Simula and Java
two of a kind

by

Øystein Myhre Andersen
Software Veteran

JavaBin/JavaZone 2020

Wikipedia says:

Simula is the name of two simulation programming languages, Simula I and Simula 67, developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. Syntactically, it is a fairly faithful superset of ALGOL 60, also influenced by the design of Simscript.

Simula has been used in a wide range of applications such as simulating very-large-scale integration (VLSI) designs, process modeling, communication protocols, algorithms, and other applications such as typesetting, computer graphics, and education. The influence of Simula is often understated, and Simula-type objects are reimplemented in C++, Object Pascal, Java, C#, and many other languages. Computer scientists such as Bjarne Stroustrup, creator of C++, and James Gosling, creator of Java, have acknowledged Simula as a major influence.

Simula 67 added new concepts to the Simula language

- Objects
- Classes
- Inheritance and subclasses
- Virtual procedures
- Coroutines
- Class libraries
- Discrete event simulation
- and features garbage collection.

Concepts that are well known today but which represented something new in 1967

Simula is considered the first object-oriented programming language. As its name suggests, the first Simula version by 1962 was designed for doing simulations; Simula 67 though was designed to be a general-purpose programming language and provided the framework for many of the features of object-oriented languages today.

The summer of 1967

(how I met Simula and fell in love)

The “Simula 67 Common Base Conference” was held at Norwegian Computing Center(NCC) in june 1967.

My role was to walk around a table and gather the pages of the report that was presented at the meeting. This was my first encounter with Simula - during my first summer job at NCC.

This conference also set up a body called "Simula Standardization Group" (SSG), with representatives from NCC and the different implementers. Its first task was to establish a formal language definition.

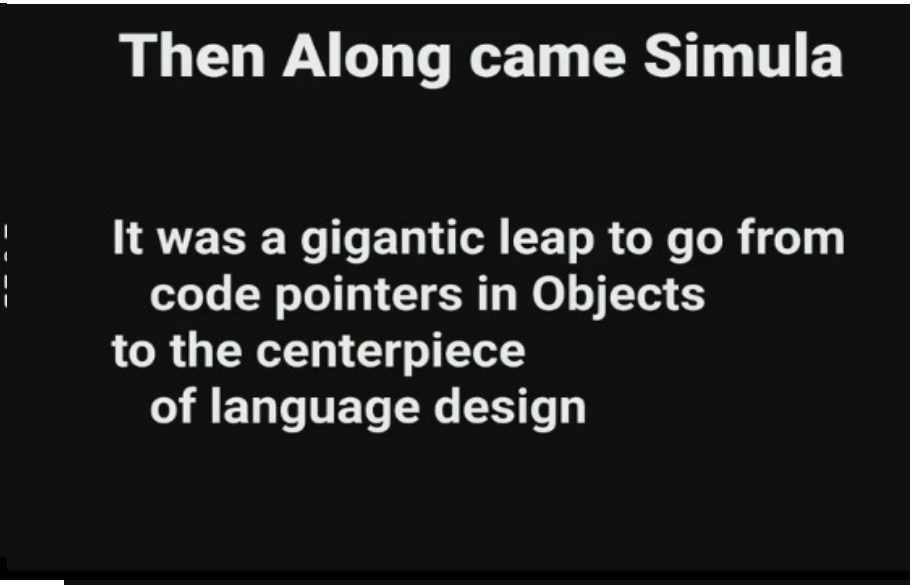
The language was formally frozen in the “Simula 67 Common Base Language” report, accepted by SSG on February 10, 1968.

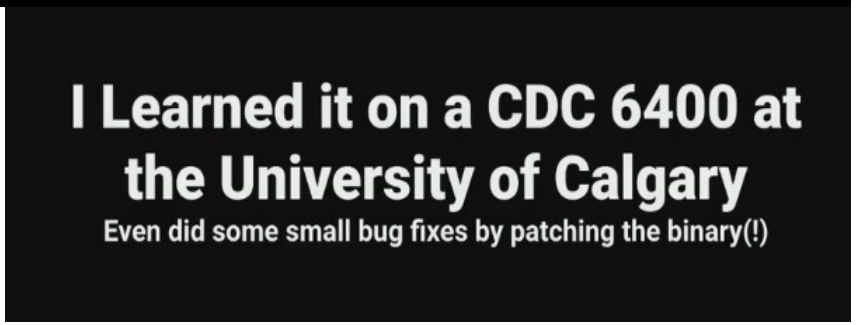
Bjørn Myhrhaug together with Ole-Johan Dahl, wrote an implementation guide for Simula 67, which was important for later implementations of the language.

The Control Data compilers were finished during the spring of 1969. In 1969 it was also decided that NCC itself should complete a compiler for the UNIVAC 1100 series and implement Simula 67 for IBM 360/370. These implementations were finished in March 1971 and May 1972 respectively.

Eventually I joined the Univac team to test the runtime system and later took over responsibility for further development.

A few years later in Calgary Canada



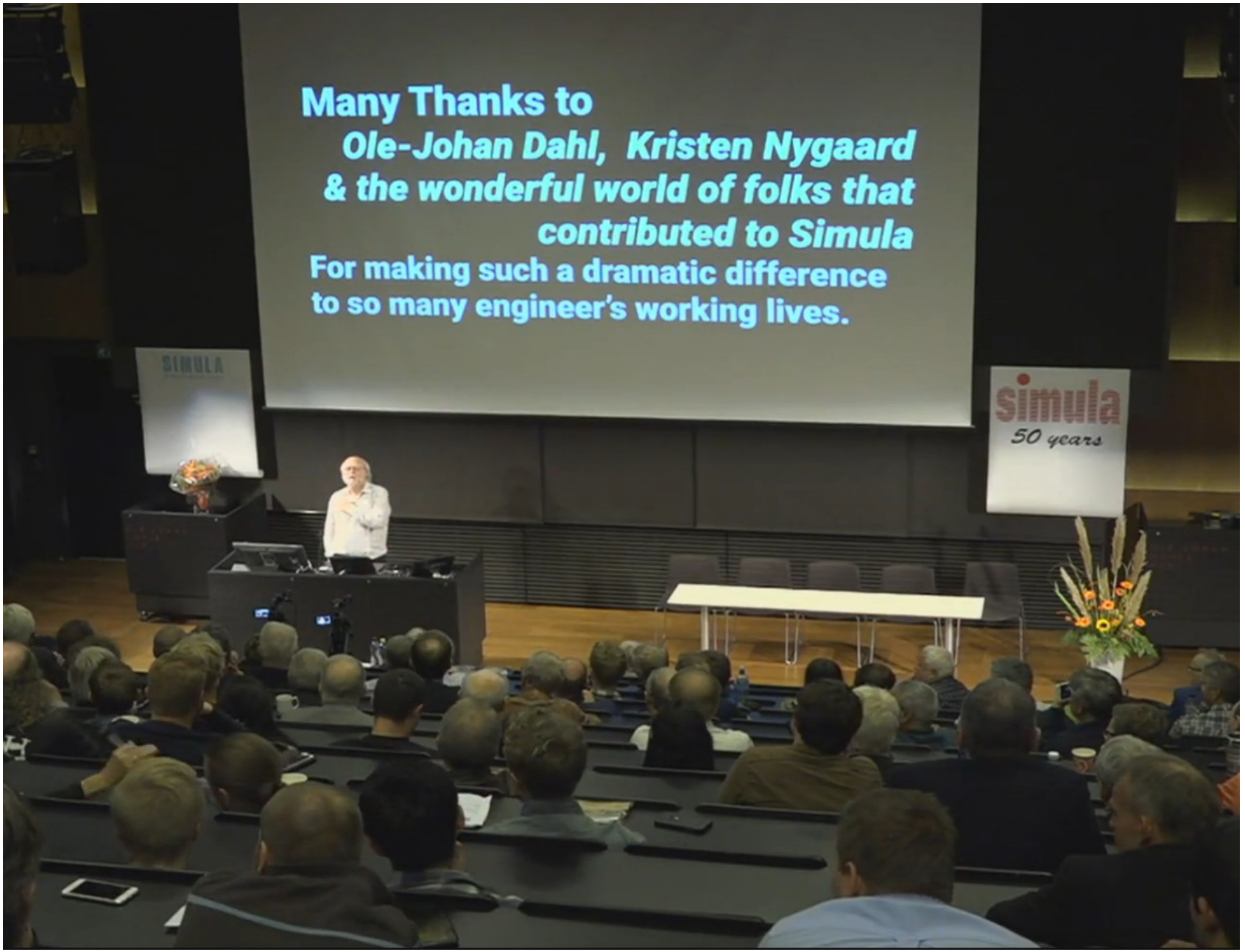


… a young student, James Gosling, obtained Control Data's distribution tape and began digging.

He found something new on that tape which turned out to be Control Data's implementation of Simula.

It is obvious that James then fell in love with Simula, which later led him to define the Java language.

The pictures on this page are from James Gosling lecture at the 50th anniversary of Simula.



James Gosling at the 50th anniversary of Simula, September 2017.

Further development at NCC

Next to maintenance of the Univac and IBM systems, two new projects were started, both of which resulted in start-up companies:

Simula as. was started as a result of the S-Port project. The idea was to develop a portable Simula System using S-Code which was specified in a collaboration with the University of Edinburgh. Mach-S's Simula compiler was modified and used as front end compiler producing S-Code. A separate language 'Simuletta' was developed to compile the runtime system to S-Code.

Back end systems for a variety of computer systems were developed, including Nord 500 and MS DOS,Windows and OS2.

SimTech was a startup company based on the Mach-S and MIKADO projects at NCC with SimX as industrial partner. The idea was to make and later produce a Simula machine in hardware. This was something many had been talking about ever since Intel had plans for its own Simula chip (iAPX 432). SimTech developed a Simula system and supplemented it with a graphics tool: class Graphics and a prototype machine were produced before they had to throw in the towel. The product was realized and sold to a large company in the United States, but then came the financial crisis of 1987 and it all faded away.

Highlight in Simula's design

Simula objects are alive

Simula objects are able to act as processes that can execute in "quasiparallel". The idea is that classes have statements of their own (like procedures), and that these are started when an object is generated.

Objects may choose to temporarily stop their execution and transfer the control to another process. If the control is later given back to the object, it will resume execution where the control last left off. A process will always retain the execution control until it explicitly gives it away.

When the execution of an object reaches the end of its statements, it will become "terminated", and can no longer be resumed (but local data and local procedures can still be accessed from outside the object).

This mechanism is used by class Simulation to model discrete events during the lifetime of processes. Note that when a process yields control a complete "reactivation point" is saved in the heap, making it possible to resume operation at some later time.

Quasi-parallel sequencing is very much like "Delimited continuations" in the upcoming Java 17.

It was a pedagogical challenge to explain this concept so that the students understood how it could be used in practical programming other than simulation.

Sequencing

(this part of Simula was omitted by James Gosling.)

Simula has three primitives to express Sequencing:

obj.Detach – Suspends the execution, saves a reactivation point and returns.

Call(obj) – Restarts a detached object at the saved reactivation point.

Resume(obj) – Suspend and restart another object in one operation.

Which, with Continuations, corresponds to:

obj.yield – Suspends the execution, saves a reactivation point and returns.

obj.run() – Startes or restarts a suspended object at the saved reactivation point.

- The detach - call pair constitutes coroutines in Simula

- The detach – resume pair establish Symmetric component sequencing used to implement discrete event simulation.

Project LOOM: Delimited continuations

(https://wiki.openjdk.java.net/display/loom)

When Java was defined, threads were used to implement AWT. Some sort of parallelism was needed for this purpose and threads had all the necessary features. Later threads were heavily used in servers, but there was a drawback, it is (too) expensive to create threads. Which led to creative solutions such as free thread-pools.

Project Loom aims to solve this problem by introducing lightweight threads. The basic building concept is called "Delimited Continuation", which is very similar to Simula's quasiparallel sequencing (QPS). E.g. both can be used to establish co-routines.

A delimited continuation is a program object representing a computation that may be suspended and resumed, very much like the detach-call pair in Simula. In both worlds we have a living object with its own stack that can be dismounted from the main stack and remounted at some later time.

The Continuation class is implemented natively in Hotspot (Java virtual machine).

Threads were used to implement QPS in the first version of my Simula system. Later, when i got the opportunity to test some simulations using the Continuation class I discovered a dramatic improvement in performance. I actually executed a simulation with a million processes without any problems on a PC.

This is good news for future server designs that need to handle millions of concurrent connections

Coroutines

Class Continuation is formally defined as:

```
public class Continuation implements Runnable {
      public Continuation(ContinuationScope scope, Runnable target)
      public static Continuation getCurrentContinuation(ContinuationScope scope)
      public final void run()
      public static void yield(ContinuationScope scope)
      public boolean isDone()
      ....
}
```

Java 17: Coroutine Example:

```
Continuation coroutine=
new Continuation(scope,new Runnable() {
       public void run() {
       trace("Part 1 - Statements");
       Continuation.yield(scope); // DETACH 1
       trace("Part 2 - Statements");
       Continuation.yield(scope); // DETACH 2
       trace("Part 3 - Statements");
}});
```

An application may look like:

```
coroutine.run(); // Will perform Part 1.
trace("Returns here after first yield");
coroutine.run(); // Will perform Part 2.
trace("Returns here after second yield");
coroutine.run(); // Will perform Part 3.
trace("Returns here after 'FINAL END'");
coroutine.run(); // IllegalStateException: Terminated
```

Simula: Same Coroutine Example:

```
ref(Coroutine) coroutin;

class Coroutine; begin
       trace("Part 1 - Statements");
       detach;
       trace("Part 2 – Statements");
       detach;
       trace("Part 3 – Statements");
end;
```

An application may look like:

```
coroutin=new Coroutine; -- Will perform Part 1.
trace("Returns here after first detach");
call(coroutin); -- Will perform Part 2.
trace("Returns here after second detach");
call(coroutin); -- Will perform Part 3.
trace("Returns here after 'FINAL END'");
call(coroutin); -- Runtime Error: Terminated
```

Object Heap and Garbage Collector

Simula uses a **single data heap** for objects and procedure frames.

A garbage collector is used to clean up this data heap.

Simula need this to implement **detach**, **call** and **resume**.

Java has both **heap** (objects) and **stack** (method frames).

When using Java to implement Simula,

we need a mechanism to **cut off and save parts of the stack**

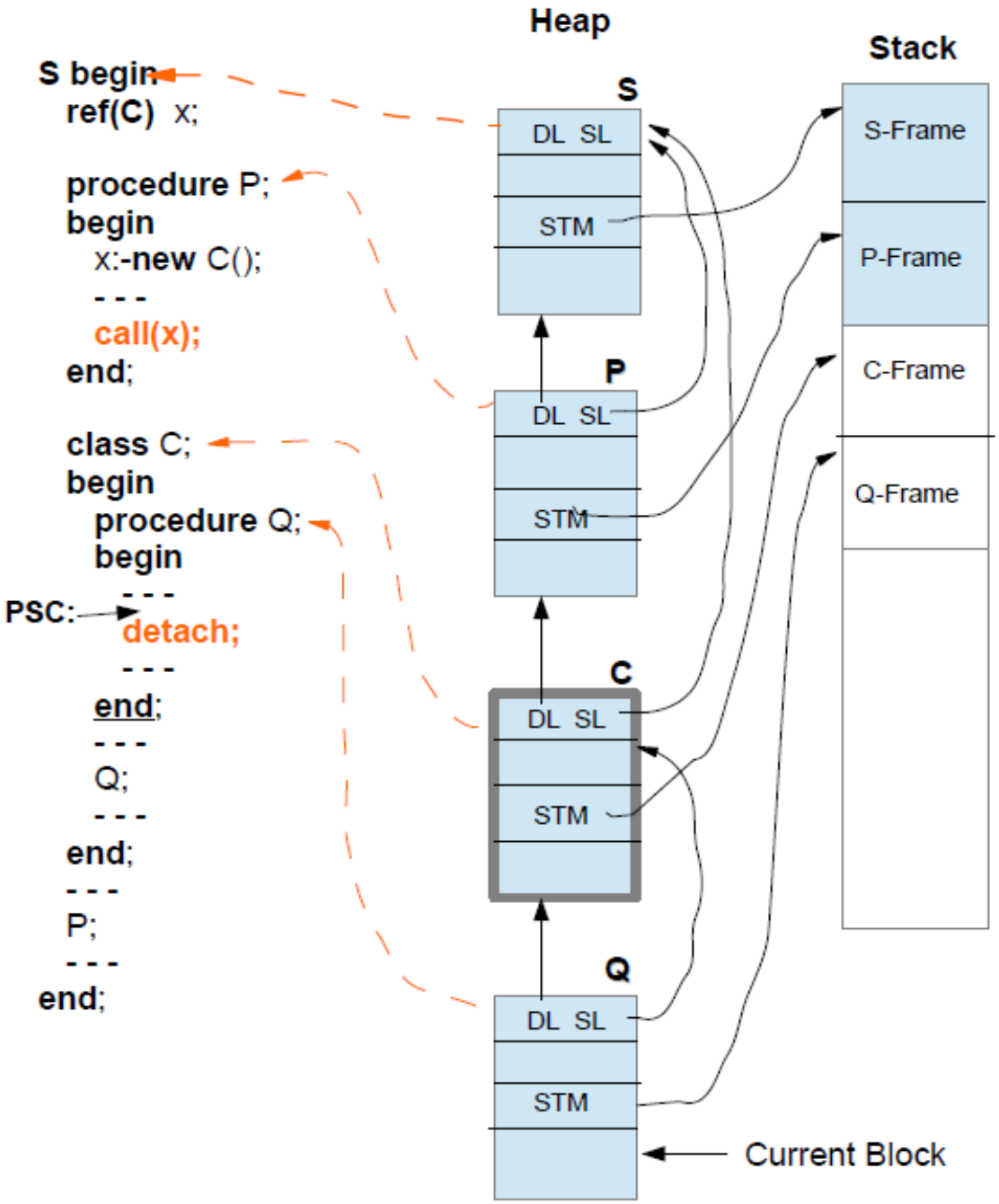


Fig. 5 Using LOOM: Just before Detach

Typical situation in Simula just before detach.

It is necessary to unmount/detach two stack-frames; C-Frame and Q-frame.

This can be achieved in several ways:

1) By popping the stack into a suitable object.
(as Kotlin does ?)

2) By letting objects like C start their own stack. (as new Thread does)

3) Let the Hotspot VM unmount the stack.

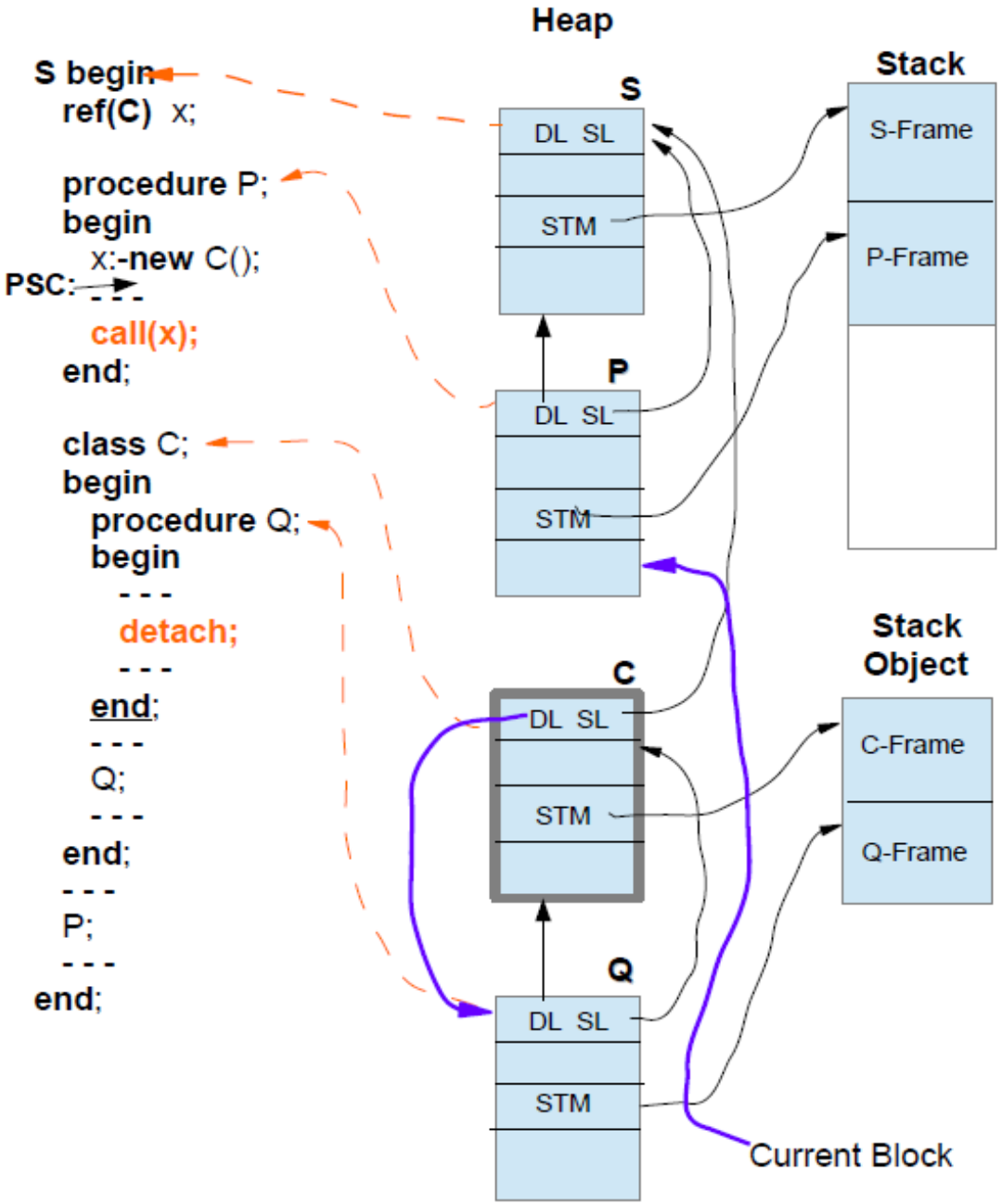


Fig. 6 Just after Detach – Dismounts part of Stack

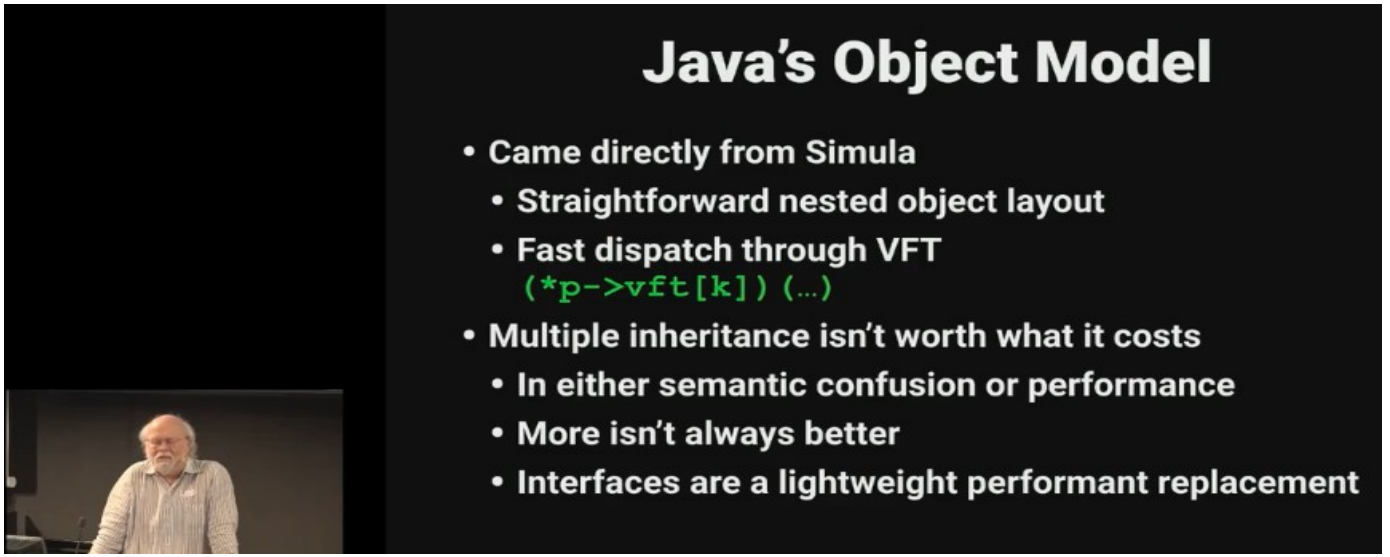
This is the situation after detach.

Procedure P gets control and the stack is ready for further computations.

The Component C is ready to be attached again and the C-Q stack object may be re-mounted onto the stack at a later time.

Making a Simula System

Making a Simula System



In the fall of 2017, I started experimenting with an implementation of Simula written in Java.

It was obvious to use Java's class concept since Java used the same object model as Simula.

Later, in the spring of 2018, an open source project was established creating a full-fledged Simula Implementation strictly after 'Simula Standard' and written in Java.

See: https://portablesimula.github.io/github.io/

Most of the code generator was rather straightforward except a few points:

- Coroutines
- Quasi-parallel Sequencing
- Simula's old fashioned goto-statement

On the following pages, I will go through some selected issues regarding mapping Simula to Java

How to goto in Java

Java does not support labels like Simula. The Java Virtual Machine (JVM), however, has labels.

A JVM-label is simply a relative byte-address within the byte-code of a method.

The Simula implementation will generate Java-code which is prepared for Byte Code Engineering.

Suppose a Simula program containing labels and goto like this:

```
Begin

   L: ...
      goto L;
      ...
End;
```

The label 'L' is declared like this:

```
final LABQNT$ L=new LABQNT$(this,1); // Local Label #1=L
```

LABQNT is a sub-class of RuntimeException.

The Statement code of a Simula block is mapped to a Java Method like this:

```
// SimulaProgram Statements
public RTObject$ STM() {
    adHoc00 THIS$=(adHoc00)CUR$;
    LOOP$:while(JTX$>=0) {
        try {
            JUMPTABLE$(JTX$); // For ByteCode Engineering
            // Statements ....
            LABEL$(1); // L
            // Statements ....
            throw(L); // GOTO EVALUATED LABEL
            // Statements ....
            break LOOP$;
        }
        catch(LABQNT$ q) {
            CUR$=THIS$;
            if(q.SL$!=CUR$) {
                CUR$.STATE$=OperationalState.terminated;
                throw(q); // Re-throw exception for non-local Label
            }
            JTX$=q.index; continue LOOP$; // GOTO Local L
        }
    }
    EBLK();
    return(null);
}
```

Byte Code Engineering

- `LABEL$(n); // Label #n`

  This method-call is used to signal the occurrence of a Simula Label. The byte-code address is collected and some instructions are removed. The parameter 'n' is the label's ordinal number.

  I.e. Try to locate the instruction sequence:

  ```
  <prev instruction>
  ICONST n
  INVOKESTATIC LABEL$
  <next instruction>
  ```

  Pick up the number 'n', remember address
  then remove the two middle instruction.

- `JUMPTABLE$(JTX$);`

  This method-call is a placeholder for where to put in a Jump-Table.

  Try to locate the instruction sequence:

  ```
  GETFIELD JTX$
  INVOKESTATIC JUMPTABLE$
  ```

  And replace it by the instruction sequence:

  ```
  GETFIELD JTX$
  TABLESWITCH ... uses the addresses collected for labels.
  ```

The ObjectWeb ASM Java bytecode engineering library from the OW2 Consortium is used to modify the byte code.

They say:

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form.

ASM provides some common bytecode transformations and analysis algorithms from which custom complex transformations and code analysis tools can be built.

ASM offers similar functionality as other Java bytecode frameworks, but is focused on performance. Because it was designed and implemented to be as small and as fast as possible, it is well suited for use in dynamic systems (but can of course be used in a static way too, e.g. in compilers).

More info at: https://asm.ow2.io
and https://www.ow2.org/

Parameter transfer "by Name" (Jensen's Device)

The basic principle is to establish an object in the calling scope. This object will have two attributes methods; get and put to read the value of the actual parameter or write back a new value, respectively. The class used for such parameter transfers looks like this:

```
abstract class Name$<T>{
        abstract T get();
        void put(T x) { error("Illegal ..."); }
}
```

Note that the 'put' method has a default definition producing an error. This enables redefinition of the 'put' method to be omitted for expression as actual parameters.

The Simula Procedure:
Is translated into this Java method:

```
procedure P(k); name k; integer k; k:=k+1;
```

```
void P(Name$<Integer>k) {
        k.put(k.get() +1); // E.g: k=k+1
}
```

A procedure call like: *P(q);*
where q is a variable, then the actual parameter is coded like this:

```
new Name$<Integer>() {
        Integer get() { return (q); }
        void put(Integer x) { q = (int) x; }
}
```

If, however, the actual parameter is an expression like (j+m*n) it is coded Like this:

```
new Name$<Integer>() {
        Integer get() { return (j +m * n); }
}
```

Note that the put method is not redefined in this later case. Thus the default definition giving an error, will be used to prevent writing into an expression.

For Statement (inherited from Algol)

The Implementation of the for-statement is a bit tricky. The basic idea is to create a ForList iterator that iterates over a set of ForElt iterators. The following subclasses of ForElt are defined:

- SingleElt<T> for basic types T control variable
- SingleTValElt for Text type control variable
- StepUntil for numeric types
- WhileElt<T> for basic types T control variable
- WhileTValElt representing For t:= <TextExpr> while <Cond>
  With text value assignment

Each of which deliver a boolean value 'CB' used to indicate whether this for-element is exhausted. All parameters to these classes are transferred 'by name'. This is done to ensure that all expressions are evaluated in the right order. The assignment to the 'control variable' is done within the various for-elements when the 'next' method is invoked. To get a full overview of all the details you are encouraged to study the generated code together with the 'FRAMEWORK for for-list iteration' found in the runtime class RTObject$.

Example, the following for-statement:

for i:=1,6,13 **step** 6 **until** 66,i+1 **while** i<80 **do** j:=j+i;

Is compiled to:

```
for(boolean CB:new ForList(
               new SingleElt<Number>(...)
              ,new SingleElt<Number>(...)
              ,new StepUntil(...)
              ,new WhileElt<Number>(...)
            )) { if(!CB) continue;
                 j=j+i;
               }
```

Optimized For-Statement

However; most of the for-statements with only one for-list element are optimized.

Single for step-until statements are optimized when the step-expression is constant.
I.e. the following for-statements:

for i:=<expr-1> step 1 until <expr-2> do <statements>
for i:=<expr-1> step -1 until <expr-2> do <statements>

for i:=<expr-1> step 6 until <expr-2> do <statements>
for i:=<expr-1> step -6 until <expr-2> do <statements>

are compiled to:

```
for(i = <expr-1>; r <= <expr-2>; r++) { <statements> }
for(i = <expr-1>; r >= <expr-2>; r--) { <statements> }

for(i = <expr-1>; r <= <expr-2>; r=r+6) { <statements> }
for(i = <expr-1>; r >= <expr-2>; r=r-6) { <statements> }
```

The other kinds of single elements are optimized in these ways:

for i:=<expr> do <statements>

for i:=<expr> while <cond> do <statements>

are compiled to:

```
i = <expr>; { <statements> }

i = <expr>;
While( <cond> ) {
      <statements>;
      i = <expr>;
}
```

Object relations: **is** and **in**

In Simula the operators **is** and **in** may be used to test the class membership of an object.

The relation "X **is** *C*" has the value **true** if X refers to an object belonging to the class *C*, otherwise the value is **false**. That is, the value is **true** if the object X is generated by class *C* and not by a subclass of *C*.

if X **is** *C* is translated to: **if**(X.getClass() == *C*.**class**)

The relation "X **in** *C*" has the value **true** if X refers to an object belonging to a class *C* or a class inner to *C*, otherwise the value is **false**. That is, the value is **true** if the object X is generated by class *C* or a subclass of *C*

if X **in** *C* is translated to: **if**(X **instanceof** *C*)

Connection Statement

The connection statement is implemented using Java's **instanceof** operator and the **if** statement. For example, the connection statement:

inspect x **do** image:-t;

Where 'x' is declared as a reference to an ImageFile, is compiled to:

```
if(x!=null) x.image=t;
```

Other examples that also use 'ref(Imagefile) x' may be:

1) **inspect** x **do** image:-t **otherwise** t:-**notext**;

2) **inspect** x
   when infile **do** t:-intext(12)
   when outfile **do** outtext(t);

3) **inspect** x
   when infile **do** t:-intext(12)
   when outfile **do** outtext(t)
   otherwise t:-**notext**;

These examples are compiled to:

1)
```
if(x!=null) x.image=t; else t=null;
```

2)
```
if(x instanceof InFile infile) t=infile.intext(12);
else if(x instanceof OutFile outfile) outfile.outtext(t);
```

3)
```
if(x instanceof InFile in) t=in.intext(12);
else if(x instanceof OutFile out) out.outtext(t);
else t=null;
```

The type Text

The type Text: The type **text** is similar to the String type in Java.

Text reference variables is a composite structure represented by a TXTREF Java Object.

```
public class TEXTREF  extends RTObject {
      TEXTOBJ OBJ; // Reference to the text object.
      int START; // Start index of OBJ.MAIN[], counting from zero.
      int LENGTH;
      int POS; // Current index of OBJ.MAIN[], counting from zero.

      ...
}
```

The special text constant **notext** is represented by Java **null**.

The object TEXTOBJ is defined as:

```
public class TEXTOBJ  extends RTObject {
      public int SIZE; // Number of characters in the text object.
      public boolean CONST; // True: Indicates a text constant
      char[] MAIN; // The characters

      ...
}
```

Attributes of Text

A text reference has a number of attribute procedures to create and manipulate text values.

```
boolean procedure constant;
integer procedure start;
integer procedure length;
text procedure main;
procedure putchar(c); character c;
procedure putint(i); integer i;
procedure putfix(i,n); integer i,n;
procedure putreal(r,n); long real r; integer n;
procedure putfrac(i,n); integer i,n;
```

```
integer procedure pos;
procedure setpos(i); integer i;
boolean procedure more;
ext procedure sub(i,n); integer i,n;
text procedure strip;
character procedure getchar;
integer procedure getint;
long real procedure getreal;
integer procedure getfrac;
```

```
begin
      comment Double space removal program,
      first version, Rob Pooley, March 1984;

      text T; ! Holds the text to be processed;

      InImage; ! Reads the text into SysIn.Image;
      inspect SysIn do ! Refer to SysIn not SysOut;
      begin
            T :- Blanks(Image.Length); ! See the next chapter;
            T:=Image; ! Copies the characters into T;
      end;
      if T.GetChar=' ' then ! First character is a space?;
      begin
            if T.GetChar=' ' then ! Second character is also?;
            T:=T.Sub(2,T.Length-1); ! Remove first character;
      end;

      comment Now write out the text;
      OutText(T);
      OutImage

end Double space remover
```

Class Libraries

Compared to today's Java class library, Simula was equipped with quite a few classes:

- Simset contains facilities for the manipulation of circular two-way lists, called "sets".
- Simulation contains facilities for discrete event simulation.
- File classes contains a variety of classes for managing files.

Class simulation can of course be written in Java.

I have an ongoing work with such an implementation.

See: https://github.com/MyhreAndersen/ClassSimulation

Basic input/output

```
ENVIRONMENT class BASICIO (INPUT_LINELENGTH, OUTPUT_LINELENGTH);
      integer INPUT_LINELENGTH, OUTPUT_LINELENGTH;
      begin ref (infile) SYSIN; ref (printfile) SYSOUT;
      ref (infile) procedure sysin; sysin :- SYSIN;
      ref (printfile) procedure sysout; sysout :- SYSOUT;
      procedure terminate_program;
      begin ... ; goto STOP end terminate_program;
      class file ..... ;
      file class imagefile ..... ;
      file class bytefile ..... ;
      imagefile class infile ..... ;
      imagefile class outfile ..... ;
      imagefile class directfile ..... ;
      outfile class printfile ..... ;
      bytefile class inbytefile ..... ;
      bytefile class outbytefile ..... ;
      bytefile class directbytefile ..... ;
      SYSIN :- new infile("..."); ! Implementation-defined
      SYSOUT :- new printfile("..."); ! files names;
      SYSIN.open(blanks(INPUT_LINELENGTH));
      SYSOUT.open(blanks(OUTPUT_LINELENGTH));
      inner;
STOP: SYSIN.close;
      SYSOUT.close
end BASICIO;
```

All these classes are implemented using Java's file classes.

Rebel Oslo
13. november kl. 15:33 ·

Vinneren av møterom-avstemningen er klar!

```
Begin
  OutText ("Hello, World!");
  Outimage;
End;
```

Da er avstemningen om hvilke norske helter som bør bli hedret med eget møterom i Rebel over. Tusen takk til alle som stemte!

Førsteplassen går til Ole-Johan Dahl og Kristen Nygaard for programmeringsspråket Simula! 😀

Simula er det opprinnelige objektorienterte programmeringsspråket for datamaskiner, utviklet av Dahl og Nygaard ved Norsk Regnesentral i 1964.

Dahl og Nygaard mottok i 2001 Turing-prisen for arbeidet. Nå hedres de to med et møterom i Rebel.

Thank you !

References:

- UiO om Simula: https://www.ub.uio.no/english/subjects/naturalscience-technology/informatics/themes/dns/index.html
- James Gosling: https://vimeo.com/237788383
- OpenSource: https://portablesimula.github.io/github.io/