Object-Oriented Languages: method polymorphism

Late binding in Smalltalk

All object-oriented languages exhibit a property called *method polymorphism*, or sometimes *late binding.* In a pure object-oriented sense, it refers to whether or not a particular object can accept a message. It is best seen in Smalltalk, a language in which everything is an object and any message may be sent to any object. Objects are instances of classes, and each class defines a type in a single-inheritance hierarchy. Every class inherits ultimately from a class called Object that sits at the top of the hierarchy. As an example, consider the following portion of this hierarchy:

```
class A supertype Object
…
class B supertype A
…
class C supertype A
```

Methods in Smalltalk can either be class methods or instance methods. Class methods, which are unique to the class, and can only be invoked by sending a message to the object which is the class itself. A typical example is the method new which creates an object of the class. Instance methods can be invoked by sending a message to an object already created from the class. Imagine such a method, m1, in class A. How can A be invoked? If an object of type A is available, it can be sent the message m1 which invokes the method of the class. E.g.

```
| anObject |
anObject <- A new.
anObject m1
```

A variable is declared between |…|. It has no type until it is assigned, for instance, by the object creation in the next line. The third line sends the message m1 to the object thus created. Since anObject is of type A, and there is a method, m1, in A, then m1 is invoked. Now imagine creating an object of type B, and sending it the message m1:

```
| anObject |
anObject <- A new.
 anObject m1.
anObject <- B new.
anObject m1
```

m1 acts as a method selector, but there is no method called m1 in class B. Smalltalk then searches the superclass of B (and, if necessary, the superclasses of it, and so on) an matches the message to the method in A, and then invokes it. Now if we override the method m1 in A with a method with the same name in C, we can invoke the method in C:

```
| anObject |
anObject <- A new.
anObject m1.
anObject <- B new.
anObject m1.
anObject <- C new.
anObject m1
```

The first two times the message m1 is sent to anObject, the method m1 in A is invoked. are different in the sense that anObject is bound to a different object of a different type. The third time m1 is sent to anObject, the method in C is selected. The binding of the message selector m1 and the methods m1 in A and in C can only be done at run-time, when the message is actually sent. This is why it is late binding – the message and methods are only determined at run-time, and can be a different binding depending on the type of the receiver. Polymorphism in this sense means "many receiver types".

Virtual functions in C++

C++ can also exhibit late binding of methods, but needs a special mechanism to do it. A typical situation is when two classes inherit from a third, and both override a function in the base class:

```
class A {
  …
public:
  void f();
};


class B : public A {
  …
public:
  void f();
};


class C : public A{
  …
public:
  void f();
};
```

If ordinary method invocation for objects is used, strong typing and object slicing ensures which version of f is called:

```
A a1;
B b1;
C c1;
a1.f();  // calls f in A
b1.f();  // calls f in B
c1.f(); // calls f in C
a1 = b1;
a1.f();  // still calls f in A
a1 = c1;
 a1.f(); // still calls f in A
```

Even if pointers are used, the same result is obtained:

```
A *pa1 = new A();
B *pb1 = new B();
C *pc1 = new C();
pa1 = pb1;
pa1->f();        // calls f in A
```

However, if we declare the overridden function in A *virtual* the situation changes. Now the assignment of the pointer pb1 to pa1 leaves pa1 pointing to an object of type B, and the call

```
pa1->f();
```

now calls f in B. Similarly for C:

```
A *pa1 = new A();
B *pb1 = new B();
C *pc1 = new C();
pa1 = pb1;
pa1->f();        // calls f in B
pa1 = pc1;
pa1->f();        // calls f in C
```

The mechanism uses RTTI to determine the type of object pointed to. The appropriate function is called depending on the type of the object pointed to, i.e. just as in the Smalltalk example, the binding of message to methods (function call to function definition) is only determined at the point of call, at run-time. (This also works with references, but not with value objects, such as the first example above.) The class declarations now look like:

```
class A {
  …
public:
  virtual void f();
};


class B : public A {
  …
public:
  void f();
};


class C : public A{
  …
public:
  void f();
};
```

If it desired that no object be created of type A, only of its subtypes, the virtual function can be declared as *abstract* with the declaration:

```
virtual void f() = 0;
```

where the = 0 notation indicates there is no body for f. Since f is abstract, the whole class is abstract. (Java actually uses the keyword abstract for this purpose.)

Late binding in Java

In Java, all methods are late-bound, i.e. the binding of call to method is done at run-time. In this sense, all methods are virtual, although Java doesn't use this terminology. Say we desire to store a number of geometric shapes in an array for drawing on a graphics window. We could define a circle, a square and a triangle as follows:

```
class Shape {
  private int posx, posy;
  public abstract void draw();
}
```

```
class Circle extends Shape {
   private int radius;
   public void draw() { // code to draw a circle … }
}
class Square extends Shape {
    private int length;
    public void draw() { // code to draw a square… }
}
class Triangle extends Shape {
    private int x2, y2;
    private int x3, y3;
    public void draw() { // code to draw a triangle… }
 }
```

We can then store these shapes in a single array of type Shape:

```
Shape[] sharr = new Shape[25];
Circle c1 = new Circle();
Square s1 = new Square();
Triangle t1 = new Triangle();
sharr[0] = c1;
sharr[1] = s1;
sharr[2] = t1;
```

All objects in Java are bound to variables as references, so no object slicing is done, as it might be in C++. The subtype principle ensures that these assignments are valid. We can then draw all the shapes in the array in one loop:

```
for (int i = 0; i < 25; i++)
     sharr[i].draw();
```

Since draw is late-bound, the appropriate draw function for the particular shape referred to by each array element is called. Note that the draw funciton in Shape is declared as abstract because it is not a real function – it is just there to be overridden in the subclasses.