35

Simula to Java and beyond: major O-O languages and environments

Encouraged by the introduction of Simula in 1967, a number of object-oriented languages have appeared on the scene, highlighting various aspects of the approach. This chapter reviews some of the languages that have attracted the most attention: Simula; Smalltalk; C++ and other O-O extensions of C; Java.

The literature still lacks an in-depth comparative study of important O-O languages. The ambition of this chapter is of necessity more modest. In particular, the space allotted to each language is not an indication of the language's practical significance, and some of the most publicized will indeed get a fairly short treatment. Our goal is to learn about *issues* and *concepts*, finding them where we can, even if that means turning our attention for a while to one of the less hyped approaches. The risk of under-representing one of the principal players is not great, since one only has to look around to pick up articles and books describing it in generous detail. The real risk would be the reverse: to miss a promising idea just because the language supporting it (say Simula) does not currently enjoy top favor. In its coverage of notable languages, then, this survey is not equal-opportunity; it is instead, in its choice of notable language traits, a case of affirmative action.

Even when the concepts are the same or similar, the terms used to denote them in official language descriptions can vary. The discussion will use the native terms when they reflect language peculiarities; for simplicity and consistency, however, it uses the terminology of the rest of this book (designed as an attempt at unification) when differences are unimportant. For example you will read about Simula routines, procedures and functions, although the corresponding terms in official Simula usage are procedure, untyped procedure and typed procedure.

35.1 SIMULA

The undisputed founder of the House of Classes (Object Palace) is Simula, whose design was completed (if we ignore a few later updates, entirely minor) in 1967. This may seem hard to believe: a full-fledged object-oriented language was around, and implemented, *before* structured programming, before Parnas had published his articles on information hiding, many years before anyone had come up with the phrase "abstract data type". The Vietnam War was still a page-4 item; barricades had not yet sprung up in the streets of Paris; a mini-skirt could still cause a stir: away by the Northern shores of the Baltic a few fortunate software developers led by a handful of visionaries were already profiting from the power of classes, inheritance, polymorphism, dynamic binding and most of the other marvels of object orientation.

Background

Simula is actually a second design. In the early sixties, a language now known as Simula 1 was developed to support the programming of discrete-event simulations. Although not quite object-oriented in the full sense of the term, it already showed some of the key insights. "Simula" proper is Simula 67, designed in 1967 by Kristen Nygaard and Ole-Johan Dahl from the University of Oslo and the Norwegian Computing Center (Norsk Regnesentral). Nygaard has explained since how the decision to keep the name was meant to ensure continuity with the previous language and the link to its user community; but an unfortunate effect was that for a long time that name evoked for many people the image of a language meant only for discrete-event simulation — a relatively narrow application area — even though Simula 67 is definitely a general-purpose programming language, whose only simulation-specific features are a handful of instructions and a *SIMULATION* library class, used by a minority of Simula developers.

The name was shortened to just Simula in 1986; the current standard is from 1987.

Availability

Simula is often presented as a respectable but defunct ancestor. In fact it is still alive and enjoys the support of a small but enthusiastic community. The language definition is maintained by the "Simula Standards Group". Compilers are available for a variety of hardware and software environments from several companies, mostly Scandinavian.

Major language traits

We will take a general look at the basic properties of Simula. To some readers Simula will be passé, and the author of this book will not feel insulted if you skip to the next section, on Smalltalk. But if you do want to gain a full appreciation of object technology you will find Simula worth your time; the concepts are there in their original form, and a few of them show possibilities that may not yet, thirty years later, have been fully exploited.

Simula is an object-oriented extension of Algol 60. Most correct Algol programs are also correct Simula programs. In particular, the basic control structures are those of Algol: loop, conditional, switch (a multiple branch instruction, low-level precursor to Pascal's **case** instruction). The basic data types (integer, real etc.) are also drawn from Algol.

Like Algol, Simula uses at the highest level a traditional software structure based on the notion of main program. An executable program is a main program containing a number of program units (routines or classes). Simula environments do support, however, a form of separate class compilation.

Simula uses full block structure in the Algol 60 style: program units such as classes may be nested within one another.

All Simula implementations support automatic garbage collection. There is a small standard library, including in particular two-way linked lists used by the *SIMULATION* class studied later in this chapter.

See "References and simple values", page 272.

As in the notation of this book, the most common entities of non-basic types denote references to class instances, rather than the instances themselves. Instead of being implicit, however, this property is emphasized by the notation. You will declare the type of such an entity as **ref** (*C*), rather than just *C*, for some class *C*; and the corresponding operations will use special symbols: :– for an assignment where integer or real operands would use :=; == rather than = for equality; =/= rather than /= for inequality. An earlier chapter presented the rationale for and against this convention.

To create an instance, you will use, rather than a creation instruction, a **new** expression:

```
ref (C) a; …; a :– new C
```

Evaluation of the **new** expression creates an instance of *C* and returns a reference to it. A class may have arguments (playing the role of the arguments to creation procedures in our notation), as in

```
class C (x, y); integer x, y
    begin … end;
```

In this case, the **new** expression must provide corresponding actual arguments:

```
a :– new C (3, 98)
```

The arguments may then be used in routines of the class; but unlike with creation instructions this gives only one initialization mechanism.

Besides routines and attributes, a class may contain a sequence of instructions, the **body** of the class; if so, the **new** call will execute these instructions. We will see how to use this possibility to make classes represents process-like computational elements rather than just passive objects as in most other O-O languages.

No assertion mechanism is provided. Simula supports single inheritance; to declare *B* as an heir of *A*, use

```
A class B;
    begin … end
```

To redefine a feature of a class in a descendant class, simply provide a new declaration; it will take precedence over the original one. (There is no equivalent to the **redefine** clause.)

The original version of Simula 67 did not have explicit information hiding constructs. In more recent versions, a feature declared as **protected** will be unavailable to clients; a protected feature which is further declared as **hidden** will also be unavailable to proper descendants. A non-protected feature may be protected by a proper descendant, but a protected feature may not be re-exported by proper descendants.

Deferred features are offered in the form of “virtual routines”, appearing in a **virtual** paragraph at the beginning of the class. It is not necessary to declare the arguments of a virtual routine; this means that different effective definitions of a virtual routine may have different numbers and types of arguments. For example, a class *POLYGON* might begin

```
class POLYGON;
    virtual: procedure set_vertices
begin
    …
end
```

allowing descendants to provide a variable number of arguments of type *POINT* for *set_vertices*: three for *TRIANGLE*, four for *QUADRANGLE* etc. This flexibility implies that some of the type checking must be done at run time.

C++ users should beware of a possible confusion: although inspired by Simula, C++ uses a different meaning for the word *virtual*. A C++ function is virtual if it is meant to be dynamically bound (it is, as we have seen, one of the most controversial aspects of C++ that you must specify this requirement explicitly). The C++ approximation to Simula’s virtual procedures is called a “pure virtual function”.

“The C++ approach to binding”, page 514.

Simula supports polymorphism: if *B* is a descendant of *A*, the assignment *a1 :– b1* is correct for *a1* of type *A* and *b1* of type *B*. (Interestingly enough, assignment attempt is *almost* there: if the type of *b1* is an ancestor of the type of *a1*, the assignment will work if the run-time objects have the proper conformance relationship — source descendant of target; if not, however, the result will be a run-time error, rather than a special value which, as with assignment attempt, the software could detect and handle.) By default, binding is static rather than dynamic, except for virtual routines. So if *f* is a non-virtual feature declared at the *A* level, *a1*.*f* will denote the *A* version of *f* even if there is a different version in *B*. You can force dynamic binding by using the **qua** construct, as in

```
(a1 qua B).f
```

This, of course, loses the automatic adaptation of every operation to its target. You may however obtain the desired dynamic binding behavior (which may largely be considered a Simula invention) by declaring polymorphic routines as virtual. In many of the examples that we have studied, a polymorphic routine was not deferred but had a default implementation right from the start. To achieve the same effect, the Simula developer will add an intermediate class where the routine is virtual.

As an alternative to using **qua**, the **inspect** instruction makes it possible to perform a different operation on an entity *a1*, depending on the actual type of the corresponding object, which must be a descendant of the type *A* declared for *a1*:

```
inspect a1
    when A do …;
    when B do …;
    …
```

This achieves the same effect but assumes that the set of descendants of a class is frozen, and runs into conflict with the Open-Closed principle

An example

Chapter 20. Compare with the final class texts in "AN OBJECT-ORIENTED ARCHITECTURE", 20.5, page 684.

The original STATE class appeared on page 686.

The following class extracts illustrate the general flavor of Simula. They are drawn from the solution to the problem of full-screen entry systems.

```
class STATE;
    virtual:
        procedure display;
        procedure read;
        boolean procedure correct;
    procedure message;
    procedure process;
begin
    ref (ANSWER) user_answer; integer choice;
    procedure execute; begin
            boolean ok;
        ok := false;
        while not ok do begin
            display; read; ok := correct;
            if not ok then message (a)
        end while;
        process;
    end execute
end STATE;
```

The original APPLICATION class appeared on page 690.

```
class APPLICATION (n, m);
        integer n, m;
begin
    ref (STATE) array transition (1:n, 0:m–1);
    ref (STATE) array associated_state (1:n);
    integer initial;
    procedure execute; begin
            integer st_number;
        st_number := initial;
        while st_number /= 0 do begin
                ref (STATE) st;
            st := associated_state (st_number); st.execute;
            st_number := transition (st_number, st.choice)
        end while
    end execute
    …
end APPLICATION
```

Coroutine concepts

Along with basic O-O mechanisms, Simula offers an interesting notion: coroutines.

The notion of coroutine was presented in the discussion of concurrency. Here is a brief reminder. Coroutines are modeled after parallel processes as they exist in operating systems or real-time software. A process has more conceptual autonomy than a routine; a printer driver, for example, is entirely responsible for what happens to the printer it manages. Besides being in charge of an abstract object, it has its own lifecycle algorithm, often conceptually infinite. The rough form of the printer process could be something like

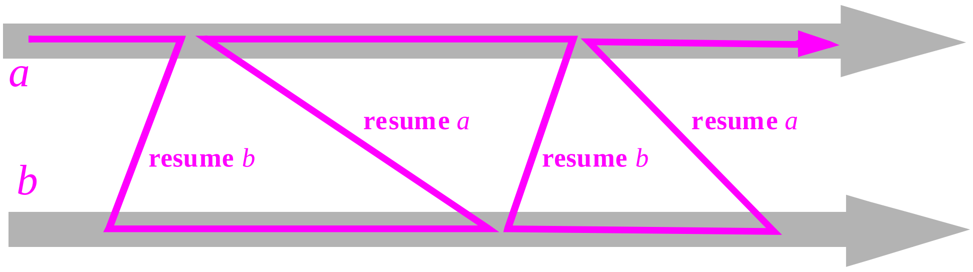
"Coroutines", page 1012.

```
from some_initialization loop forever
    "Obtain a file to be printed"; "Print it"
end
```

For a more complete description of a printer process see "Processes programmed", page 960.

In sequential programming, the relationship between program units is asymmetric: a program unit calls another, which will execute completely and return to the caller at the point of call. Communication between processes is more equal: each process pursues its own life, interrupting itself to provide information to, or get information from another.

Coroutines are similarly designed, but for execution on a single thread of control. (This sequential emulation of parallel execution is called *quasi-parallelism*.) A coroutine that "resumes" another interrupts its own execution and restarts its colleague at its last point of interruption; the interrupted coroutine may itself be later resumed.



Coroutine sequencing

(*This figure appeared originally on page 1012.*)

Coroutines are particularly useful when each of several related activities has its own logic; each may be described as a sequential process, and the master-slave relationship implied by routines is not adequate. A frequent example is an input-to-output transformation in which different constraints are placed on the structure of the input and output files. Such a case will be discussed below.

Simula represents coroutines as instances of classes. This is appropriate since coroutines almost always need persistent data, and often have an associated abstract object. As we noted earlier, a Simula class has a **body**, made of one or more instructions. In a class representing a passive data abstraction, it will only serve as initialization of the class instances (the equivalent of our creation procedure); but in a coroutine it will be the description of a process. The body of a coroutine is usually a loop of the form

```
while continuation_condition do begin
        … Actions…;
        resume other_coroutine;
        …Actions …
end
```

For some of the coroutines in a system the *continuation_condition* is often *True* to yield the equivalent of an infinite process (although at least one coroutine should terminate).

A system based on coroutines generally has a main program that first creates a number of coroutine objects, and then resumes one of them:

```
corout1 :– new C1; corout2 :– new C2; …
resume corout_i
```

On the parallel scheme see "A multi-launcher", page 988.

The evaluation of each **new** expression creates an object and starts executing its body. But the quasi-parallel nature of coroutines (as opposed to the true parallelism of processes) raises an initialization problem: with processes, each **new** would spawn off a new process and return control to the caller; but here only one coroutine may be active at any given time. If the **new** expression started the coroutine's main algorithm, the above main thread would never recapture control; for example it would never get a chance to create *C2* after spawning off *C1*.

Simula addresses this problem through the **detach** instruction. A coroutine may execute a **detach** to give control back to the unit that created it through a **new**. Coroutine bodies almost always begin (after initialization instructions if needed) with a **detach**, usually followed by a loop. After executing its **detach**, the coroutine will become suspended until the main program or another coroutine **resume**s it.

A coroutine example

Here is an illustration of the kind of situation in which coroutines may prove useful. You are requested to print a sequence of real numbers, given as input; but every eighth number (the eighth, the sixteenth, the twenty-fourth etc.) is to be omitted from the output. Furthermore, the output must appear as a sequence of lines, with six numbers per line (except for the last line if there are not enough numbers to fill it). So if i_n denotes the n-th input item, the output will start as

```
i1    i2    i3    i4    i5    i6
i_7   i_9   i_10  i_11  i_12  i_13
i_14  i_15  i_17  etc.
```

Finally, the output should only include the first 1000 numbers thus determined.

This problem is representative of coroutine use because it conceptually involves three processes, each with its specific logic: the input, where the constraint is to skip every eighth item; the output, where the constraint is to go to the next line after every sixth item; and the main program, which is required to process 1000 items. Traditional control structures are not good at combining such processes with widely different constraints. A coroutine solution, on the other hand, will work smoothly.

Following the preceding analysis, we may use three coroutines: the **producer** (input), the **printer** (output) and the **controller**. The general structure is:

```
begin
    class PRODUCER begin … See next … end PRODUCER;
    class PRINTER begin … See next … end PRINTER;
    class CONTROLLER begin … See next … end CONTROLLER;
    ref (PRODUCER) producer; ref (PRINTER) printer; ref (CONTROLLER) controller;
    producer :– new PRODUCER; printer :– new PRINTER; controller :– new CONTROLLER;
    resume controller
end
```

This is a main program, in the usual sense; it creates an instance of each of the three coroutine classes and **resume**s one of them, the controller. Here are the classes:

```
class CONTROLLER; begin
        integer i;
    detach;
    for i := 1 step 1 until 1000 do resume printer
end CONTROLLER;
class PRINTER; begin
        integer i;
    detach;
    while true do
        for i := 1 step 1 until 8 do begin
            resume producer;
            outreal (producer.last_input);
            resume controller
        end;
        next_line
    end
end PRINTER;
class PRODUCER; begin
        integer i; real last_input, discarded;
    detach;
    while true do begin
        for i := 1 step 1 until 6 do begin
            last_input := inreal; resume printer
        end;
        discarded := inreal
    end
end PRODUCER;
```

This scheme will not work if the program runs out of input before having printed 1000 output items. See exercise E35.1, page 1139.

Each class body begins with **detach** to allow the main program to proceed with the initialization of other coroutines. Procedure *outreal* prints a real number; function *inreal* reads and returns the next real on input; the extract assumes a procedure *next_line* that goes to the next line on input.

Coroutines fit well with the other concepts of object-oriented software construction. Note how decentralized the above scheme is: each process minds its own business, with limited interference from the others. The producer takes care of generating candidates from the input; the printer takes care of the output; the controller takes care of when to start and finish. As usual, a good check of the quality of the solution is the ease of extension and modification; it is indeed straightforward here to add a coroutine that will check for end of input (as requested by an exercise). Coroutines take decentralization, the hallmark of O-O architectures, one step further.

On the use of a concurrency mechanism to describe coroutines see "Coroutines", page 1012.

The architecture could be made even more decentralized. In particular, the processes in the above structure must still activate each other by name; ideally they should not have to know about each other except to communicate requested information (as when the printer obtains *last_input* from the producer). The simulation primitives studied below allow this; after that, the solution is to use a full concurrency mechanism, such as described in an earlier chapter. As you will remember, its platform-independence means that it will work for coroutines as well as true parallelism.

Sequencing and inheritance

"Synchronization for concurrent O-O computation", page 980.

Even if it does not use coroutine mechanisms (**detach**, **resume**), a Simula class may have a body (a sequence of instructions) in addition to its features, and so may take on the behavior of a process in addition to its usual role as an abstract data type implementation. When combined with inheritance, this property leads to a simpler version of what the discussion of concurrency called the *inheritance anomaly*, to which Simula, thanks to its limitation to single rather than multiple inheritance and coroutines rather than full parallelism, is able to provide a language solution.

For a class C let $body_C$ be the sequence of instructions declared as body of C and $actual\_body_C$ the sequence of instructions executed for every creation of an instance of C. If C has no parent, $actual\_body_C$ is just $body_C$. If C has a parent A (it can have at most one) then $actual\_body_C$ is by default the sequence of instructions

$actual\_body_A;\ body_C$

In other words, ancestors' bodies are executed in the order of inheritance. But this default may not be what you want. To supersede it, Simula offers the **inner** instruction which denotes the heir's body, so that the default policy is equivalent to having an **inner** at the end of the parent's body. If instead you write the body of A as

$instructions_1$; **inner**; $instructions_2$

then (assuming A itself has no parent) the execution of C will execute not its $body_C$ as written in the class but its $actual\_body_C$ defined as

$instructions_1;\ body_C;\ instructions_2$

Although the reasons for this facility are clear, the convention is rather awkward:

- In many cases descendants would need to create their instances differently from their ancestors. (Remember *POLYGON* and *RECTANGLE*.)
- Bodies of descendants, such as *C* here, become hard to understand: just reading $body_C$ does not really tell you what the execution will do.
- In addition, of course, the convention would not transpose easily to multiple inheritance, although this is not an immediate concern in Simula.

Such difficulties with **inner** are typical of the consequences of making objects active, as we found out when discussing concurrency.

"Active objects clash with inheritance", page 959.

Almost all object-oriented languages after Simula have departed from the **inner** convention and treated object initialization as a procedure.

Simulation

True to its origins, Simula includes a set of primitives for discrete-event simulation. It is no accident, of course, that the first O-O language was initially meant for simulation applications; more than in any other area, this is where the modeling power of the object-oriented method can illustrate itself.

A **simulation** software system analyzes and predicts the behavior of some external system — an assembly line, a chemical reaction, a computer operating system, a ship…

A **discrete-event simulation** software system simulates such an external system as having, at any time, a *state* that can change in response to *events* occurring at discrete instants. This differs from **continuous** simulation, which views the state as continuously evolving. Which of these two modeling techniques is best for a given external system depends not so much on whether the system is inherently continuous or discrete (often a meaningless question) as on what models we are able to devise for it.

Another competitor to discrete-event simulation is **analytical** modeling, whereby you simply build a mathematical model of the external system, then solve the equations. This is a very different approach. With discrete-event simulation, you run a software system whose behavior simulates the behavior of the external system: to get more significant results, you will increase the length of the period that you simulate in the external system's life, and so you will run the simulation longer. This is why analytical models are usually more efficient. But many physical systems are too complex to admit realistic yet tractable mathematical models; then simulation is the only possibility.

Many external systems lend themselves naturally to discrete event simulation. An example is an assembly line, where typical events may include a new part being entered into the line, a worker or machine performing a certain operation on one or more parts, a finished product being removed from the line, a failure causing the line to stop. You may use the simulation to answer questions about the modeled physical systems: how long does it take (average, minimum, maximum, standard deviation) to produce a finished product? How long will a given piece of machinery remain unused? What is the optimum inventory level? How long does it take to recover from a power failure?

The input to a simulation is a sequence of events with their occurrence times. It may come from measurements on the external systems (when the simulation is used to reconstruct and analyze past phenomena, for example a system failure); more commonly, it is produced by random number generators according to some chosen statistical laws.

A discrete-event model must keep track of external system time, also called **simulated time**, representing the time taken by external system operations such as performing a certain task on a certain part, or the instants at which certain events such as equipment failure will occur. Simulated time should not be confused with the **computing time** needed to execute the simulation system. For the simulation system, simulated time is simply a non-negative real variable, which the simulation program may only increase by discrete leaps. It is available in Simula through the query *time*, managed by the run-time system and modifiable through some of the procedures seen next.

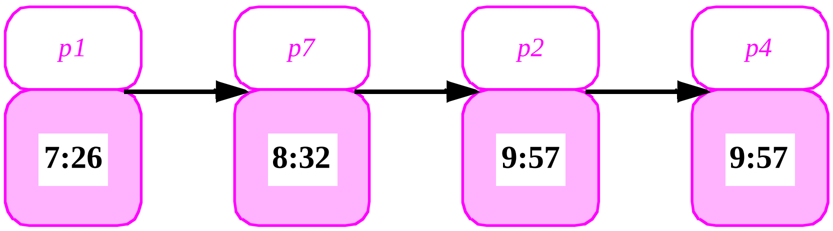
Feature *time* and other simulation-specific features come from a library class *SIMULATION*, which may be used as parent by another class. Let us call "simulation class" any class that is a descendant of *SIMULATION*.

In Simula, you may also apply inheritance to blocks: a block written under the form *C* **begin** … **end** has access to all the features declared in class *C*. *SIMULATION* is often used in this way as parent of a complete program rather than just a class. So we can also talk of a "simulation program".

First, *SIMULATION* contains the declaration of a class *PROCESS*. (As noted earlier, Simula class declarations may be nested.) An instance of *PROCESS* represents a process of the external system. A simulation class can declare descendants of *PROCESS*, which we will call "process classes", and their instances just "processes". Among other properties, a process may be linked to other processes in a linked list (which means that *PROCESS* is a descendant of the Simula equivalent of class *LINKABLE*). A process may be in one of the following four states:

- **Active**, or currently executing.
- **Suspended**, or waiting to be resumed.
- **Idle**, or not part of the system.
- **Terminated**.

Any simulation (that is to say, any instance of a descendant of *SIMULATION*) maintains an **event list**, containing **event notices**. Each event notice is a pair <*process*, *activation_time*>, where *activation_time* indicates when the *process* must be activated. (Here and in the rest of this section any mention of time, as well as words such as "when" or "currently", refer to simulated time: the external system's time, as available through *time*.) The event list is sorted by increasing *activation_time*; the first process is active, all others are suspended. Non-terminated processes which are not in the list are idle.



An event list

The basic operation on processes is activation, which schedules a process to become active at a certain time by inserting an event notice into the event list. Apparently for syntactical reasons, this operation is not a call to a procedure of class *SIMULATION*, but a specific instruction using the keyword **activate** or **reactivate**. (A procedure call would seem to be a more consistent approach; in fact the standard defines the semantics of *activate* through a fictitious procedure text.) The basic form of the instruction is

activate *some_process scheduling_clause*

where *some_process* is a non-void entity of type conforming to *PROCESS*. The optional *scheduling_clause* is of one of

at *some_time*

delay *some_period*

before *another_process*

after *another_process*

The first two forms specify the position of the new event notice by its activation time (the sorting criterion for the event list); the new activation time is *max* (*time*, *some_time*) in the **at** form and *max* (*time*, *time* + *some_period*) in the **delay** form. The new event notice will be inserted after any other already present in the list with the same activation time, unless you specify **prior**. The last two forms specify the position with reference to another process in the list. A missing *scheduling_clause* is equivalent to **delay** *0*.

A process may activate itself at a later time by specifying itself as the target process *some_process*. In this case the keyword should be **reactivate**. This is useful to represent an external system task that takes some simulated time — but of course no computer time. So if you want to simulate a task that a worker takes three minutes (180 seconds) to perform, you can let the corresponding process *worker* execute the instruction

reactivate *worker* **delay** *180*

This case is so common as to justify a special syntax, avoiding explicit self-reference:

hold (*180*)

with exactly the same effect.

Procedure hold is part of the SIMULATION class.

As you may have guessed, processes are implemented as coroutines; the simulation primitives internally use the coroutine primitives that we have reviewed. The effect of *hold* (*some_period*), for example, may be approximately described (in syntax similar to the notation of this book but extended with **resume**) as

```
        -- Insert new event notice into event list at position determined by its time:
    my_new_time := max (time, time + some_period)
    !! my_reactivation_notice.make (Current, my_new_time)
    event_list.put (my_reactivation_notice)

        -- Get first element of event list and remove it:
    next := event_list.first; event_list.remove_first

        -- Activate chosen process, advancing time if necessary:
    time := time. max (next.when); resume next.what
```

assuming the following declarations:

```
    my_new_time: REAL; my_reactivation_notice, next: EVENT_NOTICE
    class EVENT_NOTICE creation make feature
        when: REAL -- i.e. time
        what: PROCESS
        make (t: REAL; p: PROCESS) is
            do when := t; what := p end
    end
```

If a process becomes suspended by reactivating itself at a later time, execution will resume the first suspended process (the one with the earliest reactivation time) and, if its reactivation time is after the current time, correspondingly advance the current time.

Exercise E35.2, page 1139.

As this example shows, the simulation primitives, although based on the coroutine primitives, belong to a higher level of abstraction; whenever possible it is preferable to use them rather than relying directly on coroutine mechanisms. In particular you may view *hold (0)* as a form of **resume** through which you let the underlying event list mechanism pick the process to be resumed, rather than specifying it explicitly.

A simulation example

Process classes and the simulation primitives provide an elegant mechanism for modeling external-world processes. Consider as an illustration a worker who may be asked to do either one of two tasks. Both may take a variable amount of time; the second requires switching on a machine *m*, which takes 5 minutes, and waiting for the machine to do its job.

The Simula notation **this** *C, used within a class C, is the equivalent of Current as used in the rest of this book.*

```
    PROCESS class WORKER begin
        while true do begin
            "Get next task type i and task duration d";
            if i = 1 then
                activate m delay 300; reactivate this WORKER after m;
            end;
            hold (d)
        end while
    end WORKER
```

The operation "get next task type and task duration" will usually obtain the requested value from a pseudo-random number generator, using a specified statistical distribution. The Simula library includes a number of generators for common statistical laws. The type of *m* is assumed to be some process class *MACHINE* representing the behavior of machines. All actors of a simulation will be similarly represented by process classes.

Simula: an assessment

Like Algol 60 before it, Simula has made its contribution less by its commercial success than through its intellectual influence. The latter is everywhere; both in theory (abstract data types) and in practice, most of the developments of the past twenty years are children or grandchildren of the Simula ideas. As to the lack of widespread commercial success, a number of reasons can be invoked, but the most important one by far is as regrettable as it is obvious: like a few major inventions before it, Simula came too soon. Although a significant community immediately recognized the potential value of the ideas, the software field as a whole was not ready.

Thirty years later, as should be clear from the preceding overview, many of these ideas are as timely as ever.

35.2 SMALLTALK

The ideas for Smalltalk were laid out around 1970 at the University of Utah by Alan Kay, then a graduate student and part of a group that was particularly active in graphics, when he was asked to look at an Algol 60 compiler that had just been delivered to the department from Norway. Poring over it, he realized that the compiler actually went beyond Algol and implemented a set of notions that seemed directly relevant to Kay's other work. The supported Algol extension was, of course, Simula. When Kay later joined the Xerox Palo Alto Research Center (PARC), he used the same principles as the basis for his vision of an advanced personal computing environment. The other two principal contributors to the early development of Smalltalk at Xerox PARC were Adele Goldberg and Daniel Ingalls.

Smalltalk-72 evolved into Smalltalk-76, then Smalltalk-80, and versions were developed for a number of machines — initially Xerox hardware but later industry-standard platforms. Today Smalltalk implementations are available from several sources.

Language style

As a language, Smalltalk combines the influence of Simula with the free, typeless style of Lisp. The emphasis is on dynamic binding. No type checking is performed: in contrast with the approach emphasized in this book, the determination of whether a routine may be applied to an object only occurs at run time.

This, by the way, is not the standard Smalltalk terminology. A routine is called a "method" in Smalltalk; applying a routine to an object is called "sending a message" to the object (whose class must find the appropriate method to handle the message).

Another important feature that distinguishes the Smalltalk style from what we have studied in this book is the lack of a clear-cut distinction between classes and objects. Everything in the Smalltalk system is an object, including the classes themselves. A class is viewed as an instance of a higher-level class called a metaclass. This allows the class hierarchy to encompass all elements in the system; at the root of the hierarchy is the highest-level class, called *object*. The root of the subtree containing only classes is the metaclass *class*. The arguments for this approach include:

- Consistency: everything in Smalltalk follows from a single concept, object.
- Environment effectiveness: making classes part of the run-time context facilitates the development of symbolic debuggers, browsers and other tools that need run-time access to class texts
- Class methods: it is possible to define methods that apply to the class rather than to its instances. Class methods may be used to provide special implementations for standard operations like **new** which allocates instances of the class.

"Metaclasses", page 168.

An earlier discussion considered the arguments for other, more static approaches, showing different ways to obtain the same results.

Messages

Smalltalk defines three main forms of messages (and associated methods): unary, keyword and binary. **Unary** messages express calls to routines without parameters, as in

acc1 balance

which sends the message *balance* to the object associated with *acc1*. This is equivalent to the notation *acc1*.*balance* used in Simula and this book. Messages may, as here, return values. **Keyword** messages represent calls to routines with arguments, as in

point1 translateBy: *vector1*
window1 moveHor: *5 Vert*: *–3*

The use of upper-case letters in the middle of a word, giving identifiers such as *translateBy*, is part of the established Smalltalk style. Note how the message name is collapsed with the keyword for the first argument. The corresponding syntax in Simula or our notation would have been *point1*.*translate* (*vector1*) and *window1*.*move* (*5*, *–3*).

Binary messages, similar to the infix functions of Ada and the notation of this book, serve to reconcile the "everything is an object" approach with more traditional arithmetic notations. Rather than

2 addMeTo: *3*

most people, at least from the older generations who learned arithmetic before object technology, still prefer to write *2*+*3*. Smalltalk's binary messages permits this latter form as essentially a synonym for the former. There is a snag, however: precedence. The expression $a + b \ast c$ means $(a + b) \ast c$. Smalltalk developers can use parentheses to re-establish standard precedence. Unary messages take precedence over binary messages, so that *window1 height* + *window2 height* has the expected meaning.

In contrast with Simula and the language of this book, Smalltalk classes may only export methods (routines). To export an attribute, you must write a function that gives access to its value. A typical example is

```
x | |
    ↑ xx

y | |
    ↑ yy

scale: scaleFactor | |
    xx <– xx * scaleFactor
    yy <– yy * scaleFactor
```

Methods *x* and *y* return the values of the instance variables (attributes) *xx* and *yy*. The up arrow ↑ means that the following expression is the value to be returned by the method to the sender of the corresponding message. Method *scale* takes an argument, *scaleFactor*. The vertical bars | | would delimit local variables if there were any.

Inheritance is an important part of the Smalltalk approach, but except for some experimental implementations it is limited to single inheritance. To enable a redefined method to call the original version, Smalltalk allows the developer to refer to the object viewed as an instance of the parent class through the name **super**, as in

```
aFunction: anArgument |…|
    … super aFunction: anArgument …
```

It is interesting to compare this approach with the techniques based on *Precursor* and repeated inheritance.

Exercise E35.5, page 1140. See "Keeping the original version of a redefined feature", page 555.

All binding is dynamic. In the absence of static typing, errors resulting from sending a message to an object that is not equipped with a proper method to handle it will cause run-time failure, rather than being caught by a compiler.

Dynamic typing also renders irrelevant some of the concepts developed earlier in this book: Smalltalk does not need language support for genericity since a generic structure such as a stack may contain elements of any type without any static coherence checks; neither are deferred routines meaningful, since if the software includes a call *x f* (the equivalent of *x*.*f*) there is no static rule requiring any particular class to provide a method *f*. Smalltalk provides, however, a run-time mechanism to raise an error if a class *C* receives a message corresponding to a method whose effective definitions only appear in proper descendants of *C*. (In the rest of this book, *C* would be a deferred class, and instances would only be created for non-deferred descendants of *C*.) For example, we could implement *rotate* in a class *FIGURE* by

```
rotate: anAngle around: aPoint | |
    self shouldNotImplement
```

The method *shouldNotImplement* is included in the general class *object* and returns an error message. The notation *self* denotes the current object.

Environment and performance

Much of Smalltalk's appeal has come from the supporting programming environments, among the first to include innovative interaction techniques (many of them devised by other Xerox PARC projects around the time of the original Smalltalk development) which have now become commonplace: multiple windows, icons, integration of text and graphics, pull-down menus and use of the mouse as a pointing and selecting device. Such staples of current O-O environment tools such as browsers, inspectors and O-O debuggers trace some of their roots to Smalltalk environments.

As with Simula, all commercial implementations support garbage collection. Smalltalk-80 and subsequent implementations are also renowned from their libraries of basic classes, covering important abstractions such as "collections" and "dictionaries", and a number of graphical concepts.

The lack of static typing has proved a formidable obstacle to the efficiency of software systems developed in Smalltalk. Although modern Smalltalk environments, no longer solely interpretative, provide some mechanisms for compiling methods, the unpredictability of run-time target types deprives most Smalltalk developers of a number of crucial optimizations that are readily available to compilers for statically typed languages (such as setting up arrays of functions references and hence ensuring constant-time resolution of dynamic binding, as discussed in the chapter on inheritance). Not surprisingly, many Smalltalk projects have reported efficiency problems. In fact, the common misconception that object technology carries a performance penalty can be attributed in part to experience with Smalltalk environments.

Smalltalk: an assessment

Smalltalk was instrumental in associating interactive techniques with the concepts of object technology, turning the abstract objects of Simula into visual objects that became suddenly comprehensible and appealing to a larger audience. Simula had impressed programming language and programming methodology experts; Smalltalk, through the famous August 1981 issue of *Byte*, dazzled the masses.

Considering how dated the concepts of Smalltalk appear today, the commercial success that it enjoyed in the early nineties is remarkable. It can be partly attributed to two independent *a contrario* phenomena:

- The "try the next one on the list" effect. Many people who were initially drawn to object technology by the elegance of the concepts were disappointed with hybrid approaches such as C++. When looking for a better embodiment of the concepts, they often went to the approach that the computer press has consistently presented as *the* pure O-O approach: Smalltalk. Many a Smalltalk developer is indeed someone who "just says no" to C or C-like development.

- The decline of Lisp. For a long time, many companies relied on Lisp variants (along with Prolog and a few other approaches grounded in Artificial Intelligence) for side projects involving quick development of prototypes and experiments. Starting in the mid-eighties, however, Lisp largely faded from the scene; Smalltalk naturally occupied the resulting vacuum.

The last observation provides a good idea of the scope of the Smalltalk approach. Smalltalk is an excellent tool for prototyping and experimentation, especially when visual interfaces are involved (it competes in this area with more recent tools such as Borland's Delphi or Microsoft's Visual Basic). But it has largely remained uninfluenced by later developments in software engineering methodology, as attested by the absence of static typing, assertion mechanisms, disciplined exception handling, deferred classes, all of which are important for mission-critical systems — or simply any system whose proper run-time behavior is important to the organization that has developed it. The performance problems noted above do not help.

The lesson is clear: it would not in my opinion be reasonable today for a company to entrust a significant production development to Smalltalk.

35.3 LISP EXTENSIONS

Like many other pre-O-O languages, Lisp has served as the basis for several object-oriented extensions; in fact many of the earliest O-O languages after Simula and Smalltalk were Lisp-based or Lisp-like. This is not surprising, since Lisp and its implementations have for many years offered mechanisms that directly help the implementation of object-oriented concepts, and have taken much longer to find their way into mainstream languages and their environments:

- A highly dynamic approach to the creation of objects.
- Automatic memory management with garbage collection.
- Ready implementation of tree-like data structures.
- Rich development environments, such as Interlisp in the seventies and its predecessors in the previous decade.
- Run-time selection of operations, facilitating the implementation of dynamic binding.

The conceptual distance to O-O concepts is, then, shorter if you start from Lisp than if you start from C, Pascal or Ada, so that the term "hybrid" commonly used for O-O extensions of these languages, such as the C-based hybrids which we will review in the next sections, is less appropriate for extensions of Lisp.

Artificial Intelligence applications, the prime application of Lisp and Lisp-like languages, have found in O-O concepts the benefits of flexibility and scalability. They have taken advantage of Lisp's uniform representation for programs and data to extend the object-oriented paradigm with notions such as "meta-object protocol" and "computational reflection" which apply some of the O-O principles not just to the description of run-time structures (objects) but also to the software structure itself (classes), generalizing the Smalltalk concept of metaclass and continuing the Lisp tradition of self-modifying software. For most developers, however, these concepts are a little far-off, and they do not blend too well with the software engineering emphasis on a strict separation between the static and dynamic pictures.

Three main contenders were vying for attention in the world of O-O Lisp in the eighties: *Loops*, developed at Xerox, initially for the Interlisp environment; *Flavors*, developed at MIT, available on several Lisp-oriented architectures; *Ceyx*, developed at INRIA. Loops introduced the interesting concept of "data-oriented programming", whereby you may attach a routine to a data item (such as an attribute). Execution of the routine will be triggered not only by an explicit call, but also whenever the item is accessed or modified. This opens the way to event-driven computation, a further step towards decentralizing software architectures.

The unification of the various approaches came with the Common Lisp Object System or CLOS (pronounced C-Los by most people), an extension of Common Lisp which was the first object-oriented language to have an ANSI standard.

35.4 C EXTENSIONS

Much of the late nineteen-eighties transformation of object technology from an attractive idea into an industrial practice can be attributed to the emergence and tremendous commercial success of languages that added object-oriented extensions to the stable stem of a widely available non-O-O language, C. The first such effort to attract widespread attention was Objective-C; the best known today is C++.

The language styles reflect two radically different approaches to the problem of "hybrid" language design, so called because it combines O-O mechanisms with those of a language based on entirely different principles. (Examples of hybrids based on languages other than C include Ada 95 and Borland Pascal.) Objective-C illustrates the *orthogonal* approach: add an O-O layer to the existing language, keeping the two parts as independent as possible. C++ illustrates the *merged* approach, intertwining concepts from both. The potential advantages of each style are clear: the orthogonal approach should make the transition easier, avoiding unexpected interferences; the merged approach should lead to a more consistent language.

Both efforts capitalized on the success of C, which had rapidly become one of the dominant languages in the industry. The appeal to managers was obvious, based on the prospect of turning C programmers into O-O developers without too much of a culture shock. The model (evoked by Brad Cox) was that of the C and Fortran preprocessors such as Ratfor which, in the seventies, enabled part of the software community to become familiar with concepts of "structured programming" while continuing to work in familiar language frameworks.

Objective-C

Designed at Stepstone Corporation (originally Productivity Products International) by Brad Cox, Objective-C is a largely orthogonal addition of Smalltalk concepts onto a C base. It was the base language for the NEXTSTEP workstation and operating system. Although obscured in part by the success of C++, Objective-C has retained an active user community.

As in Smalltalk, the emphasis is on polymorphism and dynamic binding, but current versions of Objective-C have departed from the Smalltalk model by offering static typing as an option (and for some of them, somewhat surprisingly, static *binding* as well). Here is an example of Objective-C syntax:

```
= Proceedings: Publication {id date, place; id articles;}
        + new {return [[super new] initialize]}
        – initialize {articles = [OrderedCollection new]; return self;}
        – add: anArticle {return [contents add: anArticle];}
        – remove: anArticle {return [contents remove:anArticle];}
        – (int) size {return [contents size];}
=:
```

Class *Proceedings* is defined as heir to *Publication* (Objective-C supports single inheritance only). The braces introduce attributes ("instance variables"). The next lines describe routines; *self*, as in Smalltalk, denotes the current instance. The name *id* denotes, in the non-statically typed variant, a general class type for all non-C objects. Routines introduced by +, known as "class methods" as in Smalltalk, are meant for the class; this is the case here with the creation operation *new*. Others, introduced by –, are normal "object methods" that send messages to instances of the class.

Stepstone's Objective-C is equipped with a library of classes initially patterned after their Smalltalk counterparts. Many other classes are also available for NEXTSTEP.

C++

Originally designed by Bjarne Stroustrup at AT&T Bell Laboratories (an organization previously renowned, among other accomplishments, for its development of Unix and C), C++ quickly gained, starting around 1986, a leading position for industrial developments aiming to obtain some of the benefits of object technology while retaining compatibility with C. The language has remained almost fully *upward-compatible* with C (meaning that a valid C program is also, in normal circumstances, a valid C++ program).

Early C++ implementations were simple preprocessors that removed O-O constructs to yield plain C, based on techniques sketched in the preceding chapter. Today's compilers, however, are native C++ implementations; it has in fact become hard to find a C compiler that is not *also* a C++ compiler, requiring the user who just wants a basic C compiler to turn on a special "no C++ constructs" compilation option. This is a measure among many of the success of the approach. Compilers are available from many sources and for many platforms.

Originally, C++ was an attempt at providing a better version of C, improved in particular through a class construct and a stronger form of typing. Here is a class example:

```
class POINT {
        float xx, yy;
        public:
                void translate (float, float);
                void rotate (float);
                float x ();
                float y ();

                friend void p_translate (POINT *, float, float);
                friend void p_rotate (POINT *, float);
                friend float p_x (POINT *);
                friend float p_y (POINT *);
        };
```

The first four routines are the normal, object-oriented interface of the class. As shown by this example, the class declaration only shows the headers of these routines, not their implementations (somewhat as in the output of the **short** command studied in earlier chapters). The routine implementations must be defined separately, which raises questions of scope for both compilers and human readers.

The other four routines are examples of "friend" routines. This notion is peculiar to C++ and makes it possible to call C++ routines from normal C code. Friend routines will need an extra argument representing the object to which an operation is applied; this argument is here of type *POINT* ∗, meaning pointer to *POINT*.

C++ offers a rich set of powerful mechanisms:

- Information hiding, including the ability to hide features from proper descendants.
- Support for inheritance. Original versions supported single inheritance only, but now the language has multiple inheritance. Repeated inheritance lacks the flexibility of sharing or replicating on a feature-by-feature basis, which from the discussion of these topics seemed quite important. Instead, you share or duplicate an entire feature set from the repeated ancestor.

"The C++ approach to binding", page 514.

- Static binding by default, but dynamic binding for functions specified as virtual; the C++ approach to this issue was discussed in depth in an earlier chapter.
- A notion of "pure virtual function", which resembles deferred features.
- Stricter typing than in traditional C, but still with the possibility of casting.
- Usually no garbage collection (because of the presence of casts and the use of pointers for arrays and similar structures), although some tools are available for suitably restrained programs.
- Because of the absence of automatic memory management by default, a notion of *destructor* for taking care of object disposal (complementing the *constructors* of a class, that is to say its creation procedures).

- Exception handling, again not part of the original definition but now supported by most compilers.
- A form of assignment attempt, "downcasting".
- A form of genericity, "templates", which suffers from two limitations: no constrained genericity; and, for reasons unclear to a non-implementer, a considerable burden on compile-time performance (known in the C++ literature as the *template instantiation problem*).

See "Efficiency considerations", page 327.

- Operator overloading.
- An *assert* instruction for debugging, but no assertions in the sense of support for Design by Contract (preconditions, postconditions, class invariants) tied to O-O constructs.
- Libraries available from various suppliers, such as the Microsoft Foundation Classes.

Complexity

The size of C++ has grown considerably since the language's first versions, and many people have complained about its complexity. That they have a point is illustrated, among many possible examples, by this little excerpt from a pedagogical article by a recognized C and C++ authority, chair of the C standards committee of the American National Standards Institute and author of several respected C++ books as well as the *Dictionary of Standard C*, from whom I was at some point hoping to learn the difference between the C++ notions of reference and pointer:

> *While a reference is somewhat like a pointer, a pointer is an object that occupies memory and has an address. Non-***const** *pointers can also be made to point to different objects at run time. On the other hand, a reference is an alias to an object and does not, itself, occupy any memory. Its address and value are the address and value of the object to which it is aliased. And while you can have a reference to a pointer, you cannot have a pointer to a reference or an array of references, nor can you have an object of some reference type. References to the* **void** *type are also prohibited.*
>
> *References and pointer are not interchangeable. A reference to an* **int** *cannot, for example, be assigned to a pointer to an* **int** *or vice versa. However, a reference to a pointer to an* **int** *can be assigned a pointer to an* **int**.

From Rex Jaeschke's C++ column in DEC Professional, *November 1991.*

I swear I tried to understand. I was almost convinced I got the hang of it, although perhaps not being quite ready for the midterm exam yet. ("Give convincing examples of cases in which it is appropriate to use: (1) A pointer only. (2) A reference only. (3) Either. (4) Neither. No notes or Web browsers allowed".) Then I noticed I had missed the start of the next paragraph:

> *From what we have seen so far, it may not be obvious as to why references indeed exist.*

Oh well. Proponents of C++ would undoubtedly state that most users can ignore such subtleties. Another school holds that a programming language, the principal tool of software developers, should be based on a reasonable number of solid, powerful, perfectly understood concepts; in other words, that every serious user should know *all* of the language, and trust all of it. But it may be impossible to reconcile this view with the very idea of hybrid language.

C++: an assessment

Booch interview: http://www.geekchic.com/replique.htm. Knuth interview: Dr. Dobb's Journal, no. 246, April 1996, pages 16-22.

C++ leaves few people indifferent. The eminent author Grady Booch lists it, in a "Geek Chic" interview, as his programming language of choice. Then, according to Donald Knuth, it would make Edsger Dijkstra "*physically ill to think of programming in C++*".

C++ here could use the answer of Junia to Nero in Racine's *Britannicus*:

I have neither deserved, in all humility,
Such excess of honor, nor such indignity.

Disappointment with C++ indeed follows from exaggerated hopes. Earlier discussions in this book have carefully analyzed some of the language's more controversial design choices — especially in the areas of typing, memory management, inheritance conventions and dynamic binding — and shown that better solutions are available. But one cannot criticize C++ as if it were the be-all and end-all of object-oriented languages. What C++ has attempted, and achieved beyond anyone's dreams, was to catch a particular moment in the history of software: the time at which a large part of the profession and its managers were ready to try object technology, but *not* ready to shed their current practices. C++ was the almost magical answer: still C enough not to scare the managers; already O-O enough to attract the forward-looking members of the trade. In seizing the circumstance, C++ was only following the example of C itself, which, fifteen years earlier, was another product of coinciding opportunities — the need for a portable machine-oriented language, the development of Unix, the emergence of personal computers, and the availability of a few decommissioned machines at Bell Labs. The merits of C++ lie in the historic boost it gave to the development of object technology, making it presentable to a whole community that might not have accepted the ideas under a less conventional apparel.

That C++ is not the ideal object-oriented language, a comment regularly made by authors and lecturers in the field, and obvious enough to anyone who has studied the concepts, should not obscure this contribution. We must not indeed look at C++ as if it were destined to remain a major tool for the software engineering community well into the twenty-first century, as it would then be overstaying its welcome. In the meantime C++ has admirably played its role: that of a transition technology.

35.5 JAVA

Introduced by a Sun Microsystems team, Java gained considerable attention in the first few months of 1996, presented as the way to help tame the Internet. According to *ComputerWorld*, the number of press mentions of Java in the first six months of 1996 was 4325 (which we may multiply by 2 or 3 since this was presumably the US press only); as a point of comparison, Bill Gates was mentioned only 5096 times.

ComputerWorld, vol. 30, no. 29, 15 July 1996, page 122.

The principal contribution of Java is in implementation technology. Building on ideas already present in many other O-O environments but taken here to a new level, Java execution rests on a **bytecode** (a low-level, portable interpretable format) whose specification is in the public domain, and a widely available **virtual machine** to interpret bytecode programs. The virtual machine is simply a program, for which versions are available for many different platforms, and can be downloaded freely through the Internet; this enables almost anyone to execute bytecode programs produced by almost anyone else. Often you do not even have to download anything explicitly: the virtual machine is built in tools such as Web browsers; and such tools will be able to recognize references to a bytecode program, for example a reference embedded in a link on a Web page, so that they will then automatically download the program and execute it on the spot.

See "Remote execution", page 955.

The explosion of the Internet has given this technology a great momentum, and Sun has been able to convince many other major players to produce tools based on this technology. As the bytecode is largely separate from the language, it stands a good chance of becoming a medium of choice for compiler output, regardless of what the source language is. Compiler writers for such notations as O-O extensions of Pascal and Ada, as well as the notation of this book, have not been slow to recognize the opportunity for developing software that will run without any change, and without even the need to recompile, across all industry platforms.

Java is one of the most innovative developments in the software field, and there are many reasons to be excited about it. Java's language is not the main one. As an O-O extension of C, it has missed some of the lessons learned since 1985 by the C++ community; as in the very first version of C++, there is no genericity and only single inheritance is supported. Correcting these early oversights in C++ was a long and painful process, creating years of havoc as compilers never quite supported the same language, books never quite gave accurate information, trainers never quite taught the right stuff, and programmers never quite knew what to think.

Just as everyone in the C++ world has finally come up to speed, Java is starting along the same road. The language does have one significant benefit over C++: by removing the notion of arbitrary pointer, especially to describe arrays, it has finally made it possible to support garbage collection. For the rest, it seems to take no account of modern software engineering ideas: no assertion support (in fact, Java went so far as to remove the modest **assert** instruction of C and C++); partial reliance on run-time type checking; a confusing modular structure with three interacting concepts (classes, nested packages, source files); and ever the cryptic syntax bequeathed from C, with such lines as the following typical examples from the designers' book on the language:

From [Arnold 1996]

```
String [] labels = (depth == 0 ? basic : extended);
while ((name = getNextPlayer()) != null) {
```

exhibiting side-effect-producing functions as a way of life, use of = conflicting with the tradition of mathematics, semicolons sometimes required and sometimes illegal etc.

See "Formats for reusable component distribution", page 79.

That the language is uninspiring should not, however, detract from the contribution that Java technology has already made to portable software development. If it can eventually solve its current efficiency problems, Java could, through its bytecode, become the closest approximation (built from software rather than hardware, although "Java chips" have also been announced) to one of the oldest dreams of the computer industry: a truly universal machine.

35.6 OTHER O-O LANGUAGES

The languages reviewed so far are some of the best known, but by no means the only ones to have attracted significant attention. Here are a few other important contributions, which would each deserve a separate chapter in a book entirely devoted to object-oriented languages, and to which you can find references (books and Web pages) in the bibliographical section:

- *Oberon* is Niklaus Wirth's O-O successor to Modula-2, part of a more general project which also involves a programming environment and even hardware support.
- *Modula-3*, originally from Digital Equipment's research laboratory, is another modular language with class-like record types, also starting from Modula-2.
- *Trellis*, also from DEC Research, was among the first to offer both genericity and multiple inheritance.
- *Sather*, drawing in part from the concepts and notation of the first edition of this book, especially assertions, has the benefit of a public-domain implementation; its *pSather* version provides an interesting concurrency mechanism.
- *Beta* is a direct descendant of Simula, designed in Scandinavia with the collaboration of Kristen Nygaard (one of Simula's original authors). It introduces the *pattern* construct to unify the concepts of class, procedure, function, type and coroutine.
- *Self* is based not on classes but on "prototypes", supporting inheritance as a relation between objects rather than types.

"FROM ADA TO ADA 95", 33.7, page 1092; "Object-oriented extensions of Pascal", page 1101.

- *Ada 95* was discussed in the Ada chapter.
- *Borland Pascal* and other O-O extensions of Pascal were cited in the discussion of Pascal.

35.7 BIBLIOGRAPHICAL NOTES

Simula

[Dahl 1966] describes an initial version of Simula subsequently known as Simula 1. The current Simula, long known as Simula 67, was initially described by [Dahl 1970], which assumed Algol 60 as a basis and only described the Simula extensions. A chapter in the famous *Structured Programming* book of Dahl, Dijkstra and Hoare [Dahl 1972] brought the concepts to a wider audience. The language description was revised in 1984, incorporating the Algol 60 elements. The official reference is the Swedish national standard [SIS 1987]. For an account of Simula's history by its designers, see [Nygaard 1981].

The best known book on Simula is [Birtwistle 1973]. It remains an excellent introduction. A more recent text is [Pooley 1986].

Smalltalk

References on the earliest versions of Smalltalk (-72 and -76) are [Goldberg 1976] and [Ingalls 1978].

A special issue that *Byte* devoted to Smalltalk [Goldberg 1981] was the key event that brought Smalltalk to prominence long before supporting environments became widely available. The basic reference on the language is [Goldberg 1983], serving both as pedagogical description and reference; complementing it is [Goldberg 1985], which describes the programming environment.

For a good recent introduction to both the Smalltalk language and the VisualWorks environment see [Hopkins 1995]; for an in-depth treatment see Lalonde's and Pugh's two-volume set [Lalonde 1990-1991].

The story of Simula's original influence on Smalltalk (the "Algol compiler from Norway") comes from an interview of Alan Kay in *TWA Ambassador* (yes, an airline magazine), exact issue number forgotten — early or mid-eighties. I am indebted to Bob Marcus for pointing out the connection between Lisp's decline and Smalltalk's resurgence.

C extensions: Objective-C, C++

Objective-C is described by its designer in an article [Cox 1984] and a book [Cox 1990] (whose first edition dates back to 1986). Pinson and Wiener have written an introduction to O-O concepts based on Objective-C [Pinson 1991].

There are hundreds of books on C++. For a personal account of the language's history by its designer, see [Stroustrup 1994]. The original article was [Stroustrup 1984]; it was extended into a book [Stroustrup 1986], later revised as [Stroustrup 1991], which contains many tutorial examples and useful background. The reference manual is [Ellis 1990].

Ian Joyner has published several editions of an in-depth “C++ critique” [Joyner 1996] available on a number of Internet sites and containing detailed comparisons with other O-O languages.

Lisp extensions

Loops: [Bobrow 1982]; Flavors: [Cannon 1980], [Moon 1986]; Ceyx: [Hullot 1984]; CLOS: [Paepcke 1993].

Java

In the few months that followed the release of Java, many books have appeared on the topic. Those by the designing team include: [Arnold 1996] for a language tutorial, [Gosling 1996] as the language reference, and [Gosling 1996a] about the basic libraries.

The address shown is for the first message in the discussion; from there you can follow links to the rest of the

A discussion about Java’s lack of assertions in the style of this book (that is to say, supporting the principles of Design by Contract), conducted on Usenet in August 1995, appears at *http://java.sun.com/archives/java-interest/0992.html*.

Other languages

Oberon: [Wirth 1992], [Oberon-Web]. Modula-3: [Harbison 1992], [Modula-3-Web]. Sather: [Sather-Web]. Beta: [Madsen 1993], [Beta-Web]. Self: [Chambers 1991], [Ungar 1992].

EXERCISES

E35.1 Stopping on short files

“A coroutine example”, page 1119.

Adapt the Simula coroutine example (printer-controller-producer) to make sure that it stops properly if the input does not have enough elements to produce 1000 output elements. (**Hint**: one possible technique is to add a fourth coroutine, the “reader”.)

E35.2 Implicit resume

(This is a exercise on Simula concepts, but you may use the notation of the rest of this book extended with the simulation primitives described in this chapter.) Rewrite the producer-printer example in such a way that each coroutine does not need to resume one of its colleagues explicitly when it has finished its current job; declare instead the coroutine classes as descendants of *PROCESS*, and replace explicit **resume** instructions by *hold (0)* instructions. (**Hints**: recall that event notices with the same activation time appear in the event list in the order in which they are generated. Associate with each process a condition that needs to be satisfied for the process to be resumed.)

E35.3 Emulating coroutines

Devise a mechanism for emulating coroutines in an O-O language of your choice (such as the notation of the rest of this book) that does not provide coroutine support. (**Hint**: write a *resume* procedure, implemented as a loop containing a conditional instruction with a branch for every **resume**. Obviously, you may not for this exercise use the concurrency mechanism of chapter 30, which among other applications supports coroutines.) Apply your solution to the producer-printer-controller example of this chapter.

E35.4 Simulation

Using the notation of this book or another O-O language, write classes for discrete-event simulation, patterned after the Simula classes *SIMULATION*, *EVENT_NOTICE*, *PROCESS*. (**Hint**: you may use the techniques developed for the previous exercise.)

E35.5 Referring to a parent's version

Discuss the respective merits of Smalltalk's **super** technique against the techniques introduced earlier in this book to enable a redefined routine to use the original version: *Precursor* construct and, when appropriate, repeated inheritance.

"Keeping the original version of a redefined feature", page 555.