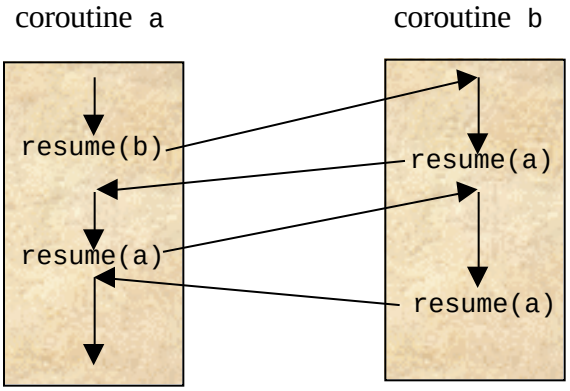
User's guide to `javaCoroutine`

This guide describes `javaCoroutine`, a Java package for coroutine sequencing. The package provides coroutine facilities similar to those provided by SIMULA [1][2]. The development of the package is described in [3].

Coroutines can be used to describe the solutions of algorithmic problems that are otherwise hard to describe [4]. Coroutines provide the means to organize the execution of a program as several sequential processes.

A coroutine may temporarily suspend its execution and another coroutine may be executed. A suspended coroutine may later be resumed at the point where it was suspended. This form of sequencing is called *alternation*. The figure below shows a simple example of alternation between two coroutines.



A coroutine program is composed of a collection of coroutines, which run in quasi-parallel with one another. Each coroutine is an object with its own execution-state, so that it may be suspended and resumed. A coroutine object provides the execution context for a method, called `body`, which describes the actions of the coroutine.

The `javaCoroutine` package provides the class `Coroutine` for writing coroutine programs. Coroutines can be created as instances of `Coroutine`-derived classes that override the abstract `body` method. As a consequence of creation, the current execution location of the coroutine is initialized at the start point of `body`.

Class `Coroutine` is sketched below.

```
public abstract class Coroutine {
    protected abstract void body();

    public static void resume(Coroutine c);
    public static void call(Coroutine c);
    public static void detach();

    public static Coroutine currentCoroutine();
    public static Coroutine mainCoroutine();
}
```

Control can be transferred to a coroutine `c` by one of two operations:

```
resume(c)
call(c)
```

Both operations cause `c` to resume its execution from its current execution location, which normally coincides with the point where it last left off.

The `call` operation furthermore establishes the currently executing coroutine as `c`'s caller. A subordinate relationship exists between the caller and the called coroutine. `c` is said to be *attached* to its caller.

The currently executing coroutine can relinquish control to its caller by means of the operation

```
detach()
```

The caller then resumes its execution from the point where it last left off.

The `currentCoroutine` method may be used to get a reference to the currently executing coroutine.

The first coroutine activated in a system of coroutines is denoted the *main coroutine*. If the main coroutine terminates, all other coroutines will terminate. A reference to this coroutine is provided through the `mainCoroutine` method.

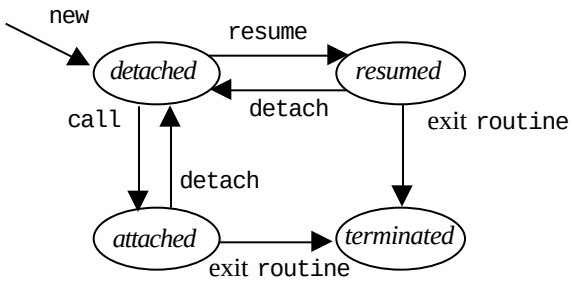
Below is shown a complete coroutine program. The program shows the use of the `resume` method for coroutine alternation as illustrated in the figure on page 1 .

```
import javaCoroutine.*;

public class CoroutineProgram extends Coroutine {
    Coroutine a, b;

    public void body() {
        a = new A();
        b = new B();
        resume(a);
        System.out.print("STOP1 ");
    }

    class A extends Coroutine {
        public void body() {
            System.out.print("A1 ");
            resume(b);
            System.out.print("A2 ");
            resume(b);
            System.out.print("A3 ");
        }
    }

    class B extends Coroutine {
        public void body() {
            System.out.print("B1 ");
            resume(a);
            System.out.print("B2 ");
            resume(a);
            System.out.print("B3 ");
        }
    }

    public static void main(String args[]) {
        resume(new CoroutineProgram());
        System.out.println("STOP2");
    }
}
```

Execution of this program produces the following (correct) output:

```
A1 B1 A2 B2 A3 STOP1 STOP2
```

A coroutine may be in one of four states of execution at any time: *attached*, *detached*, *resumed* or *terminated*. The figure below shows the possible state transitions of a coroutine.



A coroutine program consists of *components*. Each component is a chain of coroutines. The head of the component is a detached or resumed coroutine. The other coroutines are attached to the head, either directly or through other coroutines.

The main program corresponds to a detached coroutine, and as such it is the head of a component. This component is called the *main component.* The head of the main component is the main coroutine.

Exactly one component is operative at any time. Any non-operative component has an associated *reactivation point,* which identifies the program point where execution will continue if and when the component is activated (by `resume` or `call`).

When calling `detach` there are two cases:

- The coroutine is attached. In this case, the coroutine is detached, its execution is suspended, and execution continues at the reactivation point of the component to which the coroutine was attached.

- The coroutine is resumed. In this case, its execution is suspended, and execution continues at the reactivation point of the main component.

Termination of a coroutine's `body` method has the same effect as a `detach` call, except that the coroutine is terminated, not detached. As a consequence, it attains no reactivation point and it loses its status as a component head.

A call `resume(c)` causes the execution of the current operative component to be suspended and execution to be continued at the reactivation point of `c`. The call constitutes an error in the following cases:

`c` is `null`
`c` is attached
`c` is terminated

A call `call(c)` causes the execution of the current operative component to be suspended and execution to be continued at the reactivation point of `c`. In addition, `c` becomes attached to the calling component. The call constitutes an error in the following cases:

`c` is `null`
`c` is attached
`c` is resumed
`c` is terminated

A coroutine program using only `resume` and `detach` is said to use *symmetric* coroutine sequencing. If only `call` and `detach` are used, the program is said to use *semi-symmetric* coroutine sequencing.

On the next pages is shown small program for testing the javaCoroutine package. The program has been adapted from [5].

```
import javaCoroutine.*;

public class CoroutineTest extends Coroutine {
    CoroutineTest(char cmd) { command = cmd; }

    Coroutine a, b, c;
    char command;

    class A extends Coroutine {
        public void body() {
            System.out.print("a1"); detach();
        System.out.print("a2"); call(c = new C());
            System.out.print("a3"); call(b);
            System.out.print("a4"); detach();
        }
    }

    class B extends Coroutine {
        public void body() {
            System.out.print("b1"); detach();
            System.out.print("b2"); resume(c);
            System.out.print("b3"); detach();
        }
    }

    class C extends Coroutine {
        public void body() {
            System.out.print("c1"); detach();
            System.out.print("c2\n");
            System.out.println("==> " + command);
            if (command == 'r')
                resume(a);
            else if (command == 'c')
                call(a);
            else
                detach();
            System.out.print("c3"); detach();
            System.out.print("c4");
        }
    }

    public void body() {
        System.out.print("m1"); call(a = new A());
        System.out.print("m2"); call(b = new B());
        System.out.print("m3"); resume(a);
        System.out.print("m4"); resume(c);
        System.out.print("m5\n");
    }
```

```
    public static void main(String args[]) {
        resume(new CoroutineTest('r'));
        resume(new CoroutineTest('c'));
        resume(new CoroutineTest('x'));
    }
}

/*

Expected output:

m1a1m2b1m3a2c1a3b2c2
==> r
b3a4m4c3m5

m1a1m2b1m3a2c1a3b2c2
==> c
b3a4c3m4c4m5

m1a1m2b1m3a2c1a3b2c2
==> x
m4c3m5

*/
```

References

1. G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug & K. Nygaard,
   SIMULA BEGIN,
   Studentlitteratur (1974).

2. *Programspråk – SIMULA*, SIS,
   Svensk Standard SS 63 61 14 (1987).

3. K. Helsgaun,
   Discrete Event Simulation in Java,
   Datalogiske skrifter, No. 89, Roskilde University (2000).

4. C. D. Marlin,
   Coroutines,
   Lecture Notes in Computer Science (1980).

5. H. B. Hansen,
   SIMULA - et objektorienteret programmeringssprog,
   Kompendium, Roskilde Universitetscenter (1990).