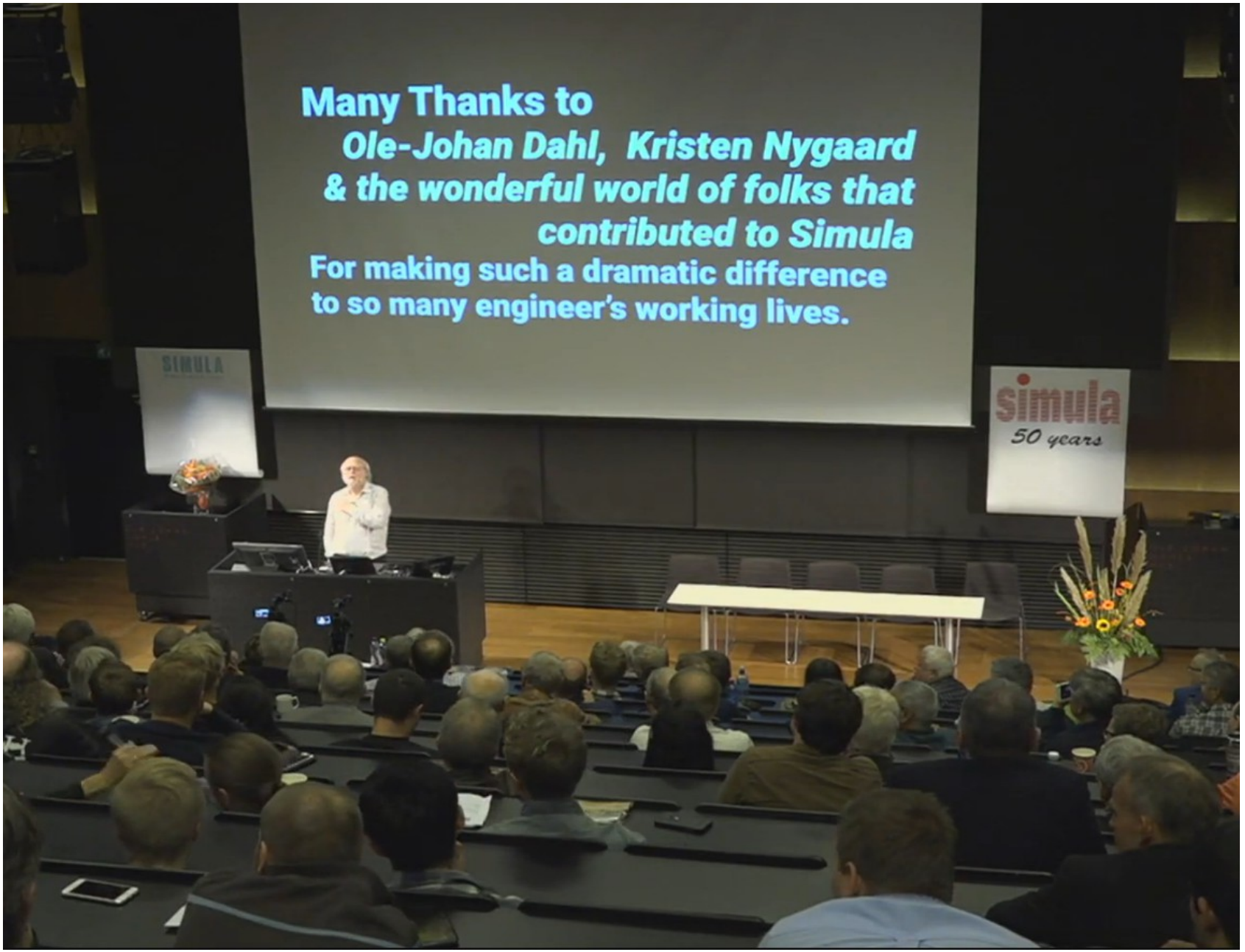
Re-inventing Simula using Java.

by

Øystein Myhre Andersen
Software Veteran

JavaZone 2019



James Gosling at the 50th anniversary of Simula.

Making a Simula System

An open source project was established creating a full-fledged Simula Implementation strictly after 'Simula Standard'.

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

I will explain how the new concept of *delimited continuations* can be used to implement co-routines and QPS,

and, if we have time for it,

how Java's exception handling together with *byte code egineering* can be used to reintroduce goto in the Java Language.

Sequencing

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

Simula has three primitives to express Sequencing:

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

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

3) Resume(obj) – Suspend and restart another object.

- The detach - call pair constitutes coroutines in Simula

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

References:

- Simula Standard sect. 7 Sequencing.
- Arne Wang og Ole-Johan Dahl. Coroutine sequencing in a block structured environment.

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

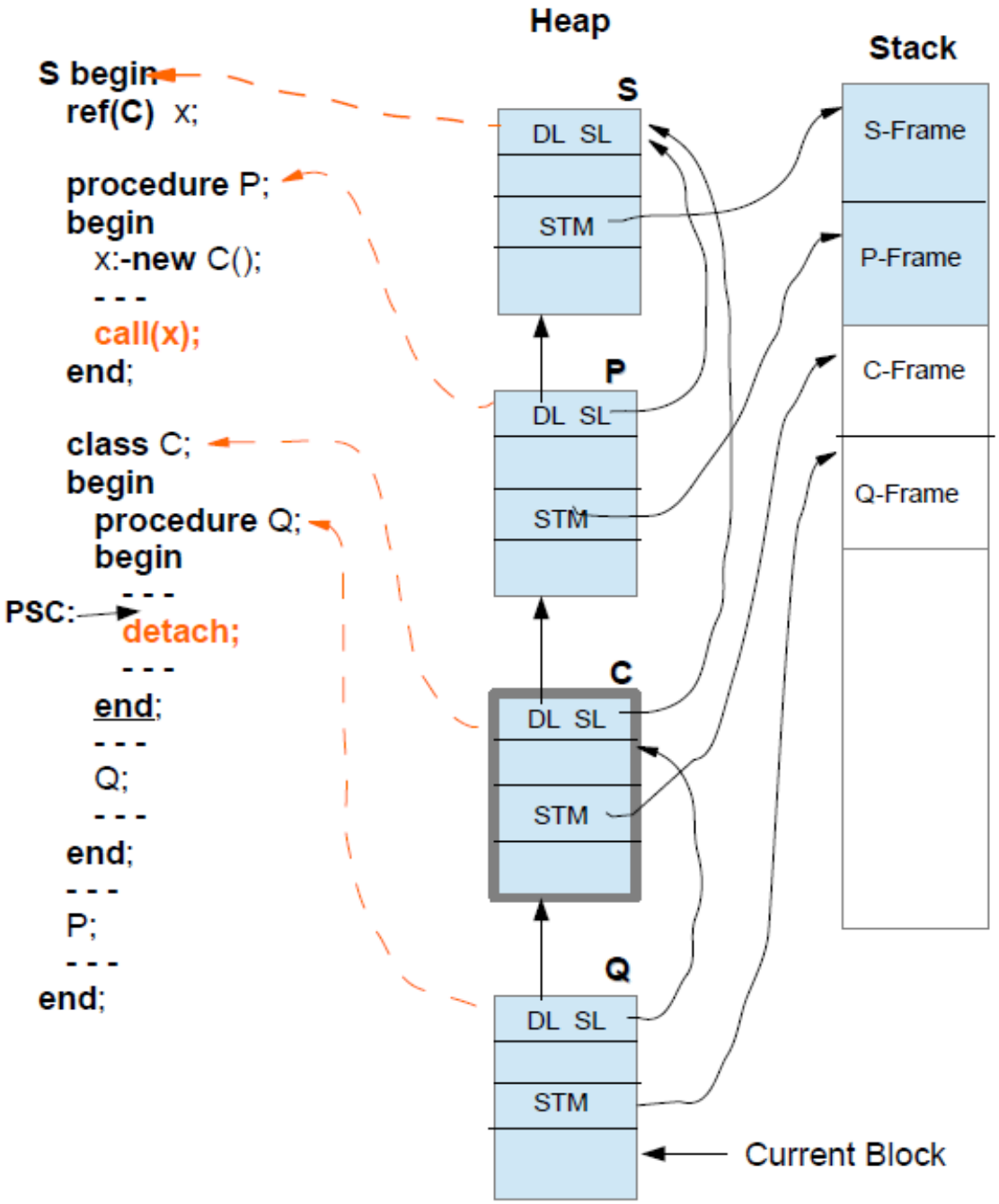


Fig. 5 Using LOOM: Just before Detach

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.

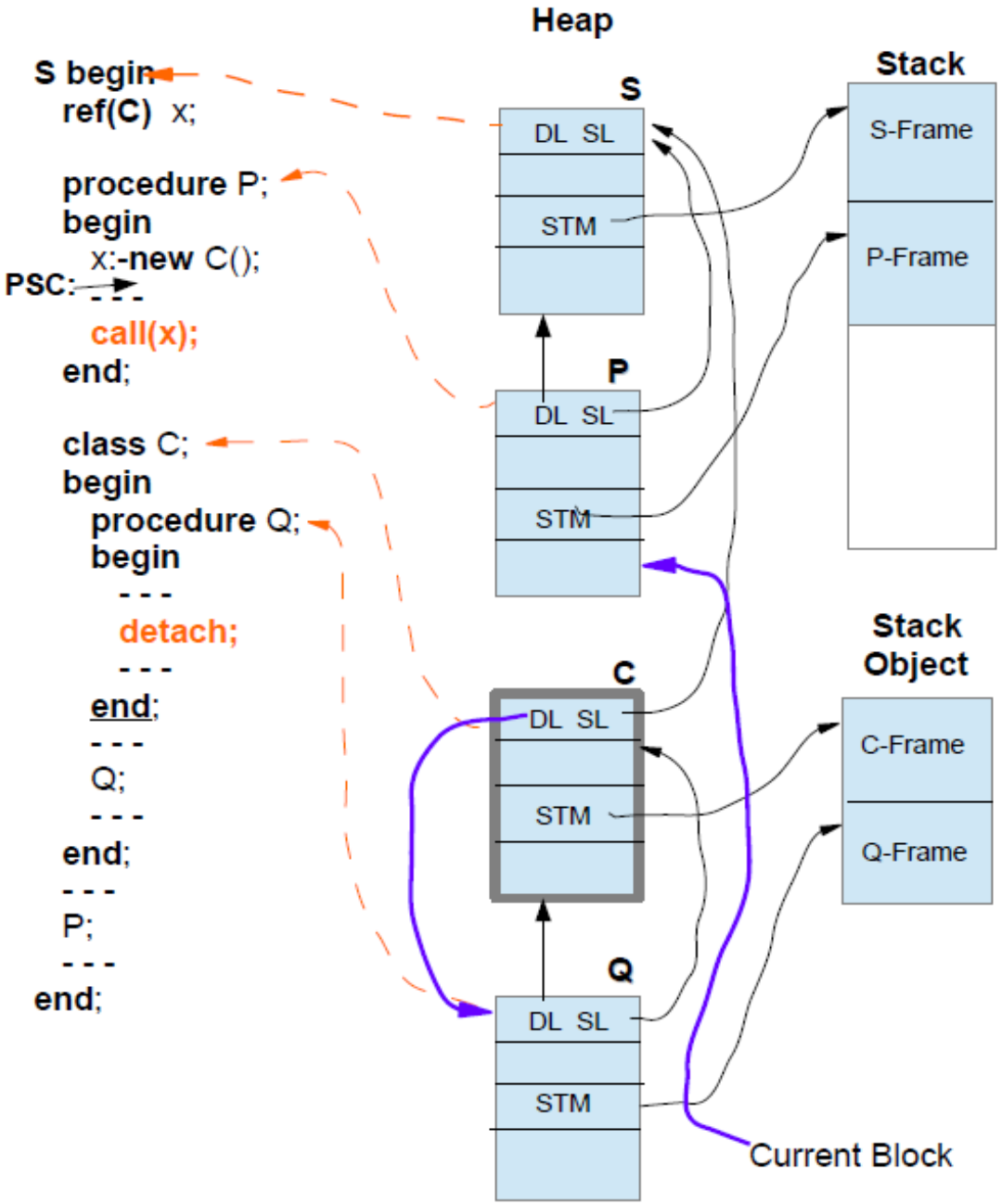


Fig. 6 Just after Detach – Dismounts part of Stack

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.

Project LOOM

(http://openjdk.java.net/projects/loom/)

Delimited continuations:

```
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()

    ....

}
```

A delimited continuation is a program object representing a computation that may be suspended and resumed

The Continuation class is implemented natively in Hotspot (except for scoping; that is implemented in Java).

Every continuation has its own stack.

From the perspective of the implementation, starting or continuing a continuation mounts it and its stack on the current thread – conceptually concatenating the continuation's stack to the thread's – while yielding a continuation unmounts or dismounts it.

Example: Co-routine with three active phases:

```
static final ContinuationScope scope=new ContinuationScope("TST");

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

This scheme is used as guideline for implementing coroutines in Simula

An application may look like this:

```
coroutine.run(); // Will perform Part 1.
System.out.println("Returns here after first DETACH(Yield)");
coroutine.run(); // Will perform Part 2.
System.out.println("Returns here after second DETACH(Yield)");
coroutine.run(); // Will perform Part 3.
System.out.println("Returns here after 'FINAL END'");
coroutine.run(); // IllegalStateException: Continuation terminated
```

We get the following output:

```
Part 1 - Statements
Returns here after first DETACH(Yield)
Part 2 - Statements
Returns here after second DETACH(Yield)
Part 3 - Statements
Returns here after 'FINAL END'
Exception: IllegalStateException: Continuation terminated
```

Example: Chain Execution (ala' Simulation)

```
static Continuation next;
static Continuation producer=null;
static Continuation consumer=null;

public static void EXECUTE(Continuation cont) {
      next=cont;
      while(next!=null) {
            Continuation n=next; next=null; n.run();
      }
}

public static void RESUME(Continuation cont) {
      next=cont; Continuation.yield(scope);
}

public static void example2() {

      producer=new Continuation(scope,new Runnable() {
            public void run() {
               System.out.println("Producer:Part 1 - Statements");
               RESUME(consumer);
               System.out.println("Producer:Part 2 - Statements");
               RESUME(consumer);
               System.out.println("Producer:Part 3 - Statements");
               RESUME(consumer);
      }});

      consumer=new Continuation(scope,new Runnable() {
            public void run() {
               System.out.println("Consumer:Part 1 - Statements");
               RESUME(producer);
               System.out.println("Consumer:Part 2 - Statements");
               RESUME(producer);
               System.out.println("Consumer:Part 3 - Statements");
      }});

      EXECUTE(producer); // Will start Producer:Part 1.
      System.out.println("End of program");
}
```

We get the following output:

```
Producer:Part 1 - Statements
Consumer:Part 1 - Statements
Producer:Part 2 - Statements
Consumer:Part 2 - Statements
Producer:Part 3 - Statements
Consumer:Part 3 - Statements
End of program
```

This scheme is used as guideline for implementing QPS in Simula

Do you want to try for yourself

- You can use LOOM early access:

http://openjdk.java.net/projects/loom/

- You can use my emulator:

https://github.com/MyhreAndersen/Continuations

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/

Thank you !