Chapter 10
The Object-Oriented Paradigm

In this chapter, we present the important aspects of object-oriented languages, a paradigm which has its roots in Simula (during the 60s) and Smalltalk (in the 70s). It achieved enormous success, including commercial success, in the two following decades (C++ and Java are only two of the most known languages among the many that are in use today). “Object oriented” is by now an abused slogan, which can be found applied to languages, to databases, operating systems, and so on. Here, we attempt to present the *linguistic* aspects which concern objects and their use, making the occasional reference to object-oriented design techniques (but we can only refer the reader to the (extended) literature for details).

Even after restricting the subject, there are many ways to proceed to the concepts that interest us. We will follow the approach we believe the simplest: we will present objects as a way to gain data abstraction in a way that is flexible and extensible. We will begin our study, then, by showing some limits to the techniques that we introduced in Chap. 9. These limits will suggest to us some concepts, which, in fact, form the basics of an object-oriented language. Having examined these characteristic aspects in detail, we will study some extensions to linguistic solutions that are available in commercial languages, referring, in particular, to the concepts of subtype, polymorphism and genericity.

Following the style that we have maintained thus far in this text, we will seek to remain independent of any specific programming language, even if this will not always be possible. The language that will mostly inspire us will be Java, for its coherence of design, which will allow us to discuss some concepts (and not linguistic details) in a clearer way and in a way that allows us to summarise the properties of other languages.

10.1 The Limits of Abstract Data Types

Abstract data types are a mechanism that guarantees the encapsulation and hiding of information in a clean and efficient manner. In particular, the have the excellent characteristic of uniting in a single construct both data and the methods for legally manipulating it. This is a very important methodological principle, which is obtained at the cost of a certain rigidity of use, which shows up when it is wanted to extend or reuse an abstraction. We will discuss these problems using a simple and certainly not realistic example, but one which exposes the most important questions.

So that we do not overload the notation with the call-by-reference, it is convenient to use a language that uses a reference model for variables. The following ADR implements a simple counter:

```
abstype Counter{
   type Counter = int;
   signature
      void reset(Counter x);
      int get(Counter x);
      void inc(Counter x);
   operations
      void reset(Counter x){
         x = 0;
      }
      int get(Counter c){
         return x;
      }
      void inc(Counter c){
         x = x+1;
      }
}
```

The concrete representation of the `Counter` type is the integer type. The only operations possible are zeroing of a counter, reading and incrementing its value.

We want now to define a counter that is enriched by some new operations. For example, we want to take into account the number of calls the `reset` operation has had in a given counter. We have a choice of 2 approaches: the first is that of defining a completely new ADT which is similar in many respects to the one just defined with the addition of new operations, as in:

```
abstype NewCounter1{
   type NewCounter1 = struct{
                        int c;
                        int num_reset = 0;
                      }
   signature
      void reset(NewCounter1 x);
      int get(NewCounter1 x);
      void inc(NewCounter1 x);
      int howmany_resets(NewCounter1 x);
   operations
      void reset(NewCounter1 x){
         x.c = 0;
         x.num_reset = x.num_reset+1;
      }
      int get(NewCounter1 x){
         return x.c;
      }
      void inc(NewCounter1 x){
         x.c = x.c+1;
      }
      int howmany_resets(NewCounter1 x){
         return x.num_reset;
      }
}
```

This is a solution that is acceptable as far as encapsulation goes, but we were required to redefine the operations what we have already defined for a simple counter (they are operations with the same name but with different argument types; names are therefore overloaded which the compiler has to differentiate using the context). In this academic example, we have only a few lines of code but the negative effect on a real situation can be quite significant.[1] More important again is what happens when, for whatever reason, it is desired to change the implementation of a simple counter. There being no relationship between `Counter` and `NewCounter`, modifications such as these are not felt by another counter: a `NewCounter` will still continue to work perfectly. But if the modification were due to reasons of usefulness or efficiency (for example, we found an excellent new way to implement an `inc`), such a modification must be performed by hand on the definition of a `NewCounter`, with all the known problems that this brings (find all the places where a variant of the old `inc` is used, do not introduce any syntactic errors, etc.).

The second possibility that we have is one that makes use of `Counter` to define an enriched counter:

```
abstype NewCounter2{
   type NewCounter2 = struct{
                        Counter c;
                        int num_reset = 0;
                      }
   signature
      void reset(NewCounter2 x);
      int get(NewCounter2 x);
      void inc(NewCounter2 x);
      int howmany_resets(NewCounter2 x);
   operations
      void reset(NewCounter2 x){
         reset(x.c);
         x.num_reset = x.num_reset+1;
      }
      int get(NewCounter2 x){
         return get(x.c);
      }
      void inc(NewCounter2 x){
         inc(x.c);
```

[1]This is a negative effect not only on the writing of the program, but also in the size of the code produced because there are *two copies* of each operation!

```
        }
        int howmany_resets(NewCounter2 x){
           return x.num_reset;
        }
}
```

The solution is clearly better than the previous one. The operations that do not have to be modified are only called from inside a NewCounter (with its usual name overloading), so that the last problem mentioned above is solved. There remains the task of performing the calls explicitly even for the operations (such as get and inc) that have not been modified: it would be preferable to have an automatic mechanism with which to *inherit* the implementations of these two operations from Counter.

There remain problems, though, of how to handle in a uniform fashion the values of Counter and of NewCounter2. Let us assume, in fact, that are dealing with a series of counters, some simple, some enriched, and that we want to reset them all to their initial value. To fix ideas, we can imagine an array of counters; we want to reset each element of this array to its initial value. A first problem arises immediately when we try to work out what the type of this array is. If we declare it as:

```
Counter V[100];
```

we cannot store NewCounter2s there because the two types are distinct. The same thing happens when the array is declared as being of type NewCounter2. To solve the problem, we need a concept of compatibility between the 2 types. Let us remember that among the various forms of compatibility we discussed in Sect. 8.7, we find the following:

> T is compatible with S when all the operations over values of type S are also possible over values of type T.

We have exactly a case of this kind: all the operations on a Counter are possible also on a NewCounter2 (on which we can apply an additional operation). It is, therefore, sensible to require that a hypothetical language should relax the rigidity of ADTs, admitting this form of compatibility. Let us therefore assume that NewCounter2 is compatible with Counter and is therefore permitted to have a vector declared as Counter V[100] in which are stored simple and extended counters. We now come to the main point, which is to reset the initial value of each one of these counters. The obvious idea is:

```
for (int i=1; i<100; i=i+1)
   reset(V[i]);
```

which does not pose type problems. Overloading is solved, interpreting the body as a call to the reset operation on Counter. All goes well with the type checker, but what can be said about the state of NewCounter2 stored in the array? We will expect that their num_reset fields have been incremented by one, but this is not the case because the reset operation defined in NewCounter2 has not been executed, only the one defined in Counter. Compatibility has solved the problem of uniformly manipulating the values of the two types but in a certain sense has destroyed encapsulation, allowing the application of an operation to a value of type `NewCounter2` which is not correct with reference to the ADT. A moment's reflection shows how the problem derives from the static solution to the overloading of `reset`. If we could decide dynamically which `reset` to apply, depending upon the "effective" type of the argument (that is on the type of the element stored in the array before the coercion connected with compatibility is applied), this problem would also be solved.

10.1.1 A First Review

Abstract data types guarantee the encapsulation and hiding of information but they are rigid when used in a design with a degree of complexity. From what has just been said, it is not unreasonable to foresee constructs such as:

- They permit the *encapsulation* and hiding of information.
- They are equipped with a mechanism which, under certain conditions, supports the *inheritance* of the implementation of certain operations from other, analogous constructs.
- They are framed in a notion of *compatibility* defined in terms of the operations admissible for a certain construct.
- They allow the *dynamic selection* of operations as a function of the "effective type" of the arguments to which they are applied.

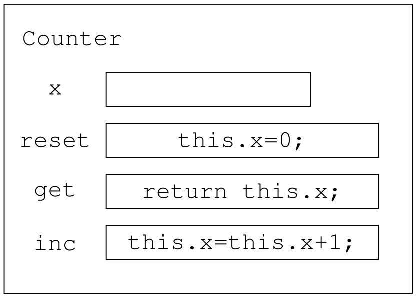
These four requirements are satisfied by the object-oriented paradigm. Rather, we can take them as essential characteristics of this paradigm which separates an object-oriented language from one which is not. We will discuss them in detail after introducing some basic concepts and once the terminology has been fixed.

10.2 Fundamental Concepts

In this section, we will establish the fundamental concepts which organise the object-oriented paradigm. We will begin by introducing some terminology and macroscopic linguistic concepts (object, class, method, overriding, etc.), so that we can return to the four aspects listed in Sect. 10.1.1. with the aim of discussing them in more detail. We will take them in the following order:

1. Encapsulation and abstraction.
2. Subtypes, that is a compatibility relation based on the functionality of an object.
3. Inheritance, that is the possibility of reusing the implementation of a method previously defined in another object or for another class.
4. Dynamic method selection.

Fig. 10.1 An object for a counter



These are distinct mechanisms, each exhibited on its own by other paradigms. As will be seen, it is their interaction that makes the object-oriented paradigm so attractive to software design, even of large systems.

10.2.1 Objects

The principal construct of object-oriented languages is (clearly) that of object. A capsule containing both data and the operations that manipulate it and which provides an interface for the outside world through which the object is accessible. The methodological idea which objects share with ADT is that data should "remain together" with the operations that manipulate it. There is, then, a big conceptual difference between them and ADTs (which is translated into a considerably different notation). Although, in the definition of an ADT, data and operations are together, when we declare a variable of an abstract type, that variable represents only the data which can be manipulated using the operations. In our counters example, we can declare a counter as `Counter c` and then increment it: `inc(c)`. *Each* object, on the other hand, is a container which (at least conceptually) encapsulates both data and operations. A counter object, defined using the same model as the definition that we have given for `Counter` could be represented as in Fig. 10.1.

The figure suggests that we may imagine an object as a record. Some fields correspond to (modifiable) data, for example the `c` field; other fields correspond to the operations that are allowed to manipulate the data (that is `reset`, `get` and `inc`).

The operations are called *methods* (or functional fields or *member functions*) and can access the data items held in the object, which are called *instance variables* (or *data members* or fields). The execution of an operation is performed by sending the object a *message* which consists of the name of the method to execute, possibly together with parameters.[2] To represent such invocations, we use the Java and C++ notation which reinforces the analogy with records:

```
object.method(parameters)
```

[2]This is mostly Smalltalk terminology. C++ expresses the same thing by saying: calling the member function of an object.

Below, we will often read the above notation as the "invocation of the method `method` (with parameters *parameters*) on the object `object`" rather than "sending the message `method(parameters)` to the object `object`" as would be formally more precise.

An aspect that is not always clear is that the object receiving the message is, at the same time, also a(n implicit) parameter to the invoked method. To increment a counter object, `o`, which has the structure of Fig. 10.1, we will write `o.inc()` ("we ask the object, `o`, to apply the message `inc` to itself"). Also, the data members are accessible using the same mechanism (if they are not hidden by capsule, obviously). If an object, `o`, has an instance variable, `v`, with `o.v` we request `o` for the value of `v`. While the notation remains uniform, we note that accessing data is a mechanism that is distinct from that of invoking a method. Invoking a method involves (or can involve) the *dynamic selection* of the method that is to be executed, while access to a data item is static, although there are exceptions to this.

The level of opacity of the capsule is defined when the object is itself created. Data can be more or less directly accessible from the outside (for example, data could be directly inaccessible, with the object, instead, providing "observers"—recall the box on page 269), some operations can be visible everywhere, others visible to some objects, while, finally, others are completely private, that is available only inside the object itself.

Together with objects, the languages of this paradigm also make available *organisational mechanisms* for objects. These mechanisms permit the grouping of objects with the same structure (or with similar structure). Although it is conceptually permitted that *every* object truly contains its own data and methods, this would result in enormous waste. In a single program, many objects would be used which are differentiated between themselves only by the value of their data and not by their structure or by their methods (for example, many counters, all analogous to the one in Fig. 10.1). Without an organisational principle, which makes explicit the similarity of all these objects, the program would loose expository clarity and would be harder to document. Furthermore, from the implementation viewpoint, it is clear that it would be appropriate for the code of each method to be stored only once, instead of being copied inside each object with similar structure. To solve these problems, every object-oriented language is based on some organisational principles. Among these principles, the one that is by far the best known is that of *classes*, although there is a whole family of object-oriented languages that lack classes (which we will briefly describe in the box on page 286).

10.2.2 Classes

A class is a model for a set of objects. It establishes what its data will be (type together with their visibility) and fixes name, signature, visibility and implementation for each of its methods. In a language with classes, every object belongs to (at least) one class. For example, an object such as that in Fig. 10.1 could be an instance of the following class:

```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}
```

It can be seen that the class contains the implementation of three methods which are declared `public` (visible to all), while the instance variable `x` is `private` (inaccessible from outside the object itself but accessible in the implementation of the methods of this class).

Objects are dynamically created by *instantiation* of their class. A specific object is allocated whose structure is determined by its class. This operation differs in fundamental ways from language to language and depends on the linguistic status of classes. In Simula, a class is a procedure which returns a pointer to an activation record containing local variables and function definitions (therefore, the objects that are instances of a class are closures). In Smalltalk, a class is linguistically an object which acts as a schema for the definition of the implementation of a set of objects. In C++ and Java classes correspond to a type and all objects that instantiate a class `A` are values of type `A`. Taking this viewpoint in Java, with its reference-based model for variables, we can create a counter object:

```
Counter c = new Counter();
```

The name `c`, of type `Counter`, is bound to a new object which is an instance of `Counter`. We can create a new object, distinct from the previous one but with the same structure, and bind it to another name:

```
Counter d = new Counter();
```

To the right of the assignment symbol, we can see two distinct operations: the creation of the objects (allocation of the necessary storage, `new` constructor) and initialisation (invocation of the class' *constructor*, represented by the name of the class and parentheses—we will return to this shortly[3]).

We can certainly assume that the code for methods is stored one time only in its class and that when an object needs to execute a specific method, this code would be looked up in the class of which it is an instance. For this to happen, the method code must correctly access the instance variables which are different for every object

[3]In C++, unlike Java, it is also possible to create an object without invoking its constructor; C++ in fact allows the creation of objects on the stack (see page 286).

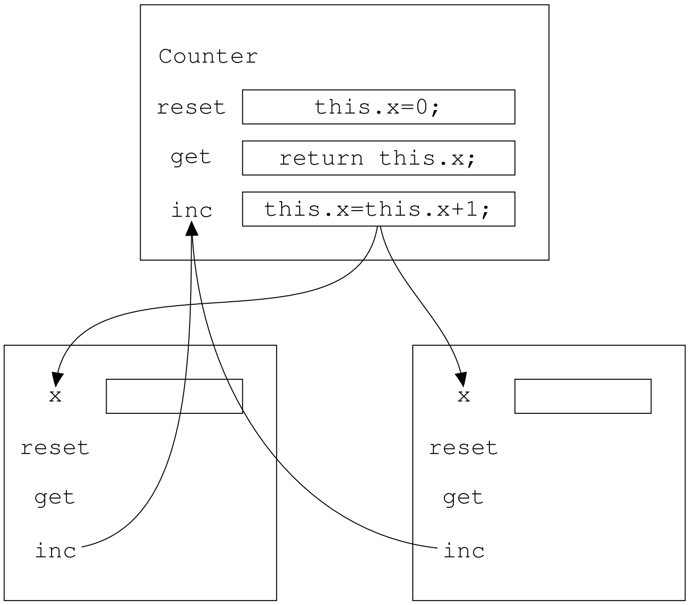


Fig. 10.2 The implementation of methods is inside classes

and are, therefore, not all stored together in the class but inside the instance (as is shown graphically in Fig. 10.2). In the figure, the methods of class `Counter` refer to its instance variables using the name `this`. We have already seen that when an object receives a message requesting the execution of a method, the object itself is an implicit method parameter. When a reference is made to an instance variable in the body of a method, there is an implicit reference to the object currently executing the method. From the linguistic viewpoint, the current object is usually denoted by a unique name, usually `self` or `this`. For example, the definition of the `inc` method could be written as follows. Here, the implicit reference to the current object is made explicit:

```
public void inc(){
   this.x = this.x+1;
}
```

In the case of a call to a method via `this`, the link between the method and the code to which it is referring is dynamically determined. This is an important aspect of this paradigm and we will discuss it in more detail in Sect. 10.2.6.

Some languages allow some variables and methods to be associated with classes (and not to their instances). They are called *class* or *static* variables and methods. Static variables are stored together in the class and static methods cannot, obviously, refer to `this` in their body because that they have no current object.

Languages based on Delegation

There also exist object-oriented languages that lack classes. In these languages, which are based on delegation (or on *prototypes*), the organising principle (see the end of Sect. 10.2.1) is not the class, but delegation; that is, the principle that one object can ask another object (its parent) to execute a method for it. Among these languages, the progenitor is Self which was developed at Xerox PARC and Stanford towards the end of the 1980s. Other languages based upon delegation are Dylan, a language for programming the Apple Newton PDA and on Javascript, which was designed to be distributed in one of the early versions of Netscape.

The fields of an object can contain values (simple data or other objects) or methods (that is, code), or references to other objects (the parent of the object). An object is created from nothing, or more often, by copying (or cloning) another object (which is called its *prototype*). Prototypes play the methodological role of classes. They provide the model for the structuring and functioning for other objects but are not special objects. Linguistically, they are ordinary objects, which, however, are used not for computation, but as a model. When a prototype is cloned, the copy maintains a reference to the prototype as its parent.

When an object receives a message but has no field of that name, it passes the message to its parent, and the process repeats. When the message reaches an object that understands it, the code associated with the method is executed. The language ensures that the reference to `self` is correctly resolved as a reference to the object that originally received the message. In general, data in the object is considered equal to methods. Access to a data item corresponds to sending a message, which the same object responds by returning the value of the field.

The delegation mechanism (and the unification of code and data) makes inheritance more powerful. It is possible and natural to create objects which share portions of data (in a language based on classes, this is possible only using static data associated with classes; however, such a template is too "static" to be profitably used). Moreover, an object can change the reference to its parent dynamically and thereby change its own behaviour.

Objects in the heap and on the stack Every object-oriented language allows the dynamic creation of objects. Where objects are created depends on the language. The most common solution is to allocate objects in the heap and access them using references (which will be real pointers in languages that permit them, but will be variables if the language supports a reference model). Some languages allow explicit allocation and deallocation of objects in the heap (C++ is one); others, probably the majority, opt instead for a garbage collector.

The option of creating objects on the stack like ordinary variables is not very common. C++ is one such language. When a declaration of a variable of class type is elaborated, the creation and initialisation of an object of that type is performed.

The object is assigned as a value to the variable.[4] The two operations contributing to the creation of an object (allocation and initialisation) occur implicitly in cases where the constructor is not explicitly called. Some languages, finally, allow objects to be created on the stack and left uninitialised.

In our pseudo-language, we assume that object creation occurs explicitly on the heap and has a reference-variable model.

10.2.3 Encapsulation

Encapsulation and information hiding represent two of the cardinal points of data abstraction. From the linguistic viewpoint, there is not much to add to what we have already said. Every language allows the definition of objects by hiding some part of them (either data or methods). In every class there are, therefore, at least two *views*: private and public parts. In the private view, everything is visible: it is the level of access possible inside the class itself (by its methods). In the public view, however, only explicitly exported information is visible. We say that the public information is the interface to a class, by analogy with ADTs.[5]

At this level no great difference appear between object-oriented programming languages and the forms of data abstraction that we have already discussed. The encapsulation possible with objects, however, is much more flexible and, more importantly, extensible than is possible with ADTs. But this will become clear only at the end of our examination of the concepts.

10.2.4 Subtypes

A class can be made to correspond, in a natural fashion, with the set of objects which are instances of that class. This set of objects is the type associated with that class. In typed languages, this relation is explicit. A class definition also introduces the definition of a type whose values are the instances of that class. In typeless languages (like Smalltalk), the correspondence is only conventional and implicit.

Among the types thus obtained, a compatibility relation is defined (Sect. 8.7) in terms of the operations possible on values of the type. The type associated with the class `T` is a subtype of `S` when every message understood (that is which can be received without generating an error) from objects of `S` is also understood by the objects of `T`. If an object is represented as a record containing data and functions,[6]

[4]Things are clearly different when a variable of type pointer-to-a-class type is declared. In this case, no object is created (without an explicit request to do so).

[5]In different languages, the term "interface" means different things. Do not confuse, in particular, our use of the term "interface" with what is meant by it in Java (where it is a particular language construct, a sort of class in which the only components are names and method signatures but do not contain their implementations).

[6]That is, as a closure.

the subtype relation corresponds to the fact that `T` is a record type containing all fields of `S`, as well, possibly, other fields. More precisely, taking account of the fact that some fields of the two types could be private, we might say that `T` is a subtype of `S` when the interface of `S` is a subset of the interface of `T` (note the inversion: a subtype is obtained when we have a bigger interface).

Some languages (like C++ and Java) use a name-based equivalence for types (see Sect. 8.6) which does not properly extend to a completely structural compatibility relation. In such languages, it is not, therefore, the only structural property between interfaces that defines the subtype relation, but it must be explicitly introduced by the programmer. This is the role of the definition of subclasses, or derived classes, which in our pseudo-language will be denoted using the neutral `extending` construct:[7]

```
class NamedCounter extending Counter{
   private String name;
   public void set_name(String n){
      name = n;
   }
   public String get_name(){
      return name;
   }
}
```

The class `NamedCounter` is a subclass of `Counter` (which, in its turn is a superclass of `NamedCounter`),[8] that is the `NamedCounter` type is a subtype of `Counter`. Instances of `NamedCounter` contain all the fields of `Counter` (even its private fields, but they are inaccessible in the subclass), in addition to having new fields introduced by the definition. In this way, structural compatibility can be guaranteed (a subclass is explicitly derived from its superclass) but this is explicitly stated in the program.

Redefinition of a method In the simple example of `NamedCounter`, subclasses are limited to extending the interface of the superclass. A fundamental characteristic of the subclass mechanism is ability of a subclass to modify the definition (the implementation) of a method present in its superclass. This mechanism is called *method overriding*.[9] Our extended counters from Sect. 10.1 can be defined as a subclass of `Counter` as:

```
class NewCounter extending Counter{
   private int num_reset = 0;
   public void reset(){
```

[7]In place of `extending`, in C++, we write ": `public`", while in Java `extends` or even `implements` is used, depending upon whether a class or an interface is being extended.

[8]In C++, `Counter` is the *base* class and `NamedCOunter` is the *derived* class.

[9]Method overriding is a mechanism which interacts in a very subtle way with other aspects of the object-oriented paradigm, in particular with the dynamic selection of methods; for the time being, we will note only this possibility, returning to this question in due course.

```
        x = 0;
        num_reset = num_reset + 1;
    }
    public int howmany_resets(){
        return num_reset;
    }
}
```

Class `NewCounter` simultaneously extends the interface of `Counter` with new fields and redefines the `reset` method. A `reset` method sent to an instance of `NewCounter` will cause the invocation of the new implementation.

Shadowing In addition to modifying the implementation of a method, a subclass can also redefine an instance variable (or field) defined in a superclass. This mechanism is called *shadowing*. For implementation reasons (to be seen below), shadowing is significantly different from overriding. For the present, we merely note that, in a subclass, an instance variable can be redefined with the same name and same type as that in the superclass. We could, for example, modify our extended counters using the following subclass of `NewCounter` where, for some (strange) reason, the initial value of `num_reset` is initialised to 2 and is incremented by 2 each time:

```
class EvenNewCounter extending NewCounter{
    private int num_reset = 2;
    public void reset(){
        x = 0;
        num_reset = num_reset + 2;
    }
    public int howmany_resets(){
        return num_reset;
    }
}
```

Using the usual notion of visibility in block-structured languages, each reference to `num_reset` inside `EvenNewCounter` refers to the local variable (initialised to 2) and not to the one declared in `NewCounter`. A `reset` message sent to an instance of `EvenNewCounter` will cause the invocation of the new implementation for `reset` which will use the new `num_reset` field. However, as we will see below, there is a big difference, both at the semantic as well as at the implementation levels, between overriding and shadowing.

Abstract classes For simplicity of exposition, we have introduced the type associated with a class as the set of its instances. Many languages, however, permit the definition of classes that cannot have instances because the class lacks the implementation of some method. In such classes, there is only the name and the type (that is, the signature) of one or more methods—their implementation is omitted. Classes of this type are called *abstract* classes. Abstract classes serve to provide interfaces and can be given implementations in subclasses that redefines (in this case,

Subtypes in Smalltalk?

Among the essential characteristics of the object-oriented paradigm, we have included subtypes. On the other hand, there are object-oriented languages which do not have types (at least do not have types over which we can perform any significant checking). Smalltalk is one of these languages. How can we have subtypes in a language without types?

When we gave the definition of subtype, we were careful to include the case of Smalltalk as well of other untyped languages. Recall that `T` is a subtype of `S` when the interface of `S` is a subset of the interface of `T`. That is, when we may freely use an object of class `T` (that is without generating errors) in place of one of class `S`. This operational definition, which is expressed in terms of the substitution of objects, makes perfect sense even in an untyped context, once we associate the set of its instances (its "type") with the class `T`.

On the other hand, when in the language there is no linguistic notion of subtype, it is only in the reflection of the designers that the presence of such a relationship becomes clear. In particular, in Smalltalk the definition of a subclass does not generate a subtype because Smalltalk allows subclasses to remove methods from their superclass.

define for the first time) the method lacking an implementation.[10] Abstract classes also correspond to types and the mechanism that provides implementations for their methods also generates subtypes.

The subtype relation In general, languages ban cycles in the subtype relation between classes, that is, we cannot have both `A` subtype `B` and `B` subtype `A`, unless `A` and `B` coincide. The subtype relation is, therefore, a partial order. In many languages, this relation has a maximal element: the type of which all other types defined by classes are subtypes. We will denote such a type with the name `Object`. Messages that an instance of `Object` accepts are fairly limited (it is an object of maximum generality). We find in general a method for cloning which returns a copy of the object, an equality method and a few others. Moreover, some of these methods could be abstract in `Object` and therefore require redefinition before they can be used.

It can be seen that it is not guaranteed that, given a type `A`, there exists a *unique* type `B` which is the *immediate supertype* of `A`. In the following example:

```
abstract class A{
   public int f();
}

abstract class B{
```

[10]A method for which there is only a signature is, in Java, said to be an *abstract* method; in C++, it is a *pure virtual* member function.

```
   public int g();
}

class C extending A,B{
   private x = 0;
   public int f(){
      return x;
   }
   public int g(){
      return x+1;
   }
}
```

The type C is a subtype of A as well as of B and there is no other type included between C and the other two. The subtype hierarchy is not a tree in general, but just an acyclic oriented graph (see Fig. 10.3).

Constructors We have already seen that an object is a complex structure which includes data and code all wrapped up together. The creation of an object is therefore also an operation of a certain complexity which consists of two distinct actions: first the allocation of necessary memory (in the heap or on the stack) and the proper initialisation of the object's data. This last action is performed by the *constructor* of the class. That is, by code associated with the class and which the language guarantees will execute at exactly the same time the instance is created. The constructor mechanism is of some complexity because the data in an object consists not only of that explicitly declared in the class whose instance is being created, but also the data declared in its superclasses. In addition, more than one constructor is often permitted for a class. We have, therefore, a series of questions about constructors which can be summarised as the following:

- *Constructor selection*. When the language permits it, there is more than one constructor for a class, so how is the choice made as to which one to use when creating a specific object? In some languages (for example C++ and Java), the name of the constructor is the same as the name of the class. Multiple constructors all have the same name and must be distinguished either by their type or the number of the arguments or both (they are therefore overloaded, with static resolution). Since C++ permits the implicit creation of objects on the stack, there are spe-

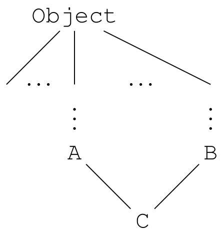


Fig. 10.3 The subtype relation is not a tree

cific mechanisms for selecting the appropriate constructor to use in each case.[11] Other languages allow the programmer to choose constructor names freely (but the constructors remain syntactically distinct from ordinary methods) and require that every creation operation (our `new`) is always associated with a constructor.

- *Constructor chaining*. How and when is the part of object that belongs to the superclass initialised? Some languages limit themselves to executing the constructor of the class whose instance they are constructing. If the programmer intends calling superclass constructors, it must be done explicitly. Other languages (among them C++ and Java) guarantee, on the other hand, that when an object is initialised, the constructor for its super class will be invoked (*constructor chaining*) before any other operations are performed by the constructor specific to the subclass. Once more, there are many issues which each language has to resolve. Among these, the two most important are determining which superclass constructor to invoke and how to determine its arguments.

10.2.5 Inheritance

We have seen that a subclass can redefine the methods of its superclass. But what happens when the subclass does not redefine them? In such cases, the subclass inherits methods from the superclass, in the sense that the implementation of the method in the superclass is made available to the subclass. For example, `NewCounter` inherits from `Counter` the data item `x` and methods `inc` and `get` (but not `reset` which is redefined).

More generally (including in our definition, phenomena also present in class-less object-oriented languages), we can characterise inheritance as a mechanism which permits the definition of new objects based on the reuse of pre-existing ones.

Inheritance permits code reuse in an extendable context. By modifying the implementation of a method, a class automatically makes the modification available to all its subclasses, with no intervention required on the part of the programmer.

Is important to understand the difference between the relation of inheritance and that of subtype. The concept of subtype has to do with the possibility of using an object in another context. It is a relation between the interfaces of two classes. The concept of inheritance has to do with the possibility of reusing the code which manipulates an object. It is a relation between the implementations of two classes. We are dealing with two mechanisms which are completely independent, even if they are often linguistically associated in several object-oriented languages. Both C++ and Java have constructs which can simultaneously introduce both relations between the two classes, but this does not mean that the concepts are the same. In the literature, we sometimes see the distinction between *implementation inheritance* (inheritance) and *interface inheritance* (our subtype relation).

Inheritance and visibility We have already seen that there are two views of each class: the private and the public, with the latter shared between all the clients of

[11] The case of a constructor which takes a single object as the argument (a copy constructor in C++ jargon) is especially tricky and the source of many programming errors.

the class. A subclass is a particular client of the superclass. It uses the methods of the superclass to extend the functionality of the superclass, but sometimes it has to access some non-public data. Many languages therefore introduce a third view of a class: one for subclasses. Taking the term from C++, we can refer to it as the *protected* view of a class.[12]

If the subclass has access to some details of the super class' implementation, the subclass depends in a much tighter way on the superclass. Every modification of the superclass will require a modification of the subclass. From the pragmatic viewpoint, this is reasonable only if the two classes are "close" to each other, for example belonging to the same package. The stronger the pairing of the two classes, the more the resulting system becomes difficult to modify and maintain. However, making protected and public interfaces coincide can be too restrictive. We may think of a class hierarchy providing data structures that gradually become more specialised. Being able to access the representation of the data structures, the subclass will be able to implement its own operations in a more efficient way.

Single and multiple inheritance In some languages, a class can inherit from a single immediate superclass. The inheritance hierarchy is therefore a tree and we say that the language has single (or simple) inheritance. Other languages, on the other hand, allow a class to inherit methods from more than one immediate superclass; the inheritance hierarchy in such a case is an acyclic oriented graph and the language has multiple inheritance.

There are only a few languages which support multiple inheritance (among which are C++ and Eiffel), because it presents problems that do not have an elegant solution at either the conceptual or implementation level. The fundamental problems relate to name clashes. We have a name clash when a class `C` simultaneously inherits from `A` and `B`, which both provide implementation for methods with the same signature. The following is a simple example:

```
class A{
   int x;
   int f(){
      return x;
   }
}
class B{
   int y;
   int f(){
      return y;
   }
}
class C extending A,B{
   int h(){
      return f();
   }
}
```

[12]Java also has a `protected` visibility modifier, which is, though, more liberal than that in C++; it grants visibility to an entire package and not just to classes.

Inheritance and Subtypes in Java

The subtype relation is introduced in Java using either the `extends` clause (to define subclasses) or the `implements` clause when a class is declared a subtype of one or more interfaces (for Java, an interface is a kind of incomplete abstract class in which only names and method signatures are included—they do not include implementations). The inheritance relation is introduced with the `extends` clause whenever the subclass does not redefine a method and therefore uses an implementation from the superclass. It should be noted there is never inheritance *from* an interface because the interface has nothing to be inherited. The language constrains every class to having a single immediate superclass (that is a single superclass can be named in an `extends`), but allows that a single class (or interface) implements more than one interface:

```
interface A{
   int f();
}
interface B{
   int g();
}
class C{
   int x;
   int h(){
      return x+2;
   }
}
class D extends C implements A,B{
   int f(){
      return x;
   }
   int g(){
      return x+1;
   }
}
```

Java therefore has single inheritance. The inheritance hierarchy is a tree organised by `extends` clauses. In addition, the inheritance relation is, in Java, always a subhierarchy of the subtype hierarchy.

The situation of the example, when at the same time we have inheritance from superclass and implementation of some abstract interface, is often called *mix-in* inheritance (because the names of the abstract methods in the interface are mixed in with the inherited implementations). As usual, current terminology is often imprecise and confuses subtypes with inheritance. In many manuals (and even in the official definition of Java ...), it is said that Java has single inheritance for classes, but multiple inheritance for interfaces. According to our terminology there is no true inheritance where interfaces are involved.

Inheritance and Subtypes in C++

The C++ mechanisms that allow for the definition of the inheritance relation, and those responsible for subtyping, are not independent. The definition of derived classes (the C++ term for subclass) introduces the inheritance relation. It also introduces a subtype relation when the derived class declares its base class (that is the superclass) as `public`; when the base class is not public, the derived class inherits from the base class but there is not subtype relation.

```
class A{
public:
   void f(){...}
   ...
}
class B : public A{
public:
   void g(){...}
   ...
}
class C : A{
public:
   void h(){...}
   ...
}
```

Both classes `B` and `C` inherit from `A` but only `B` is a subtype of `A`.

Since the subtype relation follows that of subclass, with the tools seen so far, we could not introduce an *interface subtype*, that is, a class derived from a base class which provides no implementations but just fixes an interface. It is to this end that C++ introduces the concept of *abstract* base class, in which some methods need not have an implementation. In such cases, as with Java interfaces, we can have subtypes without inheritance.

Which of the two methods named `f` is inherited in `C`? We can solve this problem in three different ways, none of which is totally satisfactory:

1. Forbid name clashes syntactically.
2. Require that any conflict should be resolved by the programmer's appropriately qualifying every reference to a name that is in conflict. For example, the body of `h` in class `C`, should be written as `B::f()` or as `A::f()`, which is the solution adopted by C++.
3. Decide upon a convention for solving the conflict, for example favouring the first-class named in the `extending` clause.

From the pragmatic point of view, it is possible to give examples of situations in which any of these solutions is unnatural and counterintuitive. As far as explicit solution is concerned, it can be seen that the conflict could observed not in `C` but in one of its subclasses (if `f` is not called inside `C`). The designer must therefore know the class hierarchy with some precision. However it might be, in cases like this, it is methodologically better to redefine `f` in `C`. For example:

```
class C extending A,B{
   int f(){
      return A::f();
   }
   int h(){
      return this.f();
   }
}
```

such that in the subclasses of `C`, there are no name clashes.

The most interesting problems of multiple inheritance are present in the so-called diamond problem. This case arises when a class inherits from two superclasses, each of which inherits from the same, single superclass. A simple situation of this kind is the following (the diamond-shaped inheritance hierarchy is shown graphically in Fig. 10.4).

```
class Top{
   int w;
   int f(){
      return w;
   }
}
class A extending Top{
   int x;
   int g(){
      return w+x;
   }
}
class B extending Top{
   int y;
   int f(){
      return w+y;
   }
   int k(){
      return y;
   }
}
class Bottom extending A,B{
   int z;
   int h(){
      return z;
   }
}
```

In this case, as well, we have the usual problem of name conflict but it is crucially the implementation question which is more relevant. That is, devise a technique allowing the correct *and efficient* selection of the code for `f` when this method is invoked on an instance of class `Bottom`. We will deal with this in Sect. 10.3.4.

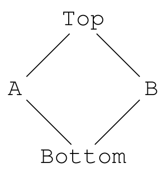


Fig. 10.4 The Diamond problem for multiple inheritance

In conclusion, multiple inheritance is a highly flexible tool for the combination of abstractions corresponding to distinct functionalities. Some of the situations in which it works are in reality better expressed using subtype relations ("inheriting" from abstract classes). There are no simple, unequivocal, and elegant solutions to the problems that multiple inheritance poses. The cost benefit balance between single and multiple inheritance is equivocal.

10.2.6 Dynamic Method Lookup

Dynamic method lookup (or dispatch) is the heart of the object-oriented paradigm. In this, the characteristics that have already been discussed combine to form a new synthesis. In particular, dynamic method lookup allows compatibility of subtypes and abstraction to coexist. This is, in particular, something that was seen to be problematic for abstract data types (page 281).

Conceptually, the mechanism is very simple. We have already seen that a method defined for one object can be redefined (overridden) in objects that belong to subtypes of the original object. Therefore when a method, `m`, is invoked on an object, `o`, there can be many versions of `m` possible for `o`. The selection of which implementation for `m` is to be used in an invocation

```
o.m(parameters);
```

is a function of the type of the object receiving the message. Note that what is relevant is the type of the *object* which receives the message, not the type of the reference (or name) to that object (which is instead static information).

Let us give an example in our pseudo-language with classes. Figure 10.5 repeats the counters discussed in Sects. 10.2.2 and 10.2.4. In the context of these class declarations, we now execute the following fragment:

```
NewCounter n = new NewCounter();
Counter c;
c = n;
c.reset();
```

Fig. 10.5 Two classes for counters

```
class Counter{
   private int x;
   public void reset(){
      x = 0;
   }
   public int get(){
      return x;
   }
   public void inc(){
      x = x+1;
   }
}
class NewCounter extending Counter{
   private int num_reset = 0;
   public void reset(){
      x = 0;
      num_reset = num_reset + 1;
   }
   public int howmany_resets(){
      return num_reset;
   }
}
```

The (static) type of the name `c` is `Counter` but it refers (dynamically) to an instance of `NewCounter`. So it will be the `reset` method of `NewCounter` that is invoked.

The canonical example is the one which (negatively) concluded Sect. 10.1. Both `Counter` and `NewCounter` are stored in a data structure whose type is their supertype:

```
Counter V[100];
```

Now we apply the `reset` method to each:

```
for (int i=1; i<100; i=i+1)
   V[i].reset();
```

Dynamic lookup assures us that the correct method will be invoked on any counter. In general, a compiler will be unable to decide what will be the type of the object whose method will be invoked, hence the dynamicity of this mechanism.

The reader should have noted a certain analogy between overloading and dynamic method lookup. In both cases, the problem is the same: that of resolving an ambiguous situation in which a single name can have more than one meaning. Under overloading, though, the ambiguity is resolved statically, based on the type of the *names* involved. In method lookup, on the other hand, the solution of the problem is at runtime and makes use of the dynamic type of the object and not of its name. It is not, however, a mistake to think of method lookup as a runtime overloading operation in which the object receiving the message is considered the first (implicit) argument of the method whose name has to be resolved.[13]

Is important to observe explicitly that dynamic method lookup is at work even when a method in an object invokes a method in the same object, as happens in the next fragment:

```
class A{
   int a = 0;
   void f(){g();}
   void g(){a=1;}
}
class B extending A{
   int b = 0;
   void g(){b=2;}
}
```

Let us now assume that we have an object `b` which is an instance of `B` and we want to invoke on `b` the method `f` inherited by `b` from `A`. Now, `f` calls `g`: which of the two implementations of `g` will be executed? Recall that the object which receives the message is also an important parameter to the method. The call of `g` in the body of `f` can be written more explicitly as `this.g()`, where, recall, `this` is a reference to the current object. The current object is `b` and therefore the method which will be invoked is that of class `B`. It can be seen that, in this way, a call to a method such as `this.g()` in the body of `f` can refer to (implementations of) methods that are not yet written and which will be available only later through the class hierarchy. This mechanism, through which the name `this` becomes bound dynamically to the current object, is called *late binding* of `self` (or of `this`).

Let us explicitly note that, unlike overriding, shadowing is a completely static mechanism. Consider, for example, the code:

```
class A{
   int a = 1;
   int f(){return -a;}
}
class B extending A{
   int a = 2;
   int f(){return a;}
}

B obj_b = new B();
print(obj_b.f());
print(obj_b.a);
```

[13]The reader will not be surprised if, once more, object-oriented programming jargon contributes to the confusion. It is not uncommon to hear (and to see written) that dynamic method lookup permits polymorphism. In our terminology there is no polymorphism because we are not in the presence of a single, uniform piece of code. It is possible to talk of polymorphism in object-oriented programming but this has to do with subtypes. We will deal with this in Sect. 10.4.1.

Dynamic Dispatch in C++

In languages like Java or Smalltalk, each method invocation happens using dynamic dispatch. C++ has, as design goals, efficiency of execution and compatibility with C, meaning by this that the use of a C construct in a C++ program must be efficient as in C.

In C++, we have, therefore, static method dispatch (analogous to function call), as well as a form of dynamic dispatch which is performed using *virtual functions*. When a method (that is a member function in C++ terminology) is defined, it is possible to specify whether it is a virtual function or not. Only the overriding of virtual functions is permitted. Dynamic dispatch is performed on virtual member functions.

Let us note, incidentally, that it is not forbidden to define, in some class, a function of the same name and signature as a non-virtual function defined in the superclass. In such a case, we do not have redefinition, but overloading, and this can be resolved statically by the compiler using the type of the *name* used to refer to that object. In the example that follows we declare a class A and a subclass B:

```
class A{
public:
   void f(){printf("A");}
   virtual void g(){printf("A");}
}
class B : public A{
public:
   void f(){printf("B");}
   virtual void g(){printf("B");}
}
```

If, now, we have a pointer a of type A*, which points to an instance of B, invocation of the function a->f() prints A, while invocation of the virtual function a->g() prints B.

```
A obj_a = obj_b;
print(obj_a.f());
print(obj_a.a);
```

where print denotes a method which outputs the integer value passed as an argument. The first two invocations of print produce the value 2 twice, as should be obvious. The third call will also print the value 2, given that, as already seen above, the method f has been redefined in class B. In fact the object created (with new B()) is an instance of B, and hence every access to it, even those through a variable of type A (as in the case of obj_a), uses the redefined method (which is obviously using the redefined fields in class B). The last occurrence of print, instead, produces the value 1. In this case, in fact, given that we are not dealing with the invocation of a method but accessing a field, it is the type of the current reference

Multimethod languages

In the languages we have considered so far, a method invocation has the form:

```
o.m(parameters);
```

Dynamic method lookup uses the run-time type of the object receiving the method, while uses the static types of the parameters. In some other languages (for example, CLOS), this asymmetry is removed. Methods are no longer associated with classes but are global names. Each method name is overloaded by a number of implementations (which in this case are called *multimethods*) and the object on which the method is invoked is passed to every multimethod. When invoking a multimethod, the code that must be executed is chosen dynamically on the basis of the (dynamic) type of both the receiver and all the arguments.

In these languages, we speak of *multiple dispatch* rather than single dispatch in languages where there exists a privileged receiver.

In languages with multiple dispatch, the analogy between (multiple) dynamic dispatch and "dynamic overloading" is more evident; multiple dispatch consists in the (runtime) resolution of an overloading on the basis of dynamic type information.

which determines which field is to be considered. Given that `obj_a` is of type `A`, when we write `obj_a.a`, the field in question is the one in class `A` (initialised to one).

This distinction between overriding and shadowing, is explained at the implementation level using the fact that the instance object of class `B` also contains all the fields of the superclasses of `B`, as we will make clear in the next section.

10.3 Implementation Aspects

Every language has its own implementation model which is optimised to the specific features that the language provides. We can, however, indicate some common lines which indicate the main problems and their solutions.

Objects An instance object of class `A` can be used as the value of any superclass of `A`. particular, it is possible to access not just the instance variables (data fields) explicitly defined in `A`, but also those defined in its superclasses. An object can be represented as if it were a record, with as many fields as there are variables in the class of which it is an instance, *in addition* to all those appearing in its superclasses. In a case of shadowing, or rather when a name of an instance variable of the class is used with the same type in a subclass, the object contains additional fields, each one corresponding to a different declaration (often, the name used in the superclass

```
class A{
   int a;
   void f(){...}
   void g(){...}
}
class B extending A{
   int b;
   void f(){...} // redefined
   void h(){...}
}
B o = new B();
```

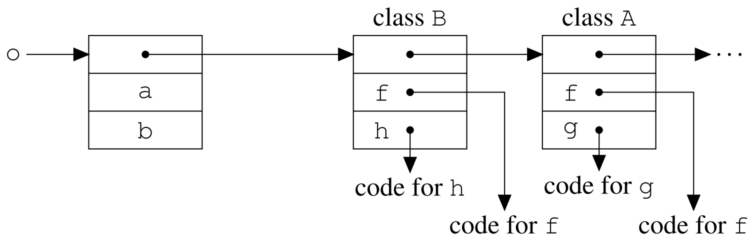


Fig. 10.6 A simple implementation of inheritance

is not accessible in the subclass, if not using particular qualifications, for example `super`). The representation also contains a pointer to the descriptor of the class of which it is an instance.

In the case of a language with a static type system, this representation allows the simple implementation of subtype compatibility (in the case of single inheritance). For each instance variable, the offset from the start of the record is statically recorded. If access to an instance object of a class is performed using a (static) reference having as type one of the superclasses of that object, static type checking ensures that access can be made only to a field defined in the superclass, which is allocated in the initial part of the record.

Classes and inheritance The representation of classes is the key to the object orientation abstract machine. The simplest and most intuitive implementation is the one which represents the class hierarchy using a linked list. Each element represents a class and contains (pointers to) the implementation of the methods that are either defined or redefined in that class. The elements are linked using a pointer which points from the subclass to the immediate superclass. When a method `m` of an object `o` which is an instance of a class `C` is invoked, the pointer stored in `o` is used to access the descriptor of `C` and determine whether `C` contains an implementation of `m`. If it does not, the pointer is set to the superclass and the procedure is repeated (see Fig. 10.6). This approach is used in Smalltalk, although it is inefficient because every method invocation requires a linear scan of the class hierarchy; the advantage of the technique is its simplicity. We will shortly discuss more efficient implementations after we have seen how to access instance variables from within a method invocation.

Late binding of self A method is executed in a way similar to a function. An activation record for the method's local variables, parameters and all the other information is pushed onto the stack. Unlike a function, though, a method must also access the instance variables of the object on which it is called; the identity of the object is not known at compile time. However, the structure of such an object is known (it depends on the class) and, therefore, the offset of every instance variable in an object's representation is statically known (subject to conditions depending upon the language). A pointer to the object which received the method is also passed as a parameter when a method is invoked. During execution of the body of the method, this pointer is the method's `this`. When the method is invoked on the same object that invokes it, `this` still is passed as a parameter. During the execution of the method, every access to an instance variable uses the offset from this pointer (instead of having an offset from a pointer into the activation records, as is the case for local variables of functions). From a logical point of view, we can assume that the `this` pointer is passed through the activation record typically with all other parameters, but this would cause a doubly-indirect access for every instance variable (one to access the pointer using the activation record pointer and one to access the variable using this). More efficiently, the abstract machine will maintain the current value of `this` in a register.

10.3.1 Single Inheritance

Under the hypothesis that the language have a static type system, the implementation using linked lists can be replaced by another, more efficient one, in which method selection requires constant time (rather than time linear in the depth of the class hierarchy).

If types are static, the set of methods that any object can invoke is known at compile time. The list of these methods is kept in the class descriptor. The list contains not just the methods that are explicitly defined or redefined in the class but also all the methods inherited from its superclasses.[14] Following C++'s terminology, we will use the term *vtable* (*virtual function table*) to refer to this data structure. Each class definition has its own vtable and all instances of the same class share the same vtable. When a subclass, `B`, of the class `A` is defined, `B`'s vtable is generated by copying the one for `A`, replacing all the methods redefined in `B` and adding the new methods that `B` defines at the bottom (this is shown in Fig. 10.7).

The fundamental property of this data structure is, that, if `B` is a subclass of `A`, `B`'s vtable contains a copy of `A`'s vtable as its initial part; redefined methods are appropriately modified. In this way, the method invocation costs only two indirect accesses, the offset of every method in the vtable being statically known. It can be

[14]We are assuming that dynamic lookup applies to all methods. In cases like C++, where only virtual methods can be redefined, the descriptor contains only the latter, while ordinary method calls are treated like normal function calls.

```
class A{
   int a;
   char c;
   void g(){...}
   void f(){...}
}
class B extending A{
   int a;
   int b;
   void h(){...}
   void f(){...}
      // redefined

}
B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```

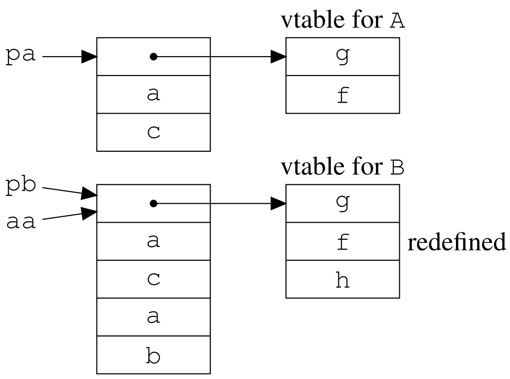


Fig. 10.7 Implementation of simple inheritance

seen that this implementation obviously takes into account the fact that it is possible to access an object with a reference that belongs statically to one of its superclasses. In the example in Fig. 10.7, when method `f` is invoked, the compiler computes an offset that remains the same whether `f` is invoked on an object of class `A` or on an object of class `B`. Different implementations of `f` defined in the two classes are located at the same offset into the vtables of `A` and `B`.

On the whole, if we have a static reference `pa` of class `A` to some object of one of its subclasses, we can compile a call to method `f` (which, we assume to be the nth in `A`'s vtable) as follows (we have assumed that the address of a method occupies w bytes):

```
R1 := pa                          // access the object
R2 := *(R1)                       // vtable
R3 := *(R2 + (n − 1) × w)         // indirect to f
call *(R3)                        // call f
```

Ignoring support for multiple inheritance, this implementation scheme is almost the same as used in C++.

Downcasting If a class' vtable also contains the name of the class itself, the implementation we have discussed allows downward casts in the class hierarchy. This is called *downcasting* and is a fairly frequently used mechanism. It permits the specialisation of the type of object by running in the opposite direction to the subtype hierarchy. A canonical example of the use of this mechanism is found in some library methods which are defined to be as general as possible. The class `Object` might use the following signature to define a method for copying objects:

```
Object clone(){...}
```

The semantics is that `clone` returns an exact copy of the object which invoked it (same class, same values in its fields, etc.). If we have an object `o` of class `A`, an invocation of `o.clone()` will return an instance of `A`, but the static type as determined by the compiler for this expression is `Object`. In order to be able meaningfully to use this new copy, we have to "force" it to type `A`, which we can indicate with a cast:

```
A a = (A) o.clone();
```

By this, we mean that the abstract machine checks that the dynamic type of the object really is `A` (in the opposite case, we would have a runtime type error).[15]

10.3.2 The Problem of Fragile Base Class

In the case of simple inheritance, the implementation of the mechanism described above is reasonably efficient, given that all the important information, except the `this` pointer is statically determined. This staticity, though, can be shown to be the source of problems in a context known as the *fragile base class problem*.

An object-oriented system is organised in terms of a very large number of classes using an elaborate inheritance hierarchy. Often some of these general classes are provided by libraries. Modifications to a class located very high in the hierarchy, can be felt in its subclasses. Some superclasses can, therefore, behave in a "fragile" manner because an apparently innocuous modification to the superclass can cause the malfunctioning of subclasses. It is not possible to identify fragility by analysing only the superclass. It is necessary to consider the entire inheritance hierarchy, something which is often impossible because whoever wrote the superclass generally does not have access to all subclasses. The problem can arise for a number of reasons. Here, we will only distinguish two main cases:

- The problem is architectural. Some subclass exploit aspects of the superclass' implementation which have been modified in the new version. This is an extremely important problem for software engineering. It can be limited by reducing the inheritance relation in favour of the subtype relation.
- The problem is implementational. The malfunctioning of the subclass depends *only* on how the abstract machine (and the compiler) has represented inheritance. This case is sometimes referred to as the problem of *fragile binary interface*.

In this text, the first case is not our main interest; the second is. A typical case shows up in the context of separate compilation where a method is added to a superclass, even if the new method interacts with nothing else. Starting from the following:

[15]This is Java notation. C++ allows the same notation to be used but, for compatibility with C, it accepts the expression at compile time without introducing dynamic checks. A cast with dynamic checks would be written `dynamic_cast<A*>(o)` in C++.

```
class Upper{
   int x;
   int f(){..}
}
class Lower extending Upper{
   int y;
   int g(){return y + f();}
}
```

the super class is modified as:

```
class Upper{
   int x;
   int h(){return x;}
   int f(){..}
}
```

If inheritance is implemented as described in Sect. 10.3.2, the subclass `Lower` (which is already compiled and linked to `Super`) ceases to function correctly because the offset used to access to method `f` has been changed, since it is determined statically. To solve this problem it is necessary to recompile all the subclasses of the modified classes, a solution which is not always easy to perform.

To obviate this question, it is necessary to compute dynamically the offset of methods in the vtable (and also the offset of instance variables in the representation of objects), in some reasonably efficient manner. The next section describes one possible solution.

10.3.3 Dynamic Method Dispatch in the JVM

In this section we present a simple description of the technique used by the Java virtual machine (JVM), the abstract machine which interprets the intermediate (bytecode) language generated by the standard Java compiler. For reasons of space, we cannot go into the details of the JVM's architecture; we note that it is a stack-based machine (it does not have user-accessible registers and all operation operands are passed on a stack contained in the activation record of the function that is currently being executed). The JVM has modules for checking operation security. We limit ourselves to discussing the broad outlines of the implementation of inheritance and method dispatching.

In Java, the compilation of each class produces a file which the abstract machine loads dynamically when the currently executing program refers to the class. This file containing a table (the *constant pool*) for symbols used in the class itself. The constant pool contains entries for instance variables, public and private methods, methods and fields of other classes used in the body of methods, names of other classes mentioned in the body of the class etc. With each instance variable and method name is recorded information such as the class where the names are defined and their type. Every time that the source code uses the name, the intermediate code of the JVM looks up the index of that name in the constant pool (to save space—it does not look up the name itself). When, during execution, reference is made to a name for the first time (using its index), it is *resolved* using the information in the constant pool, the necessary classes (for example the one in which the names are introduced) are loaded, visibility checks are performed (for example the invoked method must really exist in the class referring to it, is not a private, etc.); type checks are also performed. At this point, the abstract machines saves a pointer to this information so that the next time the same name is used, it is not necessary to perform resolution a second time.

The representation of methods in a class descriptor can be thought of as being analogous to that in the vtable.[16] The table for a subclass starts with a copy of the one for its superclass, where redefined methods have their new definitions instead of the old ones. Offsets, however, are not statically calculated. When a method is to be executed, four main cases can be distinguished (which correspond for distinct bytecode instructions):

1. The method is static. This is for a method associated with a class and not an instance. No reference (explicit or implicit) can be made to `this`.
2. The method must be dispatched dynamically (a "virtual" method).
3. The method must be dispatched dynamically and is invoked using `this` (a "special" method).
4. The method comes from an interface (that is from a completely abstract class which provides no implementation—an "interface" method).

Ignoring the last case for the moment, the 3 other cases can be distinguished primarily by the parameters passed to the method. In the first case, only the parameters named in the call are passed. In the second case, a reference is passed to the object on which the method is called. In case 3, a reference to `this` is passed. Let us therefore assume that we can invoke method `m` on object `o` as:

```
o.m(parameter)
```

Using the reference to `o`, the abstract machine accesses the constant pool of its class and extracts the *name* of `m`. At this point, it looks up this name in the class' vtable and determines its offset. The offset is saved in case of a future use of the same method on the same object.

However the same method could also be invoked on other objects, possibly belonging to subclasses of the one which `o` belongs to (e.g., a `for` loop in whose body a call to `m` is repeated on all the objects in an array). To avoid calculating the offset each time (which would be the same independent of the effective class of which the

[16] In reality, the specification of the JVM does not prescribe any particular representation and limits itself to requiring that the search for the method to be executed is semantically equivalent to the dynamic search for the method name in the list of subclasses described above. The most common implementation, however, is of a table that is very similar to the vtable.

method `m` is called on is an instance), the JVM interpreter uses a "code rewrite" technique. It substitutes for the standard lookup instruction generated by the compiler an optimised form which takes as its argument the offset of the method in the vtable. In order to fix ideas (and simplifying a great deal), when translating the invocation of a virtual method `m`, the compiler might generate the byte code instruction:

```
invokevirtual index
```

where `index` is the index of the name `m` in the constant pool. During execution of this instruction, the JVM interpreter calculates the offset `d` of `m` in its vtable and *replaces* the instruction above with the following:

```
invokevirtual_quick d np
```

Here, `np` is the number of the parameters that `m` expects (and which will be found on the stack that is part of its activation record). Every time that the flow of control returns to this instruction, `m` is invoked without overhead.

There remains the case of the invocation of an interface method (that is, case 4 above). In this case, the offset might not be the same in 2 invocations of the same method on objects in different classes. Consider the following situation (in which we use Java syntax rather than pseudocode):

```
interface Interface{
   void foo();
}
public class A implements Interface{
   int x;
   void foo(){...}
}
public class B implements Interface{
   int y;
   int fie(){...}
   int foo(){...}
}
```

Both `A` and `B` implement `Interface` and therefore are its subtypes. The offset of *foo* is different in the two classes, though. Consider our usual loop:

```
Interface V[10];
...
for (int i = 0; i<10; i=i+1)
   V[i].foo();
```

At runtime, we do not know whether the objects contained in `V` will be instances of `A`, `B` or some other class which implements `Interface`. The compiler could have generated the JVM instruction for the body of loop:

```
invokeinterface index, 0
```

(the 0 serves to fill a byte that will be used in the “quick” version). It would not be correct directly to replace this instruction by a “quick” version which returns only the offset because changing the object could also change the class of which it is an instance. What we can do is to save the offset but we cannot destroy the original name of the method; we thus rewrite the instruction as:

```
invokeinterface_quick nome_di_foo, d
```

Here `name_of_foo` is suitable information with which to reconstruct the name and the signature of `foo`; `d` is the offset determined beforehand. When this instruction is executed again, the interpreter accesses the vtable using the offset `d` and checks that there is a method with the requested name and signature. In the positive case, it is invoked; in the negative case, it searches for the method by name in the vtable, as it would were this the first time it had been seen, and determines a new offset `d′`, then writes this value in the code in place of `d`.

10.3.4 Multiple Inheritance

The implementation of multiple inheritance poses interesting problems and imposes a non-negligible overhead. The problems are of 2 orders: on the one hand, there is the problem of identifying how it is possible to adapt the vtable technique to handle method calls; on the other (and the problem is more interesting and also has an impact on the language) there is the need to determine what to do with the data present in the superclasses. This will lead to 2 different interpretations of multiple inheritance, which we can refer to as replicated and shared inheritance. We will take these problems successively.

Vtable Structure We will consider the example in Fig. 10.8. It is clear that is not possible to organise the representation of an instance of `C`, nor a vtable for `C` in such a way that the initial part coincides with the corresponding structure in both `A` and `B`.

In order to represent an instance of `C`, we may begin with the fields of `A` and add then the fields of `B`. Finally, we need to list the fields specific to `C` (Fig. 10.9). We know that, using the subtype relation, we can access an instance of `C` using a static reference to any of the three types `A`, `B` and `C`. There are 2 distinct cases corresponding to the 2 different views of an instance of `C`. If it is an access with a static reference of type `C` or of type `A`, the technique described for simple inheritance works perfectly (except that the static offsets of the real instance variables in `C` must take into account the fact that the static variables belonging to `B` are in the middle).

When, on the other hand, an instance of `C` is seen as an object of `B`, it is necessary to take into account the fact that the variables of `B` are not at the start of the record but at a distance, d, from its start that is statically determined. When, therefore, we access an instance of `C` through a reference with static type `B`, it is necessary to add the constant, d, to the reference.

Fig. 10.8 An example of multiple inheritance

```
class A{
   int x;
   int f(){
      return x;
   }
}
class B{
   int y;
   int g(){
      return y;
   }
   int h(){
      return 2*y;
   }
}
class C extending A,B{
   int z;
   int g(){
      return x + y + z;
   }
   int k(){
      return z;
   }
}
```

Similar problems arise for the structure of the vtable. A vtable for `C` is divided into two distinct parts: the first contains the methods of `A` (possibly as redefinitions) and the methods really belonging to `C`. A second part comprises the methods of `B`, possibly with their redefinitions. In the representation of an instance object of `C`, there are two pointers to the vtable. Corresponding to the "view as `A` and `C`", it will have pointers to the vtable with methods from `A` and `C`. Corresponding to the "view as `B`", it will have pointers to the vtable with the methods of `B` (note that this is a vtable for class `C`; the class `B` has *another* vtable, which is used for its own instances. In general, every superclass of a class under multiple inheritance has its own special part in the vtables of its subclasses). Invocation of a method belonging to `C`, or which was redefined or inherited from `A`, follows the same rules as simple inheritance. To invoke a method inherited (or redefined) from `B`, the compiler must take into account that the pointer to the vtable for this method does not reside at the beginning of the object but is displaced by *d* positions. In the example in the figure, to call the method `h` of an instance of `C` seen as an object of type `B` (of which we have therefore a static name, `pb`): add *d* to the reference `pb`; using an indirect access, obtain the address of the start of the second vtable (that for `B`), then using the appropriate static offset invoke method `h`. This call needs one additional operation (the first add) in addition to that required for simple inheritance.

We have, however, neglected the binding for `this`. Which current-object reference should we pass to the methods of `C`? If we are dealing with methods in the first vtable (to which access is performed using a `this` which points to the start of the object, that is with the "`A` and `B` view"), it is necessary to pass the current value of `this`. This would be wrong for methods in the second vtable (to which

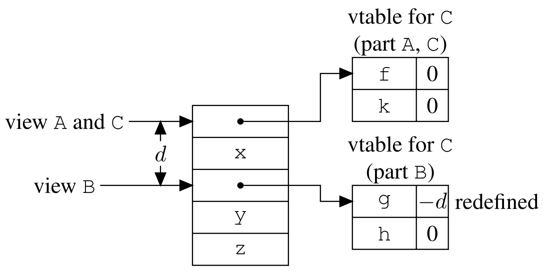


Fig. 10.9 Representation of objects and vtables for multiple inheritance

access is performed using the pointer to `this` plus the offset d), however. We must distinguish two cases:

- The method is inherited from `B` (the case of `h` in the figure). In such a case, it is necessary to pass the view of the object through which we have found vtable.
- The method is redefined in `C` (the case with `g`). In this case, the method might refer to instance variables of `A`, and so it is necessary to pass it a view of the superclass.

The situation is awkward because dynamic method lookup requires that this correction of the value of `this` is done at runtime. The simplest solution is to store this correction in the vtable, together with the name of the method. When the method is invoked, this correction will be added to the current value of `this`. In our example, the corrections are shown in Fig. 10.9 next to the name of the associated method. The correction is added to the view of the object through which the vtable was found.

Overall, if we have a reference, `pa`, to a view of class `C` of some object, we can compile a call to method `h` (which we can assume to be the nth in the vtable of `B`) as follows (the address of a method and the correction each occupy w bytes):

```
R1 := pa                                      // view A
R1 := R1 + d                                  // view B
R2 := *(R1)                                   // vtable for B
R3 := *(R2 + (n − 1) × 2 × w)                 // address of h
R2 := *(R2 + (n − 1) × 2 × w + w)             // correction
this := R1 + R2
call *(R3)                                    // call h
```

We have three instructions and an indirect access in addition to the sequence for calling a method using single inheritance.

Replicated multiple inheritance The preceding subsection has dealt with the case in which a class inherits from 2 superclasses. These superclasses, however, can themselves inherit from a common superclass, producing a diamond, as we discussed in Sect. 10.2.5:

```
class Top{
   int w;
   int f(){
      return w;
   }
}
class A extending Top{
   int x;
   int g(){
      return w+x;
   }
}
class B extending Top{
   int y;
   int f(){
      return w+y;
   }
   int k(){
      return y;
   }
}
class Bottom extending A,B{
   int z;
   int h(){
      return z;
   }
}
```

The instances and the vtable for A have a initial part that is a copy of the structure corresponding to Top. The same is the case for instances and the replicated vtable for B. Under replicated multiple inheritance, Bottom is constructed using to the approach we have already discussed, and therefore consists of two copies of the instance variables and methods of Top, as shown in Fig. 10.10.

The implementation does not pose other problems in addition to those already discussed. Name conflicts must be resolved explicitly (in particular it will not be possible to invoke the method f of Top on an object of class Bottom, nor assign an instance of Bottom to a static reference of type Top, because we would not know which of the two copies of Top to choose).

Shared multiple inheritance Replicated multiple inheritance is not always the conceptual solution which a software engineer has in mind when designing a diamond situation. When the class at the bottom of the diagram contains a single copy of the class at the top, we speak of replicated multiple inheritance. In such a case both A and B possess their own copy of Top, but Bottom has only one copy of it.

C++ allows shared inheritance using *virtual* base classes. When a class is defined as virtual, all of its subclasses always contain of a single copy of it, even if there is more than one inheritance path, as there is in the case of the diamond. With this mechanism, a class is declared virtual or nonvirtual once and for all. If we have a non-virtual class and, later, we discover that we require inheritance with sharing,

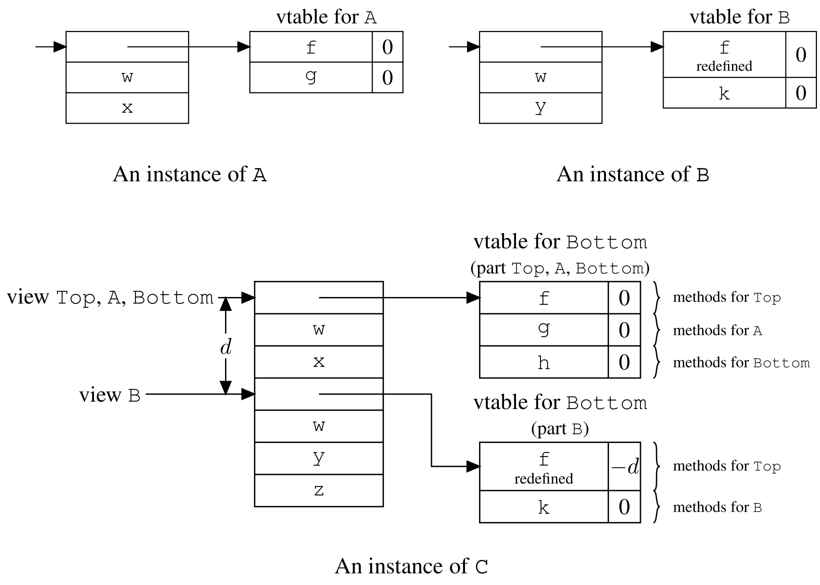


Fig. 10.10 Implementation of replicated multiple inheritance

there is nothing else to do but to rewrite the class and recompile the entire system. Worse, a class is virtual for all its subclasses, even if in some cases we wish that it were virtual for some and nonvirtual (that is replicated) for others. In such cases, it is necessary to define two copies of the class, one virtual and the other non virtual.

With inheritance with sharing there is usually a name-conflict problem. The problem is solved in arbitrary ways by different languages. In C++, for example, in the case of a virtual base class method that is redefined in a subclass, it is required that there is always a redefinition that dominates all the others in the sense that it appears in a class that is a subclass of all the other classes where this method is defined. In our example, for method `f`, the dominant redefinition is the one in class `B`. It is therefore the one inherited by `Bottom`. If both of the classes `A` and `B` redefine `f`, the definitions of these classes would be illegal in C++ because there would not be a dominant redefinition. Other languages allow inheriting classes to choose along which path the method is to be inherited; alternatively, they permit a complete qualification of names so as explicitly to choose the desired method. As usual, when dealing with multiple inheritance, there is no elegant solution that is clearly better than any other.

We now come to the implementation of the vtable for shared multiple inheritance. In Fig. 10.11, the implementation used in C++ is depicted schematically. C++ is a language in which a virtual class is shared by all of its subclasses. Since `Bottom` contains a single copy of `Top`, it is no longer possible to represent sub- and superclass in a contiguous way. To each class in the diamond there corresponds a specific view of the object. Corresponding to each view, there is both a pointer to the corresponding vtable, and a pointer to the shared part of the object. Access to instance

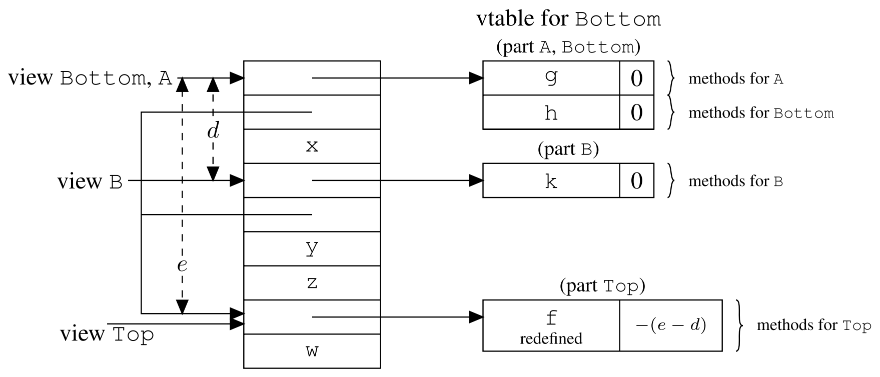


Fig. 10.11 Implementation of multiple inheritance with sharing

variables and methods of `Bottom`, `A` and `B` is the same as in the case of multiple inheritance without sharing. To access instance variables, or to invoke a method of `Top`, a preliminary indirect access instead is required (let us assume that `f` is the *n*th method of the vtable for `Top` and every address is w bytes wide):

```
R1 := pa                                  // view Bottom
R1 := *(R1 + w)                           // view Top
R2 := *(R1)                               // vtable for Top
R3 := *(R2 + (n − 1) × 2 × w)             // address of f
R2 := *(R2 + (n − 1) × 2 × w + w)         // correction
this := R1 + R2
call *(R3)                                // call h
```

The value necessary to correct the value of `this` is the difference between the view of the class in which the method is declared and the view of the class in which it is redefined.

Languages which allow more elaborate situations (for example, a single superclass is replicated in some classes and shared in others) require more sophisticated techniques that we cannot explain in this book.

10.4 Polymorphism and Generics

Having studied the characteristic aspects of the object-oriented paradigm and its associated implementations, in this section we will discuss the concept of subtype, in particular the relations between polymorphism and generics. We will begin the discussion in a general fashion; later, we will go into detail on generics in Java (which supports them in a very flexible manner).

10.4.1 Subtype Polymorphism

We introduced the concept of polymorphism in Sect. 8.8, where we also distinguished between two radically different forms of it: overloading (or *ad hoc* polymorphism) and universal polymorphism (where a single value has an infinite number of different types obtained by instantiating a general type scheme). We have also seen the concept of subtype polymorphism: a value exhibits subtype polymorphism when it has an infinite number of different types which are obtained by instantiating a general type scheme, substituting for some parameter the subtypes of an assigned type.

When a language has a structural notion of compatibility between types, as a relationship of some kind between classes, it is clear that there is a form of polymorphism, even though it is not as completely general as universal parametric polymorphism. For the subtype relation, each instance of a class `A` has, as type, all the superclasses of `A`. This property is particularly interesting in the case of methods. Let us consider a method (for simplicity we will restrict the discussion to a method with a single argument):

```
B foo(A x){...}
```

By virtue of subtype compatibility, `foo` can receive as an argument a value of *any subclass of* `A`. The code for `foo` does not need to be adapted to specific subclasses. The structure of classes (and the associated implementations) ensure that an operation that can be performed on values of type `A` can also be performed on values of its subclasses. The reader will certainly recognize polymorphism here. We are dealing with subtype polymorphism because it is not completely general but is *limited to subtypes of* `A`.

Using the symbol "`<:`" (introduced in Sect. 8.8) to denote the subtype relation, we can read `C<:D` as "`C` sub `D`" (`C` is a subtype of `D`). We also use the concept of universal quantification introduced in the box on page 239. With these notations, we can write the type of method `foo` as:

```
∀T<:A.T->B
```

which expresses the fact that this method can be applied to a value of any subclass whatsoever of `A`. This is a form of implicit subtype polymorphism because instantiation occurs automatically when the method is applied to a value of the subclass.

In reality, the notation that we are using is much more expressive than the pseudo-language that we have used so far. Let us consider a very simple method that returns its argument, of class `A`:

```
A Ide(A x){return x;}
```

From the semantic viewpoint, we can say that `Ide` has the type:

```
∀T<:A.T->T
```

Fig. 10.12 Stack class based on `Object`

```
class Elem{
   Object info;
   Elem prox;
}
class Stack{
   private Elem top = null; // empty stack
   boolean isEmpty(){
      return top==null;
   }
   void push (Object o){
      Elem ne = new Elem();
      ne.info = o;
      ne.prox = top;
      top = ne;
   }
   Object pop() {
      Object tmp = top.info;
      top = top.prox;
      return tmp;
   }
}
```

because it returns unaltered the object passed as its argument. However, assuming that `C<:A` and that `c` is an instance of `C`, the following assignment would be rejected by a *static* type checker:

```
C cc = Ide(c);
```

notwithstanding the fact that it is perfectly sensible from a semantic viewpoint. The point is that common languages have a type system that is incapable of recognizing that `∀T<:A.T->T` is a correct typing for `Ide` (as the ML typing system would do, see box on page 243), nor do they have a linguistic mechanism with which the programmer can express that the result type of a method *depends on the argument type*. However, this argument guarantees that the dynamic correction of a downcast:

```
C cc = (C) Ide(c);
```

will never cause runtime type errors.

The notation that we have used for types, in summary, allows us to express relationships between the type of the argument and the result type which are not possible in our pseudo-language (nor in the fragment of C++ or Java that we have analysed so far).

Before looking at a way in which the language can be extended, we give another, more meaningful, example of this limited form of polymorphism which it is possible through the tools we have developed so far. We want to implement a stack of elements represented as a linked list of elements. We do not immediately wish to fix the type of the objects that will appear in the list, so we declare them to be of type `Object`. Figure 10.12 shows one possible definition for a `Stack` class which contains generic elements. What interests us is that the pop method returns an Object (and can do nothing else, given that we have not made any assumptions about the type of the stack's elements). Now we can use our stack, but we must exercise some care. Let C be any class:

```
C c;
Stack s = new Stack();
s.push(new C());
c = s.pop();            // type error
```

The last line is incorrect because we are seeking to assign an Object to a variable of type C. It is it is necessary to force the assignment with a (dynamically checked) cast:

```
c = (C) s.pop();        // dynamic check
```

After these examples, we may recall what we said in Sect. 8.2.1: languages usually impose restrictions on the types which are stricter than those which are semantically reasonable, so that they can guarantee efficient static checking. The next section, however, will discuss language extensions which allow more powerful *explicit* subtype polymorphism.

10.4.2 Generics in Java

In Sect. 8.7, we discussed the concept of the C++ *template*. A program fragment where some types are indicated by *parameters*, which can then be appropriately instantiated by "concrete" types. This leads to an interesting form of polymorphism. Java introduces a similar concept (but with very different potential and implementation), giving it the name *generic*. This Java construct will be presented in this section with the objective of discussing its relationship with subtyping.

In Java, type definitions can be generic (that is, classes and interfaces can be generic), as well as methods. The syntax used is similar to that in C++ and uses angle brackets to denote parameters. Figure 10.13 shows the generic version of Fig. 10.12. The type <A> is the *formal type parameter* of the generic declaration and will have to be a instantiated later. A specific version of Stack is obtained by specifying which types must be substituted for A. For example stacks of strings or integers (Integer is a class which allows us to see an integer as an object and is different from the type int, which is formed of ordinary integers):

```
Stack<String> ss = new Stack<String>();
Stack<Integer> si = new Stack<Integer>();
```

```
class Elem<A>{
   A info;
   Elem<A> prox;
}
class Stack<A>{
   private Elem<A> top = null; // empty stack
   boolean isEmpty(){
      return top==null;
   }
   void push (A o){
      Elem<A> ne = new Elem<A>();
      ne.info = o;
      ne.prox = top;
      top = ne;
   }
   A pop() {
      A tmp = top.info;
      top = top.prox;
      return tmp;
   }
}
```

Fig. 10.13 Classes for a generic stack

Fig. 10.14 Generic pairs

```
class Pair<A,B>{
   private A a;
   private B b;
   Pair(A x, B y){    // constructor
      a=x; b=y;
      }
   A First(){
      return a;
      }
   B Second(){
      return b;
      }
}
```

The types which are supplied as the actual parameters must be class or array types.[17] The dynamic cast which had to be added in the non-generic version is now no longer necessary:

```
Stack<String> ss = new Stack<String>();
ss.push(new String("pippo"));
String s = ss.pop();
```

Figure 10.14 shows another simple example, this time of pairs of elements of any two types. A pair can obviously be instantiated by specific types:

[17]They must be reference types for implementation reasons that we have indicated on page 242.

```
Integer i = new Integer(3);
String v = new String ("pippo");
Pair<Integer,String> c = new Pair<Integer,String>(3,v);
String w = c.Second();
```

Methods too can be generic. Let us assume, for example, that we wanted to define a method for constructing diagonal pairs (that is with two identical components). A first attempt might be one using the subtype polymorphism already present in the language, so we define:

```
Pair<Object,Object> diagonal(Object x){
      return new Pair<Object,Object>(x,x);
}
```

However, in a way similar to the `pop` method above, if we apply `diagonal` to a string, the result is just a pair of `Objects` and not `Strings`:

```
Pair<Object,Object> co = diagonal(v);
Pair<String,String> cs = diagonal(v); // compilation error
```

It is necessary to parameterise the definition of `diagonal`, in particular the type of its result, as a function of the type of the argument:

```
<T> Pair<T,T> diagonal(T x){
      return new Pair<T,T>(x,x);
      }
```

The first pair of angle brackets introduces a type variable, the (formal) parameter to be used in the definition of the method. The reader will have recognized that `<T>` is a universal quantifier written using a different notation. The definition states that `diagonal` has type:

```
∀T.T->Coppia<T,T>.
```

Our diagonal can also be used without explicit instantiation:

```
Pair<Integer,Integer> ci = diagonal(new Integer(4));
Pair<String,String> cs = diagonal(new String("pippo"));
```

The compiler performs a genuine type inference (Sect. 8.8), in general more complex than the elementary one required in the example. In our case, it determines that in the first call `T` must be replaced by `Integer`, while in the second, by `String`.

A very important aspect of the type parameters (either in the type or method definition) is that they can have *bounds*. That is, they can specify that only a subtype of some classes is permitted. Let us illustrate this feature using an example.

Assume we have available an interface for geometric shapes which can be drawn. Then we have different specific classes which implements it:

```
interface Shape{
   void draw();
}
class Circle implements Shape{
   ...
   public void draw(){....}
}
class Rhombus implements Shape{
   ...
   public void draw(){....}
}
```

Let us now make use of a standard Java library (`java.util`). We have a list of shapes, that is objects of type `List<Shape>` and want to invoke the `design` method on each element of the list. The first idea will be to write[18]

```
void drawAll(List<Shape> forms){
   for(Shape f : forms)
      f.draw();
}
```

The definition is correct, but the method can be applied only to arguments which are of type `List<Shape>` (and not, for example, to `List<Rhombus>`). The reason is that the type `List<Rhombus>` is *not* a subtype of `List<Shape>`, for reasons that we will discuss shortly in Sect. 10.4.4.[19]

To obviate the problem, we can make the definition of the `drawAll` method parametric. Clearly, we cannot allow as arguments an arbitrary list because it must have to be composed of elements on which to call the `draw` method. The language allows us to specify this fact in the following way:

```
<T extends Shape> void drawAll(List<T> forms){
   for(Shape f : forms)
      f.draw();
   }
```

In this case, the formal type parameter is not an arbitrary type, but a type which extends `Shape` (here "extends" is used as a synonym of "is a subtype of"; by this, `Shape` extends itself). Using our notation of universal quantification, the type of `drawAll` becomes:

```
∀T<:Shape.List<T> -> void
```

[18]The body the method is an example of a *for-each* (see Sect. 6.3.3), an iterative construct which applies the body to all elements of the collection (list, array, etc.). In this case, for every `f`, in `forms`, it calls the method `draw`.

[19]For the moment, the reader must content themselves by knowing that, if `A<:B` and `DefPara<T>` is any parametric type definition (like `List<T>`), `DefPara<A>` and `DefPara<B>` are unrelated in the subtype hierarchy.

Now `drawAll` can be called on a list whose elements belong to any subtype of `Shape` (`Shape` is obviously a subtype of itself):

```
List<Rhombus> lr = ...;
List<Shape> lf = ...;
drawAll(lr);
drawAll(lf);
```

The bound mechanism for type variables is fairly sophisticated and flexible. In particular, a type variable can appear in its own bound. We restrict ourselves to a single example of this case, and we refer the reader to the bibliography for a deeper discussion.[20]

The elements of a class are comparable if the class implements the `Comparable` interface. We want to define a method which, given a list of elements of generic type as its argument, returns the maximum element of this list. What is the signature we can give to this `max` method? A first attempt is:

```
public static <T extends Comparable<T>>
   T max(List<T> list)
```

This expresses the fact that the elements of the list must be comparable with elements of the same type. We now try to use `max`. We have a type `Foo` which allows us to compare objects:

```
class Foo implements Comparable<Object>{...}
List<Foo> cf = ....;
```

We now invoke `max(cf)`: each element in `cf` (is a `Foo` and therefore) is comparable with any object, in particular with every `Foo`. But the compiler signals an error, because `Foo` does not implement `Comparable<Foo>`. In reality it is sufficient that `Foo` is comparable with one of its own supertypes:

```
public static <T extends Comparable<? super T>>
   T max(List<T> list)
```

Now, under the same conditions as before, `Max(cf)` is correct because `Foo` implements `Comparable<Object>`.

10.4.3 Implementation of Generics in Java

Unlike templates in C++, which are resolved at link time using code duplication and specialisation, Java generics always exist in a single copy. In this way, the idea

[20]The possibility that a type variable appears in its own bound is known in the literature as *F-bounded* polymorphism and is used in particular for typing *binary* methods, that is methods which have a parameter of the same type as the object which receives the method.

Wildcard

The `drawAll` method could be written in a more compact and elegant way using *wildcards*. The "?" character (which is read "unknown") stands for any type and can be used in generic definitions. For example, a value of type `List<?>` is a list of elements whose type is unknown. It might be thought that writing `List<?>` is the same thing as writing `List<Object>` but this is not the case because `List<Object>` is not a supertype, for example, of `List<Integer>`, while `List<?>` is really the supertype of all types `List<A>` for all `A`.

Using wildcards, the method `drawAll` could be written:

```
void drawAll(List<? extends Shape> forms){
   for(Shape f : forms)
      f.draw();
   }
```

In general, every wildcard can be always replaced by an explicit parameter. From the pragmatic viewpoint, for code clarity, a wildcard will be used where the parameter would be used only once (as in the case of `drawAll`), while an explicit parameter should be used when the type variable is used more than once (as in the case of `diagonal`, where the variable is used in the method's *result* type).

If two wildcards appear in the same construct, they must be considered distinct variables.

in parametric polymorphism that there exists a single value (a single class, a single method, etc.) which belongs to many different types (and, in the case of methods, works uniformly on different types) is respected by the implementation.

Generics are implemented by the compiler using an *erasure* mechanism. A program which contains generics is first subjected to type checking. When this static semantic checking has determined that everything is correct, the original program is transformed into a similar one in which all generics have been removed. All the information between angle brackets is eliminated (for example, every `List<Integer>` becomes `List`). All the other uses of type variables are replaced by the upper limit of the same variable (in general, this means `Object`). In the end, if, after these transformations, the cancelled program is incorrect with respect to types, appropriate (dynamic) casts are inserted. With this procedure, the use of generics does not worsen the code (in the majority of cases), either in terms of size or execution time. Perhaps more important are the other two consequences of this implementation:

- The underlying abstract machine (the JVM) requires no modification. The addition of generics does not imply modifications that need to be implemented on different architectures, but is localised to the compiler.
- It is possible to mix generic and non-generic code in a relatively simple way and, above all, in a way that is safe. Possible holes in the static type system that might

be locally produced by the simultaneous use of generic and non-generic code will be detected at runtime by the effect of the dynamic casts inserted by the compiler during the erasure process.

10.4.4 Generics, Arrays and Subtype Hierarchy

We have already observed that in Java, a generic type definition `DefPara<T>` does *not* preserve the subtype hierarchy. For any 2 types `A` and `B`, with `A<:B`, `DefPara<A>` and `DefPara<B>` are not related in the subtype hierarchy. We want now to clarify some of the reasons for this choice and we will consider the contrasting behaviour that Java exhibits for arrays: if `A<:B`, `A[]` *is* a subtype of `B[]`! After all, arrays are a primitive form of generic construct (the unique construct "array" is specialised to arrays of specific type). Why do the 2 mechanisms behave in such different ways?

We begin by discussing why generic definitions do not preserve subtypes. Let us consider the following fragment which uses the generic definitions from Fig. 10.13:

```
Stack<Integer> si = new Stack<Integer>();
Stack<Object> so = si;                      // Incorrect in  Java
```

Let us assume, contrary to the facts, that generic definitions preserve types, that is that `Stack<Integer> <: Stack<Object>`. The second line of the fragment is now legal. We have two different references (and of different types) to a single stack (integers). Let us continue our code:

```
so.push(new String("pluto"));
Integer i = si.pop();                      // danger!
```

Under these hypotheses, both lines are type correct. Since `so` is a stack of `Object`, we can store any object at all in it, for example a string. On the other hand, since `si` is a stack of integers, by randomly selecting an element, an integer is obtained. But this amounts to a clear violation of type security. Since the source of the problem is that `Stack<Integer> <: Stack<Object>`, it is just this relation that must be abandoned. The type `Stack<Object>` is therefore not the common supertype of all specific stacks `Stack<A>`. On the other hand, the pragmatics suggests that such a supertype must exist in the language, because otherwise too many programming examples would be difficult, if not impossible, to write. It is for this reason that the wildcard ("?") was introduced. `Stack<?>` *is* the supertype of every specific stack.

The intuitive idea that `A <: B` implies that `DefPara<A> <: DefPara<B>` is erroneous because it does not take into account the fact that collections can change in time. Once a `Stack<Integer>` has become a `Stack<Object>`, it is no longer possible to check statically that its modifications are consistent with its original structure.

Co- and Contravariant Functions

Let D be a set on which a pre-order, written as $\leq$ (recall the box on page 231), is defined. A function $f : D \to D$ is *covariant* when f respects the pre-order, that is $x \leq y$ implies that $f(x) \leq f(y)$. A function is *contravariant*, whenever the pre-order is reversed, that is $x \leq y$ implies $f(x) \geq f(y)$.

If the relation is a partial order, the most common terminology in mathematics is that of a monotone increasing function (for covariant) and anti-monotone, or monotone decreasing (for contravariant). In the context of types and subtypes, the covariant-contravariant terminology is always used (it is taken from category theory).

Now we come to the problem of arrays. If we substitute arrays for stacks in our fragment of a few paragraphs ago, we obtain, *mutatis mutandis*:

```
Integer[] ai = new Integer[10];
Object[] ao = ai;              // correct: Integer[] <: Object[]
ao[0] = new String("pluto");   // correct; error at runtime
```

The fragment is statically correct, because arrays in Java preserve subtypes (technically, it is said that they are a *covariant* construct), but the compiler, in order to guarantee type security, is forced to insert dynamic type checks in cases like the last line which, in the example, will cause a runtime error because we are trying to store a string in a variable of type `Integer`.

Why then add covariant arrays to the language if they impose dynamic checks which, however, remain counterintuitive in the light of the non-covariance of generics? The point is that covariant arrays allow some limited form of polymorphism. Let us consider, for example, the problem of exchanging the first 2 elements of an arbitrary array. A possible solution is:

```
public swap(Object[] vect){
   if (vect.length > 1){
            Object temp = vect[0];
            vect[0] = vect[1];
            vect[1] = temp;
   }
}
```

The method `swap` can be called on an arbitrary arrays because of its covariance.

Covariant arrays were present in Java since the earliest days of the project, well before designers considered the problem of generics. In retrospect however, in the light of the introduction of generics, covariant arrays must be considered one of the less successful aspects of the Java project.

10.4.5 Covariant and Contravariant Overriding

Let us conclude the study of the subtype relation by discussing the types permitted in the redefinition of methods. Java and C++ require that, when we have redefinition, the arguments to the redefined method must be the same as those in the one being redefined. Let us take up again our pseudo-language and let `C` be a fixed type. Given a class such as:

```
class F{
    C fie (A p) {...}
}
```

a subclass of `F` can redefine (overwrite) `fie` only with a method which takes as its argument an `A`. If the types are different, as for example in:

```
class G extending F{
    C fie (B p) {...}
}
```

we do not have redefinition, but only the definition of 2 overloaded methods.

The same does not happen to the *result* type of the method. Both C++ and Java[21] allow the type of the result of the method redefined in subclass to be a subtype of the corresponding type in the superclass. Let us assume that `D` is a subtype of `C` (`D<:C`):

```
class E extending F{
    D fie (A p) {...}
}
```

In `E` (which is a subtype of `F`: `E<:F`) the method `fie` is redefined with respect to `F`. The reader will have no difficulty convincing themselves on their own that this extension is semantically legitimate and not the cause of type errors. We can however help in this argument with some general considerations.

Supertypes and Views In a language with subtype polymorphism, a type of an object can be interpreted as a particular *view* of the object, or, in a more colourful way, as a disguise for the object. Figure 10.15 represents this idea graphically in the case of two types `G <: F` (`F` is a class of objects which contains the methods $\mathtt{c}_1$ and $\mathtt{c}_2$; the objects of `G` respond to all the methods of `F`, plus $\mathtt{c}_3$ and $\mathtt{c}_4$).

If we apply this idea (which is informal, but which completely respects subtype semantics) to the redefinition of the method, we obtain the situation shown in Fig. 10.16, where our method `fie`, which is defined in `F` with the signature `fie: A -> C`, is redefined in a subclass with signature `fie: A -> D`. The figure immediately reveals the observation that this is correct with respect to types exactly when `D<:C`. In fact, the redefined `fie` produces a value of type `D`. If `D<:C`, this value is also a value of `C` and therefore the types are respected.

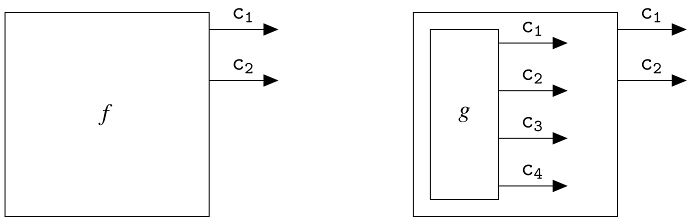
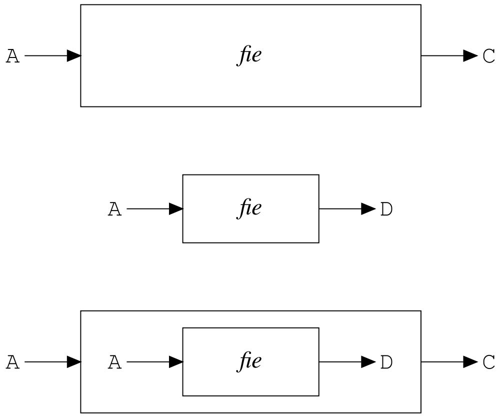


Fig. 10.15 An instance f of `F` and an instance g of `G` disguised as `F`

Fig. 10.16 Redefinition of a method: covariance of the results type



We can summarise the discussion with a general principle: with respect to the result type of a method, overriding is semantically correct (and permitted in many languages) when it is covariant. The result of the method that has been redefined in a subclass, is a subtype of the result of the original method.

This reasoning, however, can be easily applied also to argument types of methods. The scheme in Fig. 10.16 can, in fact, be generalised to that in Fig. 10.17, where method `fie` is redefined with the signature `fie: H->D`. Which relation must be valid between `H` and `A` so that the situation is semantically correct? The figure gives an immediate answer: it must be `A<:H`. With the respect to the argument type of a method, overloading is semantically correct when it is contravariant. The argument of the redefined method in the subclass, is a supertype of the argument of the original method.

From the semantic viewpoint therefore, the type of the methods `S->T` is a subtype of S'`->` T' (`S->T<:S'->T'`) whenever `S<:S'` and `T'<:T`: the type of the methods is covariant in the type of result and contravariant in the type of the arguments.

[21] Up to version 4, Java requires also that result types of a redefined method must coincide with the result type of the original method.

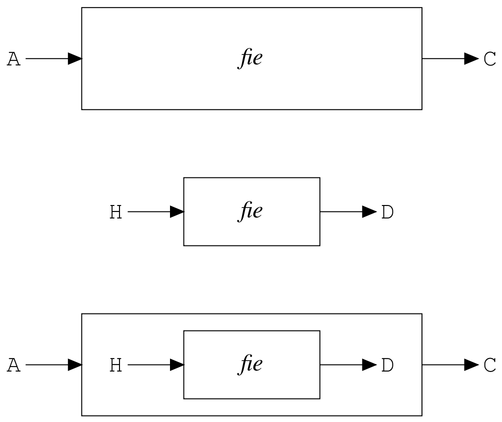


Fig. 10.17 Redefinition of a method: contravariance of argument type

Although semantically correct, contravariant overriding (of method arguments) is counterintuitive in some situations. The most important cases are those of *binary methods*, that is, methods with a parameter of the same type as the object which receives the method. The typical case is that of an equivalence or comparison method. Assuming we have a class such as the following:

```
class Point{
   ...
   boolean eq(Point p){...}
}
```

and we wish now to specialise it into the subclass:

```
class ColoredPoint extending Point{
   ...
   boolean eq(ColoredPoint p){...}
}
```

According to the contravariance rule, the `ColoredPoint` class is *not* a subclass of `Point` because the argument of `eq` would be a subtype (and not a supertype) of the type of the argument to `eq` in `Point`.

For these (and others) reasons, contravariant overriding is little used (one of the few languages which use it is Emerald), while many of the most common languages (with C++ and Java being the most common) require type identity for the argument.

There are however languages that would sacrifice static type security and only use the rule of covariant overriding for method argument types (which is semantically incorrect). Among these are some fairly well-known language, such as Eiffel and the language O_2 (one of the most common object-oriented database systems). The reason is assumed greater naturalness, especially for binary methods. Extended experimentation with O_2, amongst others, show that the semantic incorrectness of type checking never causes problems in practice (that is in situations that are not constructed *ad hoc* to violate the type system).

10.5 Chapter Summary

The chapter has been a general but deep, examination of the object-oriented paradigm, introduced as a way of obtaining abstractions in the most flexible and extensible way.

We have characterised the object-oriented paradigm when in the presence of:

- *Encapsulation* of data.
- A compatibility relation between types which we call *subtype*.
- A way to reuse code, which we refer to as *inheritance* and which can be divided into single and multiple inheritance.
- Techniques for the *dynamic dispatch* of methods.

These four concepts were viewed principally in the context of languages based on *classes* (even if we briefly looked at languages based on delegation). We then discussed the implementation of single and multiple inheritance and dynamic method selection.

The chapter was concluded with a study of some aspects relevant to the type systems in object oriented languages:

- Subtype polymorphism.
- Another form of parametric polymorphism which is possible when the language admits generics (which we studied in Java).
- The problem of overriding of co- and contravariant methods.

The object-oriented paradigm, in addition to represent a specific class of languages, is also a general development method for system software codified by semi-formal rules based on the organisation of concepts using objects and classes. Obviously the 2 aspects, the linguistic and the methodological, are not completely separate, given that the object-oriented development methodology finds its natural application in the use of object-oriented languages. However, within software engineering, methodological aspects are usually treated in an autonomous fashion without referring to any specific language. In this book, therefore, we did not go into the methodological aspects of object-oriented programming (we refer the reader to software engineering texts—see also the bibliographical notes below).

10.6 Bibliographical Notes

The first object-oriented language was Simula 67 [2, 9] which we cited in the previous chapter. It is Smalltalk (which explicitly depends on Simula), however, which had the greatest influence on succeeding languages and introduced the anthropomorphic metaphor of messages sent to objects [5].

On C++ the canonical reference is [13]. The Java language definition can be found in [6] and the most revealing text by its principal author [1]. An introduction to the Java Virtual Machine (and therefore to how the various mechanisms are implemented in Java) can be found in [8]; for the official specification of the JVM see [7].

Our description of the implementation of multiple inheritance is based on that of C++ [12]. The introduction of generics into Java is the result of much research work into subtype polymorphism. An introduction to generics as they appear in Java is to be found in [3], which we have closely followed in our presentation.

For the Self project, see the original article [14]; the retrospective [11] is a good source of information about the criteria for an innovative programming language project. For the diatribe on covariant and contravariant overriding, see [4].

Finally as far as object-oriented development methodologies are concerned, consult any general text on software engineering, for example, the classic [10].

10.7 Exercises

1. Consider the classes in Fig. 10.8 and the final definition:

```
class E{
   int v;
   void n(){...}
   }
class D extending E,C{
   int w;
   int g(){return x + y + v;} // redefinition with respect to C
   void m(){...}
}
```

   Draw the representation for an instance object of D, as well as the structure of the vtable for such a class, indicating for each method the appropriate value required to correct the value of this.

2. Given the definitions in our pseudo-language:

```
abstract class A {
   int val = 1;
   int foo (int x);
}
abstract class B extending A {
    int val = 2;
}
class C extending B {
    int n = 0;
    int foo (int x){ return x+val+n;}
}
class D extending C {
    int n;
    D(int v){n=v;}
    int foo (int x){return x+val+n;}
}
```

Consider now the following program fragment

```
int u, v, w, z;
A a;
B b;
C c;
D d = new D(3);
a = d;
b = d;
c = d;
u = a.foo(1);
v = b.foo(1);
w = c.foo(1);
z = d.foo(1);
```

Give the values of `u`, `v`, `w` and `z` at the end of execution.

3. Given the following Java definitions:

```
interface A {
    int val=1;
    int foo (int x);
}
interface B {
    int z=1;
    int fie (int y);
}
class C implements A, B {
    int val = 2;
    int z =2;
    int n = 0;
    public  int foo (int x){ return x+val+n;}
    public int fie (int y){ return z+val+n;}
}
class D extends C {
    int val=3;
    int z=3;
    int n=3;
    public int foo (int x){return x+val+n;}
    public int fie (int y){ return z+val+n;}
}
```

Consider now the following program fragment:

```
int u, v, w, z;
A a;
B b;
D d = new D();
a = d;
b = d;
System.out.println(u = a.foo(1));
System.out.println(v = b.fie(1));
System.out.println(w = d.foo(1));
System.out.println(z = d.fie(1));
```

Give the values of u, v, w and z at the end of execution.

4. Is the following fragment of Java correct? In the positive case, the method fie is redefined (overridden)? What does it print?

```
class A {
    int x = 4;
    int fie (A p) {return p.x;}
}

class B extends A{
    int y = 6;
    int fie (B p) {return p.x+p.y;}
}


public class binmeth {
    public static void main (String [] args) {
        B b = new B();
        A a = new A();
        int zz = a.fie(a)+ b.fie(a) ;
        System.out.print(zz);
    }
}
```

References

1. K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*, 4th edition. Addison-Wesley Longman, Boston, 2005.
2. G. Birtwistle, O. Dahl, B. Myhrtag, and K. Nygaard. *Simula Begin*. Auerbach Press, Philadelphia, 1973. ISBN: 0-262-12112-3.
3. G. Bracha. Generics in the Java programming language. Technical report, Sun Microsystems, 2004. Disposable on-line at java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
4. G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. Prog. Lang. Syst.*, 17(3):431–447, 1995. citeseer.ist.psu.edu/castagna94covariance.html.
5. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman, Boston, 1983. ISBN: 0-201-11371-6.
6. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3/E*. Addison Wesley, Reading, 2005. Disposable on-line a http://java.sun.com/docs/books/jls/index.html.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Sun and Addison-Wesley, Reading, 1999.
8. J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, Sebastopol, 1997.
9. K. Nygaard and O.-J. Dahl. The development of the SIMULA languages. In *HOPL-1: The First ACM SIGPLAN Conference on History of Programming Languages*, pages 245–272. ACM Press, New York, 1978. doi:10.1145/800025.808391.
10. R. Pressman. *Software Engineering: A Practitioner's Approach*, 7th edition. McGraw Hill, New York, 2009.
11. R. B. Smith and D. Ungar. Programming as an experience: The inspiration for Self. In *ECOOP '95: Proc. of the 9th European Conf. on Object-Oriented Programming*, pages 303–330. Springer, Berlin, 1995. ISBN: 3-540-60160-0.

12. B. Stroustrup. Multiple inheritance for C++. In *Proc. of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987. citeseer.ist.psu.edu/stroustrup99multiple.html.
13. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, Boston, 1997.
14. D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conf. Proc. on Object-Oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, New York, 1987. ISBN: 0-89791-247-0. doi:10.1145/38765.38828.