floating-point addition; yet programmers tend to think of the operations as a single entity, the "addition" function.

In this example it is important to point out that overloading may not be the only activity taking place. A semantically separate operation, *coercion*, is also usually associated with arithmetic operations. It occurs when a value of one type is converted into one of a different type. If mixed-type arithmetic is permitted, the addition of two values may be interpreted in at least three different ways.

- There may be four different functions, corresponding to integer + integer, integer + real, real + integer, and real + real. In this case, there is overloading but no coercion.
- There may be two different functions for integer + integer and real + real. In integer + real and real + integer, the integer value is coerced by being changed

into a real value. In this situation there is a combination of overloading and coercion.

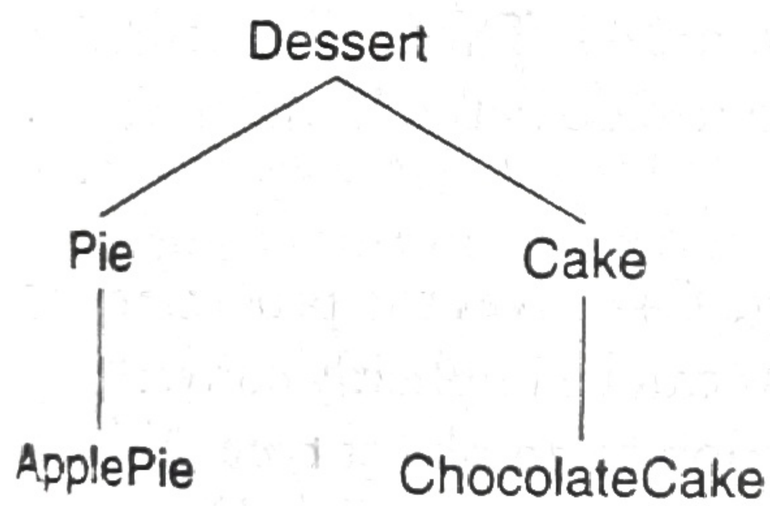
- There may be only one function for real + real addition. All arguments are coerced into being real. In this case there is coercion only, with no overloading.

Substitution as conversion. If we interpret the word *coercion* broadly as "a change in type," then the principle of substitution introduces a form of coercion not found in conventional languages. This occurs when a value from a child class is used as an actual parameter in a method that is defining the corresponding formal parameter with the parent class type. A similar change in type can occur with an interface and an instance of a class that implements the interface.

The following example from Java illustrates some of the subtle interactions between overloading and inheritance. In Java, if two or more methods have the same name and the same number of parameters, the compiler uses the following algorithm to determine the match.

1. Find all the methods that could possibly apply to the invocation—that is, all methods for which the arguments could legally be assigned to the parameter types. If one method exactly matches the argument types used in the invocation, then execute that method.
2. If any method in the set identified in Step 1 has parameter types that are *all* assignments to any other method in the set, then remove the second method from the set. Repeat until no further eliminations can be performed.
3. If exactly one method remains, that method is the most specific and will be invoked. If more than one method remains, the invocation is ambiguous, and a compiler error will be reported.

To illustrate this process, imagine we have five classes structured in the following inheritance hierarchy.



Now suppose some scope includes the following three overloaded methods, each taking a pair of Dessert parameters.

```
void order (Dessert d, Cake c);
void order (Pie p, Dessert d);
void order (ApplePie a, Cake c);
```

Imagine that we have a set of variables, each representative of a different class. Variable aCake holds a value of type Cake, anApplePie a value of type ApplePie, and so on. Finally, consider a series of invocations, both legal and illegal.

```
order (aDessert, aCake); // example 1, legal
order (anApplePie, aDessert); // example 2, legal
order (aDessert, aDessert); // example 3, illegal
order (anApplePie, aChocolateCake); // example 4, legal
order (aPie, aCake); // example 5, illegal
```

The first invocation is legal because Dessert and Cake match the type signature of the first method directly. The second invocation similarly matches the second overloaded method, since it is the only method for which the arguments could legally be assigned to the parameters. (Assigning the Dessert argument to the Cake parameter is illegal in both the first and third methods.)

The third example is illegal. After Step 1 the set of possible candidate methods is empty, since a Dessert value cannot be arbitrarily downcast to a more specific type. Therefore, a compiler error will be reported.

The forth invocation is more subtle. Initially all three methods are candidates for execution. So the method selection algorithm moves to Step 2. In this step the first overloaded method would be eliminated, since ApplePie is a more specific class than Dessert (an ApplePie can be assigned to a Dessert, but not vice-versa), and the second argument is the same. The second method would be removed for the same reason (both arguments are less specific; Pie is less specific than ApplePie and Dessert less specific than Cake). This leaves one candidate, and the fourth example will execute the third method.

Like the third example, the final invocation is also invalid. After Step 1 only the last method has been eliminated, leaving two elements in the set of possible candidates. But the first method cannot be subsumed by the second, since a Dessert cannot be assigned to a Pie. But neither can the second method be subsumed by the first, since a Dessert cannot be converted into a Cake. Thus, two methods remain after Step 2, and with the compiler unable to decide between them, an error will be reported.

Conversion operators in C++. The language C++ gives the programmer a great deal of control over how an instance of a class can be implicitly converted from or to another type. We consider first the conversion to an object type. A constructor that takes a single argument will be interpreted by the compiler as a rule for converting from the argument type to the class type. We could incoporate this into the Fraction data type discussed earlier by, for example, defining a constructor that takes a single integer argument.

```
class Fraction {
public:
  Fraction (int top, int bottom) { t = top; b = bottom; }
```

```
Fraction (int top) { t = top; b = 1; }

int numerator() { return t; }
int denominator() { return b; }

Fraction operator * (Fraction & right)
{
  Fraction result (numerator() * right.numerator(),
    denominator() * right.denominator());
  return result;
}


private:
  int t, b;
};
```

We have also given the implementation of the operator that will handle the multiplication of two fractional values. In an expression that involves the multiplication of a fraction and an integer, a temporary value will be formed that represents the conversion of the integer into a fraction. The constructor will be used to initialize this temporary object.

```
Fraction a(2, 3);

a = a * 3; // right argument will become 3/1
```

In the other direction is the conversion of an object value into a different type. Here C++ has an interesting feature that gives the programmer the ability to explicitly state how the conversion should be performed. A type name can be used as an operator. As with all operators in C++, the programmer can provide a new meaning by defining the operator as a method. For instance, suppose the programmer is continuing to develop the Fraction data type. The following class definition shows a new operator being defined. The "operator" double is invoked when a Fraction must be converted into a double precision value.

```
class Fraction {
public:
  Fraction (int top, int bottom) { t = top; b = bottom; }

  Fraction (int top) { t = top; b = 1; }

  int numerator() { return t; }
  int denominator() { return b; }

  Fraction operator * (Fraction & right)
```

```
    {
      Fraction result (numerator() * right.numerator(),
        denominator() * right.denominator());
      return result;
    }

    operator double ()
    {
      // cast is necessary to make double precision division
      return numerator() / (double) denominator();
    }

  private:
    int t, b;
  };
```

The following example illustrates the use of the conversion operator. Two declaration statements are used to construct two fractional values. In the third statement a fraction is formed using the multiplication operator defined in the class. In the fourth and final statement a multiplication is performed using a fraction and a double precision number. Since the fraction does not define a method for this situation, the fraction is converted into a double, using the conversion method supplied by the class, and the normal double precision multiplication is performed.

```
Fraction one(2,3);
Fraction two(3,4);
Fraction three = one * two; // multiply two fractions
double four = one * 3.14159; // multiply fraction and double
```

Needless to say, when implicit conversion, explicit conversion, and the substitution of object-oriented values are all combined in one statement, the algorithm used to resolve an overloaded function name can become quite complex. Typically it must include at least the following steps.

1. If there is a method with an exact match for the actual arguments, then call that version.
2. Otherwise, see if there is a match that uses standard type promotions (such as converting a short integer to a standard integer).
3. Otherwise, see if there is a match that uses standard conversions (such as interpreting a child type as an instance of the parent type).
4. Otherwise, see if a match can be found using user-supplied conversions.

5. Otherwise, if no match can be found or if more than one method seems to match, then issue a compile time error.

15.4 ⊡ Redefinition

A *redefinition* occurs when a child class defines a method using the same name as a method in the parent class but with a different type signature. The change in type signature is what differentiates redefinition from simple overriding. Redefinition would not be notable, except that languages use two different techniques to resolve a redefined name, and thus the unwary programmer can easily be surprised if they are not cognizant of the differences. These two models might be termed the *merge* model of redefinition and the *hierarchical* model.

The programming language Java uses the merge model. Here the various different meanings found in all currently active scopes are merged together to form a single collection. To match an invocation that uses an overloaded method name to a specific method body, all the possible alternatives are examined, and the closest match is the one selected.

The programming language C++ uses a different approach. In this language each surrounding scope is maintained on its own separate list. To match a name to a method, each scope will be examined in turn—hence the designation *hierarchical*. However, when a scope is found in which the name is defined, the closest match *in that scope* will be the one selected.

An example will help illustrate this distinction. Suppose a parent class defines a method with one integer argument, and the child class redefines the method using the same name but this time with two integer arguments.

```
class Parent {
  public void example (int a)
    { System.out.println("in parent method"); }
}

class Child extends Parent {
  public void example (int a, int b)
    { System.out.println("in child method"); }
}
```

Now imagine we create an instance of the child and try to execute the method using only one argument.

```
Child aChild = new Child();
aChild.example(3);
```

In the programming language Java, the method from the parent class will be selected as the best fit. The same is true in C#. The equivalent program in

C++, on the other hand, will produce a compilation error. This is because the scope corresponding to the child class is examined first, before the scope for the parent class. A method with the given name is found in the child scope, but upon examination the arguments for this method do not match those of the invocation. The C++ programmer can achieve the same effect as the Java model but only by redefining both methods in the child class.

```
class Child : public Parent {
  public void example (int a) { Parent::example(a); }
  public void example (int a, int b)
    { cout << "in child method"; }
};
```

The Delphi Pascal language offers the programmer control over which model to use. If a child class declares a method and uses the overload modifier, then the definitions in the child class will be merged with those inherited from the parent class. If the modifier is omitted, then the child class definition replaces that of the parent class. The following example illustrates this behavior. One child class redefines the method as an overloading, and the second does not. When an attempt is made to invoke the method from the parent, one class will produce a compile time error, but the second will not.

```
type
  Parent = class
  public
        procedure Example(A: Integer);
  end;

  ChildWithOneMethod = class (Parent)
  public
    procedure Example(A, B: Integer);
  end;

  ChildWithTwoMethods = class (Parent)
  public
    procedure Example(A, B: Integer); overload;
  end;

var
  C1: ChildWithOneMethod;
  C2: ChildWithTwoMethods;
begin
  C1 := ChildWithOneMethod.Create;
```

```
    C1.Example(42); // error: not enough parameters
    C2.Example(42); // okay
end
```

*15.5 ⊡ Polyadicity

A *polyadic* function is one that can take a variable number of arguments. In CLOS, for example, the addition operator is polyadic, so each of the following will evaluate to an integer value.

```
(+ 2 3)
(+ 2 3 4 5 6 7)
(+ 2)
```

Polyadic functions are found in a large number of different languages. The built-in function writeln in Pascal or the library function printf in C and C++ are examples.

```
printf("2 + 5 is %d", (2+5));
printf("the average of %d and %d is %g", 3, 7, (3+7)/2.0);
```

While it is easy to write an invocation of a polyadic function, it is much more difficult to write the body of the function itself. For example, how do you name an indefinite number of arguments? The language C++ uses a somewhat *ad hoc* device consisting of an ellipsis, a data type named va_list (for *variable argument* list), and a trio of methods va_start, va_arg, and va_end. The following example illustrates their use, defining a function that will return the sum of a list of integer values, where the first argument must represent the number of integers in the list.

```
# include <stdarg.h>
int sum (int argcnt, ...)
{
  va_list ap;
  int answer = 0;
  va_start(ap, argcnt);
  while (argcnt > 0) {
    answer += va_arg(ap, int);
    argcnt--;
  }
  va_end(ap);
  return answer;
}
```

The language C# provides a more elegant solution to this problem. The keyword **params** can be used in conjunction with an array argument. If no other matching method can be found, as a final alternative the compiler will check to see if all remaining values can be assigned as array elements. If so, then a temporary internal array is created and passed as parameter to the method.

```
class ParamsExample {
  public void Write (int x) {
    // use this with one argument
    WriteString("Example one ");
    WriteString(x.ToString());
  }

  public void Write (double x, int y) {
    // use this with two arguments
    WriteString("Example two ");
    WriteString(x.ToString());
    WriteString(y.ToString());
  }

  public void Write (params object [ ] args) {
    // use this with any other combination of arguments
    WriteString("Example three ");
    for (int i = 0; i < args.GetLength(0); i++)
      WriteString(args[i].ToString());
  }
}
```

Here are some example invocations and their associated results:

```
  ParamsExample p;
  p.Write(42);
Example one 42
  p.Write(3.14,159);
Example two 3.14159
  p.Write(1,2,3,4);
Example three 1234
  p.Write(3.14);
Example three 3.14
  p.Write(3,"abc");
Example three 3abc
```

15.5.1 Optional parameters

A slightly easier to understand technique for writing polyadic functions is found in C++ and also in Delphi Pascal. This is the concept of an optional or default parameter. The basic idea is to provide an implicit value for one or more parameters. If the invoking expression explicitly provides a value for these parameters, then the default values are ignored; otherwise, if the invoking expression omits them, the default values will be used. Optional parameters must appear at the end of the argument list.

The following simple program illustrates this technique being used in a Delphi Pascal procedure.

```
function Count (A, B : Integer; C : Integer = 0; D : Integer = 0);
begin
    (* Result is a pseudo-variable used *)
    (* to represent result of any function *)
  Result := A + B + C + D;
end

begin
  Writeln (Count(2, 3, 4, 5)); // can use four arguments
  Writeln (Count(2, 3, 4)); // or three
  Writeln (Count(2, 3)); // or two
end
```

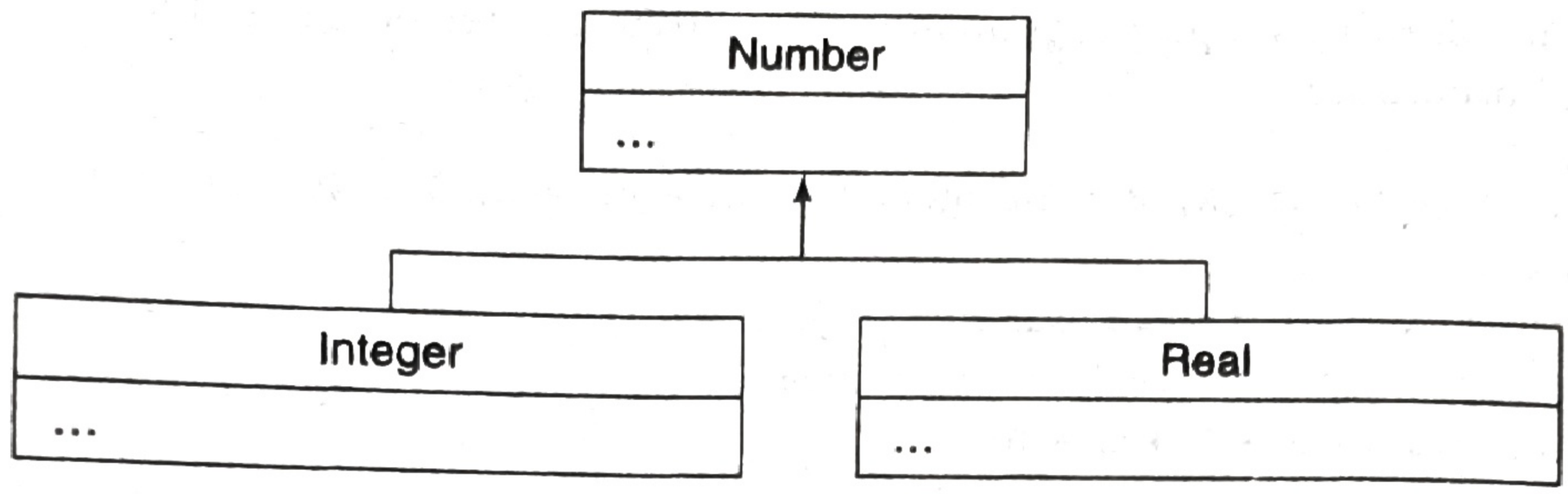
C++ and Ada have similar facilities. Judicious use of optional parameters can often eliminate the need to write multiple overloaded methods. This is particularly useful with constructors. When optional parameters are used with a function that is itself overloaded, the type signatures of all functions must be such that a unique function body can be determined once the argument values used by a call are known. (Note that a function that uses optional parameters will have several different type signatures.)

*15.6 ▣ Multi-Methods

Both overloading and overriding match an invocation by selecting one method out of many. But as we noted at the beginning of the chapter, overloading is performed at compile time, whereas overriding is performed at run time. Overriding, you will recall, uses the dynamic (or run-time) type of the receiver to determine which method should be executed. The types of any other arguments passed along with a message generally play no part in the overriding selection mechanism.

Many people have wondered whether it would be possible to combine these two techniques, allowing an overloaded function to be selected based on the types of all arguments and not just the receiver. The situations where this would be useful are common. Unfortunately, the problems that such a mechanism would introduce, particularly for statically typed languages, are difficult to solve. In this section we will illustrate these problems by means of a simple example.

Imagine that we have a statically typed object-oriented language in which Integers and Real numbers are both child classes of a more general class Number.



To implement addition as a function it is necessary to overload the add function with four different definitions.

```
function add (Integer a, Integer b) : Integer { ... }
function add (Integer a, Real b) : Real { ... }
function add (Real a, Integer b) : Real { ... }
function add (Real a, Real b) : Real { ... }
```

Notice that the return type for the first definition is different from the return type for the remaining three.

The basic idea of a multi-method is that any argument, and not just the receiver, should be allowed to be polymorphic. Furthermore, it should be the run-time value and not the static compile-time value of this variable that determines which function to execute. For example, consider the following execution sequence.

```
Number x = ...; // variable x and y are given
Number y = ...; // some unknown values
Real r = 2.134;

Real r2 = add(r, x); // which method to execute?
Real r3 = add(x, y); // is the assignment type-safe?
```

Variables x and y are numbers. By the principle of substitution, they could be assigned values that are either Integer or Real. Variable r, on the other hand, is known to be an instance of class Real. Now consider the first execution of the method add. We know the type of the first argument, but that still leaves two potential candidates for execution, depending on whether the variable x is integer

or real. Which one should be selected? A run-time test must be performed to determine the most appropriate function.

Now consider the second invocation. This one is much more difficult to handle. Here, any of the four candidate function bodies can be selected, based on the run-time types of the variables x and y. But only three of these return a value that can be assigned to a variable of type Real. If x and y are both integer, then the assignment is not type-correct, and an error should be reported. So it is necessary to perform a run-time test on both the arguments (in order to select the correct function to execute) and the result (in order to ensure type safety in the assignment).

So arguments against multi-methods are based on two complaints. The first is that the execution time cost of the function selection algorithm is too high, and second, in a statically typed language the typing issues introduce too many complications. It should be noted that the only widely used language to incorporate multi-methods is CLOS, a dynamically typed language in which the second issue is not important.

Double dispatch. One common solution to the problem just described is a technique termed *double-dispatch*. The key insight behind the double-dispatch approach is that a message can be used to determine the type of the receiver, since each object knows its own type. Determining the type of two independent values can therefore be accomplished by using each value as receiver in turn.

Imagine, for example, that the classes Integer and Real are written in Smalltalk, and each implements three methods. In Integer we have the following:

```
" class Integer "
{add:} arg
  ↑ arg addToInteger: self

{addToInteger:} anInteger
  ... " whatever is necessary to add two integers "

{addToReal:} aReal
  ... " whatever is necessary to add an integer and a real "
```

In class Real the methods are similar, except that the add: method turns around and passes the message addToReal: to its argument. Now imagine we are adding two variables, anInteger and aReal, with the types indicated by their name. The message add: given to anInteger will take its argument (which at this point is of indeterminent type) and pass it the message addToInteger. The class Real, receiving this message, knows that it is holding a real value and that the argument is integer, and it can act accordingly.

15.6.1 Overloading Based on Values

There is another curious language feature that, like multi-methods, appears to be found only in Lisp-based languages (such as CLOS or Dylan). This is the ability to overload a function based on argument values and not just types. A parameter specifies that its argument must be exactly the value specified in the function definition. For example, we could see the following two function definitions.

```
function sum (a : integer, b : integer) { return a + b; }

function sum (a : integer = 0, b : integer) { return a; }
```

The second function will only be invoked if the first argument is the constant value zero; otherwise, the more general function will be executed. Value constraints on parameters can be a very useful programming language feature. However, like multi-methods, they increase significantly the run-time execution overhead involved in method selection.

Summary ▫

Overloading is the compile time matching of a function invocation to one of many similarly named methods. There are two major categories of overloading. Overloading can be based on different scopes, where the class of the receiver determines the function to execute. Alternatively, overloading can be based on type signatures, where the type associated with the parameters that accompany the invocation is used to determine the correct method. Overloading should not be confused with the similar concepts of conversion or redefinition. An alternative to overloading is the creation of polyadic functions, functions that can take a variable number of arguments.

Further Information ▫

More detailed information regarding the conversion rules used by C++ can be found in [Ellis 1990] and [Stroustrup 1986].

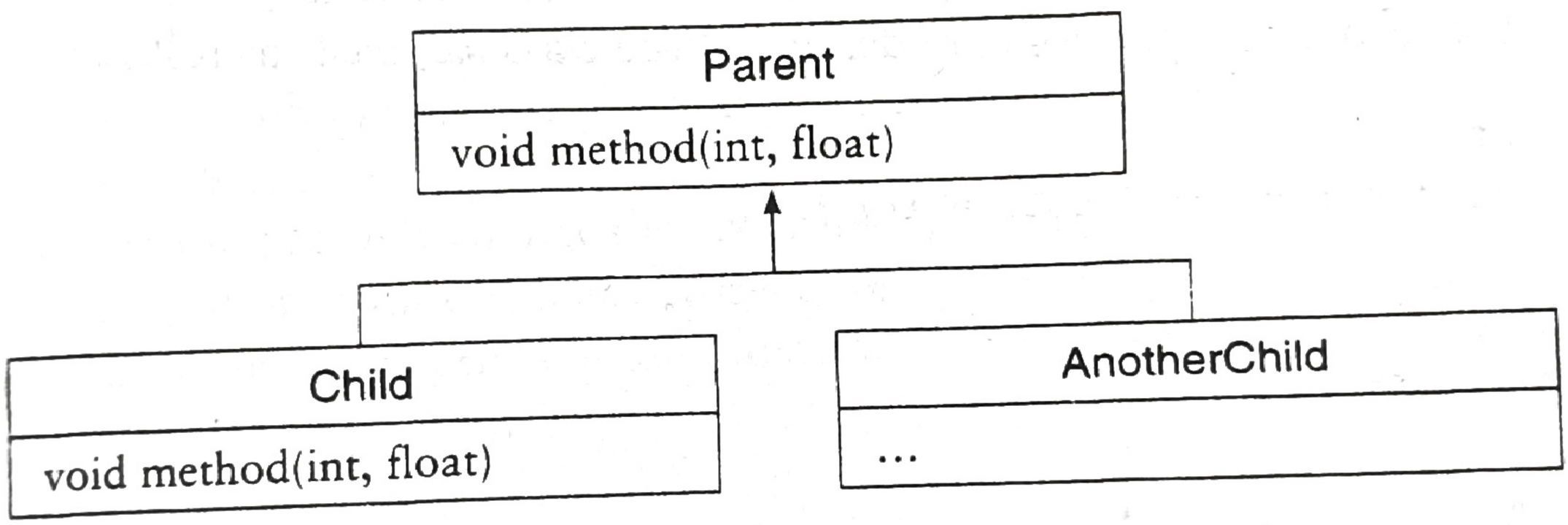
Craig [Craig 2000] provides an analysis of polyadicity in various languages.

The designer of C++ has discussed his reasons for rejecting multi-methods in [Stroustrup 1994]. Castagna [Castagna 1997] describes an algorithm that could be used in the implementation of multi-methods for a dynamically typed language.

Double dispatching was first described in Ingalls [Ingalls 1986], who called the technique multiple polymorphism. Later authors have preferred the term *double*

Overriding

We say that a method in a child class *overrides* a method in the parent class if the two methods have the same name and type signature.[1]



Overriding (also called *inclusion polymorphism*) is important when it is combined with substitution. Recall that a polymorphic variable can, using the principle of substitution, be declared as one type (for example, the parent type) but in reality hold a value of another type (typically a child type). When a message that corresponds to an overridden method is passed to such a value, the procedure that will be executed is that given by the child class and not by the parent.

In a certain sense, method overriding might be considered to be a special case of overloading, since as with overloading we again have one method name and two or more method bodies. But having made that observation we reach the end

1. As we will note in a moment, some languages require an additional keyword before overriding will take place. The feature common to all languages, however, is the parent/child relationship and the matching type signatures.

of the similarities. On the other hand, the differences between overloading and overriding include the following.

- In method overriding the classes in which the methods appear must stand in a parent/child inheritance relationship to each other, something that was not a requirement for simple overloading.
- For overriding to occur, the method type signatures must match.
- Overloaded methods are always separate, but when overriding, two methods are sometimes combined to perform the actions of both together.
- And finally, and most importantly, while overloading is typically resolved at compile time, overriding is a run-time mechanism. This means it may not be possible to predict what actions it will perform in response to any specific message until a program is actually executing.

Overriding in Smalltalk class Magnitude. An interesting example of overriding occurs in the class Magnitude in the Little Smalltalk system. Magnitude is an abstract superclass dealing with quantities that possess at least a partial, if not a total, ordering. Numbers are perhaps the most common example of objects that have magnitude, although time and date can also be ordered, as can characters, points in a two-dimensional coordinate plane, and words in a dictionary.

The six relational operators are defined in the class Magnitude as follows.

```
<= arg
  ↑ self < arg or: [ self = arg ]

>= arg
  ↑ arg <= self

< arg
  ↑ self <= arg and: [ self ~= arg ]

> arg
  ↑ arg < self

= arg
  ↑ self == arg

~= arg
  ↑ (self = arg) not
```

Note that the definitions appear to be circular, each one depending on some number of the others. How, then, is an infinite loop to be avoided if any of them are invoked? The answer is that subclasses of class Magnitude must override and

redefine at least one of the six relational messages. We leave it as an exercise for the reader to show that if the message = and either < or <= are redefined, all the remaining operators can be executed without falling into an infinite loop.

Overriding of a method contributes to code sharing, insofar as instances of the classes that do *not* override the method can all share one copy of the original. It is only in situations where this method is not appropriate that an alternative code fragment is provided. Without overriding, it would be necessary for all subclasses to provide their own method to respond to the message, even though many of these methods are identical.

16.1 ▫ Notating Overriding

Languages differ in how they require the possibility of overriding to be documented in the code itself. Some languages, such as Smalltalk, Java, or Objective-C, say nothing at all. In these languages it is only the similarity of type signatures between the parent and child classes that indicates overriding. In other languages, such as C++, a notation must be placed in the parent class to indicate that overriding can potentially take place (although the mark does not guarantee that overriding *must* take place). In other languages, such as Object Pascal, the documentation is placed in the child class. In Delphi Pascal and in C# a keyword is required in both parent and child. These differences are described in more detail in Figure 16.1.

Overriding cannot change the accessibility of a method. If the method is public in the parent, it cannot be made private in the child or vice-versa. (An exception to this rule is the use of private inheritance in C++, where public features in a parent become private in the child.)

C++ Keyword required in parent class.

```
class Parent {
public:
    virtual int example(int a) { ... }
};

class Child : public Parent {
public:
    int example(int a) { ... }
};
```

▫ Figure 16.1 —— Different ways to describe overriding

C# Keyword required in both parent and child classes.

```
class Parent {
   public virtual int example (int a) { ... }
}

class Child : Parent {
   public override int example (int a) { ... }
}
```

Delphi Pascal

```
type
  Parent = class (TObject)
     function example (int) : integer; virtual;
  end;

  Child = class (Parent)
     function example (int) : integer; override;
  end;
```

Java No keyword in either parent or child. Smalltalk, Objective-C are similar.

```
class Parent {
    public void example(int a) { ... }
}

class Child extends Parent {
    public void example(int a) { ... }
}
```

Object Pascal Keyword in child class, not in parent class.

```
type
  Parent = object
     function example (int) : integer;
  end;

  Child = object (Parent)
     function example (int) : integer; override;
  end;
```

■ Figure 16.1 — Continued

16.2 ▫ Replacement versus Refinement

While we have talked about a child class overriding the method in a parent class, in actual fact there are two different ways to interpret the process of overriding:

Replacement A method *replacement* totally overwrites the method in the parent class during execution. That is, the code in the parent class is never executed when instances of the child class are manipulated.

Refinement A method *refinement* includes, as part of its behavior, the execution of the method inherited from the parent class. Thus, the behavior of the parent is preserved and augmented.

The first interpretation of the meaning of overriding is often called *American semantics* because it is usually associated with languages of American origin (such as Smalltalk and C++). The second is known as *Scandinavian semantics* because it is most frequently associated with Simula, the original object-oriented language, and with the later language Beta, both of Scandinavian origin.

In reality, both forms of overriding are useful, and often both can be found in the same programming language. For example, almost all languages (even those solidly in the American school) use refinement semantics for constructors. That is, a constructor for a child class will *always* invoke the constructor for the parent class, thereby insuring that the data fields for both the parent and child will be properly set.

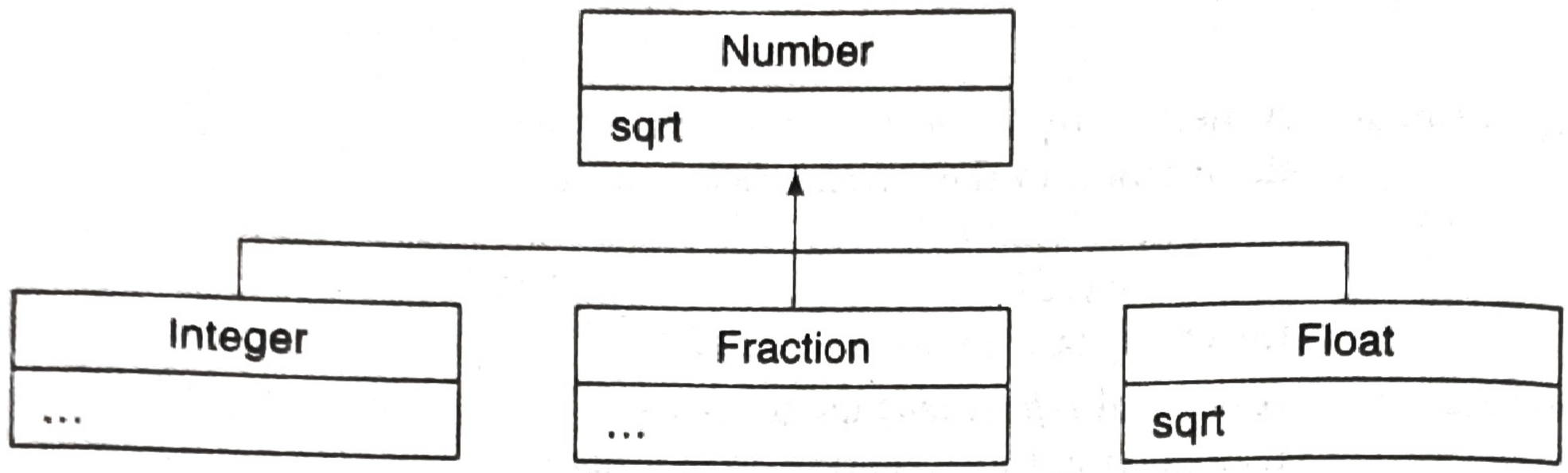
We will give two examples to illustrate the concepts of replacement and refinement. The replacement example uses Smalltalk, whereas the refinement example will be based on Beta.

16.2.1 Replacement in Smalltalk

Replacement is used in many different places in the standard Smalltalk class library. The two major reasons for using replacement are in support of code reuse and as a technique for optimization. We will illustrate both.

First consider code reuse. In Smalltalk, integers and floating-point numbers are objects; they are instances of class Integer and class Float, respectively. In turn,

both of these classes are subclasses of a more general class, Number. There are several other types of objects that are also numbers, such as class Fraction.



Now suppose we have a variable, aNumber, that currently contains a Smalltalk integer, and we send to aNumber the square-root-generating message sqrt. There is no method corresponding to this name in class Integer, so class Number is searched, and the following method is discovered.

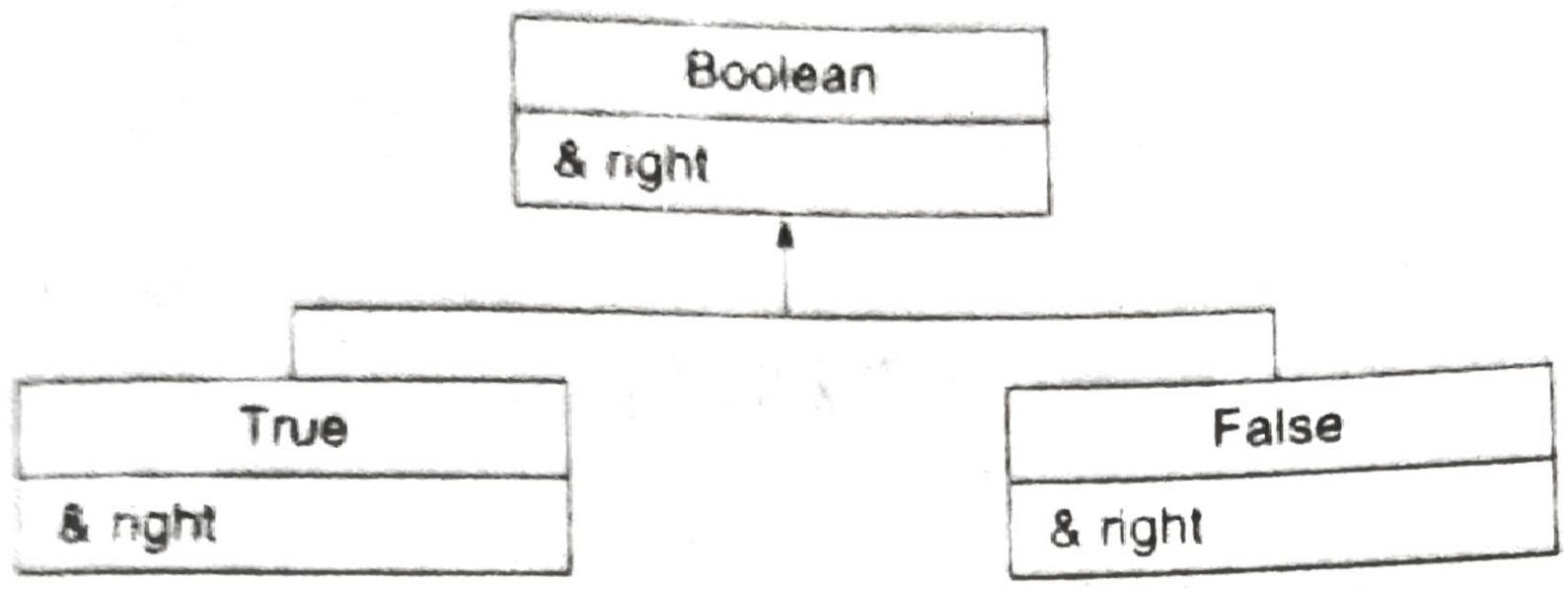
```
"class Number"
  {sqrt}
    " convert to float then compute square root "
    ↑ self asFloat sqrt
```

This method passes the message asFloat to self, which, as you will recall from Chapter 5, represents the receiver for the sqrt message. The asFloat message results in a floating-point value with the same magnitude as that of the integer number. The message sqrt is then passed to this value.

This time, the search for a method begins with class Float. It so happens that class Float contains a different method named sqrt, which for floating-point values *overrides* the method in class Number. That method (which is not shown here) computes and returns the expected floating-point value.

The ability to override and totally *replace* the method sqrt means that many kinds of numbers can share the single default routine found in class Number. This sharing avoids the need to repeat this code for each of the different subclasses of Number (which includes not only integers and floats but infinite-precision integers and fractions). Classes, such as Float, that require a behavior different from the default can simply override the method and substitute the alternative code.

Another Smalltalk example illustrates using overriding as a means of optimization. The Boolean constants true and false are instances of the classes True and False, respectively. These, in turn, are both subclasses of Boolean.



The methods & (logical *and*) and | (logical *or*) are implemented in the parent class with a general mechanism that defines the Booleans using conditional statements. This technique is correct for any combination of values.

```
"class Boolean"
  {&} right
    self ifTrue: [ right ifTrue: [ ↑ true ] ].
    ↑ false

  {|} right
    self ifTrue: [ ↑ true ].
    right ifTrue: [ ↑ true ].
    ↑ false
```

Of course, in the child classes the implementation can take advantage of additional information. That is, we know the truth or falsity of the left argument; we can therefore use this information to define much more efficient implementations of these operations. For example, here is the code for the logical operators in class True.

```
"class True"
  {&} right
    ↑ right

  {|} right
    ↑ true
```

In the case of the logical *and* operator, the entire condition depends only on the right argument. In the case of the logical *or* operator, there is no need to even consider the right argument, since a true value *or* anything else is also true. Here are the definitions of the same operations in class False.

```
"class False"
  {&} right
    ↑ false

  {|} right
    ↑ right
```

In both the classes True and False, the child classes have an implementation that is much more efficient than is possible in the parent class.

16.2.2 Refinement in Beta

A refinement occurs when the code in the child class is combined with the code in the parent class, and both are ultimately executed. As we have previously noted, in almost all object-oriented languages that support the mechanism, constructors perform a refinement even if other methods use replacement semantics. This is because it is important that any initialization required by the parent class be included as part of the process of initializing an instance of the child class.

Refinement can be simulated in those languages that use replacement semantics by having the child class explicitly invoke the method in the parent. Various different notations are used for this purpose, as illustrated in Figure 16.2.

Refinement semantics occur in the earliest object-oriented language, Simula, which was developed in the early 1960s. This idea was carried forward into the later language Beta, also of Scandinavian origin. In both these languages, code from the parent class is always executed *first*, before any code from the child class. If the code in the parent class includes the special statement

```
inner;
```

then at that point in execution the code from the child class will be executed.

We will illustrate how this mechanism works with an example modeled on Beta. The syntax we will use is different from Beta in order to more clearly point out the role of refinement. However the basic underlying principles are the same. Imagine a class includes a method designed to print out a World Wide Web anchor.

```
class Anchor {
  public void printAnchor () {
    print('<A href="http:');
    inner;
    print('">');
  }
}
```

C++ Qualified name formed using class name, two colons, then method name.

```
void Parent::example (int a) {
    cout << "in parent code\n";
}

void Child::example (int a) {
    Parent::example(12); // do parent action
    cout << "in child code\n"; // then child action
}
```

Java Pseudo-variable super refers to parent class. Smalltalk and Objective-C use a similar idea. C# uses keyword base.

```
class Parent {
   void example (int a) {
       System.out.println("in parent");
   }
}

class Child {
   void example (int a) {
       super.example(12);
       System.out.println("in child");
   }
}
```

Object Pascal and Delphi Pascal Keyword inherited invokes method in parent class.

```
procedure Parent.example (int a);
begin
   writeln("in parent code");
end;

procedure Child.example (int a);
begin
   inherited example(12); (* do parent action *)
   writeln("in child code"); (* then child action *)
end;
```

Figure 16.2 — Simulation of refinement using replacement

The inner keyword does not indicate that a subclass *must* be formed. If no subclass is present, the inner statement performs no operation.

```
Anchor anAnchor = new Anchor();
anAnchor.printAnchor();
<A href=http:"">
```

Note carefully that some operations have occurred before the call on inner, and some have occurred after the call.

Now imagine that we create a subclass—say, a class that will print an anchor for Web pages on a specific machine.

```
class OSUAnchor extends Anchor {
  public void printAnchor () {
    print('//www.cs.orst.edu/');
    inner;
  }
}
```

If we now create an instance of the child class and execute the method, the code for the parent class will be executed first, and only when the inner statement is encountered will the code from the child class be executed.

```
Anchor anAnchor = new OSUAnchor();
anAchor.printAnchor();
<A href=http:"//www.cs.orst.edu/">
```

We can continue this process indefinitely. Imagine we create a third class that will print pages in an individual's personal Web directory.

```
class BuddAnchor extends OSUAnchor {
  public void printAnchor () {
    print('~budd/');
    inner;
  }
}
```

Creating an instance of this class and executing the method will then have the effect of first invoking the code from the "grandparent" class Anchor, then from the parent class OSUAnchor, and only last from the class BuddAnchor.

```
Anchor anAnchor = new BuddAnchor();
anAnchor.printAnchor();
<A href=http:"//www.cs.orst.edu/~budd/">
```

Note that in Beta it is the parent that is first given control, and behavior in the parent can wrap around the code provided by the child. When simulating

refinement in American languages (as shown in Figure 16.2), it is the child's code that is executed first, and this can wrap around the code in the parent. It is surprisingly difficult to simulate the Beta-style behavior using the mechanism of replacement. (We explore this in an exercise at the end of the chapter.)

16.2.3 Refinement and the subclass/subtype distinction

Advocates for those languages that use refinement argue that the mechanism is conceptually elegant in that it makes it almost impossible to write a subclass that is not also a subtype. This is because refinement guarantees that the behavior of the parent will be preserved, and thus whatever actions the parent performs must also be part of the child. In a language that uses replacement there is no such guarantee. An extreme illustration, but one that is not prohibited by the language definitions, would be for a parent class to use the message sqrt to compute a square root of a number and the child class overriding the message to compute a logarithm.

Advocates for replacement semantics argue that such errors are not common and are easily detected. Adopting replacement semantics as the default means you cannot use the mechanism to support code reuse or optimization of the form described by the earlier examples in Smalltalk. Furthermore, it is easy to simulate at least the most important parts of refinement using replacement (as shown in Figure 16.2), but the reverse is not true.

16.2.4 Wrappers in CLOS

Another interesting variation on refinement semantics occurs in the Lisp dialect, CLOS. In CLOS a subclass may override a method in a parent class and specify a *wrapping method*. A wrapping method can be a *before method*, an *after method*, or an *around method*. According to the type, the method is executed before, after, or surrounding the method in the parent class.

For example, suppose we define the following parent and child classes with three methods named atest, btest, and rtest. In the parent class these simply print out a message.

```
(defclass parent () () )

(defclass child (parent) () )

(defmethod atest ((x parent)) (print "atest parent"))
(defmethod btest ((x parent)) (print "btest parent"))
(defmethod rtest ((x parent)) (print "rtest parent"))
```

In the child class, we redefine the methods, one as a before method, one as an after method, and one as an around method. In the last case, a special statement, call-next-method, invokes the method in the parent class. This is similar to the way in which refinement is simulated in languages such as C++ and Object Pascal.

```
(defmethod atest :after ((x child)) (print "atest child"))
(defmethod btest :before ((x child)) (print "btest child"))
(defmethod rtest :around ((x child))
  (list "rtest child before" (call-next-method) "rtest child after"))
```

If we make an instance of the child class and execute the three methods, we see the before methods execute the child class before the parent, the after methods execute the child class after the parent, and the around methods do both.

```
  (defvar aChild (make-instance 'child))
  (atest aChild)
"atest child"
"atest parent"
  (btest aChild)
"btest parent"
"btest child"
  (rtest aChild)
("rtest child before" "rtest parent" "rtest child after")
```

16.3 ▫ Deferred Methods

We say a method is *deferred* if in the parent class the method is defined but not implemented. Deferred methods are also sometimes called *abstract* methods, and in C++ they are usually called a *pure virtual method*.

One advantage of deferred methods is conceptual in that their use allows the programmer to think of an activity as associated with an abstraction at a higher level than may actually be the case. For example, in a collection of classes representing geometric shapes, we can define a method to draw the shape in each of the subclasses Circle, Square, and Triangle. We could have defined a similar method in the parent class Shape, but such a method cannot, in actuality, produce any useful behavior, since the class Shape does not have sufficient information to draw the shape in question. Nevertheless, the mere presence of this method permits the user to associate the concept *draw* with the single class Shape and not with the three separate concepts Square, Triangle, and Circle.

There is a second, more practical reason for using deferred methods. In statically typed object-oriented languages, such as C++ and Object Pascal, a programmer is permitted to send a message to an object only if the compiler can

C++

```
class Shape {
public:
    virtual void draw () = 0;
};
```

Java (C# and Delphi are similar.)

```
abstract class Shape {
    abstract public void draw ();
}
```

Smalltalk (Objective-C is similar.)

```
draw
    " child class should override this "
    ^ self subclassResponsibility
```

Figure 16.3 — Syntax for deferred classes

determine that there is in fact a corresponding method that matches the message selector. Suppose the programmer wishes to define a polymorphic variable of class Shape that will, at various times, contain instances of each of the different shapes. Such an assignment is possible, according to our rule of substitutability. Nevertheless, the compiler will permit the message draw to be used with this variable only if it can ensure that the message will be understood by any value that may be associated with the variable. Assigning a method to the class Shape effectively provides this assurance, even when the method in class Shape is never actually executed.

Figure 16.3 illustrates some of the ways that deferred methods are documented in various languages. In both Java and C# a deferred method is declared as abstract. In C++ the notation looks like one is assigning the method the value zero. In Smalltalk and Objective-C the parent class implements the deferred method but invokes an error message generating routine (subclassResponsibility). The child class overrides the method, thereby avoiding calling the error message.

Deferred methods are almost always found in conjunction with other methods that are not intended to be overridden. The idea is to balance code reuse (the non-overridden methods) with specialization (the overridden methods). The non-overridden methods (sometimes called *foundational* methods) provide functionality that is modified for different situations by means of the deferred methods. We will see several examples of this when we explore the idea of software Frameworks in Chapter 21.

16.4 ▫ Overriding versus Shadowing

Because they have superficial syntactic similarities, it is easy to confuse the mechanism of overriding and the related programming language concept of shadowing. To explain this, let us first examine a slightly simpler form of shadowing. Consider the following class and method.

```
class Silly {
  private int x; // an instance variable named x

  public void example (int x) { // x shadows instance variable
    int a = x+1;
    while (a > 3) {
      int x = 1; // local variable shadows parameter
      a = a - x;
    }
  }
}
```

In this example class definition there are three different variables named x. The first is an instance variable. This variable is normally accessible from within any method defined as part of the class. However, in the method example there is a parameter defined using the same name. The parameter name hides, or shadows, access to the instance variable. This means that within the method, the name x is matched to the parameter and not to the instance variable. (In this case the instance variable can still be accessed using the construct `this.x`.)

Within the while loop another variable named x is declared. This is a local variable, and it again shadows access to the parameter x. Declaring variables in this fashion is not something to be encouraged, but nothing in the language definition prohibits it. (Some compilers will issue warning messages.)

Keep this simple example in mind when we next consider how shadowing can occur in conjunction with inheritance. A good example is with instance variables. The following is legal in Java, although it is probably not something to be encouraged.

```
class Parent {
  public int x = 12;
}

class Child extend Parent {
  public int x = 42; // shadows variable from parent class
}
```

The declaration of x in the child class will shadow that of the parent. The most important feature that distinguishes shadowing from overriding is that, like overloading, shadowing is resolved at compile time based on static types and does not require any run-time mechanism.

```
Parent p = new Parent();
System.out.println(p.x);
12
Child c = new Child();
System.out.println(c.x);
42
p = c; // be careful here!
System.out.println(p.x);
12
```

Many of those languages in Figure 16.1 that require an explicit indication of overriding will perform a shadowing if no keyword is provided. This can be illustrated by the following classes in C++.

```
class Parent {
public:
    // note, no virtual keyword here
  void example () { cout << "Parent" << endl; }
};

class Child : public Parent {
public:
  void example () { cout << "Child" << endl; }
};
```

As was true with the Java example of instance variables, a match of message to method is determined by the static declaration and not by a dynamic value.

```
  Parent * p = new Parent();
  p->example()
Parent
  Child * c = new Child();
  c->example()
Child
  p = c; // be careful here!
  p->example()
Parent
```

Like C++, Delphi will shadow a method if the override directive is omitted, but the compiler will issue a warning. This warning can be eliminated by using the

reintroduce directive, which explicitly indicates that the programmer intended the new method to shadow the earlier one.

```
type
  TParent = class
    public
      procedure example (); virtual;
    end;

  TChild = class (TParent)
    public
      procedure example (); reintroduce;
    end;
```

The ideas of overriding, shadowing, and redefinition (discussed in the previous chapter) should not be confused. The following table summarizes the differences.

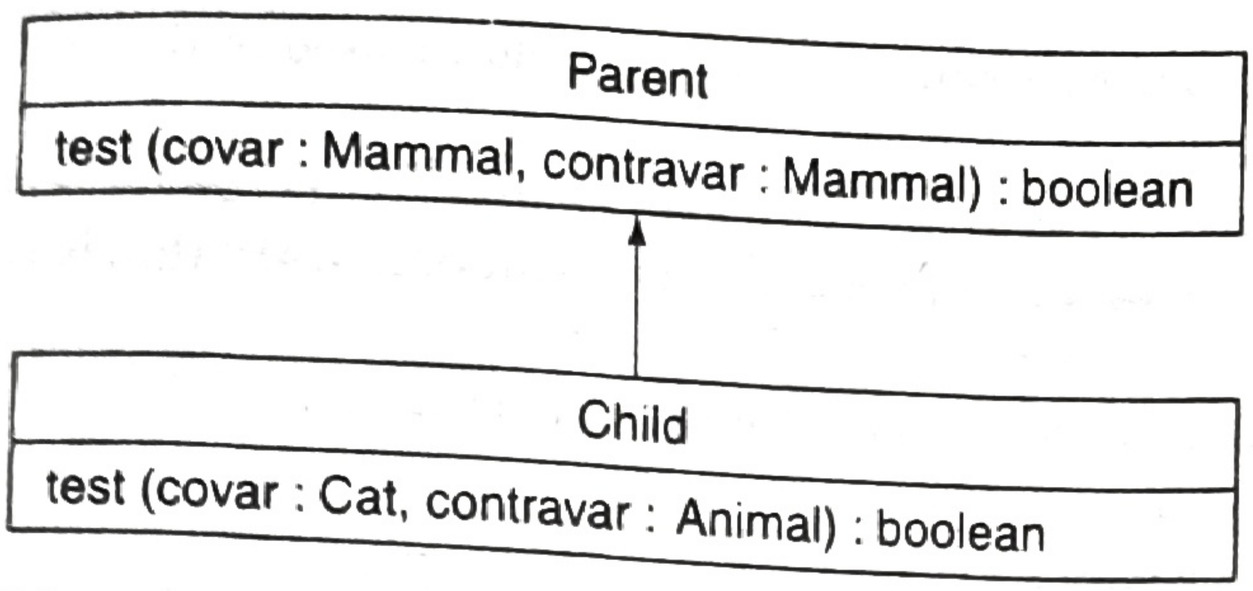
overriding	The type signatures are the same in both parent and child classes, and the method is declared as virtual in the parent class.
shadowing	The type signatures are the same in both parent and child classes, but the method was not declared as virtual in the parent class.
redefinition	The type signature in the child class differs from that given in the parent class.

16.5 ▫ Covariance and Contravariance

Frequently it seems as if it would be useful if overriding could be used but the type signature of a method in a child class could be different from that found in the parent class. A common example (one we will explore in another section) is a method equals used to compare two objects. Since normally one is only interested in comparing objects of the same type, it would seem to make sense that the argument in the child class should be the child class type. Unfortunately, the whole concept of changing type signatures is fraught with subtle difficulties, as we will explore in this section.

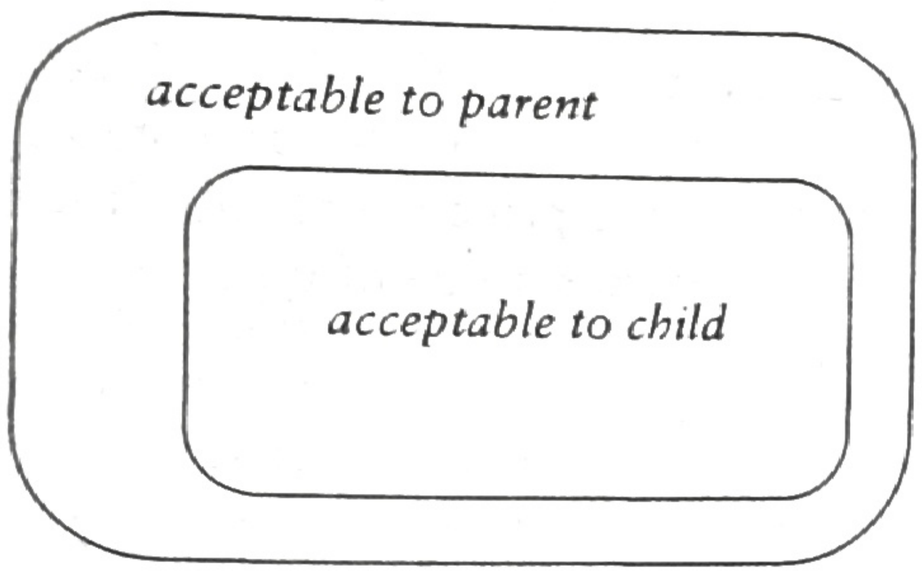
One seldom wants to change the type signatures arbitrarily but typically move either up or down the type hierarchy. The term *covariant* change is used when a type moves down the type hierarchy in the same direction as the child class. The term *contravariant* is used for the opposite—when a type moves up the class hierarchy in the opposite direction from subclassing. These two are shown in the following class hierarchy, where in the child class the first argument (named covar)

moves from Mammal to the more specific type Cat, while the second argument is changed in a contravariant fashion to the more general type Animal.



The impact of covariant or contravariant change is complicated by the fact that types can be used in a variety of different ways. The effect of changing a pass-by-value parameter is different from the effect of changing a pass-by-reference parameter, which is in turn different from the effect of changing the return type for a procedure. For this reason it is slightly easier to think about the problem using sets and consider the relationships between the set of values that are acceptable (for example, as an argument) to the parent in relationship to the set of values that are acceptable to the child.

We first consider what can happen if the set of values acceptable to the child is smaller than the set of values acceptable to the parent. This can happen, for example, in a covariant change to a pass-by-value parameter. (The parent class has a parameter of type Mammal, and the child class restricts the same parameter to the type Cat.)



A problem occurs when this covariant change runs into the principle of substitution. According to the principle of substitution, we should be able to declare a variable using the parent type but assign it a value from the child class.

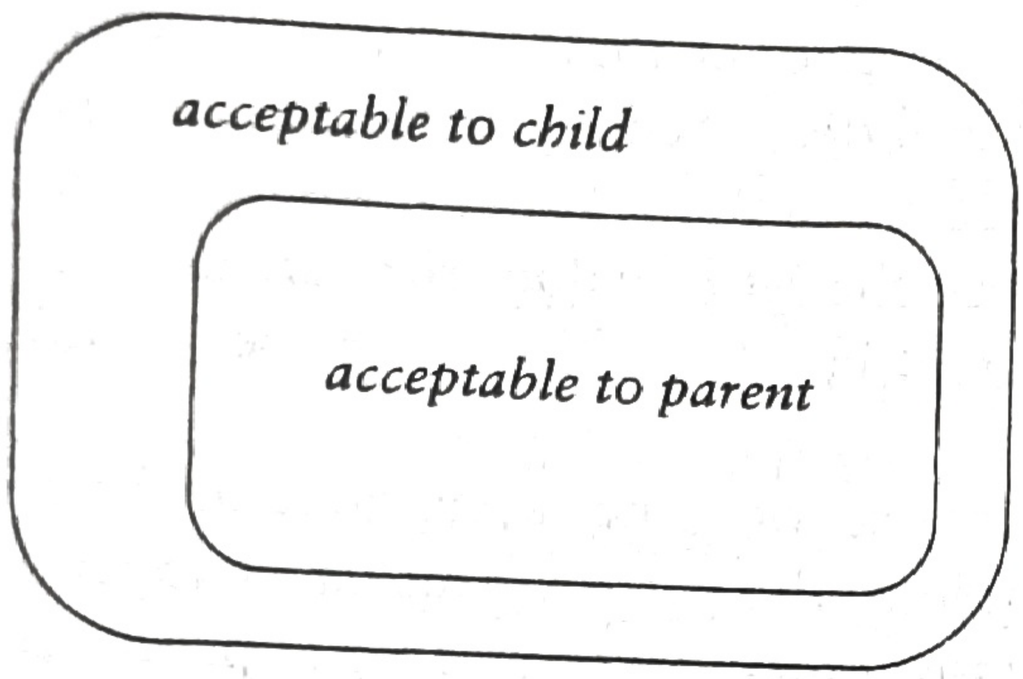
```
Parent aValue = new Child();
aValue.test(new Dog(), new Mammal()); // note type of first argument
```

As far as the compiler is concerned, the first argument is perfectly acceptable, since the declaration insists only that the value be type Mammal. But the invocation

will bind the message to the method in the child class, which is prepared only to accept values of type Cat. The consequence will almost certainly be a completely erroneous and catastrophic outcome.

It is occasionally proposed that run-time checks could be used to detect this condition, allowing at least a graceful error reporting, albeit at run time rather than at compile time. However, note that such checks are never necessary when the child is being used as an instance of the child class, but only when the child is being used as an instance of the parent class. Thus, in a large percentage, perhaps the majority, of cases such run-time checks would be superfluous.

An error will not occur in the opposite case, where a contravariant change to a by-value parameter increases the range of values the child class is willing to handle. In this case neither the parent class nor the child class can pass an unacceptable value to the method.



However, that is only true for pass-by-value parameters. When we consider a change to the result type of a method, the situation is exactly the reverse. Suppose a method in the parent class returns a value of type Mammal, and the child class includes a contravariant change that extends this to Animal, thereby making the set of values acceptable to the child *larger* than the set of values acceptable to the parent. Once again we run into problems with the principle of substitution. It would be perfectly legal for the child class to return a value of type Bird, since locally a bird satisfies the typing restrictions.

```
class Parent {
  Mammal test ( ) {
    return new Cat();
  }
}

class  Child extends Parent {
  Animal test () {
    return new Bird();
  }
}
```

But it is also legal for a variable of the parent class to hold an instance of the child class and for the result of executing the method in question to be assigned to a variable of type Mammal, since as far as the compiler is concerned, the result of the method fits that class designation.

```
Parent aValue = new Child();
Mammal result = aValue.test();   // error—a bird is not a mammal
```

The consequence will be the variable of type Mammal holding a non-mammal value, with no typing error having been reported.

A pass-by-reference parameter can be used to pass information both into and out of a procedure. Therefore, both of these errors could arise if any change whatsoever is permitted in such a value.

It is possible to specify a language so that both covariant and contravariant overriding is permitted in certain situations. Examples of languages that permit this include Eiffel and Sather. (See Section 12.4.2 for a discussion of LIKE CURRENT, which is one approach to covariant overriding.) C++ allows a covariant change in signature on the return types. This is typically used in a method such as the following.

```
class Parent {
public:
  Parent * clone () { return new Parent(); }
};

class Child : public Parent {
public:
  Child * clone () { return new Child(); }
};
```

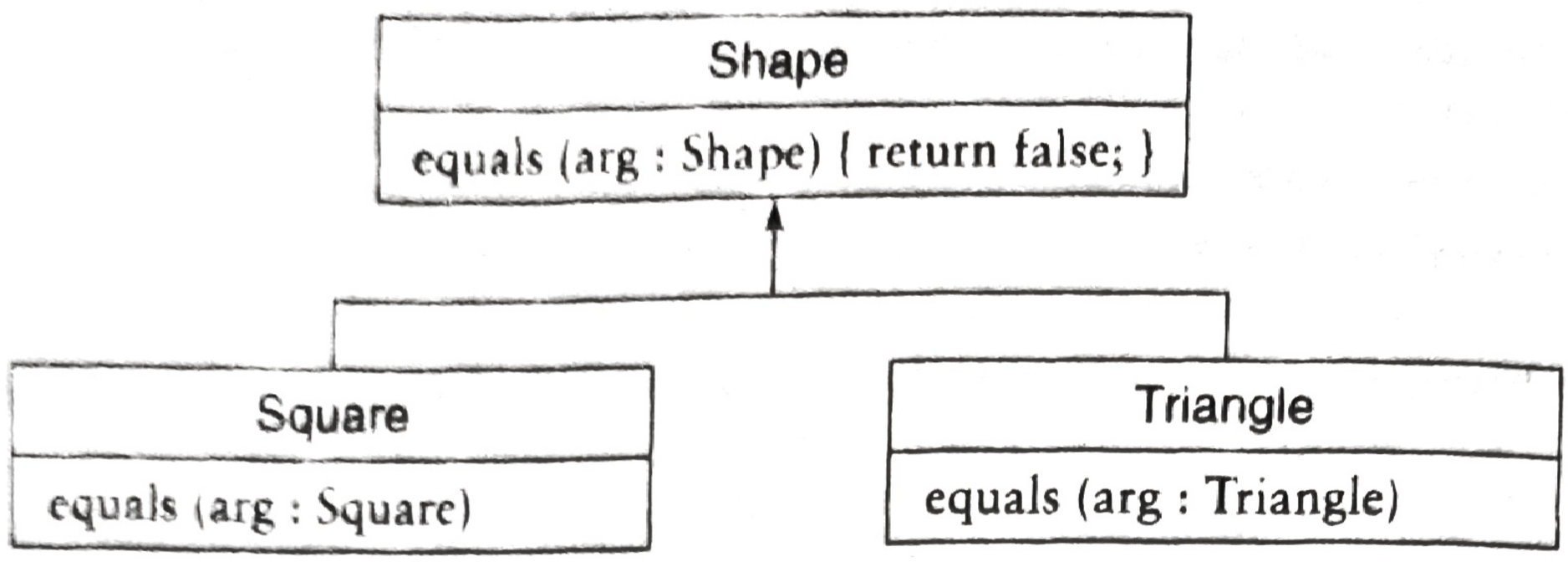
As we noted earlier, the restriction of the set of types the child can return cannot result in a type error, since any value the child can yield will still match the specifications of the parent.

However, most language designers have opted to avoid this problem altogether by using a rule that might be termed *novariance*—namely, that a child class is not allowed to change the type signature of an overridden method in any fashion.

Equality testing. The example where covariant modification of type signatures often seems most compelling is in the implementation of equality and comparison tests. Examining this situation will provide a concrete illustration of the problems that can arise.

Imagine we have the following class hierarchy where the equals method in the parent class always returns false, whereas the overridden method in the child

classes correctly handles the comparison of triangles to triangles and of squares to squares.



In the imaginary language we envision here, consider what meaning might be assigned to an attempt to compare a triangle to a square.

```
Triangle aTriangle;
Square aSquare;

if aTriangle.equals(aSquare) ...
```

There are two possibilities.

- The search for a method is based solely on the receiver, a triangle, and yields the triangle method, which requires a triangle as argument. Thus, the use of a square as argument results in a compiler error.
- The search for a method is based both on the receiver and on the argument type signatures. Since the argument does not match the method in class Triangle, the method in the parent class Shape is then invoked.

Selecting either interpretation leads to trouble as soon as we consider the principle of substitution. Suppose we create a shape variable and assign it the triangle value. Comparing the triangle and the shape is then effectively comparing a value to itself. But the first interpretation results in an error, and the second interpretation results in the nonsensical result that the value is not equal to itself.

```
Shape aShape = aTriangle;
if aTriangle.equals(aShape) ...
```

A similar incongruity occurs if the parent variable is used as the receiver. Since the only reasonable method to use is the one defined in the parent class, the test yields an unexpected false value.

```
if aShape.equals(aShape) ...
```

Both because the implementation of either covariant or contravariant overriding is complex and because the semantics are cloudy, almost all object-oriented

languages prohibit any modification of argument types in overridden methods. To get around this restriction, programmers most often resort to explicit tests and casts, as in the following C++ example.

```
bool Triangle.equals (Shape * aShape)
{
  Triangle * right = dynamic_cast<Triangle *>(aShape);
  if (right) { // it was a triangle
    // ⋮ do triangle comparison
  } else // it was not a triangle
    return false;
}
```

Again it is important to note that we here are discussing only overloading. Equality can be implemented using a redefinition (as described in the previous chapter), but redefinition is resolved at compile time based on static types and not at run time using dynamic values.

*16.6 ⊡ Variations on Overriding

In this section we will explore a few of the language-specific variations on the theme of overriding.

16.6.1 Final methods in Java

While the virtual keyword *permits* overriding, and the abstract keyword *requires* it, it is sometimes desirable to *prohibit* overriding. That is, the programmer might like to insist that the method defined in a parent class is the final and definitive version and no further changes can be made by a child class. In Java this can be accomplished through the use of the final modifier.

```
class Parent {
  public final void aMethod (int) { ... }
}

class Child extends Parent {
    // compiler error, not allowed to override final method
  public void aMethod (int) { ... }
}
```

Java also permits the keyword to be applied to an entire class, in which case the class cannot be subclassed.

The language C# has a similar keyword, sealed, that can be applied to classes but not to individual methods.

```
sealed class Parent {
    :
    :
}

class  Child : Parent { // generates a compiler error
    :
    :
}
```

Final methods and sealed classes can be exploited by an optimizing compiler, since the use of such features will often eliminate the need for a run-time message resolution.

16.6.2 Versioning in C#

As software changes over time, it is not uncommon for new methods to be introduced into class hierarchies. Sometimes this can have unfortunate consequences. For example, suppose that a child class implements a virtual method on its own—that is, one that is not inherited from its parent.

```
class Parent {
    :
    :
}

class Child : Parent {
  public virtual void aMethod(int) { ... }
}
```

Later a new revision of the parent class is released, and this new revision (which may have been created in an entirely different organization from that developing the child class) includes a method using the same name.

```
class Parent {
  public virtual void aMethod(int) { ... }
}

class Child : Parent {
  public virtual void aMethod(int) { ... }
}
```

Almost all object-oriented languages will consider these to be the same method. However, the original intent was for them to be different. In C# the programmer can add the new keyword in the child class. The presence of this keyword changes an overloading into a redefinition. (That is, the method in the child class is considered to be independent of the method in the parent class.)

```
class Parent {
  public virtual void aMethod(int) { ... }
}

class Child : Parent {
  public new virtual void aMethod(int) { ... }
}
```

A difficulty in using this feature is that the programmer cannot predict ahead of time which methods should be declared as new, since it is only necessary after a change has been introduced in the parent.

Summary

- An *override* occurs when a method in a child class uses the same name and type signature as a method in the parent class.
- While technically an override is an overloading (there are two method bodies with the same name), overriding is resolved at run time, whereas overloading is resolved at compile time.
- The effect of an override can be handled in two possible ways. A *replacement* completely replaces the code in the parent class; a *refinement* combines the code in the child with that of the parent.
- A *deferred method* is a form of overriding where the parent provides no implementation and the child is solely responsible for implementing the specified behavior.
- A name can *shadow* another use of the same name if it temporarily hides access to the previous meaning. Some languages permit both shadowing and overriding.
- A covariant change in parameter or return type is a change the moves down the class hierarchy in the same direction as the child class. A contravariant change moves a parameter or return type up the class hierarchy in the opposite direction from the child class. The semantics of both types of changes can be subtle.