11

History of Objects: Simula and Smalltalk

Objects were invented in the design of Simula and refined in the evolution of Smalltalk. In this chapter, we look at the origin of object-oriented programming in Simula, based on the concept of a procedure that returns a pointer to its activation record, and the development of a purely object-oriented paradigm in the Smalltalk project and programming language. Twenty years after its development, Smalltalk provides an important contrast with C++ and Java both in simplicity of concept and in the way that its implementation provides maximal programming flexibility.

11.1 ORIGIN OF OBJECTS IN SIMULA

As the name suggests, the *Simula* programming language was originally designed for the purpose of simulation. The language was designed by O.-J. Dahl and K. Nygaard at the Norwegian Computing Center, Oslo, in the 1960s. Although the designers began with a specific interest in simulation, they eventually produced a general-purpose programming language with widespread impact on the field of computing.

Simula has been extremely influential as the first language with classes, objects, dynamic lookup, subtyping, and inheritance. It was an inspiration to the Xerox Palo Alto Research Center (PARC) group that developed Smalltalk and to Bjarne Stroustrop in his development of C++. Although Simula 67 had important object-oriented concepts, much of the popular mystique surrounding objects and object-oriented design developed later as a result of other efforts, most notably the Smalltalk work of Alan Kay and his collaborators at Xerox PARC.

11.1.1 Object and Simulation

You may wonder what objects have to do with simulation. A partial answer may be seen in the outline of an event-based simulation program. An event-based simulation is one in which the operation of a system is represented as a sequence of events. Here is pseudocode representing a generic event-based simulation program:

KRISTEN NYGAARD

A driving force behind Simula was Kristen Nygaard, an operations research specialist and a political activist. He wanted a general modeling language that could be used to describe complex dynamic social and industrial systems in a simple way. Nygaard's interest in modeling motivated the development of innovative computing techniques, including objects, classes, inheritance, and quasi-parallel program execution allowing every object to have an optional action thread.

Nygaard was concerned by the social consequences of computing and expressed some of his reservations as follows:

> I could see that SIMULA was being used to organize work for people and I could see that it would contribute to major changes in this area: More routine work, less demand for knowledge and a skilled labor force, less flexibility at the work place, more pressure. . . . I gradually came to face a moral dilemma. . . . I realized that the technology I had helped to develop had serious consequences for other people. . . The question was what to do about it? I had no desire to un-invent SIMULA .. because I was convinced that in the society I wanted to help build, computers would come to play an immensely important role. [Translation by J.R. Holmevik]

In addition to the ACM Turing Award and the IEEE John von Neumann Medal, Kristen Nygaard, received the 1990 Norbert Wiener Award for Professional and Social Responsibility from Computer Professionals for Social Responsibility (CPSR), "For his pioneering work in Norway to develop 'participatory design,' which seeks the direct involvement of workers in the development of the computer-based tools they use." Historical information about Nygaard and the development of Simula can be found in an article by J.R. Holmevik, (*Annals of the History of Computing* Vol. 16, Number 4, 1994; pp. 25–37). More recent information may be found on Nygaard's home page.

```
Q := make_queue(first_event);
repeat
    remove next event e from Q
    simulate event e
    place all events generated by e on Q
until Q is empty
```

This form of simulation requires a data structure (some form of queue or priority queue) that may contain a variety of kinds of events. In a typed language, the most natural way to obtain such a structure is through subtyping. In addition, the operation simulate e must be written in some generic way or involve case statements that branch according to the kind of event that e actually represents. Objects help because dynamic lookup for simulate e can determine the correct code automatically. Inheritance arises when we consider ways of implementing related kinds of events.

As this quick example illustrates, event-based simulations can be programmed in a general-purpose object-oriented language. The designers of Simula discovered this when they tried to make a special-purpose language, one tailored to simulation only. Apparently, when someone once asserted that Simula was not a general-purpose language, Nygaard's response was that, because Simula had all of the features of Fortran and Algol, and more, what would he need to remove before Simula could be called general purpose?

11.1.2 Main Concepts in Simula

Simula was designed as an extension and modification of Algol 60. Here are short lists of the main features that were added to and removed from Algol 60:

- *Added to Algol 60:*
  - class concepts and reference variables (pointers to objects),
  - pass-by-reference,
  - char, text, and input–output features,
  - coroutines, a mechanism for writing concurrent programs.
- *Removed from Algol 60:*
  - changed default parameter passing mechanism from pass-by-name to a combination of pass-by-value and pass-by-result,
  - some initialization requirements on variables,
  - own variables (which are analogous to C static variables),
  - the Algol 60 string type (in favor of a text type).

In addition to objects, concurrency was an important development in Simula. This arose from an interest in simulations that had several independent parts, each defining a sequence of events. Before representing events by objects, the designers experimented with representing independent sequences of events by independent processes, with the main simulation loop alternating between these processes to allow the simulation to progress. Here is a short quote from Nygaard on the incorporation of processes into the language:

> In the spring of 1963, we were almost suffocated by the single-stack structure of Algol. Then Ole-Johan developed a new storage management scheme [the multistack scheme] in the summer and autumn of 1963. The preprocessor idea was dropped, and we got a new freedom of choice. In Feb, 1964 the process concept was created, which [led to] Simula 67's class and object concept.

This quote appears in "The Development of the Simula Languages" by K. Nygaard and O.-J. Dahl (published in *History of Programming Languages*, R. L. Wexelblat, ed., Academic, New York, 1981.)

11.2 OBJECTS IN SIMULA

Objects arise from a very simple idea: After a procedure call is executed, it is possible to leave the procedure activation record on the run-time stack and return a pointer to it. A procedure of this modified form is called a *class* in Simula and an activation record left on the stack is an *object*:

> *Class*: A procedure returning a pointer to its activation record.
>
> *Object*: An activation record produced by call to a class, called an instance of the class.

Because a Simula activation record contains pointers to the functions declared in the block and their local variables, a Simula object *is* a closure! Pointers, missing from Algol 60, were needed in Simula because a class returns a pointer to an activation record. In Simula terminology, a pointer is called a ref.

Although the concept of object begins with the idea of leaving activation records on the stack, returning a pointer to an activation record means that the activation record cannot be deallocated until the activation record (object) is no longer used by the program. Therefore, Simula implementations place objects on the heap, not the run-time stack used for procedure calls. Simula objects are deallocated by the garbage collector, which deallocates objects only when they are no longer reachable from the program that created them.

11.2.1 Basic Object-Oriented Features in Simula

Simula contains most of the main object-oriented features that we use today. In addition to classes and objects, mentioned in the preceding subsection, Simula has the following features:

- *Dynamic lookup*, as operations on an object are selected from the activation record of that object,
- *Abstraction* in later versions of Simula, although not in Simula 67,

- *Subtyping,* arising from the way types were associated with classes,
- *Inheritance*, in the form of class prefixing, including the ability to redefine parts of a class in a subclass.

Although Simula 67 did not distinguish between public and private members of classes, a later version of the language allowed attributes to be made "protected," which means that they are accessible for subclasses (but not other classes), or "hidden," in which case they are not accessible to subclasses either.

In addition to the features just listed, Simula contains a few object-related features that are not found in most object-oriented languages:

- Inner, which indicates that the method of a subclass should be called in combination with execution of superclass code that contains the inner keyword,
- Inspect and qua, which provide the ability to test the type of an object at run time and to execute appropriate code accordingly. Inspect is a class (type) test, and qua is a form of type cast that is checked for correctness at run time.

All of these features are discussed in the following subsections. Some features that are found in other languages but not in early Simula are multiple inheritance, class variables (as in Smalltalk), and the self / super mechanism found in Smalltalk.

11.2.2 An Example: Points, Lines, Circles

Here is a short program example, taken from a classic book describing early Simula (Birtwistle, Dahl, Myhrhaug, and Nygaard, *Simula Begin*, Auerbach, 1973). This example illustrates some characteristics of Simula and shows how closely early object-oriented programming in Simula resembles object-oriented programming today.

Problem

Given three distinct points p, q, and r in the plane, find the center and the radius of the circle passing through p, q, and r. The situation is drawn in Figure 11.1.

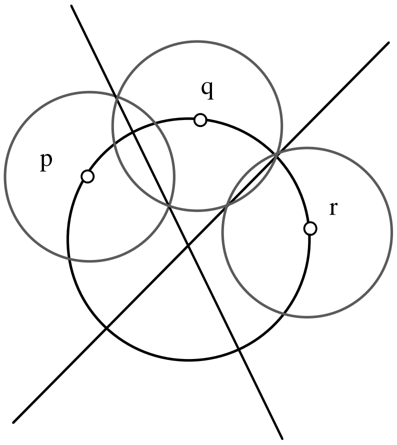


Figure 11.1. Points, circles, and their lines of intersection.

Algorithm

An algorithm for solving this problem is suggested by 11.1. The algorithm has the following steps:

1. Draw intersecting circles Cp and Cq, centered at points p and q, respectively.
2. Draw intersecting circles Cq' and Cr, centered at q and r, respectively. For simplicity, we assume that Cq and Cq' are the same circle.
3. Draw line $L1$ through the points of intersection of Cp and Cq.
4. Draw line $L2$ through the points of intersection of Cq and Cr.
5. The intersection of $L1$ and $L2$ is the center of the desired circle.

This method will fail if the three points are colinear, as there is no circle passing through three colinear points.

Methodology

We can code this algorithm by representing points, lines, and circles as objects and equipping each class of objects with the necessary operations. Here is a sketch of the classes and operations we need:

Point

Representation

x, y coordinates

Operations

```
equality(anotherPoint) : boolean
distance(anotherPoint) : real
```

Line

Representation

All lines have the form $ax + by + c = 0$. We may store a, b, and c, normalized so that all three numbers are not too large. When we call the Line class to build a Line object, we will normalize the values of a, b, and c.

Operations

```
parallelto(anotherLine) : boolean
meets(anotherLine) : ref(Point)
```

The parallelto operation is used to see if two lines will intersect. The meets operation is used to find the intersection of two lines that are not parallel.

Circle

Representation

```
center : ref(Point)
```

Operations

```
intersects(anotherCircle) : ref(Line)
```

It should be clear how to solve the problem with these classes of objects. Given two points, we can pass one to the distance function of the other and obtain the distance between the two points. This lets us calculate the radius for intersecting circles centered at the two points. Given two circles, the intersects function finds the line passing through the two points of intersection, and so on.

11.2.3 Sample Code and Representation of Objects

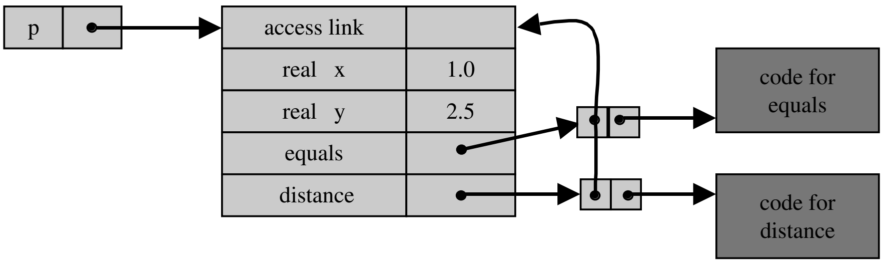
In Simula, the Point class can be written and used to create a new point, as follows:

```
class Point(x,y); real x,y;
begin
    boolean procedure equals(p); ref(Point) p;
        if p =/= none then
          equals := abs(x - p.x) + abs(y - p.y) < 0.00001;
    real procedure distance(p); ref(Point) p;
        if p == none then error else
            distance := sqrt(( x - p.x )**2 + (y - p.y) ** 2);
end ***Point***
p :- new Point(1.0, 2.5);
```

Because all objects are manipulated by refs (the Simula term for pointers), the type of an object variable has the form ref(Class), where Class is the class of the object. Because the equals procedure requires another point as an argument, the formal parameter p is declared to have type ref(Point). Simula references are initialized to a special value none and :- is used for pointer assignment. The test p =/= none (read "p not-equal none") tests whether the pointer p refers to an object.

In the body of the Point class, we access parts of the object, which are locally declared variables and procedures, simply by naming them. Parts of other objects, such as the object passed as a parameter to distance, are accessed with a dot notation, as in p.x and p.y.

When the statement at the bottom of the code example above is executed, it produces a run-time structure that we may draw as follows:



This Point object contains pointers to the closures for Point class procedures and the environment pointer of each closure points to the activation record that is the object. (This means the codes for equals and distance can be shared among all points, but the closure pairs must be different for each object.) When one of these procedures is called, as in p.equals(q), an activation record for the call is created and its access link is set according to the closure for the procedure. This way, when the code for equals refers to x, the x that will be used is the x stored inside the object p.

There are several ways that this representation of objects may be optimized; you should assume only that this representation captures the behavior of Simula objects, not that every Simula compiler actually uses precisely this storage layout. Some useful optimizations are shown in the chapters on Smalltalk and C++, in which each object stores only a pointer to a table storing pointers to the methods.

To give a little more sample code, the line class may be written in Simula as follows:

```
class Line(a,b,c); real a,b,c;
begin
    boolean procedure parallelto(l); ref(Line) l;
        if l =/= none then
            parallelto := abs(a*l.b - b* l.a) < 0.00001;
    ref(Point) procedure meets(l); ref(Line) l;
    begin real t;
        if l =/= none and ~parallelto(l) then
            begin
            t := 1/(l.a * b - l.b * a);
            meets :- new Point(. . . , . . . );
            end;
    end; ***meets***
    real d;
    d := sqrt(a**2 + b**2);
    if d = 0.0 then error else
        begin
        d := 1/d;
        a := a * d; b := b * d; c := c * d;
        end;
end *** Line***
```

The procedure meets invokes another procedure of the same object, parallelto. The code following the procedure meets is initialization code; it is executed whenever a Line object is instantiated. You might want to think about how this initialization code, which is written as if a class were an ordinary procedure, corresponds to constructor code in object-oriented languages you are familiar with.

11.3 SUBCLASSES AND SUBTYPES IN SIMULA

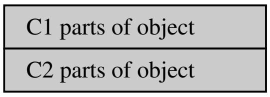
11.3.1 Subclasses and Inheritance

The classes in a Simula program are arranged in a hierarchy. One class is a *superclass* of a second if the second is defined by inheritance from the first. We also say that a class is a *subclass* of its superclass.

Simula syntax for a class C1 with subclasses C2 and C3 is

```
class C1
     <declarations_1> ;
C1 class C2
     <declarations_2>
C1 class C3
     <declarations_3>
```

When we create a C2 object, for example, we do this by first creating a C1 object (activation record) and then appending a C2 object (activation record). In a picture, a C2 object looks like this:



The structure is essentially the same as if procedure called C2 was declared and called within C1: The access links of the second activation record refer to the first.

Here is an example code that uses Simula class prefixing to define a colored point subclass of the Point class defined in Subsection 11.2.3:

```
Point class ColorPt(c); color c;        ! List new parameter only
begin
    boolean procedure equals(q); ref(ColorPt) q;
         ...;
end ***ColorPt***
ref(Point) p;                           ! Class reference variables
ref(ColorPt) cp;
p :- new Point(2.7, 4.2);
cp :- new ColorPt(3.6, 4.9, red);       ! Include parent class parameters
```

The ColorPt class adds a color field c to points. Because Simula 67 did not hide fields, the c field of a ColorPt object can be accessed and changed directly by use of the dot notation. For example, cp.c := green changes to color of the point named by cp. The ColorPt class redefines equals so that cp.equals can compare color as well as x and y.

The statement p :- New Point(2.7, 4.2) causes an activation record to be created with locations for parameters x and y. These are set to 2.7 and 4.2, respectively and

https://doi.org/10.1017/CBO9780511804175.012 Published online by Cambridge University Press

then the body of the Point class is executed. Because the body of the Point} class is empty, nothing happens at this stage for points. After the body is executed, a pointer to the activation record is returned. The activation record contains pointers to function values (closures) equals and distance.

A prefixed class object is created by a similar sequence of steps that involves calls to the parent class before the child class. More specifically, an activation record is created for the parent class and an activation record is created for the child class. Then parameter values are copied to the activation records, parent class first, and the class bodies are executed, parent class first. Some additional details are considered in the exercises.

11.3.2 Object Types and Subtypes

All instances of a class are given the same type. The name of this type is the same as the name of the class. For example, if p and q are variables referring to objects created by the Point class, then they will have type ref(Point). As mentioned in Section 11.2, ref arises because all Simula objects are manipulated through pointers.

The class names (types of objects) are arranged in a *subtype* hierarchy corresponding exactly to the subclass hierarchy. In other words, the only subtype relations that are recognized in Simula 67 are exactly those that arise from inheritance: If class A is derived from class B, then the Simula type checker treats type A as a subtype of type B.

There are some interesting subtleties regarding assignment and subtyping that also apply to other languages. For example, look carefully at the following legal Simula code, in which Simula syntax :- is used for reference (pointer) assignment:

```
class A;
A class B;   /* B is a subclass of A */
ref (A) a;
ref (B) b;
a :- b;      /* legal since B is a subclass of A */
...
b :- a;      /* also legal but checked at run-time to make sure a points to a B
                object*/
```

Both assignments are accepted at compile time and satisfy Simula's notion of type compatibility. However, a run-time check is needed for the second assignment to guarantee type safety. The same run-time checking also appears in Beta, a cultural descendant of Simula, and in Java array array assignment. The reason for the run-time test is that the object reference a might point to an object of class B or it might point to an object of class A that is not an object of class B. In the first case, the assignment is OK; in the second case, it is not. If an assignment causes an object reference b with static type ref(B) to point to an object that is *not* from class B or some subclass of B, this is a type error.

It is possible to rewrite the assignment to b in the code we just looked at by using Simula inspect:

```
inspect a
    when B do b :- a
    otherwise .../* some appropriate action */
```

If a does not refer to a B object, then the otherwise clause will catch the error and let the programmer take appropriate action. However, in the case in which the programmer knows that a refers to a B object, the syntax with an implicit run-time test is obviously simpler.

There was an error in the original Simula type checker surrounding the relationship between subtyping and inheritance. This is illustrated in the following code, extracted from a running DEC-20 Simula program written by Alan Borning. (The DEC-20 version of Simula uses := instead of :- for pointer assignment.)

```
class A;
A class B; /* B is a subclass of A */
ref (A) a;
ref (B) b;
proc assignA (ref (A) x)
    begin
    x := a
    end;
assignA(b);
```

In the terminology of Chapter 10, Simula subclassing produces the subtype relation B <: A. Simula also uses the semantically incorrect principle that, if B <: A, then ref (B) <: ref (A). Therefore, this code will statically type check, but will cause a type error at run time. The same type error occurs in the original implementation of Eiffel, a language designed many years later.

11.4 DEVELOPMENT OF SMALLTALK

The Smalltalk project at Xerox PARC in the 1970s streamlined the object metaphor and drew attention to object-oriented programming. Although the central object-oriented concepts in Smalltalk resemble their Simula ancestors, Smalltalk was not an incremental modification of any previously existing language. It was a completely new language, with new terminology and an original syntax.

This is a list of some of the main advances of Smalltalk over Simula:

- The object metaphor was extended and refined
    - In Smalltalk, everything is an object, even a class.
    - All operations are therefore messages to objects.

  - Objects and classes were shown to be useful organizing concepts for building an entire programming environment and system.
- Abstraction was added, using the distinction between
  - private *instance variables* (data associated with an object) and
  - public *methods* (code for performing operations)

These short lists, however, do not do justice to the Smalltalk project. Smalltalk was conceived as part of an innovative and forward-looking effort that designed an entire computing system around a new programming language.

Smalltalk is more flexible and more powerful than Simula or other languages that preceded it, and many Smalltalk fans consider it a better language than most that followed. Smalltalk is untyped, allowing greater flexibility in the modification of programs and greater flexibility in the use of general-purpose algorithms on a variety of data objects. In addition, the importance of treating everything as an object cannot be underestimated. This is the source of much of the flexibility of Smalltalk. Smalltalk has many interesting and powerful facilities, including a mechanism that allows objects to detect a message they do not understand and to respond to the message.

Dynabook

Part of the inspiration for the Smalltallk language and system came from Alan Kay's concept of a *Dynabook*. The Dynabook was to be a small portable computer that was capable of storing useful and interesting personal information and running programs for a variety of applications. It was a laptop computer with business and entertainment software, and sufficient storage space to store telephone numbers, business data, or a copy of a popular novel. It is almost impossible to explain, in the present day, how utterly improbable this sounded in the 1970s. At the end of the 1970s, a minicomputer, typically shared by 15 or 20 people, required the same floor space as a small laundromat and had roughly the same appearance – rows of machines, some with visible spinning drums. Very-large-scale integrated (VLSI) circuits were not a reality until the late 1970s, so there was really no way the Dynabook could conceivably have been manufactured at the time it was imagined. To give another historical data point, the Xerox Alto of the 1970s was an advanced personal computer that cost $32,000 at the time, more than the annual salary of a computer professional. It had removable disks, which each person would use to store personal working data. Each disk was 2ft in diameter, a couple inches thick, and weighed 5–10 lb! Hardly the sort of thing you could carry around in your pocket or even imagine inserting into a portable computer. Not only was Kay's prediction of hardware miniaturization revolutionary, but the idea that people would use a computer for a wide range of personal activities was shocking.

Smalltalk was intended to be the operating system interface as well as the standard programming language for Dynabook; all of a user's interaction with the Dynabook would be through Smalltalk. With this in mind, the syntax of Smalltalk was designed to be used with a special-purpose Smalltalk editor. Because the Xerox group believed that hardware would eventually catch up with the need for computing power, the Smalltalk implementation emphasized flexibility and ease of use over efficiency.

11.5 SMALLTALK LANGUAGE FEATURES

11.5.1 Terminology

The Smalltalk project developed precise terminology for the main Smalltalk concepts and constructs. Here is a list of the main object-related terms and brief descriptions:

- *Object*: A combination of private data and functions. Each object is an *instance* of some class.
- *Class*: A template defining the implementation of a set of objects.
- *Subclass*: Class defined by inheriting from its superclass.
- *Selector*: The name of a message (analogous to a function name).
- *Message:* A selector together with actual parameter values (analogous to a function call).
- *Method*: The code in a class for responding to a message.
- *Instance Variable*: Data stored in an individual object (instance of a class).

The way these terms were used in the Smalltalk language and documentation is close to current informal use among many object-oriented professionals, but not entirely universal. For example, some people use "method" to refer to both the name of a method (which Smalltalk would call a selector) and the code used to respond to a message.

11.5.2 Classes and Objects

The Smalltalk environment included a special editor designed for the Smalltalk language. With this editor, classes were defined by filling in a table of a certain form, illustrated with the definition of a Point class in Figure 11.2.

In Smalltalk terminology, a class variable is a variable that is defined in the class and shared among all objects of the class. In this example, the value of pi may be used in geometric calculations involving points. Therefore, the class includes a variable called pi. An instance variable is a variable that is repeated for each object. Each point object, for example, will have two instance variables, x and y, as each point may

class name	Point
super class	Object
class var	pi
instance var	x y
class messages and methods	
⟨…names and code for methods…⟩	
instance messages and methods	
⟨…names and code for methods…⟩	

Figure 11.2. Definition of Point class.

be located at a different position in the x–y plane. Methods are functions that may be used to manipulate objects, and messages contain the names and parameters of methods.

Here are some example class messages and methods for point objects (subsequently explained):

```
newX:xvalue Y:yvalue ||
                    ^ self new x: xvalue y: yvalue
newOrigin ||
                    ^ self new x: 0 y: 0
                    initialize ||
                         pi <- 3.14159
```

and some example instance messages and methods:

```
x: xcoord y: ycoord ||
         x <- xcoord
         y <- ycoord
moveDx: dx Dy: dy ||
         x <- dx + x
         y <- dy + y
x || ^x
y || ^y
draw ||< < . . . code to draw point . . . > >
```

These are written with Smalltalk syntax, which includes the following:

```
||   —place to list local declarations for method body
^ - unary prefix operator giving return value of method
<- - assignment
```

Smalltalk message names include positions for message arguments, indicated by colons.

The first class method, newX:Y:, is a two-argument method that creates a new point. The two arguments are the x coordinate and the y coordinate of the new point. A new point at coordinates (3, 2) is created when the message

```
newX: 3 Y:2
```

is sent to the Point class, with x coordinate 3 following the first colon and y coordinate 2 following the second coordinate. When we look at inheritance and other situations in which the name of a method is important, it will be useful to remember that the official Smalltalk name of a message contains all the words and colons, as in "newX:Y:."

The newX:Y: method is a class method of the Point class, which means that if you want to create a new point with specific *x* and *y* coordinates, you may send a newX:Y: message to the Point class (which is itself an object). The newX:Y: method has no local variables, as any local declarations would appear between the vertical bars following the name. The effect of the method is indicated by the expression

```
^ self new x: xvalue y: yvalue
```

As previously mentioned, the first symbol, ^, means "return." The value that is returned is the object produced when the message new x: xvalue y: yvalue is sent to the Point class. (Self refers to the object that contains the method; this is discussed in Subsection 11.6.3.) This message is handled by a predefined class method, provided by the system, that allows any class to create instances of it.

To continue the example, suppose we execute the following code:

```
p <- Point newOrigin
p moveDx: 3 Dy: 4
```

This will create a point at the origin, (0, 0), and then move the point to position (3, 4). Thereafter, the value of expression

```
p x
```

that sends the message x to the object p will be the object 3 and the value of

```
p y
```

that sends the message y to the object p will be the object 4.

Run-Time Representation of Objects

The run-time representation of a Smalltalk object has a number of parts. Each point object, for example, may have a different *x* coordinate and *y* coordinate. Therefore, each object must contain space for an *x* coordinate and a *y* coordinate. The methods, on the other hand, are shared among all point objects. To save space and take advantage of the similarity between objects of the same class, each object contains a pointer to a data structure containing information associated with its class. The basic run-time structure for points is illustrated in Figure 11.3. This illustration is intended to show the basic functionality of Smalltalk classes and objects; actual implementations might store things differently or perform various optimizations to improve performance.

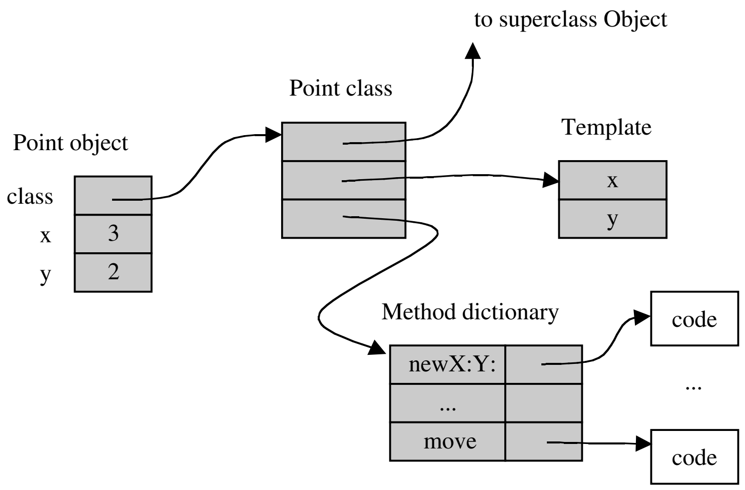


Figure 11.3. Run-time representation of `Point` object and class.

When a message, such as `move`, is sent to a point object, the code for `move` is located by following a pointer to the class of the object, following another pointer to the method dictionary, and then looking up the code in this data structure. If the code for a method is not found in the method dictionary for the class, the method dictionaries of superclasses are searched, as described in the next subsection on inheritance.

11.5.3 Inheritance

We look at Smalltalk inheritance by considering a subclass of `Point` that implements colored points. The main difference between `ColoredPoint` and `Point` is that each colored point has an associated color. If we want to provide methods for the same geometric calculations, then we can make colored points a subclass of points. In Smalltalk, the `ColoredPoint` class can be defined as shown in Figure 11.4.

class name	ColoredPoint
super class	Point
class var	
instance var	color
class messages and methods	
newX:xv Y:yv C:cv	⟨ … code … ⟩
instance messages and methods	
color	|| ^color
draw	⟨ … code … ⟩

Figure 11.4. Definition of `ColoredPoint` class.

The second line of Figure 11.4 shows that Point is the superclass of ColoredPoint. As a result, all instance variables and methods of Point become instance variables and methods of ColoredPoint by default. In Smalltalk terminology, ColoredPoint *inherits* instance variables *x* and *y*, methods x:y:, moveDx:Dy:, and so on. In addition, ColoredPoint adds an instance variable color and a method color to return the color of a ColoredPoint.

Although ColoredPoint inherits the draw method from Point, the draw method for ColoredPoint must be implemented differently. When we draw a ColoredPoint, the point should be drawn in color. Therefore, Figure 7.1 shows a draw method, overriding the one inherited from Point. (Redefinition of a method is sometimes called *overriding* the method.) An option that is available in Smalltalk, but not in most other object-oriented languages, is to specify that a superclass method should be undefined on a subclass.

Because colored points should have a color specified when they are created, the ColoredPoint class will have a class method that sets the color of a colored point:

```
newX: xvalue Y:yvalue C:cvalue ||
        ^ self new x: xvalue y:yvalue color:cvalue
```

As an example use of ColoredPoint, suppose we execute the following code:

```
cp <- ColoredPoint newX:1 Y:2 C:red
cp moveDx: 3 Dy: 4
```

Then the value of the expression

```
cp x
```

that sends the message x to object cp will be the object 4, and the value of the expression

```
cp color
```

that sends the message color to the object cp will be the object red. Note that even though move is an inherited method, defined originally for points without color, the result of moving a ColoredPoint is a ColoredPoint.

Run-Time Structure to Support Inheritance

The run-time structures for a ColoredPoint object and ColoredPoint class are illustrated in Figure 11.5. The illustration shows a ColoredPoint object, with the object's class pointer pointing to the ColoredPoint class object. The ColoredPoint class contains a superclass pointer that points to the Point class object. Like the Point class, the

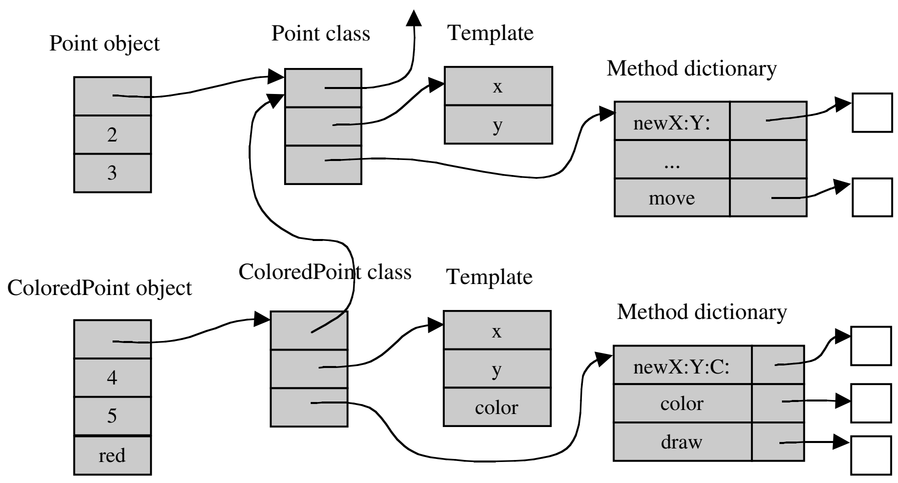


Figure 11.5. Run-time representation of ColoredPoint object and classs.

ColoredPoint class also has a pointer to a template, which contains the names of the instance variables, and a method dictionary, which contains pointers to code that implements each of the methods. Like Figure 11.3, this figure is schematic. An actual compiler could store things slightly differently or perform optimizations to improve performance.

If a draw message is sent to a ColoredPoint object, then we find the code for executing this method by following the pointer in the object to the ColoredPoint class structure and by following the pointer there to the method dictionary for the ColoredPoint class. The dictionary is then searched for a method with the right name. If a message other than new, color, and draw is sent, then the method name will not be found in the ColoredPoint method dictionary. In this case, the superclass pointer in the ColoredPoint class is used to locate the Point class and the Point method dictionary. Then the search continues as if the object were an instance of its superclass.

It should be clear from these data structures that looking up a method at run time can involve a large number of steps. If class A inherits from class B and class B inherits from class C, for example, then searching for an inherited method could involve searching all three method dictionaries at run time. If the operation performed by this method is simple, then it might take more time to find a method than to execute it.

This run-time cost of Smalltalk method lookup illustrates one of the basic principles of programming language design: *You can't get something for nothing*. Dynamic lookup is a powerful programming feature, but it has its costs. Fortunately, there are optimizations that reduce the cost of method lookup in practice. One common optimization (also used in Java) is to cache recently found methods at the method invocation site in a program. If this is done, then a second invocation of the same method can be executed by a jump directly to the code location stored in the cache. As discussed in the next chapter, static type information can also be used to significantly reduce the cost of method lookup. This makes lookup more efficient in C++. However, optimizations based on static type information do not work in Smalltalk because Smalltalk has no static type system.

There are some interesting details surrounding the way that instance variables are accessed by methods. Because the code for the draw method in the ColoredPoint class is compiled at the same time that the layout of ColoredPoint objects is determined, the ColoredPoint method can be compiled so that if it needs to access instance variables of a ColoredPoint object, it can locate them easily (given a pointer to the object). Specifically, the offset of the x coordinate, y coordinate, and color, relative to the starting address of the ColoredPoint object, is known at compile time and can be used in any ColoredPoint method. The same is true for Point methods. However, because a Point method can be inherited by ColoredPoint and invoked on a ColoredPoint object, it is necessary for Point instance variables to appear at the same relative offset in both Point and ColoredPoint objects. More generally, accessing instance variables by use of offsets known at compile time requires the layout of a subclass object to resemble the layout of a superclass object. You might want to think about the advantages and disadvantages of compiling methods in this way compared with run-time search for instance variables by using the class templates. One consequence of using compile-time offsets is that, when a superclass is changed and a new instance variable is added, all subclasses must be recompiled. (This was the case for the Xerox PARC Smalltalk compiler.)

11.5.4 Abstraction in Smalltalk

In most object-oriented languages, a programmer can decide which parts of an object are visible to clients of the class and which parts are not. Smalltalk makes this decision for the programmer by using particularly simple rules:

- *Methods are Public:* Any code with a pointer to an object may send any message to that object. If the corresponding method is defined in the class of the object, or any superclass, the method will be invoked. This makes all methods of an object visible to any code that can access the object.
- *Instance Variables are Protected:* The instance variables of an object are accessible only to methods of the class of the object and to methods of its subclasses.

This is not a fully general approach to abstraction. In particular, there are practical situations in which it would be useful to have private methods, accessible only to other methods of the same class. However, the Smalltalk visibility rules are simple, easy to remember, and sufficient in many situations.

11.6 SMALLTALK FLEXIBILITY

11.6.1 Dynamic Lookup and Polymorphism

Advocates of object-oriented languages often speak about the "polymorphism" inherent in object-oriented code. When they say this, they mean that the same message name invokes different code, depending on the object that receives the message. In this book, we refer to this phenomenon as dynamic lookup, as explained in Chapter 10.

Here is an example that illustrates the value of dynamic lookup. Consider a list of objects, some of which are points and others of which are colored points. If we want to display every point on the list, we may traverse the list, sending each object the draw method. Because of dynamic lookup, the Point objects will execute the Point draw method and the ColoredPoint objects will execute the ColoredPoint draw method. Dynamic lookup supports code reuse, as one routine will display lists of Points, lists of ColoredPoint, and lists that have both Points and ColoredPoints.

The run-time structures used for Smalltalk classes and objects support dynamic lookup in two ways. First, methods are selected through the receiver object. Second, method lookup starts with the method dictionary of the class of the receiver and then proceeds upward. The first method found with the appropriate name is selected.

The following code sends a draw message to a Point and a ColoredPoint:

```
p <- Point newOrigin
p draw
p <- ColoredPoint newX:1 Y:2 C:red
p draw
```

Be sure you understand how and why different code is used in lines two and four to draw the two different kinds of objects.

11.6.2 Booleans and Blocks

An assumption that many people make when learning about Smalltalk is that it has the same kind of syntax, with specific reserved words and special forms, as other languages.

However, many constructs that would be handled with a special-purpose statement form in other languages can be handled uniformly in Smalltalk with the principle that *everything is an object*. A good example is the Smalltalk treatment of Booleans, which uses the built-in construct of blocks.

A Smalltalk Boolean object (true or false) has a selector named ifTrue:ifFalse:. The method associated with this selector requires two parameters, one to be executed if the object is true and the other if the object is false. The arguments to this method are generally Smalltalk *blocks*, which are objects that have a message, "execute yourself." Although blocks may be defined by special in-line syntax, for convenience, they are not essentially different from other Smalltalk objects.

The reason for using blocks in Smalltalk Booleans is easy to understand if we think about conditionals in other languages. In general, the construct

```
if ... then ... else ...
```

cannot be defined as an ordinary function, as an ordinary function call would involve evaluating all arguments before calling the function. If we evaluated A, B, and C first in

```
if A then B else C
```

this is inefficient and incorrect if B and C have side effects or fail to terminate. We want to evaluate B only in the case that A is true and evaluate C only in the case that A is false.

For example, consider the following expression:

```
i < j ifTrue: [i add: 1] ifFalse: [j subtract: 1]
```

This is the Smalltalk way of writing

```
if i < j then i+1 else j-1
```

The first part of the expression, i < j, is a < j message to object i, asking it to compare itself with j. The result will be either the object true or the object false, in either case an object that understands an ifTrue:ifFalse: message. Therefore, after the comparison, i < j, one of the two subsequent blocks will be executed, depending on the result of the comparison.

The way Smalltalk actually works is that the class Boolean has two subclasses, True and False, each with one instance, called true and false, respectively. In these classes, the ifTrue:ifFalse: method is implemented as follows:

```
True class . . .
    ifTrue: trueBlock ifFalse: falseBlock
         trueBlock value
False class . . .
    ifTrue: trueBlock ifFalse: falseBlock
         falseBlock value
```

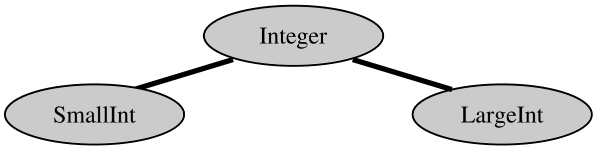
where value is the message you send to a block if you want it to evaluate itself and return its value.

There is a little bit of "special treatment" here in that the system will not allow more instances of classes True and False to be created. This is to guarantee correctness of an optimization. For efficiency, the standard idiom ifTrue: [. . .] ifFalse: [. . .] is optimized by the compiler in most Smalltalk implementation. However, this is again simply an optimization for efficiency. In principle, Booleans could be defined within the language. In addition, the unoptimized methods previously given are called if any arguments are variables that might hold blocks.

11.6.3 Self and Super

The special symbol self may be used in the body of a Smalltalk method. The special property of self is that it always refers to the object that contains this method, whether directly or by inheritance. The way this works is most easily explained by example.

There are several classes of numbers in Smalltalk. Three are arranged in the following hierarchy:



Even if the Integer class does not create any objects, we can define the following method in the Integer class:

```
factorial ||
    self <= 1
       ifTrue: [^ 1]
       ifFalse: [^(self - 1) factorial * self]
```

This method computes the factorial of the integer object to which it belongs. The method first asks the object to compare itself with the predefined number object 1. If it is less-than-or-equal, the result will be the object true and the block [^1] will be evaluated, giving the result 1. If the object is greater than 1, then the second block [^(self - 1) factorial * self] will be evaluated, causing a recursive call to the factorial method on a smaller integer.

There are two operations in the ifFalse case of the factorial method that could be implemented in different ways in the two subclasses of Integer. The first is - (minus). Because small integers may have a different representation from large integers, subtraction may be implemented differently in the two classes. Multiplication, *, may also be implemented differently in the two classes. These methods do not have to return objects of the same class. For example, n*m could be a SmallInt if n and m are both small, or could be a LargeInt if both are SmallInts and their product is larger than the maximum size for SmallInt.

In Smalltalk, we begin a message to self by beginning the search for each method at the object where the first message is received. For example, suppose we want to compute the factorial of a small integer n. The computation will begin by sending the factorial message to n. Because the first step of factorial is self <= 1, the message <= 1 will be sent to the object n. It is important that dynamic lookup is used here, as the correct method for comparing n with 1 is the <= method defined in the SmallInt class. In fact, the Integer class does not need to have a <= method defined in order for the calculation of factorial to work properly. To appreciate the value of self, you might think about whether you could write a factorial method in class Integer that would work properly on SmallInt and LargeInt without using self.

The special symbol super is similar to self, except that, when a message is sent to super, the search for the appropriate method body starts with the superclass of the object instead of the class of the object. This mechanism provides a way of accessing a superclass version of a method that has been overidden in a subclass.

11.6.4 System Extensibility: The Ingalls Test

Dan Ingalls, a leader of the Smalltalk group, proposed the following test for determining whether a programming language is truly object oriented:

> Can you define a new kind of integer, put your new integers into rectangles, ask the system to blacken a rectangle, and have everything work?

This test is probably the most extreme test of an object-oriented language. Essentially he is asking "Is *everything* an object?" Or, more specifically, "Is every operation determined by dynamic lookup through objects?"

Why is This Important?

The main issue addressed by this test is the extensibility of programs and systems. If we build a graphic-oriented system that manipulates and displays various kinds of objects, then the design and implementation of basic objects like points and coordinates might be determined very early in the design process. If a complex system is built around one basic construct, like standard language-supplied integers, then how easy will it be to modify the system to use another form of integers?

The answer, of course, is that it depends on how you build the system. In Smalltalk, however, the only way of building a system is by representing basic concepts as objects. Because all operations in Smalltalk are done by sending messages and every message results in run-time lookup of the appropriate method body, a lot of flexibility is inherently built into any system. In C++ and Java, by contrast, it is possible to build a system with integers that are not objects. As a result, it is possible to build systems that will not immediately accept other forms of integer without some change to the code. In particular, most Java or C++-based systems will not be able to handle a rectangle that has some coordinates given by built-in integers that are not objects and some coordinates given by objects.

What Are the Implementation Costs?

If a system is built so that every operation on integers is accompanied by method lookup, then an expression `x add y` sending a message to an integer cannot be safely optimized to a static function call such as `integer_add(x,y)`. If the compiler or programmer replaces `x add y` with `integer_add(x,y)`, then we cannot replace a built-in integer `x` with a user-defined integer object `x`.

If a program uses integers repeatedly and each integer operation involves a run-time search for an appropriate method, this is a *substantial run-time cost*, even if the best-known optimizations are used to reduce running time as much as possible. In fact, for essentially this reason, the efficiency of many Smalltalk programs is 5 to 10 times less than corresponding programs in C. Of course, program running time is not everything. If a system will be used in many ways, and flexibility and adaptability are important, then flexibility and adaptability may be more important than running speed.

11.7 RELATIONSHIP BETWEEN SUBTYPING AND INHERITANCE

11.7.1 Interfaces as Object Types

Although the Smalltalk language does not use any static type checking, there is an implicit form of type that every Smalltalk programmer uses in some way. The type of an object in Smalltalk is its interface, the set of messages that can be sent to the object without receiving the error “message not understood.” The interface of an object is determined by its class, as a class lists the messages that each object will answer. However, different classes may implement the same messages, as there are no Smalltalk rules to keep different classes from using the same selector names.

The Smalltalk library contains a number of implemented classes for a variety of purposes. One part of the library that provides interesting examples of interfaces and inheritance is the collection-class library. The collection classes implement general collections like Set, Bag, Dictionary, LinkedList, and a number of more specialized kinds of collections. Some example interfaces from the collection classes are the following sets of selector names:

```
Set_Interface = {isEmpty, size, includes:, removeEvery:}
Bag_Interface = {isEmpty, size, includes:, occurrencesOf:, add:, remove:}
Dictionary_Interface = {at:put:, add:, values, remove:}
Array_Interface = {at:put:, atAllPut:, replaceFrom:to:with:}
```

Remember that a colon indicates a position for a parameter. For example, if s is a set object, then s includes:3 sends the message includes:3 to s. The result will be true or false, depending on whether the set s contains the element 3.

Here is a short explanation of the intended meaning of each of the selectors:

isEmpty: return true if the collection is empty, else false
size: return number of elements in the collection
includes:x: return true if x is in the collection, false otherwise
removeEvery:x: remove all occurrences of x from the collection
occurrencesOf:x: return a number telling how many occurrences of x are in the collection
add:x: add x to the collection
remove:x: remove x from the collection
at:x put:y: insert value y into an indexed collection at position indexed by x
values: return the set of indexed values in the indexed collection (e.g., list all the words in a dictionary)
atAllPut:x: in finite updateable collection such as an array, set every value to x
replaceFrom:x t:y with:v: in range from x to y, set every indexable value to v (e.g., for array A, set A[x]=A[x+1]= ... =A[y]=v)

11.7.2 Subtyping

Because Smalltalk is an untyped language, it may not seem relevant to compare subtyping and inheritance. However, if we recall the definition of a subtype,

Type A is a subtype of a type B if any context expecting an expression of type B may take any expression of type A without introducing a type error,

we can see that subtyping is about substitutivity. If we build a Smalltalk program by using one class of objects and then try to use another class in its place, this can work if the interface of the second class contains all of the methods of the first, but not necessarily otherwise. Therefore, it makes sense to associate subtyping with the *superset* relation on Smalltalk class interfaces. Subtyping is sometimes called conformance when discussing interfaces instead of types of some typed programming language.

An example of subtyping arises if we consider extensible collections. To be specific, let us define ExtensibleCollection_Interface by

```
ExtensibleCollection_Interface = {isEmpty, size, includes:, add:, removeEvery:}
```

This is the same as Set_Interface, with the method add: added. Using <: for subtyping, we have

```
ExtensibleCollection_Interface <: Set_Interface
```

In words, this means that every object that implements the ExtensibleCollection interface also implements the Set interface. Semantically, this makes sense because a program that uses Set objects can send any of the messages listed in the Set interface to Set objects. Because each one of these messages is also implemented in every ExtensibleCollection object, we can substitute ExtensibleCollection objects for Set objects without obtaining "message not understood" errors.

11.7.3 Subtyping and Inheritance

In Smalltalk, the interface of a subclass is often a subtype of the interface of its superclass. The reason is that a subclass will ordinarily inherit all of the methods of its superclass, possibly adding more methods. This is certainly the case for Point and ColoredPoint, as described earlier in this chapter. Points have the five methods x:y:, moveDx:Dy:, x,y, and draw. Colored points have these five methods and a color method, so every colored point has all of the point methods.

In general, however, subclassing does not always lead to subtyping in Smalltalk. The simplest way that subtyping may fail is when a subclass does not implement all of the methods of the superclass. In Smalltalk, there is simply a way of saying, in a subclass, that one or more superclass methods should not be defined on subclass objects. Because it is possible to delete a method from a superclass, a subclass may not produce a subtype.

On the other hand, it is easy to have subtyping without inheritance. Because subtyping, as we have defined it, is based only on the names of methods, two classes could be defined independently (neither a subclass of the other) but have exactly the same methods. Then, according to our definitions, each is a subtype of the other. But there is no inheritance.

William Cook, a researcher at Hewlett-Packard Labs and later a developer at Apple Computer, did an empirical study of the collection classes in the Smalltalk-80 class library to determine which relationships between subtyping and inheritance arise in Smalltalk practice (Interfaces and Specifications for the Smalltalk-80 Collection Classes, ACM Conf. Object.Oriented Programming: Systems, Languages, and Applications, ACM SIGPLAN Notices 27(10), 1992). Among a relatively small number of classes, he found all possible relationships between subtyping and inheritance. A diagram from his study is shown in Figure 11.6. In this diagram, solid lines indicate subtyping between interfaces, and dotted curves show actual places where inheritance is used in the construction of the collection classes of the Smalltalk-80 library. In studying this diagram, it is important to keep in mind that there are more interfaces listed here than were actually implemented in the library. To show the structural relationship between classes more clearly, Cook included some interfaces that are intersections of the interfaces of classes in the library.

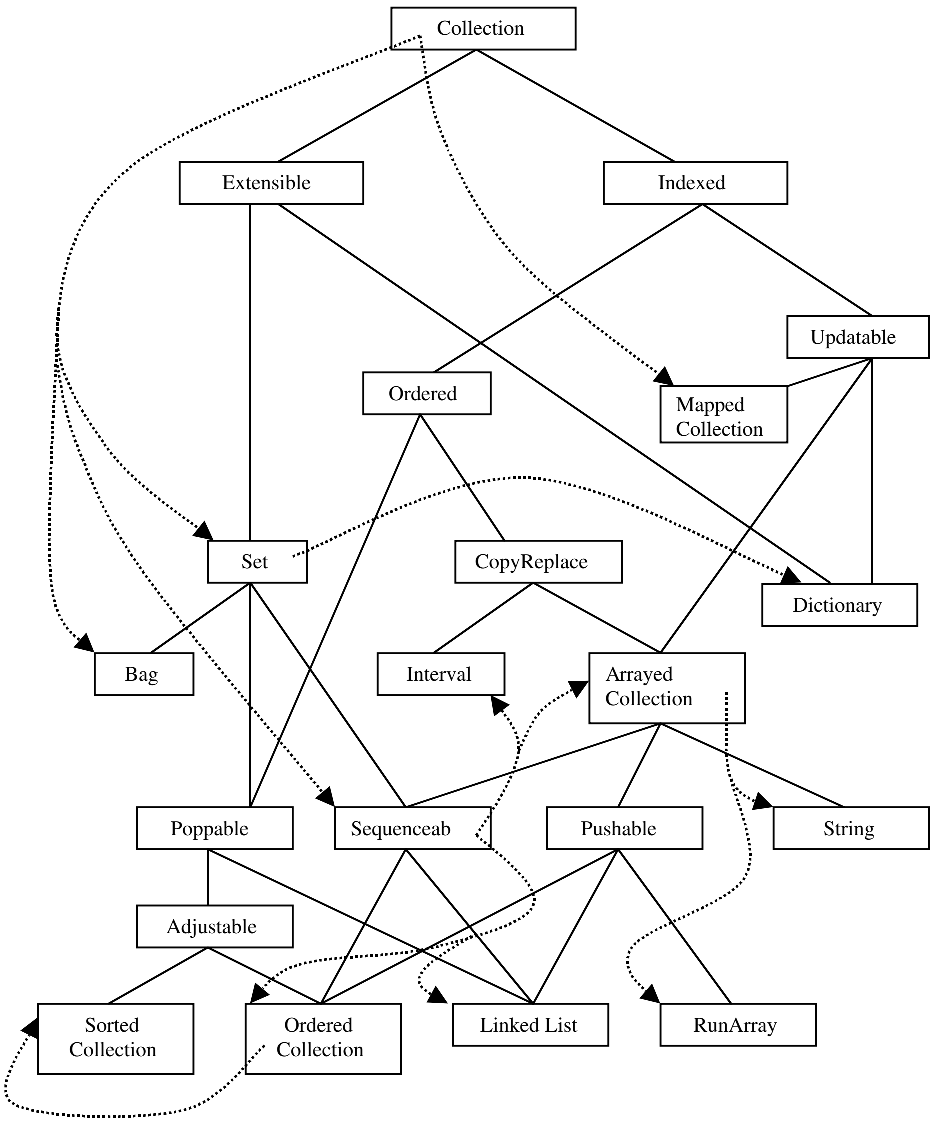


Figure 11.6. Smalltalk collection class hierarchy.

Generally speaking, the subtyping relation between a set of classes is independent of how the classes are implemented, but the inheritance relation is not. Subtyping is a relation between types, which is a property that is important to users of an object, whereas inheritance is a property of implementations. Imagine a group of people sitting down to design the Smalltalk collection classes. Once they have agreed on the names of the classes and the operations on each class, they have agreed on the subtyping relation between the classes. However, the inheritance relation is not determined at this point. If each person implemented one of the classes independently, then all of the classes could be built without using inheritance at all. At the other extreme, it is often possible to factor the functionality of a set of related classes so that inheritance is used extensively. To give a specific example, arrays and dictionaries are similar kinds of collections. Abstractly, both are a form of function from a finite set to an arbitrary set. Therefore, it is conceivable that a programmer could build a general finite-function class and use this as the dictionary class. This programmer can inherit from dictionary to produce the special case of arrays in which the index set is always a range of integers. On the other hand, a different programmer could first build arrays and then use arrays to implement dictionaries by hashing each index word to an integer value and using this to store the indexed value in a specific array location. This illustrates the independence of subtyping and inheritance in Smalltalk.

11.8 CHAPTER SUMMARY

In this chapter, we studied Simula, the first object-oriented language, and Smalltalk, an influential language that generalized and popularized object-oriented programming.

Simula was designed in Norway, beginning as part of a project to develop a simulation language. Smalltalk was designed at Xerox PARC (in Palo Alto, California) and was intended to accompany a futuristic concept called the Dynabook. An important unifying concept in Smalltalk is that everything should be an object.

Simula

- *Objects:* A Simula object is an activation record produced by call to a class.
- *Classes:* A Simula class is a procedure that returns a pointer to its activation record. The body of a class may initialize the object it creates.
- *Abstraction:* Hiding was not provided in 1967, but was added later and used as the basis for C++.
- *Subtyping:* Objects are typed according to the classes that create them. Subtyping is determined by class hierarchy.
- *Inheritance:* A Simula class may be defined as an extension of a class that has already been defined.

Smalltalk

- *Objects:* A Smalltalk object is created by a class. At run time, an object stores its instance variables and a pointer to the instantiating class.
- *Classes:* A Smalltalk class defines class variables, class methods, and the instance methods that are shared by all objects of the class. At run time, the class data

structure contains pointers to an instance variable template, a method dictionary, and the superclass.

- *Abstraction:* Abstraction is provided through protected instance variables. All methods are public but instance variables may be accessed only by the methods of the class and methods of subclasses.
- *Subtyping:* Smalltalk does not have a compile-time type system. Subtyping arises implicitly through relations between the interfaces of objects. Subtyping depends on the set of messages that are understood by an object, not the representation of objects or whether inheritance is used.
- *Inheritance:* Smalltalk subclasses inherit all instance variables and methods of their superclasses. Methods defined in a superclass may be redefined in a subclass or deleted.

Because Smalltalk has no compile-time type system, a variable may point to any object of any type at run time. This makes it more difficult to implement efficient method lookup algorithms for Smalltalk than for some other object-oriented languages. You may wish to review the data structure and algorithms for locating a method of an object as illustrated in Subsection 11.5.3. Be sure you understand how search for a method proceeds from subclass to superclass in the event that the method is not defined in the subclass.

EXERCISES

11.1 Simula Inheritance and Access Links

In Simula, a class is a procedure that returns a pointer to its activation record. Simula prefixed classes were a precursor to C++ -derived classes, providing a form of inheritance. This question asks about how inheritance might work in an early version of Simula, assuming that the standard static scoping mechanism associated with activation records is used to link the derived class part of an object with the base class part of the object.

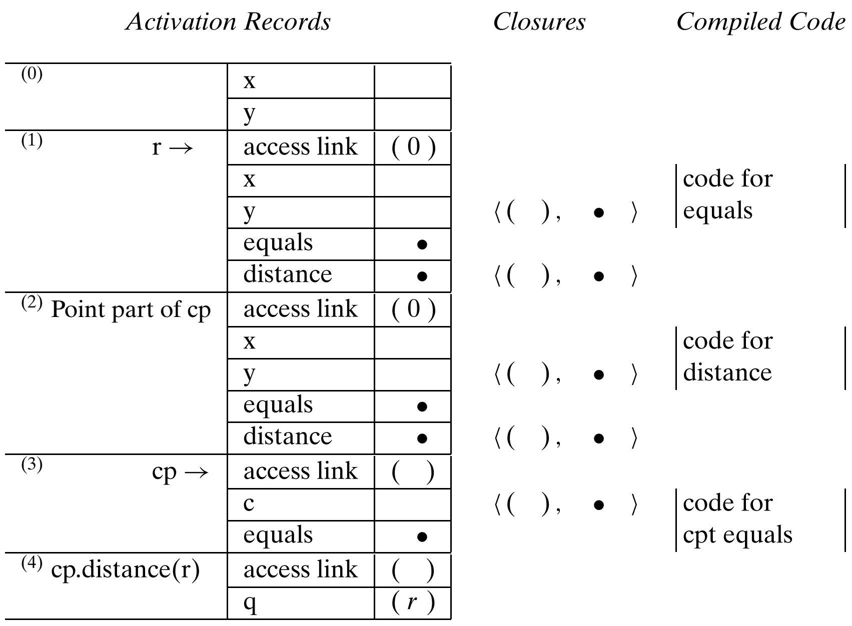
Sample `Point` and `ColorPt` classes are given in the text. For the purpose of this problem, assume that if `cp` is a `ColorPt` object, consisting of a `Point` activation record followed by a `ColorPt` activation record, the access link of the parent-class (`Point`) activation record points to the activation record of the scope in which the class declaration occurs, and the access link of the child-class (`ColorPt`) activation record points to activation record of the parent class.

(a) Fill in the missing information in the following activation records, created by execution of the following code:

```
ref(Point) p;
ref(ColorPt) cp;
r :- new Point(2.7, 4.2);
cp :- new ColorPt(3.6, 4.9, red);
cp.distance(r);
```

Remember that function values are represented by closures and that a closure is a pair consisting of an environment (pointer to an activation record) and a compiled code.

In the following illustration, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. Because the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled “access link.” The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.



(b) The body of distance contains the expression

```
sqrt(( x - q.x )**2 + (y - q.y) ** 2)
```

that compares the coordinates of the point containing this distance procedure to the coordinate of the point q passed as an argument. Explain how the value of x is found when cp.distance(r) is executed. Mention specific pointers in your diagram. What value of x is used?

(c) This illustration shows that a reference cp to a colored point object points to the ColorPt part of the object. Assuming this implementation, explain how the expression cp.x can be evaluated. Explain the steps used to find the right x value on the stack, starting by following the pointer cp to activation record (3).

(d) Explain why the call cp.distance(r) needs access to only the Point part of cp and not the ColorPt part of cp.

(e) If you were implementing Simula, would you place the activation records representing objects r and cp on the stack, as shown here? Explain briefly why you might consider allocating memory for them elsewhere.

11.2 Loophole in Encapsulation

A priority queue (also known as a heap) is a form of queue that allows the minimum element to be retrieved. Assuming that Simula has a list type, a Simula pq class might look something like this:

```
class pq();
begin
    (int list) contents;
    bool procedure isempty();
       < . . . return true if pq empty, else false . . . >
    procedure insert(x); int x;
       < . . . put x at appropriate place in list . . . >
    int procedure deletemin();
       < . . . delete first list elt and return it . . . >
    contents := nil
end
```

where the sections <. . .> would be filled in with appropriate executable Simula code. We would create a priority queue object by writing

```
h :- new pq();
```

and access the components by writing something like

```
h.contents;
h.isempty;
h.insert(x);
h.deletemin();
```

In ML we can write a function that produces a closure with essentially the same behavior as that of a Simula priority queue object. This is subsequently written, together with the associated exception declaration:

```
exception Empty;

fun new_pq() =
    let val contents = (ref nil) : int list ref
        fun insert(x,nil)    = [x]
        |   insert(x, y::l) = if x<y then x::(y::l)
                                  else y::insert(x, l)
    in
        {
         emp_meth = fn () => !contents = nil,
         ins_meth = fn x =>  (contents := insert(x, !contents)),
         del_meth = fn () => case !contents of
                                  nil => raise Empty
                                |   y::l => (contents := l; y)
         }
    end;
```

This question asks you to think about a "loophole" in the Simula object system that allows a client program to interfere with the correct behavior of a priority queue object.

(a) The main property of a priority queue is that if n elements are inserted and k are removed (for any $k < n$), then these k elements are returned in increasing order. Explain in two or three sentences why both the Simula and ML forms of priority queue objects should exhibit this behavior in a "reasonable" program.

(b) Explain in a few sentences how a "devious" program that uses a Simula priority queue object h could cause the following behavior: After 0 and 1 are inserted into the priority queue by calls h.insert(0) and h.insert(1), the first deletemin operation on the priority queue returns some number other than 0. (This will not be possible for an ML priority queue object.) Do not use any pointer arithmetic or other operations that would not be allowed in a language like Pascal.

(c) Write a short sequence of priority queue operations of the form

```
h :- new pq();
h.insert(0);
h.insert(1);
...
h.deletemin();
```

where the ellipsis (...) may contain h operations, but not insert or deletemin, demonstrating the behavior you described in part (b).

11.3 Subtyping of Refs in Simula

In Simula, the procedure call assignA(b) in the following context is considered statically type correct:

```
class A ... ;     /* A is a class */
A class B ... ;   /* B is a subclass of A */

ref (A) a;        /* a is a variable pointing to an A object */
ref (B) b;        /* b is a variable pointing to a B object */

proc assignA (ref (A) x)
   begin
      x := a
   end;

assignA(b);
```

(a) Assume that *if B <: A then ref (B) <: ref (A)*. Using this principle, explain why both the procedure assignA and the call assignA(b) can be considered statically type correct.

(b) Explain why actually executing the call assignA(b) and performing the assignment given in the procedure may lead to a type error at run time.

(c) The problem is that the "principle," *if B <: A then ref (B) <: ref (A)*, is not semantically sound. However, we can make type checking by using this principle sound by inserting run-time tests. Explain the run-time test you think a Simula compiler should insert in the compiled code for procedure assignA. Can you think of a reason why the designers of Simula might have decided to use run-time tests instead of disallowing ref subtyping in this situation? (You do not have to agree with them; just try to imagine what rationale might have been used at the time.)

11.4 Smalltalk Run-Time Structures

Here is a Smalltalk Point class whose instances represents points in the two-dimensional Cartesian plane. In addition to accessing instance variables, an instance method allows point objects to be added together.

Class name	Point
Superclass	Object
Class variables	*Comment: none*
Instance variables	*x y*
Class messages and methods	*Comment: instance creation* newX: xValue Y: yValue | | ↑ self new x: xValue y: yValue
Instance messages and methods	*Comment: accessing instance vars* x: xCoordinate y: yCoordinate | | x ⟵ xCoordinate y ⟵ yCoordinate x | | ↑ x y | | ↑ y *comment: arithmetic* + aPoint | | ↑ Point newX: (x + aPoint x) Y: (y + aPoint y)

(a) Complete the top half of the illustration of the Smalltalk run-time structure shown in Figure P.11.4.1 for a point object with coordinates (3, 4) and its class. Label each of the parts of the top half of the figure, adding to the drawing as needed.

(b) A Smalltalk programmer has access to a library containing the Point class, but she cannot modify the Point class code. In her program, she wants to be able to create points by using either Cartesian or polar coordinates, and she wants to calculate both the polar coordinates (radius and angle) and the Cartesian coordinates of points. Given a point (x, y) in Cartesian coordinates, the radius is ((x * x) + (y * y)) squareRoot and the angle is (x/y) arctan. Given a point (r, θ) in polar coordinates, the x coordinate is r * (θ cos) and the y coordinate is r * (θ sin).

  i. Write out a subclass PolarPoint, of Point and explain how this solves the programming problem.

  ii. Which parts of Point could you reuse and which would you have to define differently for PolarPoint?

(c) Complete the drawing of the Smalltalk run-time structure by adding a PolarPoint object and its class to the bottom half of the figure you already filled in with Point structures. Label each of the parts and add to the drawing as needed.

11.5 Smalltalk Implementation Decisions

In Smalltalk, each class contains a pointer to the class template. This template stores the names of all the instance variables that belong to objects created by the class.

(a) The names of the methods are stored next to the method pointers. Why are the names of instance variables stored in the class, instead of in the objects (next to the values for the instance variables)?

(b) Each class's method dictionary stores the names of only the methods explicitly written for that class; inherited methods are found by searching up the superclass pointers at run time. What optimization could be done if each subclass

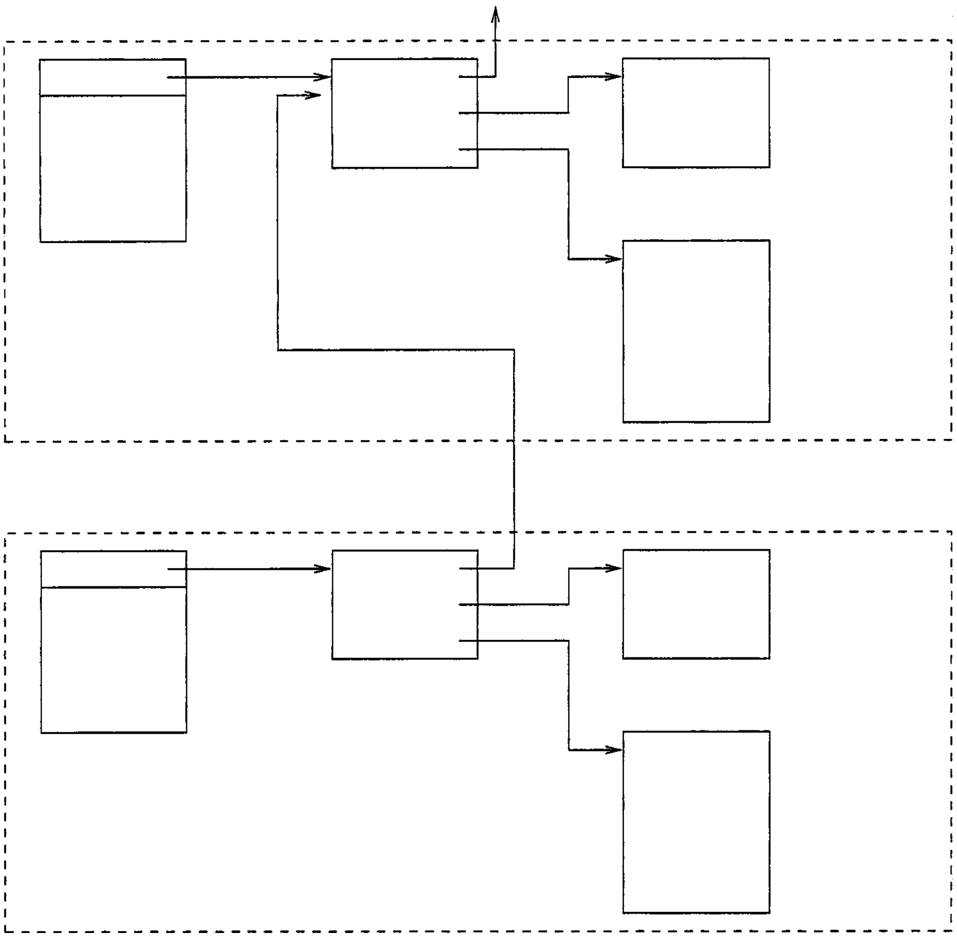


Figure P.11.4.1. Smalltalk run-time structures for Point and PolarPoint.

contained all of the methods of its superclasses in its method dictionary? What are some of the advantages and disadvantages of this optimization?

(c) The class template stores the names of all the instance variables, even those inherited from parent classes. These are used to compile the methods of the class. Specifically, when a subclass is added to a Smalltalk system, the methods of the new class are compiled so that, when an instance variable is accessed, the method can access this directly without searching through the method dictionary and without searching through superclasses. Can you see some advantages and disadvantages of this implementation decision compared with looking up the relative position of an instance variable in the appropriate class template each time the variable is accessed? Keep in mind that in Smalltalk, a set of classes could remain running for days, while new classes are added incrementally and, conceivably, existing classes could be rewritten and recompiled.

11.6 Protocol Conformance

We can compare Smalltalk interfaces to classes that use *protocols*, which are lists of operation names (selectors). When a selector allows parameters, as in at: put: , the selector name includes the colons but not the spaces. More specifically, if dict is an updatable collection object, such as a dictionary, then we could send dict a message by writing dict at:'cross' put:'angry'. (This makes our dictionary definition of "cross" the single word "angry.") The protocol for updatable collections will therefore contain the seven-character selector name at:put: . Here are some example protocols:

```
          stack : {isEmpty, push:, pop }
          queue : {isEmpty, insert:, remove }
 priority_queue : {isEmpty, insert:, remove }
        dequeue : {isEmpty, insert:,insertFront:, remove, removeLast }
simple_collection : {isEmpty }
```

Briefly, a stack can be sent the message isEmpty, returning true if empty, false otherwise; push: requires an argument (the object to be pushed onto the stack), pop removes the top element from the stack and returns it. Queues work similarly, except they are first-in, first-out instead of first-in, last-out. Priority queues are first-in, minimum-out and dequeues are doubly ended queues with the possibility of adding and removing from either end. The simple_collection class just collects methods that are common to all the other classes. We say that the protocol for A *conforms* to the protocol for B if the set of A selector names contains the set of B selector names.

(a) Draw a diagram of these classes, ordered by protocol conformance. You should end up with a graph that looks like William Cook's drawing shown in Figure 11.6.

(b) Describe briefly, in words, a way of implementing each class so that you make B a subclass of A only if the protocol for B conforms to the protocol set for A.

(c) For some classes A and B that are unrelated in the graph, describe a strategy for implementing A as a subclass of B in a way that keeps them unrelated.

(d) Describe implementation strategies for two classes A and B (from the preceding set of classes) so that B is a subclass of A, but A conforms to B, not the other way around.

11.7 Removing a Method

Smalltalk has a mechanism for "undefining" a method. Specifically, if a class A has method m, then a programmer may cancel m in subclass B by writing

```
m:
    self shouldNotImplement
```

With this declaration of m in subclass B, any invocation of m on a B object will result in a special error indicating that the method should not be used.

(a) What effect does this feature of Smalltalk have on the relationship between inheritance and subtyping?

(b) Suppose class A has methods m and n, and method m is canceled in subclass B. Method n is inherited and not changed, but method n sends the message m to self. What do you think happens if a B object b is sent a message n? There are two possible outcomes. See if you can identify both, and explain which one you think the designers of Smalltalk would have chosen and why.

11.8 Subtyping and Binary Methods

This question is about the relationship between subtyping and inheritance. Recall that the main principle associated with subtyping is substitutivity: If A is a subtype of B, then wherever a B object is required in a program, an A object may be used instead without producing a type error. For the purpose of this question, we will use *Message not understood* as our Smalltalk type error. This is the most common error message resulting from a dynamic type failure in Smalltalk; it is the object-oriented analog of the error *Cannot take car of an atom* that every Lisp programmer has generated at some time. Remember that Smalltalk is a dynamically typed language. This question asks you to show how substitutivity can fail, using the fact that a method given in a superclass can be redefined in a subclass.

If there are no restrictions on how a method (member function) may be redefined in a subclass, then it is easy to redefine a method so that it requires a different number of arguments. This will make it impossible to meaningfully substitute a subclass object for a superclass object. A more subtle fact is that subtyping may fail when a method is redefined in a way that appears natural and (unless you've seen this before) unproblematic. This is illustrated by the following Point class and ColoredPoint subclass:

Class name	Point
Class variables	
Instance variables	xval yval

Instance messages and methods	xcoord ↑ xval
	ycoord ↑ yval
	origin
	  xval ⟵ 0
	  yval ⟵ 0
	movex: dx movey: dy
	  xval ⟵ xval + dx.
	  yval ⟵ yval + dy
	equal: pt
	  ↑ (xval = pt xcoord & yval = pt ycoord)

Class name	ColoredPoint
Class variables	
Instance variables	color
Instance messages and methods	color ↑ color changecolor: newc color ⟵ newc equal: cpt ↑ (xval = cpt xcoord & yval = cpt ycoord & color = cpt color)

The important part here is the way that equal is redefined in the colored point class. This change would not be allowed in Simula or C++, but is allowed in Smalltalk. (The C++ compiler does not consider this an error, but it would not treat it as redefinition of a member function either.) The intuitive reason for redefining equal is that two colored points are equal only if they have the same coordinates and are the same color.

Problem: Consider the expression `p1 equal:p2`, where `p1` and `p2` are either `Point` objects or `ColoredPoint` objects. It is guaranteed not to produce *Message not understood* if both `p1` and `p2` are either `Points` or `ColoredPoints`, but may produce an error if one is a `Point` and the other a `ColoredPoint`. Consider all four combinations of `p1` and `p2` as `Points` and `ColoredPoints`, and explain briefly how each message is interpreted.

11.9 Delegation-Based Object-Oriented Languages

In this problem, we explore a delegation-based object-oriented language, Self, in which objects can be defined directly from other objects. Classes are not needed and not supported. Self has run-time type checking and garbage collection.

Here is the Self language description:

(extracted from D. Ungar and R. B. Smith, Self: The power of simplicity, ACM SIGPLAN Notices 22(12), 1987, pp. 227–242.)

> In Smalltalk, . . . everything is an object and every object contains a pointer to its class, an object that describes its format and holds its behavior. In Self too, everything is an object. But, instead of a class pointer, a Self object contains named slots with may store either state or behavior. If an object receives a message and it has no matching slot, the search continues via a *parent* pointer. This is how Self implements inheritance. Inheritance in Self allows objects to share behavior, which in turn allows the programmer to alter the behavior of many objects with a single change. For instance, a [Cartesian] point object would have slots for its non-shared characteristics: x and y. Its parent would be an object that held the behavior shared among all points: +, etc.
>
> In Self, there is no direct way to access a variable: instead, objects send messages to access data residing in name slots. So to access its “x” value, a point sends itself the “x” message. The message finds the “x” slot, and evaluates the object found therein. . . . In order to change contents of the “x” slot to, say, 17, instead of performing an assignment like “x←17,” the point must send itself the “x:” message with 17 as the argument. The point object must contain a slot named “x:” containing the assignment [function].

(a) Using this description, draw the run-time data structure of the Self version of a `(3,4)` `Point` object, as described in the last two sentences of the first paragraph, and it parent object. Assume that assignments to the `x` and `y` characteristics are permitted.

(b) Self’s lack of classes and instance variables make inheritance more powerful. For example, to create two points sharing the same `x` coordinate, the `x` and `x:` slots can be put in a separate object that is a parent of each of the two points. Draw a picture of the run-time data structures for two points sharing the same `x` coordinate.

(c) To create a new `Point` object, the `clone` message is sent to an existing point. The language description continues:

> Creating new objects . . . is accomplished by a simple operation, copying. . . . [In Smalltalk,] creating new objects from classes is accomplished by instantiation, which includes the interpretation of format information in a class. Instantiation is similar to building a house from a plan.

Cloning an object does not clone its parent, however.

If a *point* object contains fields `x` and `y` and methods `x:`, `y:`, `move`, then cloning the object will create another object with two fields and three methods. Each point will have a parent pointer, two fields, and three methods – six entries in all. The SELF point will be twice the size of the corresponding Smalltalk point.

Explain how you would structure your SELF program so that each point can be cloned without cloning its methods.

(d) SELF also allows a *change-parent* operation on an object. The parent pointer of an object can be set to point to any other object. (An exception is that the first object must not be an ancestor of the second – we do not want a cycle.)

The change-parent operation is useful for objects that change behavior in different states. For example, a window can be in the *visible* or *iconified* (minimized) state. When iconified, mouse clicks and window display work differently than when the window is visible. A window's parent pointer can be set to `VisibleWindow` initially, then changed to `IconifiedWindow` when the "minimized" button is pressed. Another example is a file object that can be in the *open* or *closed* state. The `open` method changes the parent pointer from `ClosedFile` to `OpenFile`.

The change-parent operation adds a lot of flexibility to SELF. Can you think of disadvantages of this feature?