

```

class ListClass
  meth isNil(_) raise undefinedMethod end end
  meth append(_ _) raise undefinedMethod end end
  meth display raise undefinedMethod end end
end

class NilClass from ListClass
  meth init skip end
  meth isNil(B) B=true end
  meth append(T U) U=T end
  meth display {Browse nil} end
end

class ConsClass from ListClass
  attr head tail
  meth init(H T) head:=H tail:=T end
  meth isNil(B) B=false end
  meth append(T U)
    U2={@tail append(T $)}
  in
    U={New ConsClass init(@head U2)}
  end
  meth display {Browse @head} {@tail display} end
end

```

Figure 7.13: Lists in object-oriented style

7.4.2 Constructing a hierarchy by following the type

When writing programs with recursion, we saw in Section 3.4.2 that it is a good idea to define first the type of the data structure, and then to construct the recursive program by following the type. We can use a similar idea to construct inheritance hierarchies. For example, consider the list type $\langle \text{List } T \rangle$, which is defined as:

$$\begin{array}{lcl}
 \langle \text{List } T \rangle & ::= & \text{nil} \\
 & | & T \cdot | \cdot \langle \text{List } T \rangle
 \end{array}$$

This says that a list is either `nil` or a list pair. Let us implement the list ADT in the class `ListClass`. Following the type definition means that we define two other classes that inherit from `ListClass`, which we can call `NilClass` and `ConsClass`. Figure 7.12 shows the hierarchy. This hierarchy is a natural design to respect the substitution principle. An instance of `NilClass` is a list, so it is easy to use it wherever a list is required. The same holds for `ConsClass`.

Figure 7.13 defines a list ADT that follows this hierarchy. In this figure, `ListClass` is an *abstract class*: a class in which some methods are left undefined. Trying to call the methods `isNil`, `append`, and `display` will raise an

```

class GenericSort
  meth init skip end
  meth qsort(Xs Ys)
    case Xs
    of nil then Ys = nil
    [] P|Xr then S L in
      {self partition(Xr P S L)}
      {Append {self qsort(S $)}
        P|{self qsort(L $)} Ys}
    end
  end
  meth partition(Xs P Ss Ls)
    case Xs
    of nil then Ss=nil Ls=nil
    [] X|Xr then Sr Lr in
      if {self less(X P $)} then
        Ss=X|Sr Ls=Lr
      else
        Ss=Sr Ls=X|Lr
      end
      {self partition(Xr P Sr Lr)}
    end
  end
end
end

```

Figure 7.14: A generic sorting class (with inheritance)

exception. Abstract classes are not intended to be instantiated, since they lack some methods. The idea is to define another class that inherits from the abstract class and that adds the missing methods. This gives a *concrete class*, which can be instantiated since it defines all the methods it calls. NilClass and ConsClass are concrete classes. They define the methods isNil, append, and display. The call {L1 append(L2 L3)} binds L3 to the concatenation of L1 and L2, without changing L1 or L2. The call {L display} displays the list. Let us now do some calculations with lists:

```

L1={New ConsClass
  init(1 {New ConsClass
    init(2 {New NilClass init}})}}
L2={New ConsClass init(3 {New NilClass init}})
L3={L1 append(L2 $)}
{L3 display}

```

This creates two lists L1 and L2 and concatenates them to form L3. It then displays the contents of L3 in the browser, as 1, 2, 3, nil.

```
class IntegerSort from GenericSort
  meth less(X Y B)
    B=(X<Y)
  end
end

class RationalSort from GenericSort
  meth less(X Y B)
     $\wedge \wedge (P\ Q)=X$ 
     $\wedge \wedge (R\ S)=Y$ 
    in B=(P*S<Q*R) end
end
```

Figure 7.15: Making it concrete (with inheritance)

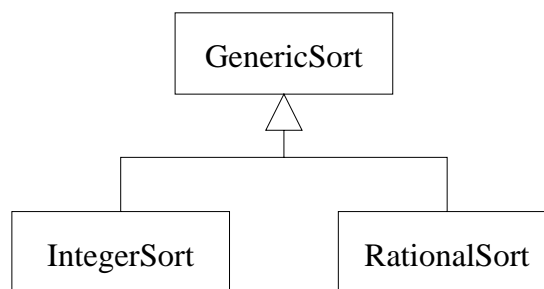


Figure 7.16: A class hierarchy for genericity

```

fun {MakeSort Less}
  class $
    meth init skip end
    meth qsort(Xs Ys)
      case Xs
      of nil then Ys = nil
      [] P|Xr then S L in
        {self partition(Xr P S L)}
        {Append {self qsort(S $)}
          P|{self qsort(L $)} Ys}
      end
    end
    meth partition(Xs P Ss Ls)
      case Xs
      of nil then Ss=nil Ls=nil
      [] X|Xr then Sr Lr in
        if {Less X P} then
          Ss=X|Sr Ls=Lr
        else
          Ss=Sr Ls=X|Lr
        end
        {self partition(Xr P Sr Lr)}
      end
    end
  end
end

```

Figure 7.17: A generic sorting class (with higher-order programming)

7.4.3 Generic classes

A *generic class* is one that only defines part of the functionality of an ADT. It has to be completed before it can be used to create objects. Let us look at two ways to define generic classes. The first way, often-used in object-oriented programming, uses inheritance. The second way uses higher-order programming. We will see that the first way is just a syntactic variation of the second. In other words, inheritance can be seen as a programming style that is based on higher-order programming.

Using inheritance

A common way to make classes more generic in object-oriented programming is to use abstract classes. For example, Figure 7.14 defines an abstract class `GenericSort` for sorting a list. This class uses the quicksort algorithm, which needs a boolean comparison operation. The boolean operation's definition depends on the type of data that is sorted. Other classes can inherit from `GenericSort`

```

IntegerSort = {MakeSort fun {$ X Y} X<Y end}

RationalSort = {MakeSort fun {$ X Y}
                  ^/^(P Q) = X
                  ^/^(R S) = Y
                  in P*S<Q*R end}

```

Figure 7.18: Making it concrete (with higher-order programming)

and add definitions of `less`, for example, for integers, rationals, or strings. In this case, we specialize the abstract class to form a *concrete class*, i.e., a class in which all methods are defined. Figure 7.15 defines the concrete classes `IntegerSort` and `RationalSort`, which both inherit from `GenericSort`. Figure 7.16 shows the resulting hierarchy.

Using higher-order programming

There is a second natural way to create generic classes, namely by using higher-order programming directly. Since classes are first-class values, we can define a function that takes some arguments and returns a class that is specialized with these arguments. Figure 7.17 defines the function `MakeSort` that takes a boolean comparison as its argument and returns a sorting class specialized with this comparison. Figure 7.18 defines two classes, `IntegerSort` and `RationalSort`, that can sort lists of integers and lists of rational numbers (the latter represented as pairs with label `^/^(`). Now we can execute the following statements:

```

ISort={New IntegerSort init}
RSort={New RationalSort init}

{Browse {ISort qsort([1 2 5 3 4] $)}}
{Browse {RSort qsort([ ^/^(23 3) ^/^(34 11) ^/^(47 17)] $)}}

```

Discussion

It is clear that we are using inheritance to “plug in” one operation into another. This is just a form of higher-order programming, where the first operation is passed to the second. What is the difference between the two techniques? In most programming languages, the inheritance hierarchy must be defined at compile time. This gives a *static* genericity. Because it is static, the compiler may be able to generate better code or do more error checking. Higher-order programming, when it is possible, lets us define new classes at run-time. This gives a *dynamic* genericity, which is more flexible.

7.4.4 Multiple inheritance

Multiple inheritance is useful when an object has to be two different things in the same program. For example, consider a graphical display that can show a variety of geometrical figures, including circles, lines, and more complex figures. We would like to define a “grouping” operation that can combine any number of figures into a single, composite figure. How can we model this with object-oriented programming? We will design a simple, fully working program. We will use multiple inheritance to add the grouping ability to figures. The idea for this design comes from Bertrand Meyer [122]. This program can easily be extended to a full-fledged graphics package.

Geometric figures

We first define the class `Figure` to model geometric figures, with methods `init` (initialize the figure), `move(X Y)` (move the figure), and `display` (display the figure):

```
class Figure
  meth otherwise(M)
    raise undefinedMethod end
  end
end
```

This is an abstract class; any attempt to invoke its methods will raise an exception. Actual figures are instances of subclasses of `Figure`. For example, here is a `Line` class:

```
class Line from Figure
  attr canvas x1 y1 x2 y2
  meth init(Can X1 Y1 X2 Y2)
    canvas:=Can
    x1:=X1 y1:=Y1
    x2:=X2 y2:=Y2
  end
  meth move(X Y)
    x1:=@x1+X y1:=@y1+Y
    x2:=@x2+X y2:=@y2+Y
  end
  meth display
    {@canvas create(line @x1 @y1 @x2 @y2)}
  end
end
```

and here is a `Circle` class:

```
class Circle from Figure
  attr canvas x y r
  meth init(Can X Y R)
    canvas:=Can
```

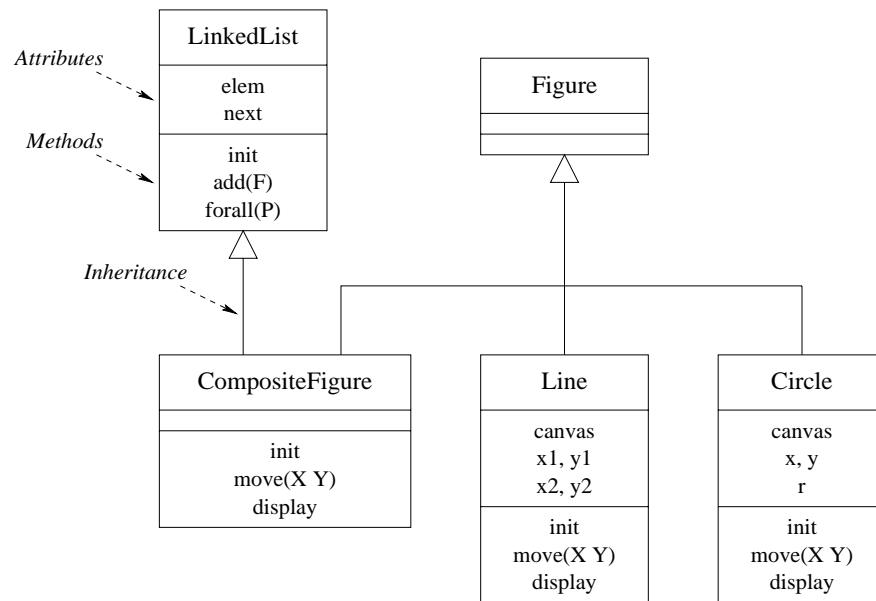


Figure 7.19: Class diagram of the graphics package

```

        x:=X y:=Y r:=R
    end
    meth move(X Y)
        x:=@x+X y:=@y+Y
    end
    meth display
        {@canvas create(oval @x-@r @y-@r @x+@r @y+@r)}
    end
end

```

Figure 7.19 shows how `Line` and `Circle` inherit from `Figure`. This kind of diagram is called a *class diagram*. It is a part of UML, the Uniform Modeling Language, a widely-used set of techniques for modeling object-oriented programs [54]. Class diagrams are a useful way to visualize the class structure of an object-oriented program. Each class is represented by a rectangle with three parts, containing the class name, the attributes it defines, and the methods it defines. These rectangles can be connected with lines representing inheritance links.

Linked lists

We define the class `LinkedList` to group figures together, with methods `init` (initialize the linked list), `add(F)` (add a figure), and `forall(M)` (execute {F M} for all figures):

```

class LinkedList
    attr elem next

```

```

meth init(elem:E<=null next:N<=null)
    elem:=E next:=N
end
meth add(E)
    next:={New LinkedList init(elem:E next:@next)}
end
meth forall(M)
    if @elem\=null then {@elem M} end
    if @next\=null then {@next forall(M)} end
end
end

```

The `forall(M)` method is especially interesting because it uses first-class messages. A linked list is represented as a sequence of instances of this class. The `next` field of each instance refers to the next one in the list. The last element has the `next` field equal to `null`. There is always at least one element in the list, called the header. The header is not an element that it seen by users of the linked list; it is just needed for the implementation, The header always has the `elem` field equal to `null`. Therefore an empty linked list corresponds to a header node with both `elem` and `next` fields equal to `null`.

Composite figures

What is a composite figure? It is *both* a figure and a linked list of figures. Therefore we define a class `CompositeFigure` that inherits from both `Figure` and `LinkedList`:

```

class CompositeFigure from Figure LinkedList
    meth init
        LinkedList,init
    end
    meth move(X Y)
        {self forall(move(X Y))}
    end
    meth display
        {self forall(display)}
    end
end

```

Figure 7.19 shows the multiple inheritance. The multiple inheritance is correct because the two functionalities are completely different and have no undesirable interaction. The `init` method is careful to initialize the linked list. It does not need to initialize the figure. As in all figures, there is a `move` and a `display` method. The `move(X Y)` method moves all figures in the linked list. The `display` method displays all figures in the linked list.

Do you see the beauty of this design? With it, a figure can consist of other figures, some of which consist of other figures, and so forth, to any number of



Figure 7.20: Drawing in the graphics package

levels. The inheritance structure guarantees that moving and displaying will always work correctly.

Example execution

Let us run this example. First, we set up a window with a graphics display field:

```
declare
W=250 H=150 Can
Window={QtK.build td(canvas(width:W height:H bg:white handle:Can))}
{Window show}
```

This uses the QtK graphics tool, which is explained in Chapter 10. For now just assume that this sets up a canvas, which is the drawing field for our geometric figures. Next, we define a composite figure F1 containing a triangle and a circle:

```
declare
F1={New CompositeFigure init}
{F1 add({New Line init(Can 50 50 150 50)})}
{F1 add({New Line init(Can 150 50 100 125)})}
{F1 add({New Line init(Can 100 125 50 50)})}
{F1 add({New Circle init(Can 100 75 20)})}
```

We can display this figure as follows:

```
{F1 display}
```

This displays the figure once. Let us move the figure around and display it each time:

```
for I in 1..10 do {F1 display} {F1 move(3 ~2)} end
```

Figure 7.20 shows the result.

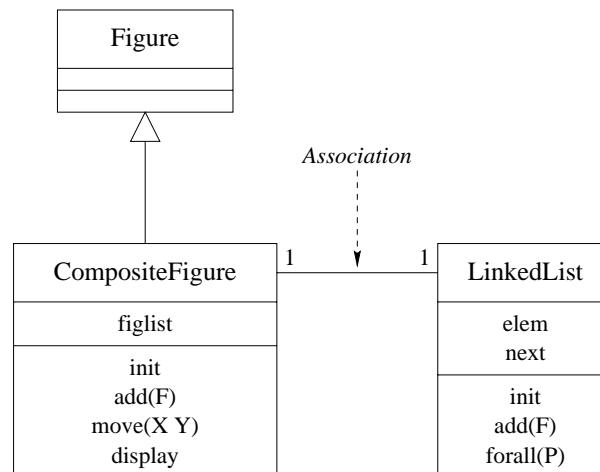


Figure 7.21: Class diagram with an association

Composite figures with single inheritance

Instead of defining `CompositeFigure` with multiple inheritance, we can define it using single inheritance by putting the list of figures in an attribute. This gives:

```

class CompositeFigure from Figure
  attr figlist
  meth init
    figlist:={New LinkedList init}
  end
  meth add(F)
    {@figlist add(F)}
  end
  meth move(X Y)
    {@figlist forall(move(X Y))}
  end
  meth display
    {@figlist forall(display)}
  end
end

```

Figure 7.21 shows the class diagram for this case. The link between `CompositeFigure` and `LinkedList` is called an *association*. It represents a relationship between the two classes. The numbers attached to the two ends are cardinalities; each number says how many elements there are for a particular instance. The number 1 on the linked list side means that there is exactly one linked list per composite figure, and similarly for the other side. The association link is a specification; it does not say how it is implemented. In our case, each composite figure has a `figlist` attribute that references a linked list.

The example execution we gave before will also work in the single inheritance case. What are the trade-offs in using single or multiple inheritance in this

example? In both cases, the figures that make up the composite figure are encapsulated. The main difference is that multiple inheritance brings the operations of linked lists up to the same level as figures:

- With multiple inheritance, a composite figure is also a linked list. All the operations of the `LinkedList` class can be used directly on composite figures. This is important if we want to do linked list calculations with composite figures.
- With single inheritance, a composite figure completely hides its structure. This is important if we want to protect the composite figure from any calculations other than those defined in its class.

Scaling it up

It is straightforward to extend this example to be a full-fledged graphics package. Here are some of the changes that should be made:

- Many more figures can be defined to inherit from `Figure`.
- In the current implementation, figures are tied to their canvas. This has the advantage that it allows figures to be spread over multiple canvasses. But usually we will not want this ability. Rather, we would like to be able to draw the same figure on different canvasses. This means that the canvas should not be an attribute of figure objects but be passed as argument to the `display` method.
- A *journaling* facility can be added. That is, it should be possible to record sequences of drawing commands, i.e., sequences of calls to figures, and manipulate the recordings as first-class entities. These recordings represent drawings at a high level of abstraction. They can then be manipulated by the application, stored on files, passed to other applications, etc.
- The `display` method should be able to pass arbitrary parameters from the application program, through the graphics package, to the underlying graphics subsystem. In the `Line` and `Circle` classes, we change it as follows:

```
meth display(...)=M
    {@canvas {Adjoin M create(line @x1 @y1 @x2 @y2)}}
end
```

The `Adjoin` operation combines two record arguments, where the second argument overrides the first in the case of conflicts. This allows arbitrary parameters to be passed through `display` to the canvas drawing command. For example, the call `{F display(fill:red width:3)}` draws a red figure of width 3.

7.4.5 Rules of thumb for multiple inheritance

Multiple inheritance is a powerful technique that has to be used with care. We recommend that you use multiple inheritance as follows:

- Multiple inheritance works well when combining two completely independent abstractions. For example, figures and linked lists have nothing in common, so they can be combined fruitfully.
- Multiple inheritance is much harder to use correctly when the abstractions have something in common. For example, creating a `WorkStudy` class from `Student` and `Employee` is dubious, because students and employees are both human beings. They may in fact both inherit from a common `Person` class. Even if they do not have a shared ancestor, there can be problems if they have some concepts in common.
- What happens when sibling superclasses share (directly or indirectly) a common ancestor class that specifies a stateful object (i.e., it has attributes)? This is known as the *implementation-sharing problem*. This can lead to duplicated operations on the common ancestor. This typically happens when initializing an object. The initialization method usually has to initialize its superclasses, so the common ancestor is initialized twice. The only remedy is to understand carefully the inheritance hierarchy to avoid such duplication. Alternatively, you should only inherit from multiple classes that do not share a stateful common ancestor.
- When name clashes occur, i.e., the same method label is used for adjacent superclasses, then the program must define a local method that overrides the conflict-causing methods. Otherwise the object system gives an error message. A simple way to avoid name clashes is to use name values as method heads. This is a useful technique for classes, such as mixin classes, that are often inherited from by multiple inheritance.

7.4.6 The purpose of class diagrams

Class diagrams are excellent tools for visualizing the class structure of an application. They are at the heart of the UML approach to modeling object-oriented applications, and as such they enjoy widespread use. This popularity has often masked their limitations. They have three clear limitations:

- They do not specify the functionality of a class. For example, if the methods of a class enforce an invariant, then this invariant does not show up in the class diagram.
- They do not model the dynamic behavior of the application, i.e., its evolution over time. Dynamic behavior is both large-scale and small-scale.

Applications often go through different phases, for which different class diagrams are valid. Applications are often concurrent, with independent parts that interact in coordinated ways.

- They only model one level in the application's component hierarchy. As Section 6.7 explains, well-structured applications have a hierarchical decomposition. Classes and objects are near the base of this hierarchy. A class diagram explains the decomposition at this level.

The UML approach recognizes these limitations and provides tools that partially alleviate them, e.g., the interaction diagram and the package diagram. Interaction diagrams model part of the dynamic behavior. Package diagrams model components at a higher level in the hierarchy than classes.

7.4.7 Design patterns

When designing a software system, it is common to encounter the same problems over and over again. The design pattern approach explicitly recognizes this and proposes solutions to these problems. A *design pattern* is a technique that solves one of these common problems. The present book is full of design patterns in that sense. For example, here are two:

- In declarative programming, Section 3.4.2 introduces the rule of *constructing a program by following a type*. A program that uses a complicated recursive data structure can often be written easily by looking at the type of the data structure. The structure of the program mirrors the type definition.
- Section 6.4.2 introduces a series of techniques for making an ADT secure by *wrapping the functionality in a secure layer*. These techniques are independent of what the ADT does; they work for *any* ADT.

Design patterns were first popularized in an influential book by Gamma, Helm, Johnson, and Vlissides [58], which gives a catalogue of design patterns in object-oriented programming and explains how to use them. The catalogue emphasizes patterns based on inheritance, using the type view. Let us look at a typical design pattern of this catalogue from the viewpoint of a programmer who thinks in terms of computation models.

The Composite pattern

Composite is a typical example of a design pattern. The purpose of Composite is to build hierarchies of objects. Given a class that defines a leaf, the pattern shows how to use inheritance to define trees. Figure 7.22, taken from Gamma *et al*, shows the inheritance diagram of the Composite pattern. The usual way to use this pattern is to plug in an initial leaf class, `Leaf`. Then the pattern defines

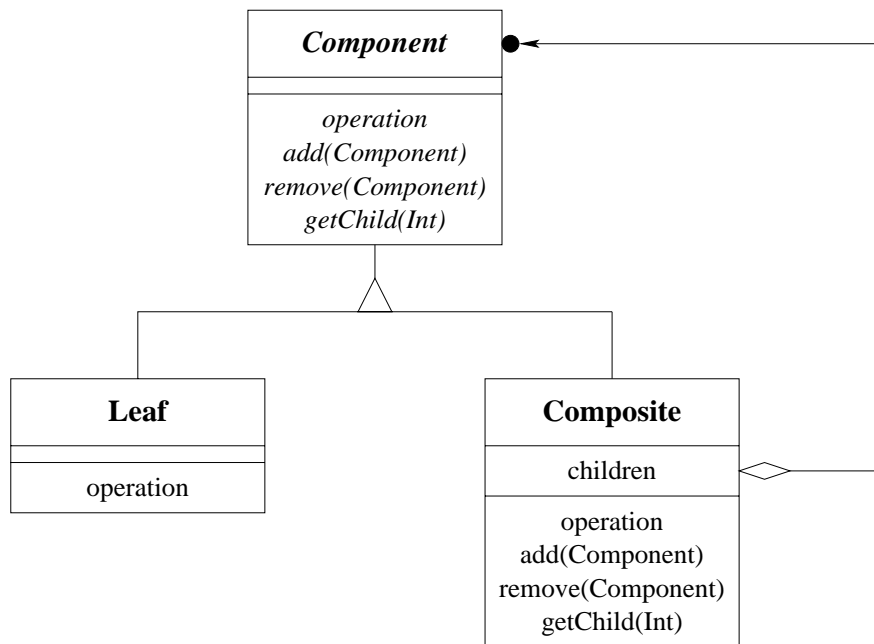


Figure 7.22: The Composite pattern

both the `Composite` and `Component` classes. `Component` is an abstract class. The hierarchy is either an instance of `Leaf` or `Composite`.

We can use the Composite pattern to define compound graphic figures. Section 7.4.4 solves the problem by combining a figure and a linked list (either with single or multiple inheritance). The Composite pattern is a more abstract solution, in that it does not assume that the grouping is done by a linked list. The `Composite` class has `add` and `remove` operations but does not say how they are implemented. They could be implemented as a linked list, but they could also be implemented differently, e.g., as a dictionary or as a declarative list.

Given a class that defines a leaf of the tree, the Composite pattern returns a class that defines the tree. When put in this way, this sounds much like higher-order programming: we would like to define a function that accepts a class and returns another class. Most programming languages, such as C++ and Java, do not allow defining this function, however. There are two reasons for this. First, most languages do not consider classes as first-class values. Second, the function defines a new *superclass* of the input class. Most languages allow defining new subclasses but not new superclasses. Yet despite these limitations we would still like to use the Composite pattern in our programs.

The usual solution to this dilemma is to consider design patterns as primarily a way to organize one's thoughts, without necessarily being supported by the programming language. A pattern might exist only in the mind of the programmer. Design patterns can then be used in languages like C++ or Java, even if they cannot be implemented as abstractions in those languages. This can be made

easier by using a source code preprocessor. The programmer can then program directly with design patterns, and the preprocessor generates the source code for the target language.

Supporting the Composite pattern

The object system of this chapter lets us support a grouping pattern very much like Composite from within the computation model. Let us implement a tree structure whose leaves and internal nodes are objects. The leaves are instances of the `Leaf` class, which is provided at run-time. The internal nodes forward all method invocations to the leaves in their subtree. The simplest way to implement this is to define a class `Composite` for the internal nodes. This class contains a list of its children, which may be instances of `Composite` or `Leaf`. We assume that all instances have the initialization method `init` and that `Composite` instances have the method `add` for adding a new subtree.

```

class Composite
  attr children
  meth init
    children:=nil
  end
  meth add(E)
    children:=E|@odelist
  end
  meth otherwise(M)
    for N in @children do {N M} end
  end
end

```

If nodes have many subnodes, then it is inefficient to remove nodes in this implementation. In that situation, using dictionaries instead of lists might be a good choice. Here is an example of how to construct a tree:

```

N0={New Composite init}
L1={New Leaf init} {N0 add(L1)}
L2={New Leaf init} {N0 add(L2)}
N3={New Composite init} {N0 add(N3)}
L4={New Leaf init} {N0 add(L4)}

L5={New Leaf init} {N3 add(L5)}
L6={New Leaf init} {N3 add(L6)}
L7={New Leaf init} {N3 add(L7)}

```

If `Leaf` is the `Figure` class of Section 7.4.4, then `Composite` defines composite figures.

Enforcing valid trees This implementation works for any `Leaf` class because of dynamic typing. The disadvantage of this solution is that the system does not

enforce all leaves to be instances of the same class. Let us add such enforcement to Composite:

```

class Composite
  attr children valid
  meth init(Valid)
    children:=nil
    @valid=Valid
  end
  meth add(E)
    if {Not {@valid E}} then raise invalidNode end end
    children:=E|@children
  end
  meth otherwise(M)
    for N in @children do {N M} end
  end
end

```

When an instance of Composite is initialized, it is given a function Valid, which is bound to the stateless attribute valid. The function Valid is used to check the validity of each inserted node.

7.5 Relation to other computation models

“The language does not prevent you from deeply nesting classes, but good taste should. [...] Nesting more than two levels invites a readability disaster and should probably never be attempted.”

– The Java Programming Language, Second Edition,
Ken Arnold and James Gosling (1998) [10]

Object-oriented programming is one way to structure programs, which is most often used together with explicit state. In comparison with other computation models, it is characterized primarily by its use of *inheritance*. From the viewpoint of multiple computation models, inheritance is not a new concept in the kernel language, but emerges rather from how the **class** linguistic abstraction is defined. This section examines how inheritance relates to other higher-order techniques. This section also examines the commonly-stated desire that “everything should be an object”, to find out what it means and to what extent it makes sense.

7.5.1 Object-based and component-based programming

Object-based programming is object-oriented programming without inheritance. This is like component-based programming with class syntax. This gives a convenient notation for encapsulating state and defining multiple operations on it. Without inheritance, the object abstraction becomes much simpler. There are no

problems of overriding and conflict in multiple inheritance. Static and dynamic binding are identical.

7.5.2 Higher-order programming

Object-oriented programming and higher-order programming are closely related. For example, let us examine the case of a sorting routine that is parameterized by an order function. A new sorting routine can be created by giving a particular order function. In higher-order programming, this can be written as follows:

```
proc {NewSortRoutine Order ?SortRoutine}
  proc {SortRoutine InL OutL}
    % ... {Order X Y} calculates order
  end
end
```

In object-oriented programming, this can be written as follows:

```
class SortRoutineClass
  attr ord
  meth init(Order)
    ord:=Order
  end
  meth sort(InL OutL)
    % ... {@ord order(X Y $)} calculates order
  end
end
```

The order relation itself is written as follows:

```
proc {Order X Y ?B}
  B=(X<Y)
end
```

or as follows:

```
class OrderClass
  meth init skip end
  meth order(X Y B)
    B=(X<Y)
  end
end
```

Instantiating the sorting routine is then written as follows:

```
SortRoutine={NewSortRoutine Order}
```

or as follows:

```
SortRoutine={New SortRoutineClass init({New OrderClass init})}
```

Embellishments added by object-oriented programming

It is clear that procedure values and objects are closely related. Let us now compare higher-order and object-oriented programming more carefully. The main difference is that object-oriented programming “embellishes” higher-order programming. It is a richer abstraction that provides a collection of additional idioms beyond procedural abstraction:

- *Explicit state* can be defined and used easily.
- *Multiple methods* that share the same explicit state can be defined easily. Invoking an object picks one of them.
- *Classes* are provided, which define a set of methods and can be instantiated. Each instance has a fresh explicit state. If objects are like procedures, then classes are like procedures that return procedures.
- *Inheritance* is provided, to define new sets of methods from existing sets, by extending, modifying, and combining existing ones. Static and dynamic binding make this ability particularly rich.
- Different *degrees of encapsulation* can be defined between classes and objects. Attributes and methods can be private, public, protected or have some other, programmer-defined encapsulation.

It is important to note that these mechanisms do not provide any fundamentally new ability. They can be completely defined with higher-order programming, explicit state, and name values. On the other hand, the mechanisms are useful idioms that lead to a programming style that is often convenient.

Object-oriented programming is an abstraction that provides a rich notation to use any or all of these mechanisms together, whenever they are needed. This richness is a double-edged sword. On the one hand, it makes the abstraction particularly useful for many programming tasks. On the other hand, the abstraction has a complex semantics and is hard to reason about. For this reason, we recommend to use object orientation only in those cases when it significantly simplifies program structure, e.g., when there is a clear need for inheritance: the program contains a set of closely-related abstract data types. In other cases, we recommend to use simpler programming techniques.

Common limitations

The object system defined in this chapter is particularly close to higher-order programming. Not all object systems are so close. In particular, the following characteristics are often absent or cumbersome to use:

- *Classes as values*. They can be created at run-time, passed as arguments, and stored in data structures.

- *Full lexical scoping.* Full lexical scoping means that the language supports procedure values with external references. This allows a class to be defined inside the scope of a procedure or another class. Both Smalltalk-80 and Java support procedure values (with some restrictions). In Java they are instances of inner classes (i.e., nested classes). They are quite verbose due to the class syntax (see section quote).
- *First-class messages.* Usually, the labels of messages and methods both have to be known at compile time. The most general way to remove this restriction is to allow messages to be values in the language, which can be calculated at run time. Both Smalltalk-80 and Java provide this ability, although it is more verbose than the usual (static) method invocations. For example, here is a generic way to add “batching” to a class C:

```
class Batcher
  meth nil skip end
  meth `|`(M Ms) {self M} {self Ms} end
end
```

Mixing in the class Batcher adds batching ability to any other class:

```
C={New class $ from Counter Batcher end init(0)}
{C [inc(2) browse inc(3) inc(4)]}
```

Section 7.8.5 gives another way to add batching.

Some object-oriented languages, e.g., C++, do not support full higher-order programming because they cannot define procedure values with lexical scoping at run time (as explained in Section 3.6.1). In these languages, many of the abilities of higher-order programming can be obtained through encapsulation and inheritance, with a little effort from the programmer:

- A procedure value can be encoded as an object. The object’s attributes represent the procedure value’s external references and the method arguments are the procedure value’s arguments. When the object is created, its attributes are initialized with the external references. The object can be passed around and called just like a procedure value. With a little bit of discipline from the programmer, this allows for programming with procedure values, thus giving true higher-order programming.
- A generic procedure can be encoded as an abstract class. A generic procedure is one that takes procedure arguments and returns a specific procedure. For example a generic sorting routine can take a comparison operation for a given type and return a sorting routine that sorts arrays of that type. An abstract class is a class with undefined methods. The methods are defined in subclasses.

Encoding procedure values as objects

Let us give an example of how to encode a procedure value in a typical object-oriented language. Assume we have any statement $\langle \text{stmt} \rangle$. With procedural abstraction, we can define a procedure with **proc** {P} $\langle \text{stmt} \rangle$ **end** and execute it later as {P}. To encode this in our object system we have to know the external references of $\langle \text{stmt} \rangle$. Assume they are x and y . We define the following class:

```
class Proc
  attr x y
  meth init(X Y) @x=X @y=Y end
  meth apply X=@x Y=@y in  $\langle \text{stmt} \rangle$  end
end
```

The external references are represented by the stateless attributes x and y . We define P by doing $P = \{\text{New Proc init}(X\ Y)\}$ and call it with $\{P\ \text{apply}\}$. This encoding can be used in any object-oriented language. With it, we can use almost all the higher-order programming techniques of this book. It has two disadvantages with respect to procedures: it is more cumbersome to write and the external references have to be written explicitly.

7.5.3 Functional decomposition versus type decomposition

How do we organize an ADT that is based on a type $\langle T \rangle$ with subtypes $\langle T \rangle_1, \langle T \rangle_2, \langle T \rangle_3$ and includes a set of operations $\langle F \rangle_1, \dots, \langle F \rangle_n$? In declarative programming, Section 3.4.2 recommends to construct functions by following the type definition. In object-oriented programming, Section 7.4.2 recommends to construct inheritance hierarchies in similar fashion, also by following the type definition. Both sections give examples based on lists. Figure 7.23 gives a rough schematic overview comparing the two approaches. They result in very different program structures, which we call functional decomposition and type decomposition. In *functional decomposition*, each function definition is a self-contained whole, but the types are spread out over all functions. In *type decomposition*, each type is a self-contained whole, but the function definitions are spread out over all types. Which approach is better? It turns out that each has its uses:

- In functional decomposition, one can modify a function or add a new function without changing the other function definitions. However, changing or adding a type requires to modify all function definitions.
- In type decomposition, one can modify a type (i.e., a class) or add a new type (including by inheritance) without changing the other type definitions. However, changing or adding a function requires to modify all class definitions.

When designing a program, it is good to ask oneself what kind of modification is most important. If the type is relatively simple and there are a large number

Type definition

$$\langle T \rangle ::= \langle T \rangle_1 \mid \langle T \rangle_2 \mid \langle T \rangle_3$$

Operations

$$\langle F \rangle_1, \langle F \rangle_2, \dots, \langle F \rangle_n$$

Functional decomposition

```

fun { $\langle F \rangle_1$   $\langle T \rangle$  ...}
  case  $\langle T \rangle$ 
  of  $\langle T \rangle_1$  then
    ...
  [ $\rangle$ ]  $\langle T \rangle_2$  then
    ...
  [ $\rangle$ ]  $\langle T \rangle_3$  then
    ...
  end
end

fun { $\langle F \rangle_2$   $\langle T \rangle$  ...}
  ...
end

...

fun { $\langle F \rangle_n$   $\langle T \rangle$  ...}
  ...
end

```

Type decomposition

```

class  $\langle T \rangle$  ... end

class  $\langle T \rangle_1$  from  $\langle T \rangle$ 
  ...
  meth  $\langle F \rangle_1$ (...)
  ...
end
  meth  $\langle F \rangle_2$ (...)
  ...
end
  ...
  meth  $\langle F \rangle_n$ (...)
  ...
end
end

class  $\langle T \rangle_2$  from  $\langle T \rangle$  ... end

class  $\langle T \rangle_3$  from  $\langle T \rangle$  ... end

```

Figure 7.23: Functional decomposition versus type decomposition

of operations, then the functional approach is usually clearer. If the type is complex, with a relatively small number of operations, then the type approach can be clearer. There are techniques that combine some of the advantages of both approaches. See, e.g., [211], which explains some of these techniques and shows how to use them to build extensible compilers.

7.5.4 Should everything be an object?

In the area of object-oriented programming, the principle is often invoked that “everything should be an object”. Often, it is invoked without a precise understanding of what it means. For example, we saw someone define it on a mailing list as “one should send messages to everything” (whatever that means). Let us examine this principle and try to discover what it is really trying to say.

Strong objects

A sensible way to define the principle is as “all language entities should be instances of ADTs with as many generic properties as possible”. In its extreme form, this implies five properties: all language entities should be defined by classes, be extensible with inheritance, have a unique identity, encapsulate a state, and be accessed with a uniform syntax. The word “object” is sometimes used for entities with all these properties. To avoid confusion, we will call them *strong* objects. An object-oriented language is called *pure* if all its entities are strong objects.

The desire for purity can lead to good things. For example, many languages have the concept of “exception” to handle abnormal events during execution. It can be quite convenient for exceptions to be objects within an inheritance hierarchy. This allows classifying them into different categories, catching them only if they are of a given class (or its subclasses), and possibly changing them (adding information) if they are stateful.

Smalltalk-80 is a good example of a language for which purity was an explicit design goal [60, 89]. All data types in Smalltalk, including simple ones like integers, are objects. However, not everything in Smalltalk is an object; there is a concept called *block* that is a procedure value used for building control abstractions.

In most languages, not all entities are strong objects. Let us give some examples in Java. An integer in Java is a pure value; it is not defined by a class and does not encapsulate a state. An object in Java can have just **final** attributes, which means that it is stateless. An array in Java cannot be extended with inheritance. Arrays behave as if they were defined in a final class. We summarize this by saying that Java is object-oriented but not pure.

Should a language have only strong objects? It is clear that the answer is no, for many reasons. First, stateless entities can play an important role. With them, the powerful reasoning techniques of declarative programming become possible. For this reason, many language designs allow them. We cite Objective Caml [32], which has a functional core, and Java [10], which has immutable objects. In addition, stateless entities are essential for making transparent distributed programming practical (see Chapter 11). Second, not all entities need a unique identity. For example, structured entities such as tuples in a database are identified by their contents, not by their names. Third, the simplicity of a uniform syntax is illusory, as we explain below.

We seem to be removing each property one by one. We are left with two principles: all language entities should be instances of ADTs and uniformity among ADTs should be exploited when it is reasonable. Some ADTs will have all the properties of strong objects; others will have only some of these properties but also have some other, completely different properties. These principles are consistent with the use of multiple computation models advocated in this book. Building a system consists primarily in designing abstractions and realizing them

as ADTs.

Objects and explicit state

Let us elaborate why stateless objects are a good idea. Given a particular object, how can one predict its future behavior? It depends on two factors:

1. Its internal state, which potentially depends on all past calls. These calls can be done from many parts of the program.
2. Its textual definition, which depends on all classes it inherits from. These classes can be defined in many places in the program text.

We see that the semantics of an object is spread out over both time and space. This makes an object harder to understand than a function. The semantics of a function is all concentrated in one place, namely the textual definition of the function. The function does not have a history; it depends only on its definition and its arguments.

We give an example that shows why objects are harder to program with when they have state. Assume that we are doing arithmetic with the IEEE floating point standard and that we have implemented the standard completely. This means, for example, that we can change the rounding mode of arithmetic operations during execution (round to nearest even, round up, round down, etc.). If we do not use this ability carefully, then whenever we do an addition $x+y$ we have no idea what it will do unless we have followed the whole execution. Any part of the program could have changed the rounding mode. This can wreak havoc on numeric methods, which depend on predictable rounding to get good results. One solution is for all numeric methods to set the rounding method initially and on each external call.

To avoid this problem as much as possible, the language should not favor explicit state and inheritance. That is, *not* using them should be easy. For inheritance, this is almost never a problem, since it is always harder to use it than to avoid it. For explicit state, it depends on the language. In the object-oriented model of this chapter, defining (stateless) functions is actually slightly easier than defining (stateful) objects. Objects need to be defined as instances of classes, which themselves are defined with a **class-end** wrapping one or more **meth-end** declarations. Functions just need **fun-end**.

In popular object-oriented languages, unfortunately, explicit state is almost always the default and functions are usually syntactically cumbersome. In Smalltalk, all attributes are stateful but function values can be defined easily. In Java, there is no syntactic support for functions and object attributes are stateful unless declared to be **final**.

Uniform object syntax

A language's syntax should help and not hinder programmers in designing, writing, and reasoning about programs. An important principle in syntax design is *form mirrors content*. Differences in semantics should be visible as differences in syntax and vice versa. For example, the while loop as used in many languages has a syntax similar to **while** $\langle \text{expr} \rangle$ **do** $\langle \text{stmt} \rangle$. By writing $\langle \text{expr} \rangle$ before $\langle \text{stmt} \rangle$, the syntax reflects the fact that the condition $\langle \text{expr} \rangle$ is evaluated before executing $\langle \text{stmt} \rangle$. If $\langle \text{expr} \rangle$ is false, then $\langle \text{stmt} \rangle$ is not executed at all. The Cobol language does things differently. It has the perform loop, which can be written **perform** $\langle \text{stmt} \rangle$ **until** $\langle \text{expr} \rangle$. This syntax is misleading since $\langle \text{expr} \rangle$ is tested before $\langle \text{stmt} \rangle$ yet is written after $\langle \text{stmt} \rangle$. The perform loop's semantics are **while not** $\langle \text{expr} \rangle$ **do** $\langle \text{stmt} \rangle$.

Should all operations on language entities have the same syntax? This does not necessarily improve readability. For example, Scheme has a uniform syntax which does not necessarily make it more readable. We find that a uniform syntax just moves the richness of the language away from the syntax and into the names of objects and classes. This adds a second layer of syntax, making the language more verbose. Let us give an example taken from symbolic programming languages. Stateless values can be given a very natural, compact syntax. A list value can be created just by mentioning it, e.g.:

```
LV=[1 2 3]
```

This is approximately the syntax used by languages that support symbolic programming, such as Prolog, Haskell, Erlang, and their relatives. This contrasts with the use of a uniform object syntax:

```
ListClass *lv= new ConsClass(1, new ConsClass(2,
                                     new ConsClass(3, new NilClass())));
```

This is C++ syntax, which is similar to Java syntax. For decomposing a list value, there is another natural notation using pattern matching:

```
case LV of X|LV2 then ... end
```

Pattern matching is commonly used in symbolic languages. This is also cumbersome to do in a uniform object syntax. There is a further increase in verbosity when doing concurrent programming in the object syntax. This is because the uniform syntax requires explicit synchronization. This is not true for the **case** syntax above, which is sufficient for concurrent programming if the computation model does implicit dataflow synchronization.

Another, more realistic example is the graphical user interface tool of Chapter 10. Inspired by this tool, Christophe Ponsard built a Java prototype of a similar tool. The Java version is more cumbersome to use than the Oz version, primarily because Java has no syntactic support for record values. Unfortunately, this verbosity is an inherent property of Java. There is no simple way around it.

7.6 Implementing the object system

The complete object system can be implemented in a straightforward way from the declarative stateful computation model. In particular, the main characteristics come from the combination of higher-order programming with explicit state. With this construction, you will understand objects and classes completely.

While the construction of this section works well and is reasonably efficient, a real implementation will add optimizations to do even better. For example, a real implementation can make an object invocation be as fast as a procedure call. This section does not give these optimizations.

7.6.1 Abstraction diagram

The first step in understanding how to build an object system is to understand how the different parts are related. Object-oriented programming defines a hierarchy of abstractions that are related to each other by a kind of “specification-implementation” relationship. There are many variations on this hierarchy. We give a simple one that has most of the main ideas. Here are the abstractions, in order from most concrete to most abstract:

- **Running object.** A running object is an active invocation of an object. It associates a thread to an object. It contains a set of environment frames (the part of the thread’s stack that is created while executing the object) as well as an object.
- **Object.** An object is a procedure that encapsulates an explicit state (a cell) and a set of procedures that reference the state.
- **Class.** A class is a wrapped record that encapsulates a set of procedures named by literals and a set of attributes, which are just literals. The procedures are called *methods*. Methods take a state as argument for each attribute and modify that state. Methods can only call each other indirectly, through the literals that name them. Often the following distinction is useful:
 - **Abstract class.** An abstract class is a class in which some methods are called that have no definition in the class.
 - **Concrete class.** A concrete class is a class in which all methods that are called are also defined.

If first-class messages are supported by the language, then invocations of the form $\{\text{Obj } M\}$ are possible where M is calculated at run time. If such an invocation exists in the program, then the distinction between abstract and concrete class disappears in the program (although it may still exist conceptually). Executing the invocation $\{\text{Obj } M\}$ raises an exception if M does not exist in Obj .

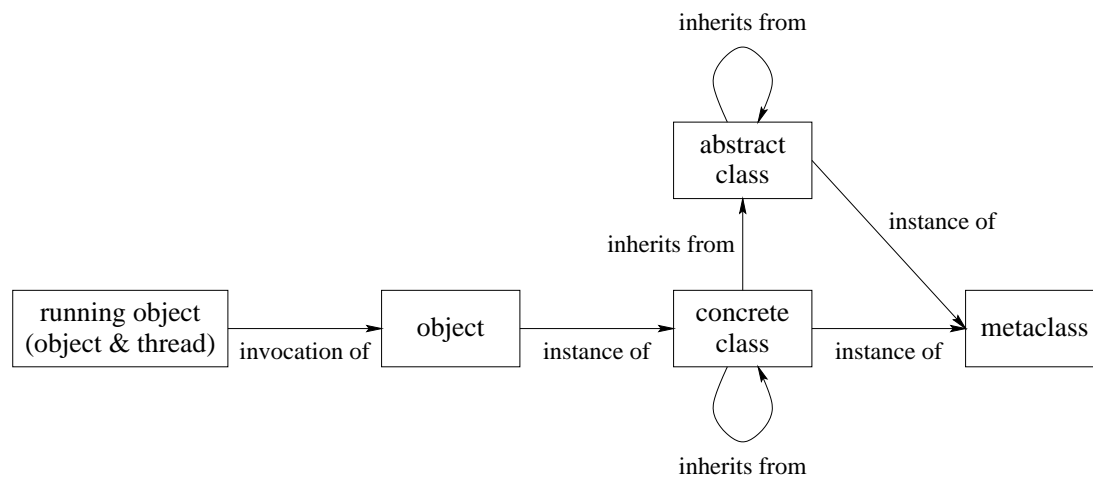


Figure 7.24: Abstractions in object-oriented programming

- **Metaclass.** A metaclass is a class with a particular set of methods that correspond to the basic operations of a class, for example: object creation, inheritance policy (which methods to inherit), method call, method return, choice of method to call, attribute assignment, attribute access, self call. Writing these methods allows to customize the semantics of objects.

Figure 7.24 shows how these concepts are related. There are three relationships, “invocation of”, “instance of”, and “inherits from”. These relationships have the following intuitive meanings:

- A running object is created when a thread invokes an object. The running object exists until the thread’s execution leaves it. Multiple invocations of the same object can exist simultaneously.
- An object can be created as an instance of a class. If the object system distinguishes between abstract and concrete classes, then it is usually only possible to create instances of concrete classes. The object exists forever.² The object encapsulates a cell that was created especially for it. Multiple instances of the same class can exist simultaneously.
- A class can be created that inherits from a list of other classes. The new class exists forever. Inheritance takes a set of methods and a list of classes and returns a new class with a new set of methods. Multiple classes that inherit from the same class can exist simultaneously. If one class can inherit from several classes, then we have multiple inheritance. Otherwise, if one class can inherit only from one class, we have single inheritance.

²In practice, until the actively running program loses all references to it. At that point, garbage collection can reclaim its memory and finalization can perform a last action, if necessary.

```
class Counter
  attr val
  meth init(Value)
    val:=Value
  end
  meth inc(Value)
    val:=@val+Value
  end
  meth browse
    {Browse @val}
  end
end
```

Figure 7.25: An example class `Counter` (again)

- A class can be created as an instance of a metaclass. The new class exists forever. The basic operations of the class are defined by particular methods of the metaclass. Multiple instances of the same metaclass can exist simultaneously.

7.6.2 Implementing classes

We first explain the class linguistic abstraction. The `Counter` class of Figure 7.25 is translated internally into the definition of Figure 7.26. This figure shows that a class is simply a value, a record, that is protected from snooping because of the wrapper `Wrap` (see Section 3.7.5). (Later, when the class is used to create objects, it will be unwrapped with the corresponding `Unwrap`.) The class record contains:

- A set of methods in a method table. Each method is a three-argument procedure that takes a message `M`, which is always a record, an extra parameter `S` representing the state of the current object, and `Self`, which references the object itself.
- A set of attribute names, giving the attributes that each class instance (object) will possess. Each attribute is a stateful cell that is accessed by the attribute name, which is either an atom or an Oz name.

This example is slightly simplified because it does not show how to support static binding (see exercises). The `Counter` class has a single attribute accessed by the atom `val`. It has a method table with three methods accessed through the features `browse`, `init`, and `inc`. As we can see, the method `init` assigns the value `Value` to the attribute `val`, the method `inc` increments the attribute `val`, and the method `browse` browses the current value of `val`.

```

declare Counter
local
  Attrs = [val]
  MethodTable = m(browse:MyBrowse init:Init inc:Inc)
  proc {Init M S Self}
    init(Value)=M
  in
    (S.val):=Value
  end
  proc {Inc M S Self}
    X
    inc(Value)=M
  in
    X=@(S.val) (S.val):=X+Value
  end
  proc {MyBrowse M S Self}
    browse=M
    {Browse @(S.val)}
  end
in
  Counter = {Wrap c(methods:MethodTable attrs:Attrs)}
end

```

Figure 7.26: An example of class construction

7.6.3 Implementing objects

We can use the class `Counter` to create objects. Figure 7.27 shows a generic function `New` that creates an object from any class. It starts by unwrapping the class. It then creates an object state, a record, from the attributes of the class. It initializes each field of this record to a cell (with an unbound initial value). This uses the iterator `Record.forAll` to iterate over all fields of a record.

The object `Obj` returned by `New` is a one-argument procedure. When called as `{Obj M}`, it looks up and calls the procedure corresponding to `M` in the method table. Because of lexical scoping, the object state is visible only within `Obj`. One can say that `Obj` is a procedure that *encapsulates* the state.

The definition of Figure 7.27 works correctly, but it may not be the most efficient way to implement objects. An actual system can use a different, more efficient implementation as long as it behaves in the same way. For example, the Mozart system uses an implementation in which object invocations are almost as efficient as procedure calls [74, 76].

The proof of the pudding is in the eating. Let us verify that the class works as claimed. We now create the `Counter` class and try out `New` as follows:

```

C={New Counter init(0)}
{C inc(6)} {C inc(6)}

```

```

fun {New WClass InitialMethod}
  State Obj Class={Unwrap WClass}
in
  State = {MakeRecord s Class.attrs}
  {Record.forAll State proc {$ A} {NewCell _ A} end}
  proc {Obj M}
    {Class.methods.{Label M} M State Obj}
  end
  {Obj InitialMethod}
  Obj
end

```

Figure 7.27: An example of object construction

```
{C browse}
```

This behaves in exactly the same way as the example of Section 7.2.1.

7.6.4 Implementing inheritance

Inheritance calculates a new class record starting from existing class records, which are combined according to the inheritance rules given in Section 7.3.1. Inheritance can be defined by the function `From`, where the call `C={From C1 C2 C3}` returns a new class record whose base definition is `C1` and which inherits from `C2` and `C3`. It corresponds to the following class syntax:

```

class C from C2 C3
  ... % The base class C1
end

```

Figure 7.28 shows the definition of `From`. It uses the set operations in the `Set` module, which can be found on the book's Web site. `From` first checks the method tables and attribute lists for conflicts. If a duplicate method label or attribute is found in `C2` and `C3` that is not overridden by `C1`, then an exception is raised. Then `From` constructs the new method table and attribute lists. Overriding is handled properly by the `Adjoin` function on the method tables (see Appendix B.3.2). The definition is slightly simplified because it does not handle static binding and because it assumes that there are exactly two superclasses.

7.7 The Java language (sequential part)

Java is a concurrent object-oriented language with a syntax that resembles C++. This section gives a brief introduction to the sequential part of Java. We explain how to write a simple program, how to define classes, and how to use inheritance. We defer talking about concurrency in Java until Chapter 8. We do not talk

```

fun {From C1 C2 C3}
  c(methods:M1 attrs:A1)={Unwrap C1}
  c(methods:M2 attrs:A2)={Unwrap C2}
  c(methods:M3 attrs:A3)={Unwrap C3}
  MA1={Arity M1}
  MA2={Arity M2}
  MA3={Arity M3}
  ConfMeth={Minus {Inter MA2 MA3} MA1}
  ConfAttr={Minus {Inter A1 A2} A3}
in
  if ConfMeth\=nil then
    raise illegalInheritance(methConf:ConfMeth) end
  end
  if ConfAttr\=nil then
    raise illegalInheritance(attrConf:ConfAttr) end
  end
  {Wrap c(methods:{Adjoin {Adjoin M2 M3} M1}
    attrs:{Union {Union A2 A3} A1})}
end

```

Figure 7.28: Implementing inheritance

about the reflection package, which lets one do much of what the object system of this chapter can do (although in a more verbose way).

Java is almost a pure object-oriented language, i.e., almost everything is an object. Only a small set of primitive types, namely integers, floats, booleans, and characters, are not objects. Java is a relatively clean language with a relatively simple semantics. Despite the syntactic similarity, there is a major difference in language philosophy between Java and C++ [184, 10]. C++ gives access to the machine representation of data and a direct translation to machine instructions. It also has manual memory management. Because of these properties, C++ is often suitable as a replacement for assembly language. In contrast, Java hides the representation of data and does automatic memory management. It supports distributed computing on multiple platforms. It has a more sophisticated object system. These properties make Java better for general-purpose application development.

7.7.1 Computation model

Java consists of statically-typed object-oriented programming with classes, passive objects, and threads. The Java computation model is close to the shared-state concurrent model, minus dataflow variables, triggers, and names. Parameter passing is done by value, both for primitive types and object references. Newly-declared variables are given a default initial value that depends on their type. There is support for single assignment: variables and object attributes can be