Concepts and terminology in the Simula Programming Language

An introduction for new readers of Simula literature

Stein Krogdahl
Department of Informatics
University of Oslo, Norway
April 2010

Introduction

Simula is the programming language in which the basic object-oriented concepts were introduced, in 1967. Its syntax and procedural constructs are more or less directly taken from the Algol 60 language. Simula came in two versions, and there has been a lot of misunderstanding around this. The language now called “Simula 1” appeared in 1965, and was especially aimed at discrete event simulation. It had no notion of classes, subclasses or virtual entities, but instead a construct *activity* of which one dynamically could generate *processes*. These concepts to some extent corresponds to *classes* and *objects* in current O-O languages, but were aimed at simulation in that they had a built-in mechanism for simulated time in which the processes could live their lives. This mechanism relied on the processes running in *quasi-parallel* (as *co-routines*) and on a queue of scheduled processes sorted on the time they next wanted to do something.

However, the designers of Simula 1 (Ole-Johan Dahl and Kristen Nygaard) soon saw (with some essential inspiration from Tony Hoare) that these concepts could be generalized and made part of a general purpose programming language. Thus, in 1967 a new language appeared that explicitly was *not* aimed at any particular area of programming. It was this language that, for the first time, had the now well known concepts of *objects, classes, subclasses* and *virtual* entities. Also, any object in the new language could act as a process, in the sense that it could run in *quasi-parallel* with other objects. But this mechanism was now *not* tied to simulation. Instead, using this general mechanism, a special “simulation package” was provided, and from the very start it became an integrated part of the language. With this package, simulation programs could be written in much the same way as in Simula 1, and thus the new language soon took over the market from Simula 1, which more or less went out of use.

There was much discussion about the name of the new language, but as the first Simula language already had gained some popularity, the unfortunate decision was taken that this new language should also be called Simula, but with the postfix “67”. As indicated above, this has caused a lot of misunderstanding, the most common one being that also Simula 67 was a language especially aimed at simulation, and therefore did not really introduce the O-O concepts in a fully general form. This is simply wrong, but one should not be surprised that such misunderstandings turn up when the languages were named in such a confusing way. The name “Simula 67” is now officially changed to only “Simula” (and the Simula 1 language is more or less forgotten).

Concepts and terminology

The terminology used for some important concepts in Simula is different from what later become the norm in O-O programming. This has had the unfortunate effect that most articles, books and papers about Simula have been difficult to read for those only knowing the "standard" terminology used today. This is a pity, and below we explain the main concepts and the terminology used in Simula to the extent that such articles should be straight forward to read.

- **Class**
  Simula has classes in very much the same way as e.g. Java later adopted. Their syntactic structure is as follows:

  ```
  class C(…);   ! Formal parameters within the parenthesis ;
  begin
       …
  end;
  ```

  Thus, Simula uses "begin … end;" in the same way as "{ … }" is used e.g. in Java. More about parameters and other details around classes is given below. As is seen, comments in Simula start with "!" and end at the first ";".

- **Prefix classes and superclasses**
  The word prefix is used in Simula in a way that most closely corresponds to "super" in e.g. Java, and it stems from the special syntax used in Simula where the name of the superclass is given as a *prefix* to the class declaration. Thus, the following Simula class declaration:

  ```
  A class B; begin … end;
  ```

  corresponds to the following Java declaration:

  ```
  class B extends A { … }
  ```

  Thus we e.g. have the following correspondences:
  - Prefix class = super class = base class
  - Multiple prefixing = multiple inheritance (but this is not offered in Simula)

- **Prefixed block**
  This is a special variation of the usage described above. In Simula, classes within classes are allowed to any depth, and one is encouraged to use classes with local classes to implement concepts that naturally comprise more than one class. Thus the concept *Graph* (in fact, *directed* graph, as the edge has a *from* and a *to* end) can in Simula be described e.g. as follows (where only some crucial variables are mentoned):

  ```
  class Graph;
  begin
      class Node; begin
          ref(Edge) firstEdgeOut; …   ! The first in a list of outgoing edges ;
      end;
      class Edge; begin
          ref(Node) from, to;
          ref(Edge) nextEdgeOut; …  ! The next in the list of outgoing edges;
      end;
  end;
  ```

When we want to use these classes we can *prefix a block*, usually the outermost, with the name of the outer class Graph. This has the effect that that block acts as if it is the body of a subclass of Graph, so that all the declarations inside class Graph become directly available in the block.

```
Graph begin
    Node class LargeNode; begin integer size;  … end;
    ref(LargeNode) rLN;
    rLN :- new LargeNode;  ! More about "reference assignments" below ;
    rLN.size := 12345;
end;
```

Regrettably, it is not allowed to prefix a block with more than one class, which to some extent reduces the usability of this mechanism, as one would often like to "import" more than one such functional unit e.g. from a library. Note that one could use the following construct:

```
Module1 begin
       Module2 begin
           … We can here refer to all declarations in Module1 and in Module2  …
       end;
end;
```

However, because of block level restrictions for subclassing (see below), we cannot make subclasses of the classes in Module1 inside the block prefixed with Module2, and this reduces the value of such a construct.

Later languages has adopted constructs similar to block prefixing, e.g. Beta and Java (as *anonymous classes*).

- **ref(C) a, b;**
  As shown above, Simula uses "**ref(C)"** to indicate that you use class **C** as a type. Thus "ref(C) a, b;" corresponds to what is written "C a, b;" in Java. For the basic value types (boolean, integer, real etc.), Simula uses the same syntax as Java (except that Simula does not allow explicit initialization). The "no reference" value is in Simula written: **none**.

- **Assignment**
  Simula uses ":=" for standard value assignment. However for *reference assignments*, that is, for variables of type "ref(C)" for some class C, the operator ":-" is used. This is *almost* totally redundant, but for texts (strings) one can use either operator and get different effect. In Simula, texts are stored in special objects called *text objects*, and one can have references to such objects in the same way as to ordinary *class objects*. Thus, if we write "text t, u;", we get two variables that can hold such references, and we can perform the following assignments: "t:= u" and "t:-u". The former will copy the text of the (text) object pointed to by u into the object pointed to by t. The latter will set t to reference the same text object as u does.

  If a reference assignment in Simula is made the "wrong way" (to a variable typed with a subclass of the type of the expression), Simula will automatically insert a run time test.

- **Boolean operators**
  In Simula the following boolean operators:
  For boolean values: *and, or, not*
  For numeric and text values: =, <>, <=, >=, <, >
  For references (also text references): ==, =/=

- **Simula uses the keyword "procedure".**
  For the standard concept *method/function/procedure* Simula uses the word *procedure* in all cases, even when a result is returned. Note that procedures can be declared anywhere in a program, not only at the outermost level within classes. This includes the outermost level of the program (global procedure), and inside another procedure. Examples of procedure declarations:

  ```
  procedure action(a, b); integer a; ref(C) b; begin … end;

  boolean procedure isGood(a, b); ref(C) a, b; begin … isGood:= false; … end;
  ```

  A *type procedure* (one that returns a result) will implicitly get a local variable typed and named as the procedure itself. The returned value is the value of this variable at procedure termination (that is, when execution passes through the "end" of the procedure). Simula has no *return* statement.

  Simula allows procedures to have procedure parameters ("formal procedures"). Also a variant of "by variable" transmission is offered, which is called "by name" transmission (an inheritance from the language Algol 60). None of these modes are allowed for classes. If a procedure doesn't have parameters, no parentheses are used. Note that in Simula parameters of type ref(C) are said to be transmitted "by reference" even when the simplest form of transmission is used, which is the same as Java's "value" transmission of references.

- **Parameters to classes**
  Classes may have parameters, and syntactically they occur after the name of the class, exactly as for procedures. However, only parameters that are transmitted by value in the Java sense are legal for classes (which includes transmission "by reference" in the Simula sense). The formal parameters of a class are in every way considered normal attributes of the class, e.g. in that they can be accessed by "dot-notation". The only difference is that they are initated to the values of the actual parameters. Subclasses will inherit the parameters of the superclass, but may itself specify additional parameters. Thus, in a "new C(…)" operation one has to give as many actual parameters as given in C and in all its superclasses together, starting with those of the outermost superclass.

- **Type-casting and "qua"**
  What in Java is written "((C)r).a" is in Simula written "r qua C.a" (where C can only be a class). Note that no parentheses are needed.

- **Co-routines and quasi-parallel execution**
  As mentioned in the introduction, "the code of" an object can run in quasi-parallel with the code of other objects. Below we give a slightly simplified explanation of how this works.

A class is syntactically very similar to a procedure, also in that after the declarations of the class, executable code may follow. That code is started immediately after a new object is generated and the parameters are transmitted. This code can be used to initialize the object, but it can also perform the very special statement "detach", which means that the execution control goes back to the "main program" (we give no detailed explanation of this concept) where it last left. Also, the object itself remembers where it (last) executed a detach. Later, one may, from the main program or from another object, say "resume" (or "call", which are technically slightly different) to the object, and the object will then resume its actions from where it last stopped. If the resume operation comes from another object, that object will now be detached so that it can later resume execution from that point.

While an objects executes in quasiparallel with others, it may get its own call stack by calling procedures, and a "detach" or a "resume" performed from within its own call stack will then work so that the object itself is detched. However, the "resume point" for this object is then set inside the procedure call at the top of the stack at the place where it performed the "detach" or "resume" operation. Thus, if this object is later resumed, it will start at this point, with the full stack preserved from last time it executed. As every object in a quasiparallel execution in this way may have its own stack, Simula is often characterized as a *multi-stack language*. Simula does not have any mechanism for running objects in real parallel.

- **Visibility regulation: hidden and protected**
  At the outset in 1967, Simula had no mechanisms for regulating the visibility of names. However, the need for this soon became clear, and around 1972 such a mechanism was added. It works so that a declaration local to a class can be declared *protected*, which means that it can only be accessed from inside an object of that class, and of subclasses. When it is declared *protected*, it can also be declared *hidden* either in the same class or in a subclass. This has the effect that it will not be visible in further subclasses.

- **The virtual mechanism**
  In Simula, not only procedures can be virtual, but also labels and switches (a sort of label arrays). However, this is now almost forgotten, and probably never used. Procedures can be declared either virtual or not (as e.g. in C++ and C#), and if virtual it can optionally be "abstract" in the sense that no implementation is given in that class. In the 1967 definition parameters to virtual procedures had to be type checked at run time (to obtain a certain freedom), but later syntax was added so that one may have them checked during compilation, as in most modern typed O-O languages.

- **Subclasses and block levels**
  In Simula, subclasses can only be declared at the same "block level" as the superclass. So, if the closest surrounding block of the subclass declaration is B (a procedure, a class, or a prefixed or simple block), also the superclass must be declared in B. A small exception is that if B is a class or prefixed block, then the superclass may be declared in the prefix (superclass) of the class/block. The rules are so strict to avoid any attempt to access variables declared in blocks that are terminated. Thus, you cannot, e.g., in some procedure where you suddenly need it, declare your own subclass of a class declared further out, and this is sometimes a pity. When you fetch a

separately compiled class e.g. from a library it is automatically given a block level so that its fits into the program.

This strict block level rule is relaxed in many later languages, e.g. in Beta and in Java. However, in Java this has resulted in some “strange” restrictions concerning variable access from inner (nested) classes.

- **Simula has no “most general class”, like Java’s “Object”**
  There is nothing in Simula corresponding e.g. to the predefined class Object in Java. However, all classes declared without a superclass implicitly get some predefined procedures.

- **Simula’s inner**
  The executable code of a class can, by the keyword “inner”, indicate that the corresponding code of the actual subclass (if one exist) should be executed here. Assume we have the following classes:

  ```
  class A; begin <code A1>; inner; <code A2> end;
  A class B; begin <code B1>; inner; <code B2> end;
  B class C; begin <code C1>; inner; <code C2> end;
  ```

  When an object of class C is generated, the execution order of the code is as follows:

  ```
  <code A1>; <code B1>; <code C1>; <code C2>; <code B2>; <code A2>
  ```

  Thus e.g. class A can control the start and termination of the execution of the code provided by its subclasses. If no *inner* is given, one is assumed before the terminating end of the class body. Some later languages also offer a similar construct (at least Beta and gbeta).

- **Simula attributes and members**
  In Simula, everything declared at the outermost level within a class (or a superclass) is said to be an *attribute* of that class or of an object of that class. The word *member* can definitely *not* be used with a meaning similar to that of *attribute*, but instead any object of a class or of a subclass is said to be a *member of that class*. Thus, a class corresponds to a *set whose members are the objects of that class (including its subclasses),* and thus the word *sub-class* get its proper meaning.

- **No exceptions, but goto and detach statements**
  Simula has nothing like the exception mechanism of e.g. Java (that can bring you backwards along the call chain in a controlled way). However, Simula has an expressive goto statement that can bring you to a statically visible *label* (written as a name followed by a colon in front of a statement). The label may be declared in any textually surrounding block, and the control will be transferred to that point, and all blocks in between will be terminated in a controlled way. Another way to obtain an exception-like effect is to use a detach statement from within a procedure that is called (directly or indirectly) from an object. This will bring you dynamically backwards to that object, this object will be detached (as explained above), and the control goes back to the “main program”. Explicit information cannot be transmitted with any of these mechanisms.

- - - o o o - - -