

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

Figure 7.1: An example class Counter (with **class** syntax)

setting, when multiple threads use the objects. In particular, object locking is explained there.

7.2 Classes as complete ADTs

The heart of the object concept is controlled access to encapsulated data. The behavior of an object is specified by a *class*. In the most general case, a class is an incremental definition of an ADT, that defines the ADT as a modification of other ADTs. There is a rich set of concepts for defining classes. We classify these concepts into two sets, according as they permit the class to define an ADT completely or incrementally:

- **Complete ADT definition.** These are all the concepts that permit a class, taken by itself, to define an ADT. There are two sets of concepts:
 - Defining the various elements that make up a class (Section 7.2.3), namely methods, attributes, and properties. Attributes can be initialized in several ways, per object or per class (Section 7.2.4).
 - Taking advantage of dynamic typing. This gives first-class messages (Section 7.2.5) and first-class attributes (Section 7.2.6). This allows powerful forms of polymorphism that are difficult or impossible to do in statically-typed languages. This increased freedom comes with an increased responsibility of the programmer to use it correctly.
- **Incremental ADT definition.** These are all the concepts related to inheritance, that is, they define how a class is related to existing classes. They are given in Section 7.3.

```

local
  proc {Init M S}
    init(Value)=M in (S.val):=Value
  end
  proc {Browse2 M S}
    {Browse @(S.val)}
  end
  proc {Inc M S}
    inc(Value)=M in (S.val):=@(S.val)+Value
  end
in
  Counter=c(attrs:[val]
             methods:m(init:Init browse:Browse2 inc:Inc))
end

```

Figure 7.2: Defining the Counter class (without syntactic support)

7.2.1 An example

To see how classes and objects work in the object system, let us define an example class and use it to create an object. We assume that the language has a new construct, the **class** declaration. We assume that classes are first-class values in the language. This lets us use a **class** declaration as either statement or expression, in similar manner to a **proc** declaration. Later on in the chapter, we will see how to define classes in the kernel language of the stateful model. This would let us define **class** as a linguistic abstraction.

Figure 7.1 defines a class referred to by the variable Counter. This class has one *attribute*, *val*, that holds a counter's current value, and three *methods*, *init*, *browse*, and *inc*, for initializing, displaying, and incrementing the counter. The attribute is assigned with the `:=` operator and accessed with the `@` operator. This seems quite similar to how other languages would do it, modulo a different syntax. But appearances can be deceiving!

The declaration of Figure 7.1 is actually executed at run time, i.e., it is a statement that creates a class value and binds it to Counter. Replace “Counter” by “\$” and the declaration can be used in an expression. Putting this declaration at the head of a program will declare the class before executing the rest, which is familiar behavior. But this is not the only possibility. The declaration can be put anywhere that a statement can be. For example, putting the declaration inside a procedure will create a new and distinct class each time the procedure is called. Later on we will use this possibility to make parameterized classes.

Let us create an object of class Counter and do some operations with it:

```

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

```

This creates the counter object C with initial value 0, increments it twice by 6,

```

fun {New Class Init}
  Fs={Map Class.attrs fun {$ X} X#{NewCell _} end}
  S={List.toRecord state Fs}
  proc {Obj M}
    {Class.methods.{Label M} M S}
  end
in
  {Obj Init}
  Obj
end

```

Figure 7.3: Creating a Counter object

and then displays the counter's value. The statement `{C inc(6)}` is called an object application. The message `inc(6)` is sent to the object, which invokes the corresponding method. Now try the following:

```

local X in {C inc(X)} X=5 end
{C browse}

```

This displays nothing at all! The reason is that the object application

```
{C inc(X)}
```

blocks inside the method `inc`. Can you see exactly where? Now try the following variation:

```

declare S in
local X in thread {C inc(X)} S=unit end X=5 end
{Wait S} {C browse}

```

Things now work as expected. We see that dataflow execution keeps its familiar behavior when used with objects.

7.2.2 Semantics of the example

Before going on to describe the additional abilities of classes, let us give the semantics of the Counter example. It is a simple application of higher-order programming with explicit state. The semantics we give here is slightly simplified; it leaves out the abilities of **class** that are not used in the example (such as inheritance and **self**). Section 7.6 gives the full semantics.

Figure 7.2 shows what Figure 7.1 does by giving the definition of the class Counter in the stateful model without any **class** syntax. We can see that according to this definition, a class is simply a record containing a set of attribute names and a set of methods. An attribute name is a literal. A method is a procedure that has two arguments, the message and the object state. In each method, assigning to an attribute ("`val:=`") is done with a cell assignment and accessing an attribute ("`@val`") is done with a cell access.

$\langle \text{statement} \rangle$	$::=$ class $\langle \text{variable} \rangle$ { $\langle \text{classDescriptor} \rangle$ } { meth $\langle \text{methHead} \rangle$ ['=' $\langle \text{variable} \rangle$] ($\langle \text{inExpression} \rangle$ $\langle \text{inStatement} \rangle$) end } end lock [$\langle \text{expression} \rangle$ then] $\langle \text{inStatement} \rangle$ end $\langle \text{expression} \rangle$ ' := ' $\langle \text{expression} \rangle$ $\langle \text{expression} \rangle$ ' , ' $\langle \text{expression} \rangle$...
$\langle \text{expression} \rangle$	$::=$ class $\text{' \$ '}$ { $\langle \text{classDescriptor} \rangle$ } { meth $\langle \text{methHead} \rangle$ ['=' $\langle \text{variable} \rangle$] ($\langle \text{inExpression} \rangle$ $\langle \text{inStatement} \rangle$) end } end lock [$\langle \text{expression} \rangle$ then] $\langle \text{inExpression} \rangle$ end $\langle \text{expression} \rangle$ ' := ' $\langle \text{expression} \rangle$ $\langle \text{expression} \rangle$ ' , ' $\langle \text{expression} \rangle$ ' @ ' $\langle \text{expression} \rangle$ self ...
$\langle \text{classDescriptor} \rangle$	$::=$ from { $\langle \text{expression} \rangle$ }+ prop { $\langle \text{expression} \rangle$ }+ attr { $\langle \text{attrInit} \rangle$ }+
$\langle \text{attrInit} \rangle$	$::=$ ([' ! '] $\langle \text{variable} \rangle$ $\langle \text{atom} \rangle$ unit true false) [' : ' $\langle \text{expression} \rangle$]
$\langle \text{methHead} \rangle$	$::=$ ([' ! '] $\langle \text{variable} \rangle$ $\langle \text{atom} \rangle$ unit true false) [' (' { $\langle \text{methArg} \rangle$ } [' ... '] ') '] [' = ' $\langle \text{variable} \rangle$]
$\langle \text{methArg} \rangle$	$::=$ [$\langle \text{feature} \rangle$ ' : '] ($\langle \text{variable} \rangle$ ' _ ' $\text{' \$ '}$) [' <= ' $\langle \text{expression} \rangle$]

Table 7.1: Class syntax

Figure 7.3 defines the function `New` which is used to create objects from classes. This function creates the object state, defines a one-argument procedure `Obj` that is the object, and initializes the object before returning it. The object state `S` is a record holding one cell for each attribute. The object state is hidden inside `Obj` by lexical scoping.

7.2.3 Defining classes

A *class* is a data structure that defines an object's internal state (attributes), its behavior (methods), the classes it inherits from, and several other properties and operations that we will see later on. More generally, a *class* is a data structure that describes an ADT and gives its partial or total implementation. Table 7.1 gives the syntax of classes. There can be any number of objects of a given class. They are called *instances* of the class. These objects have different identities

and can have different values for their internal state. Otherwise, all objects of a given class behave according to the class definition. An object `Obj` is called with the syntax `{Obj M}`, where `M` is a record that defines the message. Calling an object is also called *sending a message* to the object. This terminology exists for historical reasons; we do not recommend it since it is easily confused with sending a message on a communication channel. An object invocation is synchronous, like a procedure's. The invocation returns only when the method has completely executed.

A class defines the constituent parts that each instance will have. In object-oriented terminology, these parts are often called *members*. There are three kinds of members:

- **Attributes** (declared with the keyword “**attr**”). An attribute, is a cell that contains part of the instance's state. In object-oriented terminology, an attribute is often called an *instance variable*. The attribute can contain any language entity. The attribute is visible only in the class definition and all classes that inherit from it. Every instance has a separate set of attributes. The instance can update an attribute with the following operations:
 - An assignment statement: $\langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$. This assigns the result of evaluating $\langle \text{expr} \rangle_2$ to the attribute whose name is obtained by evaluating $\langle \text{expr} \rangle_1$.
 - An access operation: $@\langle \text{expr} \rangle$. This accesses the attribute whose name is obtained by evaluating $\langle \text{expr} \rangle$. The access operation can be used in any expression that is lexically inside the class definition. In particular, it can be used inside of procedures that are defined inside the class.
 - An exchange operation. If the assignment $\langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$ is used as an expression, then it has the effect of an exchange. For example, consider the statement $\langle \text{expr} \rangle_3 = \langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$. This first evaluates the three expressions. Then it unifies $\langle \text{expr} \rangle_3$ with the content of the attribute $\langle \text{expr} \rangle_1$ and atomically sets the new content to $\langle \text{expr} \rangle_2$.
- **Methods** (declared with the keyword “**meth**”). A method is a kind of procedure that is called in the context of a particular object and that can access the object's attributes. The method consists of a head and body. The head consists of a label, which must be an atom or a name, and a set of arguments. The arguments must be distinct variables, otherwise there is a syntax error. For increased expressiveness, method heads are similar to patterns and messages are similar to records. Section 7.2.5 explains the possibilities.
- **Properties** (declared with the keyword “**prop**”). A property modifies how an object behaves. For example:

- The property `locking` creates a new lock with each object instance. The lock can be accessed inside the class with the `lock ... end` construct. Locking is explained in Chapter 8.
- The property `final` makes the class be a final class, i.e., it cannot be extended with inheritance. Inheritance is explained in Section 7.3.

Attributes and methods are literals. If they are defined with atom syntax, then they are atoms. If they are defined with identifier syntax (e.g., capitalized), then the system will create new names for them. The scope of these names is the class definition. Using names gives a fine-grained control over object security, as we will see. Section 7.2.4 shows how to initialize attributes.

In addition to having these kinds of members, Section 7.3 shows how a class can *inherit* members from other classes. An instance of a class is created with the operation `New`:

```
MyObj={New MyClass init}
```

This creates a new object `MyObj` of class `MyClass` and passes `init` as the first message to the object. This message is used to initialize the object.

7.2.4 Initializing attributes

Attributes can be initialized in two ways: per instance or per class.

- **Per instance.** An attribute can be given a different initial value per instance. This is done by not initializing it in the class definition. For example:

```
class OneApt
  attr streetName
  meth init(X) @streetName=X end
end
Apt1={New OneApt init(drottninggatan)}
Apt2={New OneApt init(rueNeuve)}
```

Each instance, including `Apt1` and `Apt2`, will initially reference a different unbound variable. Each variable can be bound to a different value.

- **Per class.** An attribute can be given a value that is the same for all instances of a class. This is done by initializing it with “:” in the class definition. For example:

```
class YorkApt
  attr
    streetName:york
    streetNumber:100
    wallColor:_
    floorSurface:wood
```

```

    meth init skip end
end
Apt3={New YorkApt init}
Apt4={New YorkApt init}

```

All instances, including Apt3 and Apt4, have the same initial values for all four attributes. This includes wallColor, even though the initial value is an unbound variable. All instances refer to the *same* unbound variable. It can be bound by binding it in one of the instances, e.g., @wallColor=white. Then all instances will see this value. Be careful not to confuse the two operations @wallColor=white and wallColor:=white.

- **Per brand.** This is another way to use the per-class initialization. A *brand* is a set of classes that are related in some way, but not by inheritance. An attribute can be given a value that is the same for all members of a brand by initializing with the same variable for all members. For example:¹

```

L=linux
class RedHat
  attr ostyle:L
end
class SuSE
  attr ostyle:L
end
class Debian
  attr ostyle:L
end

```

Each instance of each class will be initialized to the same value.

Since an attribute is stateful, its initial reference can be changed.

7.2.5 First-class messages

The principle is simple: messages are *records* and method heads are *patterns* that match a record. As a consequence, the following possibilities exist for object calls and method definitions:

- In the *object call* {Obj M}, the following is possible:
 1. **Static record as message.** In the simplest case, M is a record that is known at compile time, e.g., like in the object call {Counter inc(X)}.
 2. **Dynamic record as message.** It is possible to call {Obj M} where M is a variable that references a record that is calculated at run time.

¹With apologies to all omitted Linux distributions.

Because of dynamic typing, it is possible to create new record types at run time (e.g., with `Adjoin` or `List.toRecord`).

- In the *method definition*, the following is possible:
 1. **Fixed argument list.** The method head is a pattern consisting of a label followed by a series of arguments in parentheses. For example:

```
meth foo(a:A b:B c:C)
    % Method body
end
```

The method head `foo(a:A b:B c:C)` is a pattern that must match the message exactly, i.e., the label `foo` and arity `[a,b,c]` must match. The features (`a`, `b`, and `c`) can be given in any order. A class can only have one method definition with a given label, otherwise there is a syntax error.

2. **Flexible argument list.** The method head is the same as in the fixed argument list except it ends in “...”. For example:

```
meth foo(a:A b:B c:C ...)
    % Method body
end
```

The “...” in the method head means that any message is accepted if it has at least the listed arguments. This means the same as the “...” in patterns, e.g., in a **case** statement. The given label must match the message label and the given arity must be a subset of the message arity.

3. **Variable reference to method head.** The whole method head is referenced by a variable. This is particularly useful with flexible argument lists, but it can also be used with a fixed argument list. For example:

```
meth foo(a:A b:B c:C ...)=M
    % Method body
end
```

The variable `M` references the full message as a record. The scope of `M` is the method body.

4. **Optional argument.** A default is given for an argument. The default is used if the argument is not in the message. For example:

```
meth foo(a:A b:B<=V)
    % Method body
end
```

The “`<=V`” in the method head means that the field `b` is optional in the object call. That is, the method can be called either with or

without the field. With the field, an example call is `foo(a:1 b:2)`, which ignores the expression `v`. Without the field, an example call is `foo(a:1)`, for which the actual message received is `foo(a:1 b:v)`.

5. **Private method label.** We said that method labels can be names. This is denoted by using a variable identifier:

```
meth A(bar:X)
    % Method body
end
```

The method `A` is bound to a fresh name when the class is defined. `A` is initially visible only in the scope of the class definition. If it has to be used elsewhere in the program, it must be passed explicitly.

6. **Dynamic method label.** It is possible to calculate a method label at run time, by using an the escaped variable identifier. This is possible because class definitions are executed at run time. The method label has to be known when the class definition is executed. For example:

```
meth !A(bar:X)
    % Method body
end
```

causes the method label to be whatever the variable `A` was bound to. The variable must be bound to an atom or a name. By using names, this technique can make methods secure (see Section 7.3.3).

7. **The otherwise method.** The method head with label `otherwise` is a catchall that accepts any message for which no other method exists. For example:

```
meth otherwise(M)
    % Method body
end
```

A class can only have one method with head `otherwise`, otherwise there is a syntax error. This method must have just one argument, otherwise a run-time “arity mismatch” error is given. If this method exists, then the object accepts *any* message. If no method is defined for the message, then the `otherwise(M)` method is called with the full message in `M` as a record. This mechanism allows to implement *delegation*, an alternative to inheritance explained in Section 7.3.4. This mechanism also allows making wrappers around method calls.

All these possibilities are covered by the syntax of Table 7.1. In general, for the call `{Obj M}`, the compiler tries to determine statically what the object `Obj` and the method `M` are. If it can, then it compiles a very fast specialized call instruction. If it cannot, then it compiles a general object call instruction. The general instruction uses caching. The first call is slower, because it looks up the

method and caches the result. Subsequent calls find the method in the cache and are almost as fast as the specialized call.

7.2.6 First-class attributes

Attribute names can be calculated at run time. For example, it is possible to write methods to access and assign any attributes:

```
class Inspector
  meth get(A ?X)
    X=@A
  end
  meth set(A X)
    A:=X
  end
end
```

The `get` method can access any attribute and the `set` method can assign any attribute. Any class that has these methods will open up its attributes for public use. This ability is dangerous for programming but can be very useful for debugging.

7.2.7 Programming techniques

The class concept we have introduced so far gives a convenient syntax for defining ADTs with encapsulated state and multiple operations. The **class** statement defines a class value, which can be instantiated to give objects. In addition to having a convenient syntax, class values as defined here keep all the advantages of procedure values. All of the programming techniques for procedures also apply for classes. Classes can have external references just like procedure values. Classes are compositional: classes can be nested within classes. They are compatible with procedure values: classes can be nested within procedures and vice versa. Classes are not this flexible in all object-oriented languages; usually some limits are imposed, as explained in Section 7.5.

7.3 Classes as incremental ADTs

As explained before, the main addition that object-oriented programming adds to component-based programming is inheritance. Object-oriented programming allows to define a class incrementally, by extending existing classes. It is not enough to say which classes are extended; to properly define a new ADT more concepts are needed. Our model includes three sets of concepts:

- The first is inheritance itself (Section 7.3.1), which defines which preexisting classes are extended.

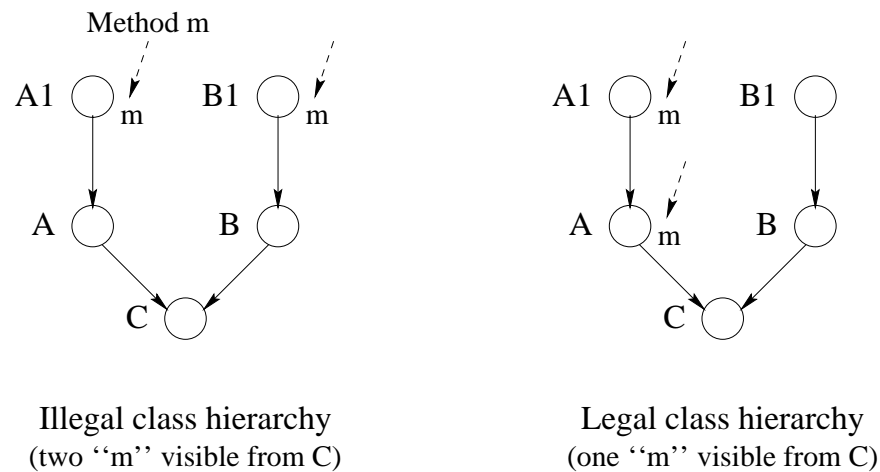


Figure 7.4: Illegal and legal class hierarchies

- The second is method access control (Section 7.3.2), which defines how to access particular methods both in the new class and in the preexisting classes. It is done with static and dynamic binding and the concept of **self**.
- The third is encapsulation control (Section 7.3.3), which defines what part of a program can see a classes’ attributes and methods.

In addition, the model can use first-class messages to implement *delegation*, a completely different way to define ADTs incrementally (see Section 7.3.4).

7.3.1 Inheritance

Inheritance is a way to construct new classes from existing classes. It defines what attributes and methods are available in the new class. We will restrict our discussion of inheritance to methods. The same rules apply to attributes. The methods available in a class C are defined through a precedence relation on the methods that appear in the class hierarchy. We call this relation the *overriding relation*:

- A method in class C overrides any method with the same label in all of C’s superclasses.

Classes may inherit from one or more classes, which appear after the keyword **from** in the class declaration. A class that inherits from exactly one class is said to use *single inheritance* (sometimes called *simple inheritance*). Inheriting from more than one class is called *multiple inheritance*. A class B is a *superclass* of a class A if:

- B appears in the **from** declaration of A, or
- B is a superclass of a class appearing in the **from** declaration of A.

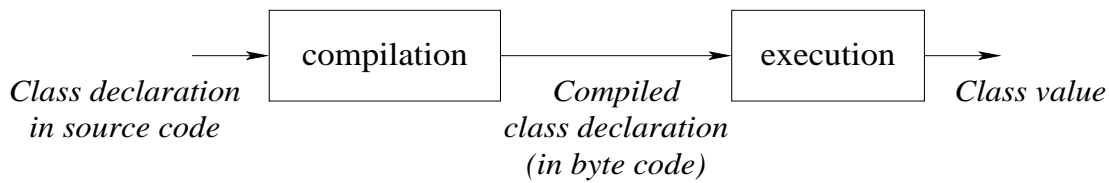


Figure 7.5: A class declaration is an executable statement

A class hierarchy with the superclass relation can be seen as a directed graph with the current class being the root. The edges are directed towards the subclasses. There are two requirements for the inheritance to be legal. First, the inheritance relation is directed and acyclic. So the following is not allowed:

```
class A from B ... end
class B from A ... end
```

Second, after striking out all overridden methods, each remaining method should have a unique label and is defined in only one class in the hierarchy. Hence, class C in the following example is illegal because the two methods labeled m remain:

```
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
```

Figure 7.4 shows this hierarchy and a slightly different one that is legal. The class C below is also illegal, since two methods m are available in C:

```
class A meth m(...) ... end end
class B meth m(...) ... end end
class C from A B end
```

Run time is all there is

If a program containing the declaration of class C is compiled in Mozart then the system will not complain. It is only when the program *executes* the declaration that the system will raise an exception. If the program does not execute the declaration then no exception is raised. For example, a program that contains the following source code:

```
fun {StrangeClass}
  class A meth foo(X) X=a end end
  class B meth foo(X) X=b end end
  class C from A B end
in C end
```

can be successfully compiled and executed. Its execution has the effect of *defining the function* StrangeClass. It is only during the call {StrangeClass} that an

```
class Account
  attr balance:0
  meth transfer(Amt)
    balance:=@balance+Amt
  end
  meth getBal(Bal)
    Bal=@balance
  end
  meth batchTransfer(AmtList)
    for A in AmtList do {self transfer(A)} end
  end
end
```

Figure 7.6: An example class Account

exception will be raised. This “late error detection” is not just a property of class declarations. It is a general property of the Mozart system that is a consequence of the dynamic nature of the language. Namely, there is no distinction between compile time and run time. The object system shares this dynamic nature. For example, it is possible to define classes whose method labels are calculated at run time (see Section 7.2.5).

The Mozart system blurs the distinction between run time and compile time, to the point where everything is run time. The compiler is part of the run-time system. A class declaration is an executable statement. Compiling and executing it creates a class, which is a value in the language (see Figure 7.5). The class value can be passed to `New` to create an object.

A programming system does not strictly need to distinguish between compile time and run time. The distinction is simply a way to help the compiler perform certain kinds of optimization. Most mainstream languages, including C++ and Java, make this distinction. Typically, a few operations (like declarations) can be executed only at compile time, and all other operations can be executed only at run time. The compiler can then execute all declarations at the same time, without any interference from the program’s execution. This allows it to do more powerful optimizations when generating code. But it greatly reduces the flexibility of the language. For example, genericity and instantiation are no longer available to the programmer as general tools.

Because of Mozart’s dynamic nature, the role of the compiler is very small. Since the compiler does not actually execute any declarations (it just converts them to executable statements), it needs very little knowledge of the language semantics. The compiler does in fact have some knowledge of language semantics, but this is an optimization that allows earlier detection of some errors and more efficient compiled code. More knowledge could be added to the compiler, for example to detect class hierarchy errors when it can deduce what the method labels are.

7.3.2 Static and dynamic binding

When executing inside an object, we often want to call another method in the same object, i.e., do a kind of recursive invocation. This seems simple enough, but it becomes slightly more complicated when inheritance is involved. A common use of inheritance is to define a new ADT that extends an existing ADT. To implement this correctly, it turns out that we need two ways to do a recursive call. They are called static and dynamic binding. We explain them by means of an example.

Consider the class `Account` defined in Figure 7.6. This class models a simple bank account with a balance. We can transfer money to it with `transfer`, inspect the balance with `getBal`, and do a series of transfers with `batchTransfer`. Note that `batchTransfer` calls `transfer` for each transfer.

Let us extend `Account` to do logging, i.e., to keep a record of all transactions it does. One way is to use inheritance, by overriding the `transfer` method:

```
class LoggedAccount from Account
  meth transfer(Amt)
    {LogObj addentry(transfer(Amt))}
    ...
  end
end
```

where `LogObj` is an object that keeps the log. Let us create a logged account with an initial balance of 100:

```
LogAct={New LoggedAccount transfer(100)}
```

Now the question is, what happens when we call `batchTransfer`? Does it call the old `transfer` in `Account` or the new `transfer` in `LoggedAccount`? We can deduce what the answer must be, if we assume that a class is an ADT. Every ADT has a set of methods that define what it does. For `LoggedAccount`, this set consists of the `getBal` and `batchTransfer` methods defined in `Account` as well as the new `transfer` defined in `LoggedAccount` itself. Therefore, the answer is that `batchTransfer` must call the new `transfer` in `LoggedAccount`. This is called *dynamic binding*. It is written as a call to **self**, i.e., as `{self transfer(A)}`.

When `Account` was defined, there was no `LoggedAccount` yet. Using dynamic binding keeps open the possibility that `Account` can be extended with inheritance, while ensuring that the new class is an ADT that correctly extends the old ADT. That is, it keeps all the functionality of the old ADT while adding some new functionality.

However, dynamic binding is usually not enough to implement the extended ADT. To see why, let us investigate closer how the new `transfer` is defined. Here is the full definition:

```
class LoggedAccount from Account
  meth transfer(Amt)
```

```

        {LogObj addentry(transfer(Amt))}
        Account,transfer(Amt)
    end
end

```

Inside the new `transfer`, we have to call the old `transfer`. We cannot use dynamic binding, since this would always call the new `transfer`. Instead, we use another technique called *static binding*. In static binding, we call a method by pinpointing the method's class. Here the notation `Account,transfer(Amt)` pinpoints the method `transfer` in the class `Account`.

Both static and dynamic binding are needed when using inheritance to override methods. Dynamic binding allows the new ADT to correctly extend the old ADT by letting old methods call new methods, even though the new method did not exist when the old method was defined. Static binding allows new methods to call old methods when they have to. We summarize the two techniques:

- **Dynamic binding.** This is written `{self M}`. This chooses the method matching `M` that is visible in the current object. This takes into account the overriding that has been done.
- **Static binding.** This is written `C, M` (with a comma), where `C` is a class that defines a method matching `M`. This chooses the method matching `M` that is visible in the class `C`. This takes overriding into account from the root class up to class `C`, but no further. If the object is of a subclass of `C` that has overridden `M` again, then this is not taken into account.

Dynamic binding is the only possible behavior for attributes. Static binding is not possible for them since the overridden attributes simply do not exist, neither in a logical sense (the only object that exists is the instance of the final class) nor in a practical sense (the implementation allocates no memory for them).

7.3.3 Controlling encapsulation

The principle of controlling encapsulation in an object-oriented language is to limit access to class members, namely attributes and methods, according to the requirements of the application architecture. Each member is defined with a *scope*. The scope is that part of the program text in which the member is visible, i.e., can be accessed by mentioning its name. Usually, the scope is statically defined, by the structure of the program. It can also be dynamically defined, namely during execution, if names are used (see below).

Programming languages usually give a default scope to each member when it is declared. This default can be altered with special keywords. Typical keywords used are *public*, *private*, and *protected*. Unfortunately, different languages use these terms to define slightly different scopes. Visibility in programming languages is a tricky concept. In the spirit of [54], we will try to bring order to this chaos.

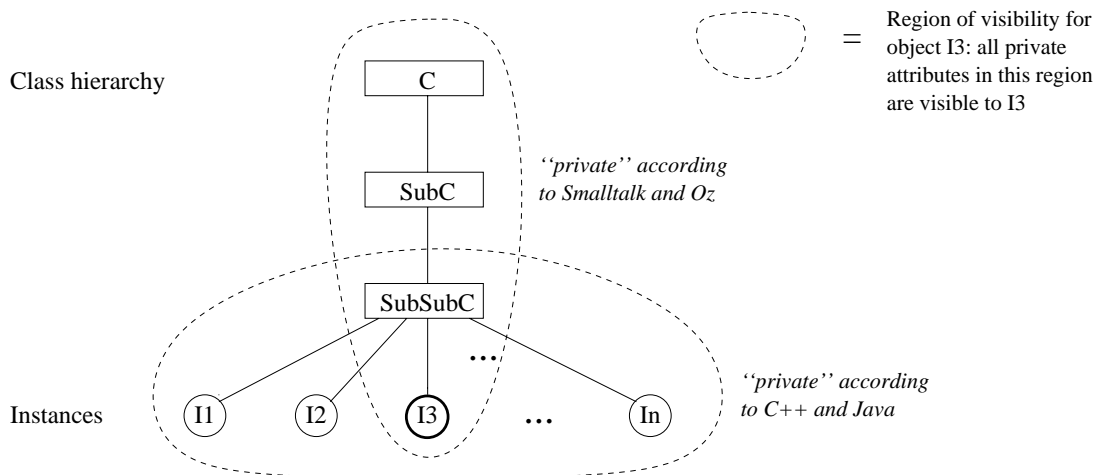


Figure 7.7: The meaning of “private”

Private and public scopes (in the ADT sense)

The two most basic scopes are *private* and *public*, with the following meanings:

- A private member is one which is only visible in the object instance. The object instance can see all members defined in its class and its superclasses. Thus private defines a kind of *vertical* visibility.
- A public member is one which is visible anywhere in the program.

In both Smalltalk and Oz, attributes are private and methods are public according to this definition.

These definitions of private and public are natural if classes are used to construct ADTs. Let us see why:

- First of all, a class is not the same thing as the ADT it defines! The class is an increment; it defines an ADT as an incremental modification of its superclasses. The class is only needed during the ADT’s construction. The ADT is not an increment; it stands on its own, with all its own attributes and methods. Many of these may come from the superclasses and not from the class.
- Second, attributes are internal to the ADT and should be invisible from the outside. This is exactly the definition of private.
- Finally, methods make up the external interface of the ADT, so they should be visible to all entities that reference the ADT. This is exactly the definition of public.

Constructing other scopes

Techniques for writing programs to control encapsulation are based essentially on two concepts: lexical scoping and name values. The private and public scopes defined above can be implemented with these two concepts. However, many other scopes can also be expressed using name values and lexical scoping. For example, it is possible to express the *private* and *protected* scopes of C++ and Java, as well as write programs that have much more elaborate security policies. The basic technique is to let method heads be *name values* instead of atoms. A name is an unforgeable constant; the only way to know a name is if someone gives you a reference to it (see Section 3.7.5 and Appendix B.2). In this way, a program can pass the reference in a controlled way, to exactly those areas of the program in which it should be visible.

In the examples of the previous sections, we have used *atoms* as method labels. But atoms are not secure: if a third party finds out the atom's print representation (either by guessing or by some other way) then he can call the method too. Names are a simple way to plug this kind of security leak. This is important for a software development project with well-defined interfaces between different components. It is even more important for open distributed programs, where code written at different times by different groups can coexist (see Chapter 11).

Private methods (in the C++ and Java sense)

When a method head is a name value, then its scope is limited to all instances of the class, but not to subclasses or their instances. This is exactly *private* in the sense of C++ and Java. Because of its usefulness, the object system of this chapter gives syntactic support for this technique. There are two ways to write it, depending on whether the name is defined implicitly inside the class or comes from the outside:

- By using a variable identifier as the method head. This implicitly creates a name when the class is defined and binds it to the variable. For example:

```
class C
  meth A(X)
    % Method body
  end
end
```

Method head A is bound to a name. The variable A is only visible inside the class definition. An instance of C can call method A in any other instance of C. Method A is invisible to subclass definitions. This is a kind of *horizontal* visibility. It corresponds to the concept of *private method* as it exists in C++ and Java (but not in Smalltalk). As Figure 7.7 shows, private in C++ and Java is very different from private in Smalltalk and Oz. In Smalltalk and Oz, private is relative to an *object* and its classes, e.g., I3 in the figure. In

C++ and Java, private is relative to a *class* and its instances, e.g., SubSubC in the figure.

- By using an *escaped* variable identifier as the method head. This syntax indicates that we will declare and bind the variable identifier outside of the class. When the class is defined then the method head is bound to whatever the variable is bound to. This is a very general mechanism that can be used to protect methods in many ways. It can also be used for other purposes than security (see Section 7.2.5). Here is an example that does exactly the same as the previous case:

```

local
  A={NewName}
in
  class C
    meth !A(X)
      % Method body
    end
  end
end

```

This creates a name at class definition time, just like in the previous case, and binds the method head to it. In fact, the previous definition is just a short-hand for this example.

Letting the programmer determine the method label allows to define a security policy at a very fine grain. The program can pass the method label to exactly those entities who need to know it.

Protected methods (in the C++ sense)

By default, methods in the object system of this chapter are public. Using names, we can construct the concept of a *protected method*, including both the C++ version and the Java version. In C++, a method is protected if it is accessible only in the class it is defined or in descendant classes (and all instance objects of these classes). The protected concept is a combination of the Smalltalk notion of private with the C++/Java notion of private: it has both a horizontal and vertical component. Let us show how to express the C++ notion of protected. The Java notion of protected is somewhat different; we leave it to an exercise. In the following class, method A is protected:

```

class C
  attr pa:A
  meth A(X) skip end
  meth foo(...) {self A(5)} end
end

```

It is protected because the attribute `pa` stores a reference to `A`. Now create a subclass `C1` of `C`. We can access method `A` as follows in the subclass:

```
class C1 from C
    meth b(...) A=@pa in {self A(5)} end
end
```

Method `b` accesses the method with label `A` through the attribute `pa`, which exists in the subclass. The method label can be stored in the attribute because it is just a value.

Attribute scopes

Attributes are always private. The only way to make them public is by means of methods. Because of dynamic typing, it is possible to define generic methods that give read and write access to *all* attributes. The class `Inspector` in Section 7.2.6 shows one way to do this. Any class that inherits from `Inspector` will have all its attributes potentially be public. Atom attributes are not secure since they can be guessed. Name attributes are secure even when using `Inspector`, since they cannot be guessed.

Atoms or names as method heads?

When should one use an atom or a name as a method head? By default, atoms are visible throughout the whole program and names are visible only in the lexical scope of their creation. We can give a simple rule when implementing ADTs: for internal methods use names and for external methods use atoms.

Most popular object-oriented programming languages (e.g., Smalltalk, C++, and Java) support only atoms as method heads, not names. These languages make atoms usable by adding special operations to restrict their visibility (e.g., `private` and `protected` declarations). On the other hand, names are practical too. Their visibility can be extended by passing around references. But the capability-based approach exemplified by names has not yet become popular. Let us look more closely at the trade-offs in using names versus atoms.

Atoms are uniquely identified by their print representations. This means they can be stored in program source files, in emails, on Web pages, etc. In particular, they can be stored in the programmer's head! When writing a large program, a method can be called from anywhere by just giving its print representation. On the other hand, with names this is more awkward: the program itself has somehow to pass the name to the caller. This adds some complexity to the program as well as being a burden for the programmer. So atoms win out both for program simplicity and for the psychological comfort factor during development.

Names have other advantages. First, it is impossible to have conflicts with inheritance (either single or multiple). Second, encapsulation can be better managed, since an object reference does not necessarily have the right to call all the

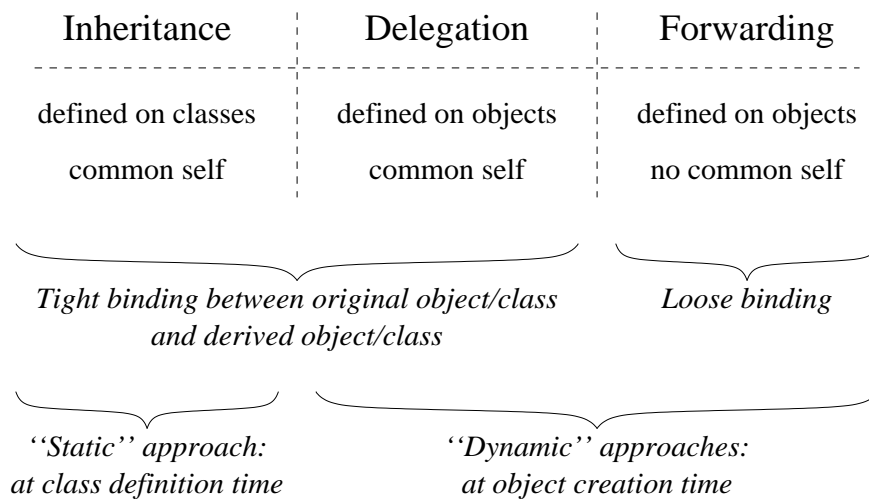


Figure 7.8: Different ways to extend functionality

object’s methods. Therefore, the program as a whole can be made less error-prone and better structured. A final point is that names can be given syntactic support to simplify their use. For example, in the object system of this chapter it suffices to capitalize the method head.

7.3.4 Forwarding and delegation

Inheritance is one way to reuse already-defined functionality when defining new functionality. Inheritance can be tricky to use well, because it implies a tight binding between the original class and its extension. Sometimes it is better to use looser approaches. Two such approaches are *forwarding* and *delegation*. Both are defined at the level of objects: if object `Obj1` does not understand message `M`, then `M` is passed transparently to object `Obj2`. Figure 7.8 compares these approaches with inheritance.

Forwarding and delegation differ in how they treat **self**. In forwarding, `Obj1` and `Obj2` keep their separate identities. A self call in `Obj2` will stay in `Obj2`. In delegation, there is just one identity, namely that of `Obj1`. A self call in `Obj2` will call `Obj1`. We say that delegation, like implementation inheritance, implies a *common self*. Forwarding does not imply a common self.

Let us show how to express forwarding and delegation. We define special object creation functions, `NewF` and `NewD`, for forwarding and delegation. We are helped in this by the flexibility of our object system: we use the `otherwise` method, messages as values, and the dynamic creation of classes. We start with forwarding since it is the simplest.

Forwarding

An object can forward to any other object. In the object system of this chapter, this can be implemented with the `otherwise(M)` method (see Section 7.2.5). The argument `M` is a first-class message that can be passed to another object. Let us define `NewF`, a version of `New` that creates objects that can forward:

```

local
  class ForwardMixin
    attr Forward:none
    meth setForward(F) Forward:=F end
    meth otherwise(M)
      if @Forward==none then raise undefinedMethod end
      else {@Forward M} end
    end
  end
in
  fun {NewF Class Init}
    {New class $ from Class ForwardMixin end Init}
  end
end

```

Objects created with `NewF` have a method `setForward(F)` that lets them set dynamically the object to which they will forward messages they do not understand. Let us create two objects `Obj1` and `Obj2` such that `Obj2` forwards to `Obj1`:

```

class C1
  meth init skip end
  meth cube(A B) B=A*A*A end
end

class C2
  meth init skip end
  meth square(A B) B=A*A end
end

Obj1={NewF C1 init}
Obj2={NewF C2 init}
{Obj2 setForward(Obj1)}

```

Doing `{Obj2 cube(10 X)}` will cause `Obj2` to forward the message to `Obj1`.

Delegation

Delegation is a powerful way to structure a system dynamically [113]. It lets us build a hierarchy among *objects* instead of among *classes*. Instead of an object inheriting from a class (at class definition time), we let an object delegate to another object (at object creation time). Delegation can achieve the same effects

```

local
  SetSelf={NewName}
  class DelegateMixin
    attr this Delegate:none
    meth !SetSelf(S) this:=S end
    meth set(A X) A:=X end
    meth get(A ?X) X=@A end
    meth setDelegate(D) Delegate:=D end
    meth Del(M S) SS in
      SS=@this this:=S
      try {self M} finally this:=SS end
    end
    meth call(M) SS in
      SS=@this this:=self
      try {self M} finally this:=SS end
    end
    meth otherwise(M)
      if @Delegate==none then
        raise undefinedMethod end
      else
        {@Delegate Del(M @this)}
      end
    end
  end
in
  fun {NewD Class Init}
    Obj={New class $ from Class DelegateMixin end Init}
  in
    {Obj SetSelf(Obj)}
    Obj
  end
end

```

Figure 7.9: Implementing delegation

as inheritance, with two main differences: the hierarchy is between objects, not classes, and it can be changed at any time.

Given any two objects `Obj1` and `Obj2`, we suppose there exists a method `setDelegate` such that `{Obj2 setDelegate(Obj1)}` sets `Obj2` to delegate to `Obj1`. In other words, `Obj1` behaves as the “superclass” of `Obj2`. Whenever a method is invoked that is not defined in `Obj2`, the method call will be retried at `Obj1`. The delegation chain can grow to any length. If there is an `Obj3` that delegates to `Obj2`, then calling `Obj3` can climb up the chain all the way to `Obj1`.

An important property of the delegation semantics is that *self* is always preserved: it is the self of the *original object* that initiated the delegation chain. It follows that the object state (the attributes) is also the state of the original object. In that sense, the other objects play the role of classes: in a first instance, it is their *methods* that are important in delegation, not the values of their attributes.

Let us implement delegation using our object system. Figure 7.9 gives the implementation of `NewD`, which is used instead of `New` to create objects. In order to use delegation, we impose the following syntactic constraints on how the object system must be used:

Operation	Original syntax	Delegation syntax
Object call	{<obj> M}	{<obj> call(M)}
Self call	{ self M}	{@this M}
Get attribute	@<attr>	{@this get(<attr> \$)}
Set attribute	<attr>:=X	{@this set(<attr> X)}
Set delegate		{<obj> ₁ setDelegate(<obj> ₂)}

These syntactic constraints could be eliminated by an appropriate linguistic abstraction. Now let us give a simple example of how delegation works. We define two objects `Obj1` and `Obj2` and let `Obj2` delegate to `Obj1`. We give each object an attribute `i` and a way to increment it. With inheritance this would look as follows:

```

class C1NonDel
  attr i:0
  meth init skip end
  meth inc(I) i:=@i+I end
  meth browse {self inc(10)} {Browse c1#@i} end
  meth c {self browse} end
end

class C2NonDel from C1NonDel
  attr i:0
  meth init skip end
  meth browse {self inc(100)} {Browse c2#@i} end
end

```

With our delegation implementation we can get the same effect by using the code of Figure 7.10. It is more verbose, but that is only because the system has no

```

class C1
  attr i:0
  meth init skip end
  meth inc(I)
    {@this set(i {@this get(i $)}+I)}
  end
  meth browse
    {@this inc(10)}
    {Browse c1#{@this get(i $)}}
  end
  meth c {@this browse} end
end
Obj1={NewD C1 init}

class C2
  attr i:0
  meth init skip end
  meth browse
    {@this inc(100)}
    {Browse c2#{@this get(i $)}}
  end
end
Obj2={NewD C2 init}
{Obj2 setDelegate(Obj1)}

```

Figure 7.10: An example of delegation

syntactic support for delegation. It is not due to the concept itself. Note that this just scratches the surface of what we could do with delegation. For example, by calling `setDelegate` again we could change the hierarchy of the program at run-time. Let us now call `Obj1` and `Obj2`:

```

{Obj2 call(c)}
{Obj1 call(c)}

```

Doing these calls several times shows that each object keeps its own local state, that `Obj2` “inherits” the `inc` and `c` methods from object `Obj1`, and that `Obj2` “overrides” the `browse` method. Let us make the delegation chain longer:

```

class C2b
  attr i:0
  meth init skip end
end
ObjX={NewD C2b init}
{ObjX setDelegate(Obj2)}

```

`ObjX` inherits all its behavior from `Obj2`. It is identical to `Obj2` except that it has a different local state. The delegation hierarchy now has three levels: `ObjX`, `Obj2`, and `Obj1`. Let us change the hierarchy by letting `ObjX` delegate to `Obj1`:


```
{ObjX setDelegate(Obj1)}
{ObjX call(c)}
```

In the new hierarchy, ObjX inherits its behavior from Obj1. It uses the `browse` method of Obj1, so it will increment by 10 instead of by 100.

7.3.5 Reflection

A system is *reflective* if it can inspect part of its execution state while it is running. Reflection can be purely introspective (only reading the internal state, without modifying it) or intrusive (both reading and modifying the internal state). Reflection can be done at a high or low level of abstraction. One example of reflection at a high level would be the ability to see the entries on the semantic stack as closures. It can be explained simply in terms of the abstract machine. On the other hand, the ability to read memory as an array of integers is reflection at a low level. There is no simple way to explain it in the abstract machine.

Meta-object protocols

Object-oriented programming, because of its richness, is a particularly fertile area for reflection. For example, the system could make it possible to examine or even change the inheritance hierarchy, while a program is running. This is possible in Smalltalk. The system could make it possible to change how objects execute at a basic level, e.g., how inheritance works (how method lookup is done in the class hierarchy) and how methods are called. The description of how an object system works at a basic level is called a *meta-object protocol*. The ability to change the meta-object protocol is a powerful way to modify an object system. Meta-object protocols are used for many purposes: debugging, customizing, and separation of concerns (e.g., transparently adding encryption or format changes to method calls). Meta-object protocols were originally invented in the context of the Common Lisp Object System (CLOS) [100, 140]. They are an active area of research in object-oriented programming.

Method wrapping

A common use of meta-object protocols is to do *method wrapping*, that is, to intercept each method call, possibly performing a user-defined operation before and after the call and possibly changing the arguments to the call itself. In our object system, we can implement this in a simple way by taking advantage of the fact that objects are one-argument procedures. For example, let us write a tracer to track the behavior of an object-oriented program. The tracer should display the method label whenever we enter a method and exit a method. Here is a version of `New` that implements this:

```
fun {TraceNew Class Init}
  Obj={New Class Init}
```

```

proc {TracedObj M}
  {Browse entering({Label M})}
  {Obj M}
  {Browse exiting({Label M})}
end
in TracedObj end

```

An object created with `TraceNew` behaves identically to an object created with `New`, except that method calls (except for calls to `self`) are traced. The definition of `TraceNew` uses higher-order programming: the procedure `TracedObj` has the external reference `Obj`. This definition can easily be extended to do more sophisticated wrapping. For example, the message `M` could be transformed in some way before being passed to `Obj`.

A second way to implement `TraceNew` is to do the wrapping with a class instead of a procedure. This traces all method calls including calls to `self`. This gives the following definition:

```

fun {TraceNew2 Class Init}
  Obj={New Class Init}
  TInit={NewName}
  class Tracer
    meth !TInit skip end
    meth otherwise(M)
      {Browse entering({Label M})}
      {Obj M}
      {Browse exiting({Label M})}
    end
  end
in {New Tracer TInit} end

```

This uses dynamic class creation, the `otherwise` method, and a fresh name `TInit` for the initialization method to avoid conflicts with other method labels.

Reflection of object state

Let us show a simple but useful example of reflection in object-oriented programming. We would like to be able to read and write the whole state of an object, independent of the object's class. The Mozart object system provides this ability through the class `ObjectSupport.reflect`. Inheriting from this class gives the following three additional methods:

- `clone(X)` creates a clone of `self` and binds it to `X`. The clone is a new object with the same class and the same values of attributes.
- `toChunk(X)` binds to `X` a protected value (a “chunk”) that contains the current values of the attributes.
- `fromChunk(X)` sets the object state to `X`, where `X` was obtained from a previous call of `toChunk`.

A *chunk* is like a record but with a restricted set of operations. It is protected in the sense that only authorized programs can look inside it (see Appendix B.4). Chunks can be implemented with procedure values and names, as explained in Section 3.7.5. Let us extend the Counter class we saw before to do state reflection:

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

We can define two objects:

```
C1={New Counter init(0)}
C2={New Counter init(0)}
```

and then transfer state from one to the other:

```
{C1 inc(10)}
local X in {C1 toChunk(X)} {C2 fromChunk(X)} end
```

At this point C2 also has the value 10. This is a simplistic example, but state reflection is actually a very powerful tool. It can be used to build generic abstractions on objects, i.e., abstractions that work on objects of *any* class.

7.4 Programming with inheritance

All the programming techniques of stateful programming and declarative programming are still possible in the object system of this chapter. Particularly useful are techniques that are based on encapsulation and state to make programs modular. See the previous chapter, and especially the discussion of component-based programming, which relies on encapsulation.

This section focuses on the new techniques that are made possible by object-oriented programming. All these techniques center around the use of inheritance: first, using it correctly, and then, taking advantage of its power.

7.4.1 The correct use of inheritance

There are two ways to view inheritance:

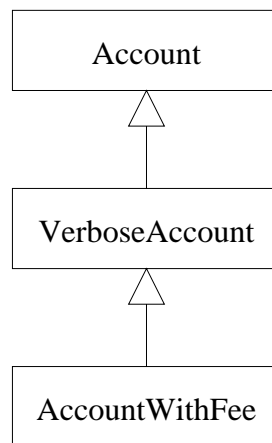


Figure 7.11: A simple hierarchy with three classes

- **The type view.** In this view, classes are types and subclasses are subtypes. For example, take a `LabeledWindow` class that inherits from a `Window` class. All labeled windows are also windows. The type view is consistent with the principle that classes should model real-world entities or some abstract versions of them. In the type view, classes satisfy the *substitution property*: every operation that works for an object of class `C` also works for objects of a subclass of `C`. Most object-oriented languages, such as Java and Smalltalk, are designed for the type view [63, 60]. Section 7.4.1 explores what happens if we do not respect the type view.
- **The structure view.** In this view, inheritance is just another programming tool that is used to structure programs. This view is **strongly discouraged** because classes no longer satisfy the substitution property. The structure view is an almost unending source of bugs and bad designs. Major commercial projects, which shall here remain anonymous, have failed for this reason. A few object-oriented languages, notably Eiffel, are designed from the start to allow both the type and structure views [122].

In the type view, each class stands on its own two feet, so to speak, as a bona fide ADT. This is even true for classes that have subclasses; from the viewpoint of the subclass, the class is an ADT, with sole access through the methods and its attributes hidden. In the structure view, classes are sometimes just scaffolding, which exists only for its role in structuring the program.

In the vast majority of cases, inheritance should respect the type view. Doing otherwise gives subtle and pernicious bugs that can poison a whole system. Let us give an example. We take as base class the `Account` class we saw before, which is defined in Figure 7.6. We will extend it in two ways. The first extension is conservative, i.e., it respects the type view:

```
class VerboseAccount from Account
```

```

    meth verboseTransfer(Amt)
      {self transfer(Amt)}
      {Browse `Balance:`#@balance}
    end
  end
end

```

We simply add a new method `verboseTransfer`. Since the existing methods are not changed, this implies that a `VerboseAccount` object will work correctly in all cases where an `Account` object works. Let us now do a second, more dangerous extension:

```

class AccountWithFee from VerboseAccount
  attr fee:5
  meth transfer(Amt)
    VerboseAccount,transfer(Amt-@fee)
  end
end

```

Figure 7.11 shows the resulting hierarchy. The open arrowhead in this figure is the usual notation to represent an inheritance link. `AccountWithFree` overrides the method `transfer`. Overriding is not a problem in of itself. The problem is that an `AccountWithFee` object does not work correctly when viewed as an `Account` object. They do not satisfy the same invariant. Consider the sequence of three calls:

```

{A getBalance(B)}
{A transfer(S)}
{A getBalance(B2)}

```

If `A` is an `Account` object, this implies $B+S=B2$. If `A` is an `AccountWithFee` object, this implies $B+S-@fee=B2$. This will break any program that relies on the behavior of `Account` objects. Typically, the origin of the break will not be obvious, since it is carefully hidden inside a method somewhere in a large application. It will appear long after the change was made, as a slight imbalance in the books. Debugging such “slight” problems is amazingly difficult and time-consuming.

The rest of this section primarily considers the type view. *Almost all* uses of inheritance should respect the type view. However, the structure view is occasionally useful. Its main use is in changing the behavior of the object system itself. For this purpose, it should be used only by expert language implementors who clearly understand the ramifications of what they are doing. A simple example is method wrapping (see Section 7.3.5), which requires using the structure view. For more information, we recommend [122] for a deeper discussion of the type view versus the structure view.

A cautionary tale

We end the discussion on the correct use of inheritance with a cautionary tale. Some years ago, a well-known company initiated an ambitious project based

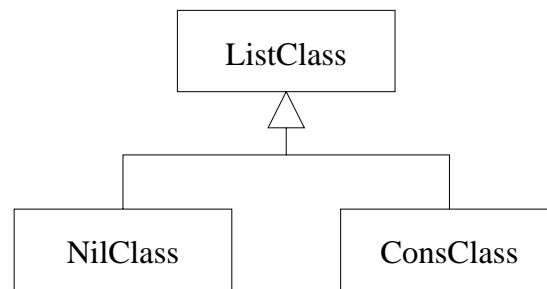


Figure 7.12: Constructing a hierarchy by following the type

on object-oriented programming. Despite a budget of several billion dollars, the project failed. Among many reasons for the failure was an incorrect use of object-oriented programming, in particular concerning inheritance. Two major mistakes were made:

- The substitution property was regularly violated. Routines that worked correctly with objects of a given class did not work with objects of a subclass. This made it much more difficult to use objects: instead of one routine being sufficient for many classes, many routines were needed.
- Classes were subclassed to fix small problems. Instead of fixing the class itself, a subclass was defined to *patch* the class. This was done so frequently that it gave layers upon layers of patches. Object invocations were slowed down by an order of magnitude. The class hierarchy became unnecessarily deep, which increased complexity of the system.

The lesson to heed is to be careful to use inheritance in a correct way. Respect the substitution property whenever possible. Use inheritance to add new functionality and not to patch a broken class. Study common design patterns to learn the correct use of inheritance.

Reengineering At this point, we should mention the discipline of *reengineering*, which can be used to fix architectural problems like these two incorrect uses of inheritance [44, 15]. The general goal of reengineering is to take an existing system and attempt to improve some of its properties by changing the source code. Many properties can be improved in this way: system architecture, modularity, performance, portability, quality of documentation, and use of new technology. However, reengineering cannot resurrect a failed project. It is more like curing a disease. If the designer has a choice, the best approach remains to prevent the disease, i.e., to design a system so that it can be adapted to changing requirements. In Section 6.7 and throughout the book, we give design principles that work towards this goal.