Discrete Event Simulation in Java

Keld Helsgaun
E-mail: keld@ruc.dk

Department of Computer Science
Roskilde University
DK-4000 Roskilde, Denmark

Abstract

This report describes `javaSimulation`, a Java package for process-based discrete event simulation. The facilities of the package are based on the simulation facilities provided by the programming language SIMULA. The design of the package follows the SIMULA design very closely. The package is easy to use and relatively efficient. In addition, Java packages for co-routine sequencing, event-based simulation and activity-based simulation are presented.

Keywords: simulation, Java, process, discrete event, coroutine

1. Introduction

The Java platform comes with several standard packages. No package, however, is provided for discrete event simulation. This is unfortunate since discrete event simulation constitutes an important application area of object oriented programming. This fact has convincingly been demonstrated by SIMULA, one of the first object-oriented programming languages [1][2][3].

SIMULA provides the standard class SIMULATION, a very powerful tool for discrete event simulation. A simulation encompasses a set of interacting *processes.* A process is an object associated with a sequence of activities ordered logically in simulated time. Each process has its own life cycle and may undergo active and inactive phases during its lifetime.

Processes represent the active entities in the real world, e.g., customers in a supermarket. Thus, the process concept may be used to describe systems in a natural way.

This report describes `javaSimulation`, a Java package for process-based discrete event simulation. The package may be seen as an implementation of class SIMULATION in Java. In addition to the simulation facilities, the package also includes the facilities for list manipulation and random number drawing as found in SIMULA.

When designing the package, great emphasis has been put into following the SIMULA design as closely as possible. The advantage of this approach is that the semantics of facilities are well-known and thoroughly tested through many years' use of SIMULA. A SIMULA user should easily learn to use the package.

The rest of this report is structured as follows.

Chapter 2 provides a short introduction to discrete event simulation by means of a concrete example, a car wash simulation. The example is used to demonstrate the use of three different approaches to discrete event simulation: *event-based, activity-based* and *process-based*. In relation to these approaches several packages have been developed. These packages are described from the user's point of view and their use is demonstrated by means of the car wash example.

Implementing a process-based simulation package in Java is not a trivial task. A process must be able to suspend its execution and have it resumed at some later time. In other words, a process should be able to act like a coroutine. Chapter 3 describes the development of a package, `javaCoroutine`, for coroutine sequencing in Java. The package provides the same coroutine facilities as can be found in SIMULA. Its implementation is based on Java's threads.

This coroutine package is then used in Chapter 4 for the implementation of a package for process-based simulation, `javaSimulation`.

`javaSimulation` is evaluated in Chapter 5, and some conclusions are made in Chapter 6.

The appendices contain Java source code and documentation.

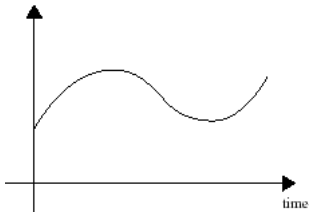
2. Discrete event simulation in Java

Simulation may be defined as the experimentation with a model in order to obtain information about the dynamic behavior of a system. Instead of experimenting with the system, the experiments are performed with a model of the system. Simulation is typically used when experimentation with the real system is too expensive, too dangerous, or not possible (e.g., if the real system does not exist).

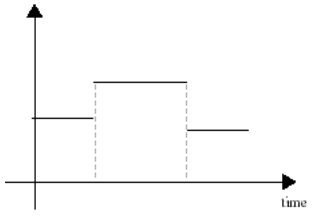
The system components chosen for inclusion in the model are termed *entities*. Associated with each entity are zero or more *attributes* that describe the state of the entity. The collection of all these attributes at any given time defines the *system state* at that time.

There are three categories of simulation models, defined by the way the system state changes:

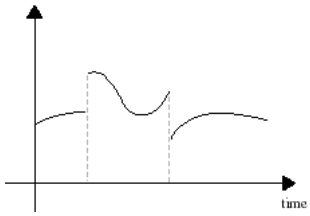
- *Continuous*: the state varies continuously with time. Such systems are usually described by sets of differential equations.



- *Discrete*: the state changes only at discrete instances of time (event times).



· *Combined continuous and discrete*: the state changes instantaneously at event times; in between consecutive event times the system state may vary continuously [4].



In this report we will only consider so-called *discrete event simulation.* In discrete event simulation the model is discrete, and the simulated clock always jumps from one event time to the most imminent event time. At each event time the corresponding action (state change) is performed, and simulated time is advanced to the next time when some action is to occur. Thus, discrete event simulation assumes that nothing happens between successive state changes.

In order to make the following description easier to comprehend, a concrete simulation example will now be presented.

2.1 The car wash problem

This example has been taken from [1].

A garage owner has installed an automatic car wash that services cars one at a time. When a car arrives, it goes straight into the car wash if this is idle; otherwise, it must wait in a queue. The car washer starts his day in a tearoom and return there each time he has no work to do. As long as cars are waiting, the car wash is in continuous operation serving on a first-come, first-served basis. All cars that have arrived before the garage closes down are washed.

Each service takes exactly 10 minutes. The average time between car arrivals has been estimated at 11 minutes.

The garage owner is interested in predicting the maximum queue length and average waiting time if he installs one more car wash and employs one more car washer.

2.2 Three approaches for discrete event simulation

There are basically three approaches that can be used for discrete event simulation: the *event-based,* the *activity-based* and the *process-based* approach [5].

(1) The event-based approach

In the event-based approach the model consists of a collection of events. Each event models a state change and is responsible for scheduling other events that depend on that event.

Each event has associated an event time and some actions to be executed when the event occurs.

In the car wash problem the *arrival of a car* is an example of an event. Actions associated with this event are the inclusion of the car into the waiting line and the scheduling of the next car arrival.

Event-based simulation is the simplest and most common implementation style of discrete event simulation because it can be implemented in any programming language.

(2) The activity-based approach

In the activity-based approach the model consists of a collection of activities. Each activity models some time-consuming action performed by an entity.

Each activity has associated a starting condition, some actions to be executed when the activity starts, the duration of the activity, and some actions to be executed when the activity finishes.

In the car wash problem the *washing of a car* is an example of an activity. The condition for starting this activity is that one of car washers is idle and the waiting line is not empty. When the activity starts, an idle car washer is removed from the tearoom, and the first waiting car is removed from the waiting line. The duration of the activity is 10 units of simulated time. When it ends, the car washer is put back into the tearoom.

Whilst the activity approach is relatively easy to understand, it normally suffers from poor execution efficiency compared to the event-based approach.

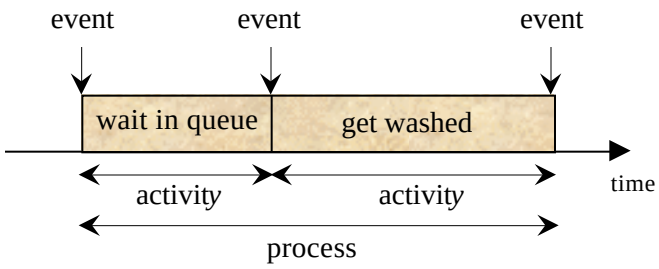
(3) The process-based approach

In the process-based approach the model consists of a collection of processes. Each process models the life cycle of an entity and is a sequence of logically related activities ordered in time.

In the car wash problem a *car* is an example of a process. Each car performs the following sequence of activities: wait in queue, get washed.

Since processes resemble objects in the real world, process-based simulation is often easy to understand. Implementation, however, is not easy and execution efficiency may be poor if the implementation is not done properly.

The figure below illustrates the relation between the concepts event, activity and process.



In the remaining part of this chapter we will show how the car wash problem can be solved in Java using each of the three simulation approaches.

2.3 Solving the car wash problem by event-based simulation

To provide a simple tool for event-based simulation a small Java package called `simulation.event` has been developed.

When using this package the events of a model are described in one or more subclasses of class `Event`. An outline of this class is shown below.

```
public abstract class Event {
    protected abstract void actions();

    public void schedule(double evTime);
    public void cancel();
    public static double time();
    public static void runSimulation(double period);
    public static void stopSimulation();
}
```

The `actions` method represents the actions associated with the event. These actions will be executed when the event occurs.

An event is scheduled to occur at a specified point in simulated time by calling its `schedule` method. The desired event time is passed as an argument to the method.

A scheduled event may be cancelled by calling its `cancel` method.

The `time` method returns the current simulated time.

The `runSimulation` method is used to run a simulation for a specified period of simulated time. Time will start at `0` and jump from event time to event time until either this period is over, there are no more scheduled events, or the `stopSimulation` method is called.

Below we will show how the package may be used for solving the car wash problem. For this purpose we will exploit two other packages, `simset` and `random`. The `simset` package provides the same facilities for list manipulation as class SIMSET of SIMULA. The `random` package provides all of SIMULA's methods for drawing random numbers. The source code and documentation of these two packages can be found in the appendices A, B, C and D.

We will represent the entities of the system (car washers and cars) by the two classes `CarWasher` and `Car`.

```
class CarWasher extends Link {}

class Car extends Link {
    double entryTime = time();
}
```

Both classes extend the `Link` class from the `simset` package. This has the effect that any object of these classes is capable of being a member of a queue. Thus, a `CarWasher` may be put into a queue of idle car washers, and a `Car` may be put into a line of waiting cars.

The attribute `entryTime` of the `Car` class is used for each `Car` to record the time it entered the garage.

The two queues are defined using the `Head` class of the `simset` package:

```
Head tearoom = new Head();
Head waitingLine = new Head();
```

Next, we define the following events:

- A car arrives
- A car washer starts washing a car
- A car washer finishes washing a car

These events are specified in three subclasses of class `Event`.

A car arrival is described in class `CarArrival` as shown below.

```
class CarArrival extends Event {
    public void actions() {
        if (time() <= simPeriod) {
            Car theCar = new Car();
            theCar.into(waitingLine);
            int qLength = waitingLine.cardinal();
            if (maxLength < qLength)
                maxLength = qLength;
            if (!tearoom.empty())
                new StartCarWashing().schedule(time());
            new CarArrival().schedule(
                time() + random.negexp(1/11.0));
        }
    }
}
```

The `actions` method specifies what happens when a car arrives at the garage. Unless the garage has closed, a `Car` is created and put into the waiting line. Next, if any car washer is idle (is waiting in the tearoom), the starting of a wash is scheduled to occur immediately. Finally the next car arrival is scheduled using the `random` package. Here, it is assumed that the number of minutes between arrivals is distributed according to a negative exponential distribution with a mean of 11 minutes.

Actually, it is not necessary for a `CarArrival` object to create a new `CarArrival` object before it finishes. It could simply reschedule itself by executing the following statement

```
schedule(time() + random.negexp(1.0/11.0));
```

The starting of a car wash is described in class `StartCarWash` shown below.

```
class StartCarWashing extends Event {
    public void actions() {
        CarWasher theCarWasher =
            (CarWasher) tearoom.first();
        theCarWasher.out();
        Car theCar = (Car) waitingLine.first();
        theCar.out();
        new StopCarWashing(theCarWasher, theCar).
            schedule(time() + 10);
    }
}
```

When this event takes place, an idle car washer is removed from the tearoom, and the first waiting car is removed from the waiting line.

A car wash takes 10 minutes. Accordingly, a `StopCarWashing` event is scheduled to occur 10 time units later.

Class `StopCarWashing` is shown below.

```
class StopCarWashing extends Event {
    CarWasher theCarWasher;
    Car theCar;

    StopCarWashing(CarWasher cw, Car c)
      { theCarWasher = cw; theCar = c; }

    public void actions() {
        theCarWasher.into(tearoom);
        if (!waitingLine.empty())
            new StartCarWashing().schedule(time());
        noOfCustomers++;
        throughTime += time() - theCar.entryTime;
    }
}
```

When a car washer has finished washing a car, he goes into the tearoom. However, if there are cars waiting to be washed, a new `StartCarWashing` event is scheduled to occur at once. So he will have a break, unless another idle car washer can do the job.

In order to make a report when the simulation has ended the following variables are updated:

`noOfCustomers`: the number of cars through the system
`throughTime`: the sum of elapsed times of the cars

The simulation program is shown below (excluding the classes described above). Note the use of inner classes.

```
import simulation.event.*;
import simset.*;
import random.*;

public class CarWashSimulation extends Simulation {
    double simPeriod = 200;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    int noOfCustomers, maxLength;
    double throughTime;

    CarWashSimulation(int n) {
        noOfCarWashers = n;
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        new CarArrival().schedule(0);
        runSimulation(simPeriod + 1000000);
        report();
    }

    void report() { ... }

    class CarWasher extends Link {}
    class Car extends Link { ... }

    class CarArrival extends Event { ... }
    class StartCarWashing extends Event { ... }
    class StopCarWashing extends Event { ... }

    public static void main(String args[]) {
        new CarWashSimulation(1);
        new CarWashSimulation(2);
    }
}
```

The `main` method of the program performs two simulations, the first with one car washer, the second with two car washers.

When a `CarWashSimulation` object is created, all car washers are put into the tearoom, the first car is scheduled to arrive immediately, and the system is simulated for a specified period of time.

The parameter passed to the `runSimulation` method has to do with finishing-off the simulation. When `simPeriod` time units have passed the garage closes and no more cars arrive, but all cars in the queue at that time will eventually be served. In the program `simPeriod` has been set to `200`.

When a simulation has finished, the method `report` is called in order to write statistics generated by the model. This method appears as follows:

```
void report() {
    System.out.println(noOfCarWashers
        + " car washer simulation");
    System.out.println("No.of cars through the system = "
        + noOfCustomers);
    java.text.NumberFormat fmt =
        java.text.NumberFormat.getNumberInstance();
    fmt.setMaximumFractionDigits(2);
    System.out.println("Av.elapsed time = "
        + fmt.format(throughTime/noOfCustomers));
    System.out.println("Maximum queue length = "
        + maxLength + "\n");
}
```

A run of the program produced the following output:

```
1 car washer simulation
No.of cars through the system = 22
Av.elapsed time = 30.22
Maximum queue length = 5

2 car washer simulation
No.of cars through the system = 22
Av.elapsed time = 10.74
Maximum queue length = 1
```

The implementation of this package is straightforward. All scheduled events are held in a list (SQS) ordered by their associated event times. As long as there are more scheduled events, and the simulation period is not over, the first event of SQS is removed, time is updated to this event time, and the actions of this event are executed.

This algorithm is implemented in the `runSimulation` method shown below.

```
public static void runSimulation(double period) {
    while (SQS.suc != SQS) {
        Event ev = SQS.suc;
        if ((time = ev.eventTime) > period) break;
        ev.cancel();
        ev.actions();
    }
    stopSimulation();
}
```

The complete source code of the package is provided in Appendix E.

2.4 Solving the car wash problem by activity-based simulation

The activity-based approach tries to capture the notion of connected start and finish events, clustering descriptions of actions to be executed at the start and finish of some time-consuming activity. The programmer must specify conditions under which such clusters of actions will occur.

Every activity should be associated with a start *condition*, a specification of the *duration* of the activity, and some start and finish actions. The *start actions* of an activity will be executed as soon as its associated condition becomes true. The *finish actions* will be executed when the activity ends (after a time period equal to the duration of the activity).

To provide a simple tool for activity-based simulation, a small Java package called `simulation.activity` has been developed.

When using this package the activities of a model are described in one or more subclasses of class `Activity`. An outline of this class is given below.

```
public abstract class Activity {
    protected abstract boolean condition();
    protected abstract void startActions();
    protected abstract double duration();
    protected abstract void finishActions();

    public static double time();
    public static void runSimulation(double period);
    public static void stopSimulation();
}
```

In order to specify an activity, all four abstract methods should be overridden in subclasses of class `Activity`.

The `time` method returns the current simulated time.

The `runSimulation` method is used to run a simulation for a specified period of simulated time. Time will start at `0` and jump from event time to event time until either this period is over, there are no more actions to be executed, or the `stopSimulation` method is called.

Below we will show how the package may be used for solving the car wash problem.

Queues, car washers and cars are represented as follows:

```
Head tearoom = new Head();
Head waitingLine = new Head();

class CarWasher extends Link {}

class Car extends Link {
    double entryTime = time();
}
```

The dynamics of the system may be described by the following activities involving the passing of time:

- Washing a car
- Waiting for the next car to arrive

These activities are specified in two subclasses of class `Activity`.

The washing of a car is described in class `CarWashing` shown below.

```
class CarWashing extends Activity {
    Car theCar;
    CarWasher theCarWasher;

    CarWashing(Car c) { theCar = c; }

    public boolean condition() {
        return theCar == (Car) waitingLine.first() &&
               !tearoom.empty();
    }

    public void startActions() {
        theCar.out();
        theCarWasher = (CarWasher) tearoom.first();
        theCarWasher.out();
    }

    public double duration() {
        return 10;
    }

    public void finishActions() {
        theCarWasher.into(tearoom);
        noOfCustomers++;
        throughTime += time() - theCar.entryTime;
    }
}
```

In order for the washing of a car to start, the car must be in front of the waiting line and there must be an idle car washer (i.e., the tearoom must not be empty). When the washing activity is started the car is removed from the waiting line and one of the idle car washers is removed from the tearoom. The wash takes 10 minutes after which the car washer goes back to the tearoom.

The class `CarArrival` shown below models the time-passing activity of waiting for the next car to arrive.

```
class CarArrival extends Activity
    public boolean condition() {
        return true;
    }

    public void startActions() {
        Car theCar = new Car();
        theCar.into(waitingLine);
        new CarWashing(theCar);
        int qLength = waitingLine.cardinal();
        if (maxLength < qLength)
            maxLength = qLength;
    }

    public double duration() {
        return random.negexp(1/11.0);
    }

    public void finishActions() {
        if (time() <= simPeriod)
            new CarArrival();
    }
}
```

A `CarArrival` activity starts immediately when created. This is accomplished by letting the `condition` method return `true`. The activity starts by inserting a new car as the last member of the waiting line and creates a `CarWashing` activity for this car. After a time period, chosen at random from a negative exponential distribution, the activity finishes by generating a new `CarArrival` activity.

The simulation program is shown below (excluding the classes described above).

```
import simulation.activity;
import simset.*;
import random.*;

public class CarWashSimulation extends Simulation {
    int noOfCarWashers;
    double simPeriod = 200;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    int noOfCustomers, maxLength;
    double throughTime;

    CarWashSimulation(int n) {
        noOfCarWashers = n;
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        new CarArrival();
        runSimulation(simPeriod + 1000000);
        report();
    }

    void report() { ... }

    class CarWasher extends Link {}
    class Car extends Link { ... }

    class CarWashing extends Activity { ... }
    class CarArrival extends Activity { ... }

    public static void main(String args[]) {
        new CarWashSimulation(1);
        new CarWashSimulation(2);
    }
}
```

The `main` method of the program performs two simulations, the first with one car washer, the second with two car washers. Each simulation is performed by the creation of an object of class `CarWashSimulation` (a subclass of class `Simulation`).

The program produces the same output as the program in Section 2.3.

The implementation of this package is straightforward. All activities waiting to start are held in a list, `waitList`. All activities that are scheduled to finish are held in a list, `SQS`, ordered by their associated finish times. At each event time, the wait list is examined to see whether any activity is eligible to start. If so, the activity is removed from the list, its start actions are executed, and a finish event is scheduled to occur when the activity finishes. When no more activities are eligible to start, time advances to the next imminent event, and the associated finish actions are executed. This continues until the simulation ends.

This algorithm is implemented in the `runSimulation` method shown below.

```
public static void runSimulation(double period) {
    while (true) {
        for (Activity a = waitList.suc;
             a != waitList;
             a = a.suc) {
            if (a.condition()) {
                a.cancel();
                a.schedule(time + a.duration());
                a.startActions();
                a = waitList;
            }
        }
        if (SQS.suc == SQS)
            break;
        Activity a = SQS.suc;
        time = a.eventTime;
        a.cancel();
        if (time > period)
            break;
        a.finishActions();
    }
    stopSimulation();
}
```

The complete source code of the package is provided in Appendix G.

2.5 Solving the car wash problem by mixed event-activity-based simulation

The different simulation approaches are not mutually exclusive. Mixed approaches may also be used.

In this section we will demonstrate how the event-approach and ingredients of the activity-approach may be combined into one single approach. For this purpose we will extend the event concept with the following definitions:

> A *time event* is an event scheduled to occur at a specified point in time.
>
> A *state event* is an event scheduled to occur when the state of the system fulfills a specified condition (a so-called *state condition*).

These two event types are used to model the dynamics of a system.

In order to provide a tool for using this simulation approach a small Java package called `simulation.events` has been developed.

When using this package the events of a model are described in one or more subclasses of the classes `TimeEvent` and `StateEvent`, which themselves are subclasses of the abstract class `Event`.

The class hierarchy is shown below.

```
public abstract class Event {
    protected abstract void actions();

    public static double time();
    public static void runSimulation(double period);
    public static void stopSimulation();
}
```

```
public abstract class TimeEvent extends Event {
    public void schedule(double evTime);
}
```

```
public abstract class StateEvent extends Event {
    protected abstract boolean condition();

    public void schedule();
}
```

The meaning and usage of the methods should be clear from the previous sections. Below we will show how the package can be used for solving the car wash problem.

Queues, car washers and cars are specified as in the previous two sections, i.e.:

```
Head tearoom = new Head();
Head waitingLine = new Head();

class CarWasher extends Link {}

class Car extends Link {
    double entryTime = time();
}
```

We will use the same three events as were used in the event-based approach:

- A car arrives (`carArrival`)
- A car washer starts washing a car (`startCarWashing`)
- A car washer finishes washing a car (`stopCarWashing`)

However, in this mixed approach we must also specify which of these events are time events, and which are state events.

It is easy to see that the arrival of a car and the finishing of a car wash are both time events. On the other hand, the starting of a car wash must be a state event, since its time of occurrence can not be predetermined.

This leads to the following class declarations for the three event types:

```
class CarArrival extends TimeEvent {
    public void actions() {
        if (time() <= simPeriod) {
            Car theCar = new Car();
            theCar.into(waitingLine);
            int qLength = waitingLine.cardinal();
            if (maxLength < qLength)
                maxLength = qLength;
            new StartCarWashing(theCar).schedule();
            schedule(time() +  random.negexp(1/11.0));
        }
    }
}
```

```
class StartCarWashing extends StateEvent {
    Car theCar;

    CarWashing(Car c) { theCar = c; }

    public boolean condition() {
        return theCar == (Car) waitingLine.first() &&
               !tearoom.empty();
    }

    public void actions() {
        theCar.out();
        CarWasher theCarWasher =
            (CarWasher) tearoom.first();
        theCarWasher.out();
        new StopCarWashing(theCarWasher, theCar).
            schedule(time() + 10);
    }
}
```

```
class StopCarWashing extends TimeEvent {
    CarWasher theCarWasher;
    Car theCar;

    StopCarWashing(CarWasher cw, Car c)
      { theCarWasher = cw; theCar = c; }

        public void actions() {
            noOfCustomers++;
            throughTime += time() - theCar.entryTime;
            theCarWasher.into(tearoom);
        }
    }
}
```

The simulation program follows the same pattern as used in the previous two sections. See Appendix J for the complete source code.

The implementation of this package is straightforward. All scheduled state events are held in a list, `waitList`, and all scheduled time events are held in a list, `SQS`, ordered by their associated event times. At each event time, the wait list is examined to see whether any state event has its condition fulfilled. If so, the event is removed from the list and its actions are executed. When no more state events occur, time advances to the next imminent time event, and the associated actions of this time event are executed This continues until the simulation ends.

This algorithm is implemented in the `runSimulation` method as shown below.

```
public static void runSimulation(double period) {
    while (true) {
        for (StateEvent a = (StateEvent) waitList.suc;
             a != waitList;
             a = (StateEvent) a.suc) {
            if (a.condition()) {
                a.cancel();
                a.actions();
                a = waitList;
            }
        }
        if (SQS.suc == SQS)
            break;
        TimeEvent ev = (TimeEvent) SQS.suc;
        time = ev.eventTime;
        ev.cancel();
        if (time > period)
            break;
        ev.actions();
    }
    stopSimulation();
}
```

The complete source code of the package is provided in Appendix I.

2.6 Solving the car wash problem by process-based simulation

The process-based approach is often the easiest to use. In this approach the active entities of a system are modeled in a very natural way. A process describes the life cycle of an entity of the system. Any process is associated with actions to be performed by the process during its lifetime. A process may be suspended temporarily and may be resumed later from where it left off.

In order to provide a tool for the process-based approach a Java package called `javaSimulation` has been developed.

When using this package the processes of a model are described in one or more subclasses of class `Process`. An outline of this class is given below. In this outline only facilities that are actually used in solving the car wash problem have been included. A more comprehensive version is given at the end of this section.

```
public abstract class Process extends Link {
    protected abstract void actions();

    public static double time();
    public static void activate(Process p);
    public static void hold(double t);
    public static void passivate();
    public static void wait(Head q);
}
```

Since `Process` is a subclass of `Link`, each process has the capability of being a member of a two-way list. This is useful, for example, when processes must wait in a queue. The `javaSimulation` package includes all the list manipulation facilities of the `simset` package.

The `actions` method represents the actions associated with a process.

The `time` method returns the current simulated time.

The `activate` method is used to make a specified process start executing its actions.

The `hold` method suspends the execution of the calling process for a specified period of time.

The `passivate` method suspends the execution of the calling process for an unknown period of time. Its execution may later be resumed by calling `activate` with the process as argument.

The `wait` method suspends the calling process and adds it to a queue.

Below we will show how the package can be used for solving the car wash problem.

First, the processes are identified and their actions are described in subclasses of class `Process` by overriding the `actions` method

A car washer is described in the following subclass of `Process`:

```
class CarWasher extends Process {
    public void actions() {
        while (true) {
            out();
            while (!waitingLine.empty()) {
                Car served =
                    (Car) waitingLine.first();
                served.out();
                hold(10);
                activate(served);
            }
            wait(tearoom);
        }
    }
}
```

The actions of a car washer are contained in an infinite loop (the length of the simulation is supposed to be determined by the main program). Each time a car washer is activated, he leaves the tearoom and starts serving the cars in the waiting line. He takes the first car out of the waiting line, washes it for ten minutes before he activates the car. The car washer will continue servicing, as long as there are cars waiting in the queue. If the waiting line becomes empty, he returns to the tearoom and waits.

A car may be described by the following subclass of `Process`:

```
class Car extends Process {
    public void actions() {
        double entryTime = time();
        into(waitingLine);
        int qLength = waitingLine.cardinal();
        if (maxLength < qLength)
            maxLength = qLength;
        if (!tearoom.empty())
            activate((CarWasher) tearoom.first());
        passivate();
        noOfCustomers++;
        throughTime += time() - entryTime;
    }
}
```

On arrival each car enters the waiting line and, if the tearoom is not empty, it activates the idle car washer in the tearoom. The car then passively waits until it has been washed. When the car has been washed (signaled by an activation by the car washer), it leaves the system.

The following subclass of Process is used to make the cars arrive at the garage with an average inter-arrival time of 11 minutes:

```
class CarGenerator extends Process {
    public void actions() {
        while (time() <= simPeriod) {
            activate(new Car());
            hold(random.negexp(1/11.0));
        }
    }
}
```

All random drawing facilities of the random package have been included in the `javaSimulation` package. In the present simulation the inter-arrival times of the cars are distributed according to a negative exponential distribution.

The simulation program is shown below (excluding the classes described above).

```
import javaSimulation.*;
import javaSimulation.Process;

public class CarWashSimulation extends Process {
    int noOfCarWashers;
    double simPeriod = 200;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    double throughTime;
    int noOfCustomers, maxLength;

    CarWashSimulation(int n) {
        noOfCarWashers = n;
    }

    public void actions() {
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        activate(new CarGenerator());
        hold(simPeriod + 1000000);
        report();
    }

    void report() { ... }

    class Car extends Process { ... }
    class CarWasher extends Process { ... }
    class CarGenerator extends Process { ... }

    public static void main(String args[]) {
        activate(new CarWashSimulation(1));
        activate(new CarWashSimulation(2));
    }
}
```

The program imports all classes of the `javaSimulation` package. Note, however, that class `Process` must be imported explicitly in order to avoid the name conflict caused by the co-existence of the class `Process` of the `java.lang` package.

The `main` method of the program performs two simulations, the first with one car washer, the second with two car washers.

Each simulation is performed by the creation and activation of an object of class `CarWashSimulation`.

Class `CarWashSimulation` is a subclass of `Process`. Thus, the `actions` method of the class may be used describe the actions associated with the main program. Here, a number of car washers and a car generator are activated before the main program waits for the simulation to finish. The variable `simPeriod` denotes the total opening time of the garage (200 minutes). All cars that have arrived before the garage closes are washed.

Before a simulation finishes, the `report` method is called. The method is identical to the `report` method given in Section 2.3. It prints the number of cars washed, the average elapsed time (wait time plus service time), and the maximum queue length. The program produces the same output as the programs of the previous sections.

The design of the `javaSimulation` package follows very closely the design of the built-in package for discrete event simulation in SIMULA, class SIMULATION.

A program is composed of a set of processes that undergo scheduled and unscheduled phases. When a process is scheduled, it has an event time associated with it. This is the time at which its next active phase is scheduled to occur. When the active phase of a process ends, it may be rescheduled, or descheduled (either because all its actions have been executed, or the time of its next active phase is not known). In either case, the scheduled process with the least event time is resumed.

The currently active process always has the least event time associated with it. This time, the simulation time, moves in jumps to the event time of the next scheduled process.

Scheduled events are contained in an event list. The processes are ordered in accordance with increasing event times. The process at the front of the event list is always the one, which is active. Processes not in the event list are either terminated or passive.

At any point in simulation time, a process can be in one (and only one) of the following four states:

(1) *active*: the process is at the front of the event list. Its actions are being executed

(2) *suspended*: the process is in the event list, but not at the front

(3) *passive*: the process is not in the event list and has further actions to execute

(4) *terminated*: the process is not in the event list and has no further actions to execute.

All the public parts of the `Process` class are shown in the class outline below.

```
public abstract class Process extends Link {
    protected abstract void actions();

    public static final Process current();
    public static final double time();
    public static final void hold(double t);
    public static final void wait(Head q);
    public static final void cancel(Process p);
    public static final Process main();

    public static final At     at;
    public static final Delay  delay;
    public static final Before before;
    public static final After  after;
    public static final Prior  prior;

    public static final void activate(Process p);
    public static final void activate(Process p,
                              At at, double t);
    public static final void activate(Process p,
                              Delay delay, double t);
    public static final void activate(Process p,
                              At at, double t, Prior prior);
    public static final void activate(Process p,
                              Delay d, double t, Prior prior);
    public static final void activate(Process p1,
                              Before before, Process p2);
    public static final void activate(Process p1,
                              After after, Process p2);

    public static final void reactivate(Process p);
    public static final void reactivate(Process p,
                              At at, double t);
    public static final void reactivate(Process p,
                              Delay delay, double t);
    public static final void reactivate(Process p,
                              At at, double t, Prior prior);
    public static final void reactivate(Process p,
                              Delay d, double t, Prior prior);
    public static final void reactivate(Process p1,
                              Before before, Process p2);
    public static final void reactivate(Process p1,
                              After after, Process p2);

    public final boolean idle();
    public final boolean terminated();
    public final double evTime();
    public final Process nextEv();
}
```

Below is given a short description of each of the methods.

`current()` returns a reference to the `Process` object at the front of the event list (the currently active process).

`time()` returns the current simulation time.

`hold(t)` schedules `Current` for reactivation at `time() + t`.

`passivate()` removes `current()` from the event list and resumes the actions of the new `current()`.

`wait(q)` includes `current()` into the two-way list `q`, and then calls `passivate()`.

`cancel(p)` removes the process `p` from the event list. If `p` is currently active or suspended, it becomes passive. If `p` is a passive or terminated process or `null`, the call has no effect.

It is desirable to have the main program participating in the simulation as a process. This is achieved by an impersonating `Process` object that can be manipulated like any other `Process` object. This object, called the *main process*, is the first process activated in a simulation.

`main()` returns a reference to the main process.

There are seven ways to activate a currently *passive* process:

`activate(p)`: activates process `p` at the current simulation time.

`activate(p1, before, p2)`: positions process `p1` in the event list *before* process `p2`, and gives it the same event time as `p2`.

`activate(p1, after, p2)`: positions process `p1` in the event list *after* process `p2`, and gives it the same event time as `p2`.

`activate(p, at, t)`: the process `p` is inserted into the event list at the position corresponding to the event time specified by `t`. The process is inserted *after* any processes with the same event time which may already be present in the list.

`activate(p, at, t, prior)`: the process `p` is inserted into the event list at the position corresponding to the event time specified by `t`. The process is inserted *before* any processes with the same event time which may already be present in the list.

`activate(p, delay, t)`: the process `p` is activated after a specified delay, `t`. The process is inserted in the event list with the new event time, and *after* any processes with the same simulation time which may already be present in the list.

`activate(p, delay, t, prior)`: the process `p` is activated after a specified delay, `t`. The process is inserted in the event list with the new event time, and *before* any processes with the same simulation time which may already be present in the list.

Correspondingly, there are seven `reactivate` methods, which work on either *active*, *suspended* or *passive* processes. They have similar signatures to their `activate` counterparts and work in the same way.

All methods described above are *class* methods of class `Process`. The following four *instance* methods are available:

`idle()` returns `true` if the process is not currently in the event list. Otherwise `false`.

`terminated()` returns `true` if the process has executed all its actions. Otherwise `false`.

`evTime()` returns the time at which the process is scheduled for activation. A runtime exception is thrown if the process is not scheduled.

`nextEv()` returns a reference to the next process, if any, in the event list.

The complete source code of class `Process` is provided in Appendix K.

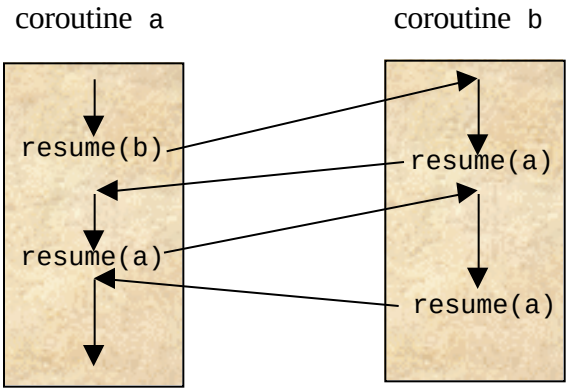
The `javaSimulation` package not only supports the process-based approach of simulation; event-based and activity-based approaches may also be used. The process-based approach encompasses the two other approaches [6][7]. This is demonstrated in appendices M and N.

3. A package for coroutine sequencing in Java

3.1 The coroutine concept

In a process-based simulation the processes undergo active and interactive phases during their lifetimes. A process may be suspended temporarily and resumed later from where it left off. Thus, a process has the properties of a *coroutine*.

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.



It is easy to see that processes may be implemented using coroutines. Below we sketch the implementation of three of the most central scheduling methods of the `javaSimulation` package: `activate`, `passivate` and `hold`.

```
void activate(Process p) {
    p.intoEventListAt(time());
    resume(current());
}
```

```
void passivate() {
    current().outOfEventList();
    resume(current());
}
```

```
void hold(double t) {
    current().intoEventListAt(time() + t);
    resume(current());
}
```

The `intoEventListAt` method inserts the process into the event list at the position corresponding to a specified event time.

The `outOfEventList` method removes the process from the event list.

The `current` method returns a reference to the process currently at the front of the event list.

As a basis for the implementation of `javaSimulation` a package for coroutine sequencing in Java has been developed. This package, called `javaCoroutine`, is based on the coroutine primitives provided by SIMULA. By supporting semi-symmetric as well as symmetric coroutine sequencing it provides more functionality than actually needed for the implementation of `javaSimulation`. Only symmetric coroutine sequencing (by means of the `resume` primitive) is needed.

The following section describes the `javaCoroutine` package from the user's point of view.

3.2 The user facilities of the `javaCoroutine` package

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 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.

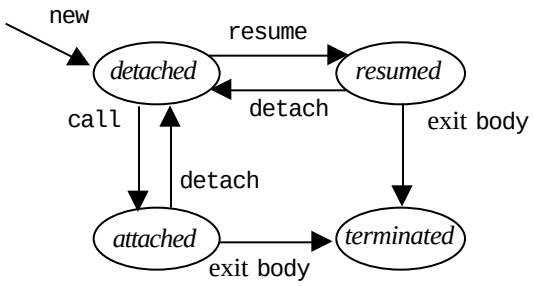
Below is shown a complete coroutine program. The program shows the use of the `resume` method for coroutine alternation as illustrated in the figure on page 30 .

```
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.

3.3 Implementation of the `javaCoroutine` package

A coroutine is characterized mainly by its execution state consisting of its current execution location and a stack of activation records. The bottom element of the stack is the activation record for the call of `body`. The remaining part of the stack contains activation records corresponding to method activations triggered by `body`.

When control is transferred to a coroutine (by means of `resume`, `call` or `detach`), the coroutine must be able to carry on where it left off. Thus, its execution state must persist between successive occasions on which control enters it. Its execution state must be "frozen", so to speak.

When a coroutine transfers from one execution state to another, it is called a *context switch*. This implies the saving of the execution state of the suspending coroutine and its replacement with the execution state of the other coroutine.

The central issue when implementing coroutines is how to achieve such context switches. The goal is to implement the primitive `enter` with the following semantics [8]:

`enter(c)` The execution point for the currently executing coroutine is set to the next statement to be executed, after which this coroutine becomes suspended and the coroutine `c` (re-)commences execution at its execution point.

Having implemented this primitive, it is easy to implement the primitives `resume`, `call` and `detach` (or similar primitives).

An implementation of `resume`, `call` and `detach` by means of `enter` is shown below. For clarity all error handling has been left out.

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

    private static Coroutine current, main;
    private Coroutine caller, callee;
    protected boolean terminated;

    public static void resume(Coroutine next) {
        if (next == current)
            return;
        while (next.callee != null)
            next = next.callee;
        next.enter();
    }

    public static void call(Coroutine next) {
        current.callee = next;
        next.caller = current;
        while (next.callee != null)
            next = next.callee;
        next.enter();
    }

    public static void detach() {
        Coroutine next = current.caller;
        if (next != null) {
            current.caller = next.callee = null;
            next.enter();
        }
        else if (main != null && current != main)
            main.enter();
    }

    private void enter() { ... }
}
```

Here `current` is a reference to the currently executing coroutine, and `main` is a reference to the main coroutine. The references `caller` and `callee` are used for chaining coroutines in a component. The boolean `terminated` is `true` when the coroutine has terminated.

The question is now how to implement the `enter` method.

As a multithreaded language Java provides support for multiple *threads* of execution (sometimes called lightweight processes). A thread can perform a task independent of other threads. Each thread has its own execution state consisting of its current execution location and a stack of activation records. In that respect, a thread is similar to a coroutine.

Threads, however, are more powerful than coroutines. Any number of threads may be executing simultaneously, whereas only one coroutine at a time may be executing.

Thus, threads may act as coroutines, if

- it is possible to control their execution in such a way that only one is executing at any time,
- control can be transferred from one thread to another.

In the following sections we will demonstrate how this is possible in Java. A series of possible implementations will be given, ending with the actual implementation of the `javaCoroutine` package.

3.3.1 Version 1: Synchronization by busy waiting

The first version, shown below, is very simple.

```
public abstract class Coroutine extends Thread {
    final public void run() {
        body();
        if (!terminated) {
            terminated = true;
            detach();
        }
    }

    abstract public void body();

    private static Coroutine current, main;

    public static void resume(Coroutine c) { ... }
    public static void call(Coroutine c) { ... }
    public static void detach() { ... }

    private void enter() {
        if (current == null) {
            current = main = this;
            start();
            return;
        }
        Coroutine old_current = current;
        current = this;
        if (!isAlive())
            start();
        if (old_current.terminated)
            return;
        while (old_current != current)
            yield();
    }
}
```

Class `Coroutine` is here defined as an extension of the `Thread` class of the standard Java libraries. When the `start` method of `Thread` is invoked, the thread begins executing its `run` method. As can be seen from the code, this has the effect that the `body` method of the coroutine starts executing. If `body` ever returns, the coroutine invokes the `detach` method and terminates.

The `enter` method performs a context switch. The call `c.enter()` resumes the coroutine `c` and suspends the calling coroutine.

When the method is invoked for the first time, there is no coroutine currently executing, and the execution of `c` is started by starting the thread associated with `c`. In all other cases, control is transferred from the currently executing coroutine to `c`. This is achieved by letting every coroutine that is not `current` execute a while-loop that only terminates if it is decided that `this` coroutine should become the next `current` (by setting `current` to `this`).

The `yield` method of `Thread` is called inside the while-loop in order to make the executing thread give up control to any other threads that are willing to execute. In this way, thread starvation is avoided.

It is easy to see that this implementation will work. It is ensured that at any time, exactly one coroutine, `current`, will be executing its `body`.

On the other hand, the implementation is very inefficient. When used for an implementation of the `javaSimulation` package it took 132 seconds to run a car wash simulation (with one car washer and `simPeriod` set to 1000000) on a 400 MHz G4 Macintosh computer running MRJ 2.2.

In comparison, it took less than 2 seconds to run the same simulation using any of the event/activity-based simulation packages of this report.

One explanation to this inefficiency is that suspended coroutines are executing code. Each suspended coroutine is constantly checking whether is has been selected as the coroutine to become the next `current`. In other words, all suspended coroutines are *busy waiting*.

3.3.2 Version 2: Synchronization by `resume` and `suspend`

An obvious idea is to use the `resume` and `suspend` methods of Java's class `Thread` for implementing the `enter` method.

The method `suspend` temporarily halts a thread; `resume` allows it to resume.

This idea is carried out in the code shown below.

```
private void enter() {
    if (current == null) {
        current = main = this;
        start();
        return;
    }
    Coroutine old_current = current;
    current = this;
    if (!isAlive())
        start();
    else
        resume();
    if (old_current.terminated)
        return;
    old_current.suspend();
}
```

The currently executing thread suspends itself after having resumed (or started) the thread that has been chosen to take over.

At first sight, this implementation seems to work. But this is not the case. A *race condition* exists. Before `old_current` has suspended, the new `current` may have had time to call its `resume` method. Resuming a thread that is not suspended, however, has no effect. The result is that both `old_current` and the next `current` will be suspended, and the coroutine system will stop.

We can solve this problem if we can assure that `old_current` will suspend only if it is not `current`. The last sentence of `enter` might be replaced by the following:

```
if (old_current != current)
     old_current.suspend();
```

However, this will not work either. A race condition still exists. Between the test `old_current != current` and the suspension of `old_current`, the next `current` may get time to set `current` to `old_current`. But without the desired effect, `old_current` still suspends itself.

We can solve the problem if we can assure that `old_current` suspends itself before the next `current` actually continues its execution. But how can we achieve such an assurance?

One solution is to give `old_current` a higher priority than the next `current`, as shown in the program fragment below.

```
old_current.setPriority(Thread.MAX_PRIORITY;
if (!isAlive())
    start();
else
    resume();
if (old_current.terminated)
    return;
if (old_current != current)
    old_current.suspend();
old_current.setPriority(Thread.NORM_PRIORITY);
```

The last line ensures that when `old_current` is resumed, its priority is set back to its original value (`NORM_PRIORITY`).

This will work, as long as the Java runtime system will let lower-priority threads run only when all higher-priority threads are blocked. However, you cannot rely on this. Some Java runtime systems might let lower-priority threads run, even when there are unblocked higher-priority threads, in order to prevent starvation.

Moreover, the use of the `Thread` methods `resume` and `suspend` is not recommended. Their use may easily result in deadlocks. For this reason these methods now have *deprecated* in Java.

Anyway, using this version of `enter` the car wash simulation program produced the correct output on these three platforms: Macintosh, PC and Sun. The CPU time was 85 seconds on the Macintosh, a reduction in CPU time of 37% in relation to version 1.

3.3.3 Version 3: Synchronization by `wait` and `interrupt`

Instead of using `resume` and `suspend` we can use the methods `wait` and `interrupt` in a similar manner. This is the shown below.

```
private void enter() {
    if (current == null) {
        main = current = this;
        start();
        return;
    }
    Coroutine old_current = current;
    current = this;
    if (!isAlive())
        start();
    else
        interrupt();
    if (old_current.terminated)
        return;
    synchronized(old_current) {
        try {
            old_current.wait();
        } catch(InterruptedException e) {}
    }
}
```

Here, `old_current` suspends itself by calling `wait`. Since the call has been enclosed in a `try` block that catches an `InterruptedException`, `old_current` will leave the try block and resume its execution if it is interrupted.

At first sight, this implementation does not seem to work. There is an apparent problem in that `old_current` may be interrupted before it has called `wait`. However, this is not so. If this happens, the interrupt will be remembered and `wait` will not be executed.

Using this version of the `enter` method the car wash simulation program produced the correct output. The CPU time, however, was now 116 seconds on the Macintosh, an increase in CPU time of 36% in relation to version 2 (the resume-suspend version), and only 12% faster than version 1 (the busy-waiting version). The interrupt mechanism of Java seems to require a considerable computational overhead.

3.3.4 Version 4: Synchronization by `wait` and `notifyAll`

A more efficient version may be obtained by using the `wait` method in combination with the `notifyAll` method.

The `wait` method is used to let one thread wait until a condition occurs, and the notification method `notifyAll` is used to tell all waiting threads that something has occurred that might satisfy that condition.

Below is shown a version of `enter` that uses these two methods.

```
private void enter() {
    if (current == null) {
        current = main = this;
        start();
        return;
    }
    Coroutine old_current = current;
    synchronized(Coroutine.class) {
        current = this;
        if (!isAlive())
            start();
        else
            Coroutine.class.notifyAll();
        if (old_current.terminated)
            return;
        try {
            while (old_current != current)
                Coroutine.class.wait();
        } catch (InterruptedException e) {}
    }
}
```

Each waiting thread waits to be selected as the next `current`. The threads are synchronized by means of the `Class` object of `Coroutine`. In the code above, a thread will be waiting to enter the synchronized statement until the lock on this object is released. This happens when a thread calls `wait`. In this way it is ensured that no thread is notified before it has called `wait`. This is important, since a notification is not remembered (in contrast to an interrupt).

Using this version of `enter` the car wash simulation program took 130 seconds on the Macintosh. This is about the same CPU time as used by version 1 (the busy-waiting version). In fact, version 3 has some busy-waiting too. When `notifyAll` is called, all waiting threads wake up temporarily and examine their condition (`old_current != current`).

3.3.5 Version 5: Synchronization by `wait` and `notify`

To increase efficiency we will use `nofity` in place of `notifyAll`. Instead of notifying all waiting threads, only the next `current` will be notified. This requires synchronization on the individual Coroutine objects.

A version of `enter` that uses this method is shown below.

```
private void enter() {
    if (main == null) {
        main = current = this;
        start();
        return;
    }
    Coroutine old_current = current;
    synchronized(old_current) {
        current = this;
        if (!isAlive())
            start();
        else
        synchronized(this) {
            notify();
        }
        if (old_current.terminated)
            return;
        try {
           old_current.wait();
        } catch(InterruptedException e) {}
    }
}
```

It is easy to prove that this version works correctly. The outer synchronization expression (`old_current`) ensures that no running thread can be notified before it has released the lock on itself, by calling `wait` or by exiting.

Using this version of `enter` the car wash simulation program took 82 seconds on the Macintosh. This is about the same CPU time as used by version 2 (the resume-suspend version).

The following versions of `enter` are all improvements on this version.

3.3.6 Version 6: Protecting the coroutines

Instead of extending class `Thread` a `Coroutine` may implement the `Runnable` interface:

```
public abstract class Coroutine implements Runnable
```

A `Coroutine` may then be executed in its own thread by passing it to a `Thread` constructor.

The following attribute is added to the Coroutine class:

```
private Thread myThread;
```

and the `enter` method is redefined as follows:

```
private void enter() {
    if (myThread == null)
        myThread = new Thread(this);
    if (current == null) {
        current = main = this;
        myThread.start();
        return;
    }
    Coroutine old_current = current;
    synchronized(old_current) {
        current = this;
        if (!myThread.isAlive())
            myThread.start();
        else
        synchronized(this) {
            notify();
        }
        if (old_current.terminated)
            return;
        try {
            old_current.wait();
        } catch(InterruptedException e) {}
    }
}
```

An advantage of this version is that the threads are protected from user manipulation.

3.3.7 Version 7: Improving the efficiency

The efficiency of the previous versions is rather low. Using any of these versions the car wash simulation program took more than 40 times longer to execute than a corresponding program that used an event-based package. This is clearly unsatisfactory.

The threads of Java seem to require a considerable overhead. How much was evaluated by running the small program shown below.

```
public class ThreadOverhead {
    static public class DummyThread extends Thread {
        public void run() {}
    }

    public static void main(String args[]) {
        for (int i = 1; i <= 90945; i++)
            new DummyThread().start();
    }
}
```

The program creates and starts 90945 threads, as many as the number of processes created by the car wash simulation program (with one car washer and `simPeriod` equal to 1000000). Each of the threads has actually nothing to do and terminates immediately after being started.

The CPU time was 78 seconds. In comparison, it took 82 seconds to run the car wash simulation program using the coroutine version of the previous section. So there is in fact a considerable overhead connected with the use of threads. Can we reduce this overhead in any way?

One possibility would be to advise the user not to use too many threads (or processes) in his program. For example, a car wash simulation program might be written which uses only a few processes; namely, the car generator and the car washers. Such a version of the program is outlined below.

```
public class CarWashSimulation extends Process {
   simPeriod = 1000000; ...

   public void actions() {
       for (int i = 1; i <= noOfCarWashers; i++)
           new CarWasher().into(tearoom);
       activate(new CarGenerator());
       hold(simPeriod + 1000000);
       report();
   }

   void report() { ... }

   class Car extends Link {
       double entryTime = time();
   }

   class CarWasher extends Process {
       public void actions() {
           while (true) {
               out();
               while (!waitingLine.empty()) {
                   Car served = (Car) waitingLine.first();
                   served.out();
                   hold(10);
                   noOfCustomers++;
                   throughTime += time() - served.entryTime;
               }
               wait(tearoom);
           }
       }
   }

   class CarGenerator extends Process {
       public void actions() {
           while (time() <= simPeriod) {
               new Car().into(waitingLine);
               int qLength = waitingLine.cardinal();
               if (maxLength < qLength)
                   maxLength = qLength;
               if (!tearoom.empty())
                   activate((CarWasher) tearoom.first());
               hold(random.negexp(1/11.0));
           }
       }
   }

   public static void main(String args[]) {
       activate(new CarWashSimulation(1));
   }
}
```

The CPU time used for running this program was only 2 seconds.

However, it is not always that easy to economize on processes. What, for example, should we do if a large number of car washers were engaged? A further reduction on the number of processes would probably make the program difficult to read.

A better method for reducing thread overhead is to let the coroutine package itself economize on the use of threads. When a coroutine has terminated, its thread is not discarded but, if necessary, reused to run other coroutines. In this way the number of generated threads is held to a minimum.

Unused threads are held in a free list. When a new coroutine starts, the first thread on the list is removed and used for coroutine execution.

Threads are represented as objects of class `Runner`, a subclass and inner class of class `Thread`. An outline of this class is shown below.

```
class Runner extends Thread {
    Coroutine myCoroutine;
    Runner next;

    public void run() {
        myCoroutine.body();
        ...
        next = firstFree;
        firstFree = this;
        ...
    }

    void go() { ... }
}
```

As long as a `Runner` is active it executes the `body` method of the coroutine referenced by `myCoroutine`. Having finished this job, the `Runner` inserts itself into the free list and waits.

The `next` reference is a link to the next `Runner` in the free list.

The `go` method is used to start or resume the execution of a `Runner`.

The following two references are declared in class `Coroutine`:

```
private Runner myRunner;
private static Runner firstFree;
```

Here `myRunner` references the `Runner` for the `Coroutine` object, and `firstFree` references the first `Runner` in the free list.

Below is given the complete code of the `Runner` class.

```
class Runner extends Thread {
    Coroutine myCoroutine;
    Runner next;

    Runner(Coroutine c) {
        myCoroutine = c;
    }

    public void run() {
        while (true) {
            myCoroutine.body();
            if (!myCoroutine.terminated) {
                myCoroutine.terminated = true;
                detach();
            }
            if (myCoroutine == Coroutine.main) {
                myCoroutine = null;
                Coroutine.main = null;
                Coroutine.current = null;
                return;
            }
            myCoroutine = null;
            next = firstFree;
            firstFree = this;
            synchronized(this) {
                try {
                    wait();
                } catch (InterruptedException e) {}
            }
        }
    }

    void go() {
        if (!isAlive())
            start();
        else
            notify();
    }
}
```

When a `Runner` has been used to execute the body of a coroutine, it waits (by calling `wait`) until it is woken up (by calling `notify` in the `go` method).

The `enter` method of class `Coroutine` now looks as follows:

```
private void enter() {
    if (myRunner == null) {
        if (firstFree != null) {
            myRunner = firstFree;
            firstFree = firstFree.next;
            myRunner.myCoroutine = this;
        } else
            myRunner = new Runner(this);
    }
    if (main == null) {
        main = current = this;
        myRunner.go();
        return;
    }
    Coroutine old_current = current;
    synchronized(old_current.myRunner) {
        current = this;
        myRunner.go();
        if (old_current.terminated)
            return;
        try {
            old_current.myRunner.wait();
        } catch(InterruptedException e) {}
    }
}
```

When a `Runner` is needed, it is taken from the free list, if possible. Otherwise, a new one is created.

When this version was used for the implementation of `javaSimulation`, the execution of the car wash simulation program took only 6 seconds on the Macintosh. Thus the recycling of threads has reduced the running time for this example substantially.

In the remaining two sections of this chapter we will improve further on this version.

3.3.8 Version 8: Mutual exclusion of main coroutines

In some applications it is convenient to have more than one coroutine system in the same program. This is, for example, the case in the car wash simulation problem where two simulations are to be performed.

```
public static void main(String args[]) {
    activate(new CarWashSimulation(1));
    activate(new CarWashSimulation(2));
}
```

First, the simulation is to be performed with one car washer, and then, with two car washers.

But since the `main` method runs in a thread, in competition with the `Runner` threads, the two simulations will be intermingled. We have to take special care to prevent this situation from arising.

We will solve the problem by letting any thread that activates a main coroutine wait until the main coroutine has finished. The main coroutine signals that it has finished by setting the reference `main` to `null`.

First, we replace this code fragment of the `enter` method

```
if (main == null) {
    main = this;
    myRunner.go();
    return;
}
```

with

```
if (main == null) {
    main = this;
    myRunner.go();
    synchronized(Runner.class) {
        try {
            while (main != null)
                Runner.class.wait();
        } catch (InterruptedException e) {}
    }
    return;
}
```

Then we replace this code fragment of the `run` method of class `Runner`

```
if (myCoroutine == Coroutine.main) {
    myCoroutine = null;
    Coroutine.current = null;
    Coroutine.main = null;
}
```

with

```
if (myCoroutine == Coroutine.main) {
     myCoroutine = null;
     Coroutine.current = null;
     synchronized(Runner.class) {
         Coroutine.main = null;
         Runner.class.notify();
     }
     return;
}
```

3.3.9 Version 9: Ending the coroutines

One last problem remains. A Java program will keep running as long as there are any *user threads* running. This will prevent any program that uses the previous coroutine version from ever stopping. Either a thread is being used for the execution of the body of a coroutine, or it is waiting in the free list for being used again.

How can we solve this problem? Very simply, just by marking all threads as *daemon* threads. Daemon threads are expendable and are stopped, when all user threads of the program have finished.

The marking is made as follows in the constructor of class `Runner`:

```
Runner(Coroutine c) {
    myCoroutine = c;
    setDaemon(true);
}
```

With this last improvement, the code of the coroutine package is complete. The complete source code of the `javaCoroutine` package is given in appendix O.

A small test program, adapted from [9], is given in Appendix P.

4. Implementation of `javaSimulation`

Equipped with the coroutine package described in the previous chapter, implementation of the `javaSimulation` package is straightforward. The SIMULA code given for class SIMULATION in reference [1] may be translated almost directly into Java.

However, during this translation process some minor problems must be solved. These problems and their solutions will be described briefly below.

(1) *How should the dynamics of processes be represented?*

Since a process behaves as a coroutine, an obvious idea is to let class `Process` be a subclass of class `Coroutine`. Below is an outline of how to implement this idea.

```
public abstract class Process extends Coroutine {
    public abstract void actions();

    protected final void body() {
        ...
        actions();
        ...
    }
}
```

The life cycle of a process is described in a subclass of `Process` by overriding the abstract method `actions`. The `body` method, inherited from class `Coroutine`, takes care of its execution.

In this way, each process acquires the capabilities of a coroutine. Thus, the resumption of a `current` in the scheduling methods may be implemented as follows:

```
resume(current());
```

This solution, however, has two flaws.

Firstly, SIMULA prescribes that `Process` must be a subclass of `Link`. But since Java does not allow multiple inheritance, `Process` cannot be a subclass of both `Link` and `Coroutine`.

Secondly, the coroutine capabilities of `Process` will not be protected from the user. He may, for example, call the `resume` method with a `Process` object as parameter. The effect of this would be disastrous for the scheduling mechanisms.

Letting `Link` be a subclass of `Coroutine` would eliminate the first of these flaws, but not the second one.

A better solution is to let each `Process` object have at its disposal a `Coroutine` object. The `Coroutine` object is responsible for executing the `actions` method. This solution is outlined below.

```
public abstract class Process extends Link {
    public abstract void actions();

    private Coroutine myCoroutine = new Coroutine() {
        protected final void body() {
            ...
            actions();
            ...
        }
    }
}
```

The resumption of `current` in the scheduling methods is now implemented as follows:

```
Coroutine.resume(current().myCoroutine);
```

This solution eliminates both flaws. Class `Process` is now a subclass of `Link`, and its coroutine capabilities can no longer be misused by the user.

(2) *How should the event list be represented?*

In the SIMULA code the event list (called the *sequencing set*, SQS) is represented as an ordered list of *event notices*. Each event notice is an object containing the event time and a reference to the process that has scheduled the event. Every time a process schedules an event, a new event notice is created and inserted into the event list. By using this implementation a lot of event notice objects are created during a simulation.

We will avoid this overhead by providing each process with the capability of being able to act as an event notice. When a process schedules an event, it does not create any event notice. It merely inserts itself in the event list. For this purpose, each `Process` has the following (private) attributes:

```
double EVTIME;
Process PRED, SUC;
```

EVTIME is the event time. PRED and SUC are the predecessor and successor of the process in the event list. If the process has no scheduled event, both PRED and SUC will be null. An auxiliary process, SQS, is used as a list head for the circular list of scheduled processes. SQS.SUC always references the currently active process, current.

(3) *How should the main process be represented?*

In SIMULATION the main process, i.e., the process corresponding to the main program, is represented by an anonymous Process object with the following body (expressed in Java):

```
while (true)
    detach();
```

Each time this process is current, it calls detach, thereby resuming the execution of the main program.

In javaSimulation we will define the *main process* as the first process activated in a simulation and obtain the desired functionality by implementing the body method of class Process as follows:

```
final public void body() {
    if (MAIN == null)
        MAIN = this;
    actions();
    TERMINATED = terminated = true;
    if (this == MAIN) {
        while (SQS.SUC != SQS)
            SQS.UNSCHEDULE(SQS.SUC);
        MAIN = null;
        return;
    }
    passivate();
}
```

When the first process in a simulation is activated, MAIN is set to reference this process. When the main process terminates, the simulation ends and the event list is emptied (to prepare for a possible subsequent simulation).

(4) *How can we make processes terminate properly?*

As seen in the code above, every terminated process calls passivate as its last action. However, if passivate merely resumes the next current, its body method never ends. This is unfortunate, since this would prevent the thread associated with a terminated process from being reused.

We can handle this situation by setting the protected boolean variable `terminated` (inherited from class `Coroutine`) to `true` when a process terminates. By this means the `enter` method is told that the associated thread is no longer needed.

(5) *How do we best imitate the syntax of SIMULA's activation statements?*

SIMULA introduces the following special keywords to be used for activation statements:

```
activate
reactivate
at
delay
before
after
prior
```

For example, the programmer can write

```
activate p at 35 prior;
```

in order to schedule an event for the process `p` to occur at system time `35`. The keyword `prior` signifies that the event be scheduled in front of any events with the same system time.

If possible, we should enable the user of `javaSimulation` to use a similar syntax.

If we provide the constants `at`, `delay`, `before`, `after` and `prior`, it is possible to get very close to the SIMULA syntax. For example, the user may express the activation statement above in Java by writing

```
activate(p, at, 35, prior);
```

We let each of these constants be a reference to an object of its own class:

```
public static final At     at;
public static final Delay  delay;
public static final Before before;
public static final After  after;
public static final Prior  prior;
```

Next, we overload the `activate` and `reactivate` methods as follows:

```
public static final void activate(Process p);
public static final void activate(Process p,
                          At at, double t);
public static final void activate(Process p,
                          Delay delay, double t);
public static final void activate(Process p,
                          At at, double t, Prior prior);
public static final void activate(Process p,
                          Delay d, double t, Prior prior);
public static final void activate(Process p1,
                          Before before, Process p2);
public static final void activate(Process p1,
                          After after, Process p2);

public static final void reactivate(Process p);
public static final void reactivate(Process p,
                          At at, double t);
public static final void reactivate(Process p,
                          Delay delay, double t);
public static final void reactivate(Process p,
                          At at, double t, Prior prior);
public static final void reactivate(Process p,
                          Delay d, double t, Prior prior);
public static final void reactivate(Process p1,
                          Before before, Process p2);
public static final void reactivate(Process p1,
                          After after, Process p2);
```

In this way, we have found a satisfactory solution to the syntax problem.

Note that a possible illegal use of the “keywords” will be detected during the program compilation.

5. Evaluation of `javaSimulation`

The `javaSimulation` package has been tested on the following platforms:

MAC:	Power Macintosh G4	(400 MHz),	Java 1.1.8
PC:	Dell PowerEdge 1300	(400 MHz),	Java 1.2.2
SUN:	Sun Enterprise 250	(300 MHz),	Java 1.2.2

Performance was measured by running the car wash simulation.

When running a simulation with one car washer and `simPeriod` set to 1000000, the following CPU times (in seconds) were measured:

MAC	PC	SUN
6	13	10

When running the program with a version of the package that *did not* reuse threads, the following CPU times were measured:

MAC	PC	SUN
80	80	42

As can be seen, the technique of reusing threads has great significance for performance.

The Java runtime system uses the underlying operating system for thread support, or its own software emulation if the operating system does not support threads. Apparently, the underlying threading system of Java on Sun is the best.

The computational overhead of threads may also be assessed by comparing the runtimes above with the following runtimes, measured when the event- and activity-based simulation packages of this report were used:

	MAC	PC	SUN
event	2	1	3
activity	2	1	4
events	2	2	6

It is well known that Java programs run slower than equivalent programs written in other programming languages. In order to examine to what extent this applies in the present case the process-based car wash simulation was executed on the Sun by use of the following software:

SIMULA:	The Lund Simula Compiler (translates into machine code)
cim:	A SIMULA compiler that produces C code
COROUTINE:	A C++ library for coroutine sequencing [10].

The C++ library exists in two versions: copy-stack and share-stack. Along with the library comes a simulation library similar to the `javaSimulation` package.

The following runtimes were measured:

	SUN
`javaSimulation`	10
SIMULA	2
cim	6
COROUTINE (copy-stack)	5
COROUTINE (share-stack)	3

As can be seen, the program based on `javaSimulation` ran 5 times slower than an equivalent SIMULA program compiled by the Lund Simula compiler. This is not a big factor, considering that the SIMULA program was compiled into highly optimized machine code.

JavaSim [11] is a Java package similar to `javaSimulation`. Both packages provide simulation facilities corresponding to those provided by SIMULA. When running the car wash simulation using JavaSim the following runtimes were measured:

MAC	PC	SUN
120	89	66

As can be seen, the efficiency of JavaSim is comparatively low.

The computational overhead of thread usage is not only of a computational nature. Threads may also allocate a substantial amount of memory. Each thread must allocate enough memory to hold its stack; actually two stacks: one for Java code and one for C code. As the default size of each of these stacks may be hundreds of kilobytes, memory consumption will be excessive for applications with many live threads.

The user must be aware of this fact and, if possible, reduce the default stack size for the threads. On many platforms this can be achieved by using the `-oss` and `–ss` options of the `java` interpreter.

We may conclude that the performance of `javaSimulation` is quite satisfactory, although not impressive. The implementation of the underlying threading system of Java plays an important role in this connection. Currently the overhead induced by using threads is considerable. However, Java is a very young language, and faster implementations are likely in the future.

6. Conclusions

This report describes `javaSimulation`, a Java package for process-based discrete event simulation. The package is based on a Java library for coroutine sequencing and contains all the simulation facilities of SIMULA.

A central implementation problem, how to make threads representing processes behave as coroutines, has been solved successfully in the present implementation.

The performance of the package is reasonably good. By recycling the threads of terminated coroutines, much of the overhead originating from the use of Java's threads has been eliminated.

References

1. O.-J. Dahl, B. Myhrhaug & K. Nygaard,
   COMMON BASE LANGUAGE,
   NNC Publication S-22 (1970).

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

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

4. K. Helsgaun,
   DISCO - a SIMULA-based language for combined continuous and discrete simulation,
   SIMULATION, Vol. 34, no. 7, pp. 1-12 (1980).

5. W. Kreutzer,
   System simulation: programming styles and languages,
   Addison Wesley (1986).

6. P. R. Hills,
   An Introduction to Simulation,
   NEC Publication S. 55, Oslo (1973).

7. W. R. Franta,
   The Process View of Simulation,
   North Holland (1977).

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

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

10. K. Helsgaun,
    A Portable C++ Library for Coroutine Sequencing,
    Datalogiske skrifter, No. 87, Roskilde University (1999).

11. R. McNab,
    SimJava: a discrete event simulation library for Java,
    University of Newcastle upon Tyne (1996).
    Available from `http://www.javaSim.ncl.ac.uk`

Appendices

A. The `simset` package

B. Source code of the `simset` package

C. The `random` package

D. Source code of the `random` package

E. Source code of the `simulation.event` package

F. Car wash simulation with `simulation.event`

G. Source code of the `simulation.activity` package

H. Car wash simulation with `simulation.activity`

I. Source code of the `simulation.events` package

J. Car wash simulation with `simulation.events`

K. Source code of `javaSimulation`

L. Process-based car wash simulation with `javaSimulation`

M. Event-based car wash simulation with `javaSimulation`

N. Activity-based car wash simulation with `javaSimulation`

O. Source code of the `javaCoroutine` package

P. Test program for `javaCoroutine`

A. The `simset` package

This package contains facilities for the manipulation of two-way linked lists. Its functionality corresponds closely to SIMULA's built-in class SIMSET.

List members are objects of subclasses of the class `Link`.

An object of the class `Head` is used to represent a list.

The class `Linkage` is a common superclass for class `Link` and class `Head`.

The three classes are described below by means of the following variables:

```
Head hd;
Link lk;
Linkage lg;
```

Class `Linkage`

```
public class Linkage {
    public final Link pred();
    public final Link suc();
    public final Linkage prev();
}
```

`lk.suc()` returns a reference to the list member that is the successor of `lk` if `lk` is a list member and is not the last member of the list; otherwise `null`.

`hd.suc()` returns a reference to the fist member of the list hd, if the list is not empty; otherwise `null`.

`lk.pred()` returns a reference to the list element that is the predecessor of `lk` if `lk` is a list member and is not the first member of the list; otherwise `null`.

`hd.pred()` returns a reference to the last member of the list `hd` if the list is not empty; otherwise `null`.

`lk.prev()` returns `null` if `lk` is not a list member, a reference to the list head if `lk` is the first member of a list; otherwise a reference to `lk`'s predecessor in the list.

`hd.prev()` returns a reference to `hd` if `hd` is empty; otherwise a reference to the last member of the list.

Class `Head`

```
public class Head extends Linkage {
    public final Link first();
    public final Link last();
    public final boolean empty();
    public final int cardinal();
    public final void clear();
}
```

`hd.first()` returns a reference to the first member of the list (`null`, if the list is empty).

`hd.last()` returns a reference to the last member of the list (`null`, if the list is empty).

`hd.cardinal()` returns the number of members in the list (`null`, if the list is empty).

`hd.empty()` returns `true` if the list `hd` has no members; otherwise `null`.

`hd.clear()` removes all members from the list.

Class `Link`

```
public class Link extends Linkage {
    public final void out();
    public final void follow(Linkage ptr);
    public final void precede(Linkage ptr);
    public final void into(Head s);
}
```

`lk.out()`	removes `lk` from the list (if any) of which it is a member. The call has no effect if `lk` has no membership.
`lk.into(hd)`	removes `lk` from the list (if any) of which it is a member and inserts `lk` as the last member of the list `hd`.
`lk.precede(lg)`	removes `lk` from the list (if any) of which it is a member and inserts `lk` before `lg`. The effect is the same as `lk.out()` if `lg` is `null`, or it has no membership and is not a list head.
`lk.follow(lg)`	removes `lk` from the list (if any) of which it is a member and inserts `lk` after `lg`. The effect is the same as `lk.out()` if `lg` is `null`, or it has no membership and is not a list head.

B. Source code of the `simset` package

```
public class Linkage {
    public final Link pred() {
        return PRED instanceof Link ? (Link) PRED : null;
    }

    public final Link suc() {
        return SUC instanceof Link ? (Link) SUC : null;
    }

    public final Linkage prev() { return PRED; }

    Linkage PRED, SUC;
}

public class Link extends Linkage {
    public final void out() {
        if (SUC != null) {
            SUC.PRED = PRED;
            PRED.SUC = SUC;
            SUC = PRED = null;
        }
    }

    public final void follow(Linkage ptr) {
        out();
        if (ptr != null && ptr.SUC != null) {
            PRED = ptr;
            SUC = ptr.SUC;
            SUC.PRED = ptr.SUC = this;
        }
    }


    public final void precede(Linkage ptr) {
        out();
        if (ptr != null && ptr.SUC != null) {
            SUC = ptr;
            PRED = ptr.PRED;
            PRED.SUC = ptr.PRED = this;
        }
    }

    public final void into(Head s) {
        precede(s);
    }
}
```

```
public class Head extends Linkage {
    public Head() { PRED = SUC = this; }

    public final Link first() { return suc(); }

    public final Link last() { return pred(); }

    public final boolean empty() { return SUC == this; }

    public final int cardinal() {
        int i = 0;
        for (Link ptr = first(); ptr != null; ptr = ptr.suc())
            i++;
        return i;
    }

    public final void clear() {
        while (first() != null)
            first().out();
    }
}
```

C. The `random` package

This package provides the same methods for random drawing as can be found in SIMULA. All methods are available in a class called `Random`. A summary of this class is shown below.

```
public class Random extends java.util.Random {
    public Random() { super(); }
    public Random(long seed) { super(seed); }

    public final boolean draw(double a);
    public final int randInt(int a, int b);
    public final double uniform(double a, double b);
    public final double normal(double a, double b);
    public final double negexp(double a);
    public final int poisson(double a);
    public final double erlang(double a, double b);
    public final int discrete(double[] a);
    public final double linear(double[] a, double[] b);
    public final int histd(double[] a);
}
```

The class is an extension of Java's standard class `java.util.Random`. Thus, all of the facilities of the latter class is also available to the user.

`public Random();`

> This constructor creates a `Random` object with the current time as its seed value.

`public Random(long seed);`

> This constructor creates a `Random` object with the given seed value.

Each of the instance methods performs a random drawing of some kind. Their semantics are as in SIMULA.

`boolean draw(double a);`

> The value is true with the probability `a`, `false` with probability `1-a`. It is always `true` if $a \geq 1$, and always `false` if $a \leq 0$.

`int randInt(int a, int b);`

> The value is one of the integers `a`, `a+1`, …, `b-1`, `b` with equal probability. If $b < a$, the call constitutes an error.

```
double uniform(double a, double b);
```

The value is uniformly distributed in the interval $a \le x < b$. If $b \le a$, the call constitutes an error.

```
double normal(double a, double b);
```

The value is normally distributed with mean `a` and standard deviation `b`.

```
double negexp(double a);
```

The value is a drawing from the negative exponential distribution with mean `1/A`. If `a` is non-positive, a runtime error occurs.

```
int poisson(double a);
```

The value is a drawing from the Poisson distribution with parameter `a`.

```
double erlang(double a, double b);
```

The value is a drawing from the Erlang distribution with mean `1/a` and standard deviation `1/(a* √b)`. Both `a` and `b` must be positive.

```
int discrete(double[] a);
```

The one-dimensional array `a` of n elements of type `double`, augmented by the element `1` to the right, is interpreted as a step function of the subscript, defining a discrete (cumulative) distribution function.

The function value satisfies

$$0 \le \texttt{discrete(a)} \le n$$

It is defined as the smallest `i` such that `a[i]` > `r`, where `r` is a random number in the interval [`0`;`1`] and `a`[n] = `1`.

```
double linear(double[] a, double[] b);
```

The value is a drawing from a (cumulative) distribution function f, which is obtained by linear interpolation in a non-equidistant table defined by `a` and `b`, such that `a[i]` = f(`b[i]`).

It is assumed that `a` and `b` are one-dimensional arrays of the same length, that the first and last elements of `a` are equal to `0` and `1`, respectively, and that `a[i]` ≥ `a[j]` and `b[i]` > `b[j]` for `i` > `j`.

```
public int histd(double[] a);
```

The value is an integer in the range [`0`;n-`1`] where n is the number of elements in the one-dimensional array `a`. The latter is interpreted as a histogram defining the relative frequencies of the values.

D. Source code of the `random` package

```
public class Random extends java.util.Random {
    public Random() { super(); }

    public Random(long seed) { super(seed); }

    public boolean draw(double a) {
        return a < nextDouble();
    }

    public int randInt(int a, int b) {
        if (b < a)
            error("randInt: Second parameter is" +
                  " lower than first parameter");
        return (int) (a + nextDouble()*(b - a + 1));
    }

    public double uniform(double a, double b) {
        if (b <= a)
            error("uniform: Second parameter is not" +
                  " greater than first parameter");
        return a + nextDouble()*(b - a);
    }

    public double normal(double a, double b) {
        return a + b*nextGaussian();
    }


    public double negexp(double a) {
        if (a <= 0)
            error("negexp: First parameter is lower" +
                  " than zero");
        return -Math.log(nextDouble())/a;
    }

    public int poisson(double a) {
        double limit = Math.exp(-a), prod = nextDouble();
        int n;
        for (n = 0; prod >= limit; n++)
            prod *= nextDouble();
        return n;
    }
```

```
public double erlang(double a, double b) {
        if (a <= 0)
            error("erlang: First parameter is not greater" +
                  " than zero");
        if (b <= 0)
            error("erlang: Second parameter is not greater" +
                  " than zero");
        long bi = (long) b, ci;
        if (bi == b)
            bi--;
        double sum = 0;
        for (ci = 1; ci <= bi; ci++)
            sum += Math.log(nextDouble());
        return -(sum + (b - (ci-1))*Math.log(nextDouble()))/
                (a*b);
    }

    public int discrete(double[] a) {
        double basic = nextDouble();
        int i;
        for (i = 0; i < a.length; i++)
            if (a[i] > basic)
                break;
        return i;
    }

    public double linear(double[] a, double[] b) {
        if (a.length != b.length)
            error("linear: arrays have different length");
        if (a[0] != 0.0 || a[a.length-1] != 1.0)
            error("linear: Illegal value in first array");
        double basic = nextDouble();
        int i;
        for (i = 1; i < a.length; i++)
             if (a[i] >= basic)
                 break;
        double d = a[i] - a[i-1];
        if (d == 0.0)
            return b[i-1];
        return b[i-1] + (b[i]-b[i-1])*(basic-a[i-1])/d;
    }
```

```
    public int histd(double[] a) {
        double sum = 0.0;
        int i;
        for (i = 0; i < a.length; i++)
            sum += a[i];
        double weight = nextDouble() * sum;
        sum = 0.0;
        for (i = 0; i < a.length - 1; i++) {
            sum += a[i];
            if (sum >= weight)
                break;
        }
        return i;
    }

    private static void error(String msg) {
        throw new RuntimeException(msg);
    }
}
```

E. Source code of the `simulation.event` package

```
public abstract class Event {
    protected abstract void actions();

    public final void schedule(double evTime) {
        if (evTime < time)
            throw new RuntimeException
                ("attempt to schedule event in the past");
        cancel();
        eventTime = evTime;
        Event ev = SQS.pred;
        while (ev.eventTime > eventTime)
            ev = ev.pred;
        pred = ev;
        suc = ev.suc;
        ev.suc = suc.pred = this;
    }

    public final void cancel() {
        if (suc != null) {
            suc.pred = pred;
            pred.suc = suc;
            suc = pred = null;
        }
    }

    public final static double time() { return time; }

    public final static void runSimulation(double period) {
        while (SQS.suc != SQS) {
            Event ev = SQS.suc;
            time = ev.eventTime;
            if (time > period)
                break;
            ev.cancel();
            ev.actions();
        }
        stopSimulation();
    }

    public final static void stopSimulation() {
        while (SQS.suc != SQS)
            SQS.suc.cancel();
        time = 0;
    }

    private final static Event SQS = new Event() {
        { pred = suc = this; }
        protected void actions() {}
    };
```

```
    private static double time = 0;
    private double eventTime;
    private Event pred, suc;
}


public class Simulation extends Event {
    protected final void actions() {}
}
```

F. Car wash simulation with `simulation.event`

```
import simulation.event.*;
import simset.*;
import random.*;

public class CarWashSimulation extends Simulation {
    int noOfCarWashers;
    double simPeriod = 1000000;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    int noOfCustomers, maxLength;
    double throughTime;
    long startTime = System.currentTimeMillis();

    CarWashSimulation(int n) {
        noOfCarWashers = n;
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        new CarArrival().schedule(0);
        runSimulation(simPeriod + 1000000);
        report();
    }

    void report() {
        System.out.println(noOfCarWashers +
                           " car washer simulation");
        System.out.println("No.of cars through the system = " +
                           noOfCustomers);
        java.text.NumberFormat fmt =
            java.text.NumberFormat.getNumberInstance();
        fmt.setMaximumFractionDigits(2);
        System.out.println("Av.elapsed time = " +
            fmt.format(throughTime/noOfCustomers));
        System.out.println("Maximum queue length = " +
                           maxLength);
        System.out.println("\nExecution time: " +
            fmt.format((System.currentTimeMillis()
                        - startTime)/1000.0) + " secs.\n");
    }

    class CarWasher extends Link {}

    class Car extends Link {
        double entryTime = time();
    }
```

```
class CarArrival extends Event {
    public void actions() {
        if (time() <= simPeriod) {
              Car theCar = new Car();
              theCar.into(waitingLine);
              int qLength = waitingLine.cardinal();
              if (maxLength < qLength)
                  maxLength = qLength;
              if (!tearoom.empty())
                  new StartCarWashing().schedule(time());
             new CarArrival().schedule
                 (time() + random.negexp(1/11.0));
          }
    }
}

class StartCarWashing extends Event {
    public void actions() {
        CarWasher theCarWasher =
            (CarWasher) tearoom.first();
        theCarWasher.out();
        Car theCar = (Car) waitingLine.first();
        theCar.out();
        new StopCarWashing(theCarWasher,
                           theCar).schedule(time() + 10);
    }
}

class StopCarWashing extends Event {
    CarWasher theCarWasher;
    Car theCar;

    StopCarWashing(CarWasher cw, Car c) {
        theCarWasher = cw; theCar = c;
    }

    public void actions() {
        theCarWasher.into(tearoom);
        if (!waitingLine.empty())
            new StartCarWashing().schedule(time());
        noOfCustomers++;
        throughTime += time() - theCar.entryTime;
    }
}


public static void main(String args[]) {
    new CarWashSimulation(1);
    new CarWashSimulation(2);
}
```

G. Source code of the `simulation.acivity` package

```
public abstract class Activity {
    protected abstract boolean condition();
    protected abstract void startActions();
    protected abstract double duration();
    protected abstract void finishActions();

    public Activity() { schedule(); }

    Activity(boolean dummy) {}

    public final static double time() { return time; }

    public final void schedule() {
        cancel();
        suc = waitList.suc;
        suc.pred = waitList.suc = this;
        pred = waitList;
    }

    public final void schedule(double evTime) {
        if (evTime < time)
            throw new RuntimeException
                ("attempt to schedule event in the past");
        cancel();
        eventTime = evTime;
        Activity a = SQS.pred;
        while (a.eventTime > eventTime)
            a = a.pred;
        pred = a;
        suc = a.suc;
        a.suc = suc.pred = this;
    }

    public final void cancel() {
        if (suc != null) {
            suc.pred = pred;
            pred.suc = suc;
            pred = suc = null;
        }
    }

    public final static void stopSimulation() {
        while (waitList.suc != waitList)
            waitList.suc.cancel();
        while (SQS.suc != SQS)
            SQS.suc.cancel();
        time = 0;
    }
```

```
    public final static void runSimulation(double period) {
        while (true) {
            for (Activity a = waitList.suc;
                 a != waitList;
                 a = a.suc) {
                if (a.condition()) {
                    a.cancel();
                    a.schedule(time + a.duration());
                    a.startActions();
                    a = waitList;
                }
            }
            if (SQS.suc == SQS)
                break;
            Activity a = SQS.suc;
            time = a.eventTime;
            a.cancel();
            if (time > period)
                break;
            a.finishActions();
        }
        stopSimulation();
    }

    private final static Activity waitList = new Activity(true) {
        { pred = suc = this; }
        public boolean condition() { return false; }
        public void startActions() {}
        public double duration() { return 0; }
        public void finishActions() {}
    };

    private final static Activity SQS = new Activity(true) {
        { pred = suc = this; }
        public boolean condition() { return false; }
        public void startActions() {}
        public double duration() { return 0; }
        public void finishActions() {}
    };

    private static double time = 0;
    private double eventTime;
    private Activity pred, suc;
}

public class Simulation extends Activity {
    protected final boolean condition() { return true; }
    protected final void startActions() {}
    protected final double duration() { return 0; }
    protected final void finishActions() {}
}
```

H. Car wash simulation with `simulation.activity`

```
import simulation.activity.*;
import simset.*;
import random.*;

public class CarWashSimulation extends Simulation {
    int noOfCarWashers;
    double simPeriod = 1000000;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    int noOfCustomers, maxLength;
    double throughTime;
    long startTime = System.currentTimeMillis();

    CarWashSimulation(int n) {
        noOfCarWashers = n;
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        new CarArrival();
        runSimulation(simPeriod + 1000000);
        report();
    }

    void report() {
        System.out.println(noOfCarWashers +
                           " car washer simulation");
        System.out.println("No.of cars through the system = " +
                           noOfCustomers);
        java.text.NumberFormat fmt =
            java.text.NumberFormat.getNumberInstance();
        fmt.setMaximumFractionDigits(2);
        System.out.println("Av.elapsed time = " +
            fmt.format(throughTime/noOfCustomers));
        System.out.println("Maximum queue length = " +
                           maxLength);
        System.out.println("\nExecution time: " +
            fmt.format((System.currentTimeMillis()
                        - startTime)/1000.0) + " secs.\n");
    }


    class CarWasher extends Link {}

    class Car extends Link {
        double entryTime = time();
    }
```

```
    class CarWashing extends Activity {
        Car theCar; CarWasher theCarWasher;

        CarWashing(Car c) { theCar = c; }

        public boolean condition() {
            return theCar == (Car) waitingLine.first() &&
                   !tearoom.empty();
        }

        public void startActions() {
            theCar.out();
            theCarWasher = (CarWasher) tearoom.first();
            theCarWasher.out();
        }

        public double duration() { return 10; }

        public void finishActions() {
            theCarWasher.into(tearoom);
            noOfCustomers++;
            throughTime += time() - theCar.entryTime;
        }
    }

    class CarArrival extends Activity {
        public boolean condition() { return true; }

        public void startActions() {
            Car theCar = new Car();
            theCar.into(waitingLine);
            new CarWashing(theCar);
            int qLength = waitingLine.cardinal();
            if (maxLength < qLength) maxLength = qLength;
        }

        public double duration() {
            return random.negexp(1/11.0);
        }

        public void finishActions() {
            if (time() <= simPeriod)
                new CarArrival();
        }
    }

    public static void main(String args[]) {
        new CarWashSimulation(1);
        new CarWashSimulation(2);
    }
}
```

I. Source code of the `simulation.events` package

```
public abstract class Event {
    protected abstract void actions();

    public final static double time() { return time; }

    public final static void runSimulation(double period) {
        while (true) {
            for (StateEvent a = (StateEvent) waitList.suc;
                 a != waitList;
                 a = (StateEvent) a.suc) {
                if (a.condition()) {
                    a.cancel();
                    a.actions();
                    a = waitList;
                }
            }
            if (SQS.suc == SQS)
                break;
            TimeEvent ev = (TimeEvent) SQS.suc;
            time = ev.eventTime;
            ev.cancel();
            if (time > period)
                break;
            ev.actions();
        }
        stopSimulation();
    }

    public final static void stopSimulation() {
        while (SQS.suc != SQS)
            SQS.suc.cancel();
        while (waitList.suc != waitList)
            waitList.suc.cancel();
        time = 0;
    }

    public final void cancel() {
        if (suc != null) {
            suc.pred = pred;
            pred.suc = suc;
            suc = pred = null;
        }
    }
```

```
    static final TimeEvent SQS = new TimeEvent() {
        { pred = suc = this; }
        protected void actions() {}
    };

    static final StateEvent waitList = new StateEvent() {
        { pred = suc = this; }
        protected boolean condition() { return false; }
        protected void actions() {}
    };

    static double time = 0;
    Event pred, suc;
}


public class Simulation extends Event {
    protected final void actions() {}
}
```

J. Car wash simulation with `simulation.events`

```
import simulation.events.*;
import simset.*;
import random.*;

public class CarWashSimulation extends Simulation {
    int noOfCarWashers;
    double simPeriod = 1000000;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    int noOfCustomers, maxLength;
    double throughTime;
    Random random = new Random(5);
    long startTime = System.currentTimeMillis();

    CarWashSimulation(int n) {
        noOfCarWashers = n;
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        new CarArrival().schedule(0);
        runSimulation(simPeriod + 1000000);
        report();
    }

    void report() {
        System.out.println(noOfCarWashers +
                           " car washer simulation");
        System.out.println("No.of cars through the system = " +
                           noOfCustomers);
        java.text.NumberFormat fmt =
            java.text.NumberFormat.getNumberInstance();
        fmt.setMaximumFractionDigits(2);
        System.out.println("Av.elapsed time = " +
            fmt.format(throughTime/noOfCustomers));
        System.out.println("Maximum queue length = " +
                           maxLength);
        System.out.println("\nExecution time: " +
            fmt.format((System.currentTimeMillis()
                        - startTime)/1000.0) + " secs.\n");
    }


    class CarWasher extends Link {}

    class Car extends Link {
        double entryTime = time();
    }
```

```
    class CarArrival extends TimeEvent {
        public void actions() {
            if (time() <= simPeriod) {
                Car theCar = new Car();
                theCar.into(waitingLine);
                int qLength = waitingLine.cardinal();
                if (maxLength < qLength) maxLength = qLength;
                new StartCarWashing(theCar).schedule();
                schedule(time() +  random.negexp(1/11.0));
            }
        }
    }

    class StopCarWashing extends TimeEvent {
        CarWasher theCarWasher; Car theCar;

        StopCarWashing(CarWasher cw, Car c)
          { theCarWasher = cw; theCar = c; }

        public void actions() {
            noOfCustomers++;
            throughTime += time() - theCar.entryTime;
            theCarWasher.into(tearoom);
        }
    }

    class StartCarWashing extends StateEvent {
        Car theCar;

        StartCarWashing(Car c) { theCar = c; }

        public boolean condition() {
            return theCar == (Car) waitingLine.first() &&
                   !tearoom.empty();
        }

        public void actions() {
            theCar.out();
            CarWasher theCarWasher =
                (CarWasher) tearoom.first();
            theCarWasher.out();
            new StopCarWashing(theCarWasher,
                               theCar).schedule(time() + 10);
        }
    }

    public static void main(String args[]) {
        new CarWashSimulation(1);
        new CarWashSimulation(2);
    }
}
```

K. Source code of `javaSimulation`

```
package javaSimulation;

public abstract class Process extends Link {
    protected abstract void actions();

    private final Coroutine myCoroutine = new Coroutine() {
        protected void body() {
            if (MAIN == null)
                MAIN = Process.this;
            actions();
            TERMINATED = terminated = true;
            if (Process.this == MAIN) {
                while (SQS.SUC != SQS)
                    SQS.SUC.cancel();
                MAIN = null;
                return;
            }
            passivate();
        }
    };

    private Process PRED, SUC;
    private double EVTIME;
    private boolean TERMINATED;

    private final static Process SQS = new Process() {
        { EVTIME = -1;  PRED = SUC = this; }
        protected void actions() {}
    };

    private static Process MAIN;

    public final boolean idle() {
        return SUC == null;
    }

    public final boolean terminated() {
        return TERMINATED;
    }

    public final double evTime() {
        if (idle())
            error("No evTime for idle process");
        return EVTIME;
    }

    public final Process nextEv() {
        return SUC == SQS ? null : SUC;
    }
```

```
public static final Process current() {
    return SQS.SUC != SQS ? SQS.SUC : null;
}

public static final double time() {
    return SQS.SUC != SQS ? SQS.SUC.EVTIME : 0;
}

public static final Process main() { return MAIN; }

private static void error(String msg) {
    throw new RuntimeException(msg);
}

public static final void hold(double t) {
    if (SQS.SUC == SQS)
       error("Hold: SQS is empty");
    Process Q = SQS.SUC;
    if (t > 0)
        Q.EVTIME += t;
    t = Q.EVTIME;
    if (Q.SUC != SQS && Q.SUC.EVTIME <= t) {
        Q.cancel();
        Process P = SQS.PRED;
        while (P.EVTIME > t)
            P = P.PRED;
        Q.scheduleAfter(P);
        resume(SQS.SUC);
    }
}

public static final void passivate() {
    if (SQS.SUC == SQS)
       error("Passivate: SQS is empty");
    Process CURRENT = SQS.SUC;
    CURRENT.cancel();
    if (SQS.SUC == SQS)
        error("passivate causes SQS to become empty");
    resume(SQS.SUC);
}

public static final void wait(Head q) {
    if (SQS.SUC == SQS)
       error("Wait: SQS is empty");
     current().into(q);
     Process CURRENT = SQS.SUC;
     CURRENT.cancel();
     if (SQS.SUC == SQS)
         error("wait causes SQS to become empty");
     resume(SQS.SUC);
}
```

```
public static final void cancel(Process p) {
    if (p == null || p.SUC == null)
        return;
    Process CURRENT = SQS.SUC;
    p.cancel();
    if (SQS.SUC != CURRENT)
        return;
    if (SQS.SUC == SQS)
        error("cancel causes SQS to become empty");
    resume(SQS.SUC);
}

private static final class At     {}
private static final class Delay  {}
private static final class Before {}
private static final class After  {}
private static final class Prior  {}


public static final At     at     = new At();
public static final Delay  delay  = new Delay();
public static final Before before = new Before();
public static final After  after  = new After();
public static final Prior  prior  = new Prior();

private static final int direct_code = 0;
private static final int at_code     = 1;
private static final int delay_code  = 2;
private static final int before_code = 3;
private static final int after_code  = 4;
```

```
private static final void activat(boolean reac, Process x,
                                  int code, double t,
                                  Process y, boolean prio) {
    if (x == null || x.TERMINATED ||
        (!reac && x.SUC != null))
        return;
    Process CURRENT = SQS.SUC, P = null;
    double NOW = time();
    switch(code) {
    case direct_code:
        if (x == CURRENT)
            return;
        t = NOW; P = SQS;
        break;
    case delay_code:
        t += NOW;
    case at_code:
        if (t <= NOW) {
            if (prio && x == CURRENT)
                return;
            t = NOW;
        }
        break;
    case before_code:
    case after_code:
         if (y == null || y.SUC == null) {
             x.cancel();
             if (SQS.SUC == SQS)
                 error("reactivate causes SQS " +
                       "to become empty");
             return;
         }
         if (x == y)
             return;
         t = y.EVTIME;
         P = code == before_code ? y.PRED : y;
    }
    if (x.SUC != null)
        x.cancel();
    if (P == null) {
        for (P = SQS.PRED; P.EVTIME > t; P = P.PRED)
            ;
        if (prio)
            while (P.EVTIME == t)
                P = P.PRED;
    }
    x.EVTIME = t;
    x.scheduleAfter(P);
    if (SQS.SUC != CURRENT)
        resume(current());
}
```

```
public static final void activate(Process p) {
    activat(false, p, direct_code, 0, null, false);
}

public static final void activate(Process p,
                      At at, double t) {
    activat(false, p, at_code, t, null, false);
}

public static final void activate(Process p,
                      At at, double t, Prior prior) {
    activat(false, p, at_code, t, null, true);
}

public static final void activate(Process p,
                      Delay delay, double t) {
    activat(false, p, delay_code, t, null, false);
}

public static final void activate(Process p,
                      Delay d, double t, Prior prior) {
    activat(false, p, delay_code, t, null, true);
}

public static final void activate(Process p1,
                      Before before, Process p2) {
    activat(false, p1, before_code, 0, p2, false);
}

public static final void activate(Process p1,
                      After after, Process p2) {
    activat(false, p1, after_code, 0, p2, false);
}
```

```
    public static final void reactivate(Process p) {
        activat(true, p, direct_code, 0, null, false);
    }

    public static final void reactivate(Process p,
                        At at, double t) {
        activat(true, p, at_code, t, null, false);
    }

    public static final void reactivate(Process p,
                        At at, double t, Prior prior) {
        activat(true, p, at_code, t, null, true);
    }

    public static final void reactivate(Process p,
                        Delay delay, double t) {
        activat(true, p, delay_code, t, null, false);
    }

    public static final void reactivate(Process p,
                        Delay d, double t, Prior prior) {
        activat(true, p, delay_code, t, null, true);
    }

    public static final void reactivate(Process p1,
                        Before before, Process p2) {
        activat(true, p1, before_code, 0, p2, false);
    }

    public static final void reactivate(Process p1,
                        After after, Process p2) {
        activat(true, p1, after_code, 0, p2, false);
    }

    private final void scheduleAfter(Process p) {
        PRED = p;
        SUC = p.SUC;
        p.SUC = SUC.PRED = this;
    }

    private final void cancel() {
        PRED.SUC = SUC;
        SUC.PRED = PRED;
        PRED = SUC = null;
    }
}
```

L. Process-based car wash simulation with `javaSimulation`

```
import javaSimulation.*;
import javaSimulation.Process;

public class CarWashSimulation extends Process {
    int noOfCarWashers;
    double simPeriod = 1000000;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    double throughTime;
    int noOfCustomers, maxLength;
    long startTime = System.currentTimeMillis();

    CarWashSimulation(int n) { noOfCarWashers = n; }

    public void actions() {
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        activate(new CarGenerator());
        hold(simPeriod + 1000000);
        report();
    }

    void report() {
        System.out.println(noOfCarWashers +
                           " car washer simulation");
        System.out.println("No.of cars through the system = " +
                           noOfCustomers);
        java.text.NumberFormat fmt =
            java.text.NumberFormat.getNumberInstance();
        fmt.setMaximumFractionDigits(2);
        System.out.println("Av.elapsed time = " +
            fmt.format(throughTime/noOfCustomers));
        System.out.println("Maximum queue length = " +
                           maxLength);
        System.out.println("\nExecution time: " +
            fmt.format((System.currentTimeMillis()
                        - startTime)/1000.0) + " secs.\n");
     }
```

```
    class Car extends Process {
        public void actions() {
            double entryTime = time();
            into(waitingLine);
            int qLength = waitingLine.cardinal();
            if (maxLength < qLength)
                maxLength = qLength;
            if (!tearoom.empty())
                activate((CarWasher) tearoom.first());
            passivate();
            noOfCustomers++;
            throughTime += time() - entryTime;
        }
    }

    class CarWasher extends Process {
        public void actions() {
            while (true) {
                out();
                while (!waitingLine.empty()) {
                    Car served = (Car) waitingLine.first();
                    served.out();
                    hold(10);
                    activate(served);
                }
                wait(tearoom);
            }
        }
    }

    class CarGenerator extends Process {
        public void actions() {
             while (time() <= simPeriod) {
                  activate(new Car());
                  hold(random.negexp(1/11.0));
             }
        }
    }

    public static void main(String args[]) {
        activate(new CarWashSimulation(1));
        activate(new CarWashSimulation(2));
    }
}
```

M. Event-based car wash simulation with `javaSimulation`

```
import javaSimulation.*;
import javaSimulation.Process;

public class CarWashSimulation extends Process {
    int noOfCarWashers;
    double simPeriod = 1000000;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    double throughTime;
    int noOfCustomers, maxLength;
    long startTime = System.currentTimeMillis();

    CarWashSimulation(int n) { noOfCarWashers = n; }

    public void actions() {
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        activate(new CarArrival());
        hold(simPeriod+1000000);
        report();
    }

    void report() {
        System.out.println(noOfCarWashers +
                           " car washer simulation");
        System.out.println("No.of cars through the system = " +
                           noOfCustomers);
        java.text.NumberFormat fmt =
            java.text.NumberFormat.getNumberInstance();
        fmt.setMaximumFractionDigits(2);
        System.out.println("Av.elapsed time = " +
            fmt.format(throughTime/noOfCustomers));
        System.out.println("Maximum queue length = " +
                           maxLength);
        System.out.println("\nExecution time: " +
            fmt.format((System.currentTimeMillis()
                        - startTime)/1000.0) + " secs.\n");
     }


    class CarWasher extends Link {}

    class Car extends Link {
        double entryTime = time();
    }
```

```
    class CarArrival extends Process {
        public void actions() {
            if (time() > simPeriod)
                 return;
            Car theCar = new Car();
            theCar.into(waitingLine);
            int qLength = waitingLine.cardinal();
            if (maxLength < qLength)
                maxLength = qLength;
            if (!tearoom.empty())
                activate(new StartCarWashing(
                             (CarWasher) tearoom.first()));
            activate(new CarArrival(),
                     delay, random.negexp(1/11.0));
        }
    }

    class StartCarWashing extends Process {
        CarWasher theCarWasher;

        StartCarWashing(CarWasher cw) { theCarWasher = cw; }

        public void actions() {
            theCarWasher.out();
            Car theCar = (Car) waitingLine.first();
            theCar.out();
            activate(new StopCarWashing(theCarWasher, theCar),
                     delay, 10);
        }
    }

    class StopCarWashing extends Process {
        CarWasher theCarWasher; Car theCar;

        StopCarWashing(CarWasher cw, Car c)
          { theCarWasher = cw; theCar = c; }

        public void actions() {
            theCarWasher.into(tearoom);
            noOfCustomers++;
            throughTime += time() - theCar.entryTime;
            if (!waitingLine.empty())
                activate(new StartCarWashing(theCarWasher));
        }
    }

    public static void main(String args[]) {
        activate(new CarWashSimulation(1));
        activate(new CarWashSimulation(2));
    }
}
```

N. Activity-based car wash simulation with `javaSimulation`

```
import javaSimulation.*;
import javaSimulation.Process;

public class CarWashSimulation extends Process {
    int noOfCarWashers;
    double simPeriod = 1000000;
    Head tearoom = new Head();
    Head waitingLine = new Head();
    Random random = new Random(5);
    double throughTime;
    int noOfCustomers, maxLength;
    long startTime = System.currentTimeMillis();

    CarWashSimulation(int n) {
        noOfCarWashers = n;
    }

     public void actions() {
        for (int i = 1; i <= noOfCarWashers; i++)
            new CarWasher().into(tearoom);
        activate(new CarArrival());
        hold(simPeriod + 1000000);
        report();
    }

    void report() {
        System.out.println(noOfCarWashers +
                           " car washer simulation");
        System.out.println("No.of cars through the system = " +
                           noOfCustomers);
        java.text.NumberFormat fmt =
            java.text.NumberFormat.getNumberInstance();
        fmt.setMaximumFractionDigits(2);
        System.out.println("Av.elapsed time = " +
            fmt.format(throughTime/noOfCustomers));
        System.out.println("Maximum queue length = " +
                           maxLength);
        System.out.println("\nExecution time: " +
            fmt.format((System.currentTimeMillis()
                       - startTime)/1000.0) + " secs.\n");
    }

    class CarWasher extends Link {}

    class Car extends Link {
        double entryTime = time();
    }
```

```
    class CarArrival extends Process {
        public void actions() {
            if (time() <= simPeriod) {
                Car theCar = new Car();
                theCar.into(waitingLine);
                activate(new CarWashing(theCar));
                int qLength = waitingLine.cardinal();
                if (maxLength < qLength)
                    maxLength = qLength;
                hold(random.negexp(1/11.0));
                activate(new CarArrival());
            }
       }
    }

    class CarWashing extends Process {
        Car theCar;
        CarWasher theCarWasher;

        CarWashing(Car c) { theCar = c; }

        public void actions() {
            if (theCar == (Car) waitingLine.first() &&
                !tearoom.empty()) {
                theCar.out();
                theCarWasher = (CarWasher) tearoom.first();
                theCarWasher.out();
                if (!waitingLine.empty())
                    activate(new CarWashing(
                                (Car) waitingLine.first()));
                hold(10);
                theCarWasher.into(tearoom);
                noOfCustomers++;
                throughTime += time() - theCar.entryTime;
                if (!waitingLine.empty())
                    activate(new CarWashing(
                                (Car) waitingLine.first()));
            }
        }
    }

    public static void main(String args[]) {
        activate(new CarWashSimulation(1));
        activate(new CarWashSimulation(2));
    }
}
```

O. Source code of the `javaCoroutine` package

```
package javaCoroutine;

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

    public static final void resume(Coroutine next) {
         if (next == null)
             error("resume non-existing coroutine");
         if (next.terminated)
             error("resume terminated coroutine");
         if (next.caller != null)
             error("resume attached coroutine");
         if (next == current)
             return;
         while (next.callee != null)
              next = next.callee;
         next.enter();
    }

    public static final void call(Coroutine next) {
        if (next == null)
            error("call non-existing coroutine");
        if (next.terminated)
            error("call terminated coroutine");
        if (next.caller != null)
            error("call attached coroutine");
        if (current != null)
            current.callee = next;
        next.caller = current;
        while (next.callee != null)
            next = next.callee;
        if (next == current)
            error("call operating coroutine");
        next.enter();
    }

    public static final void detach() {
        Coroutine next = current.caller;
        if (next != null) {
            current.caller = next.callee = null;
            next.enter();
        }
        else if (main != null && current != main)
            main.enter();
    }
```

```
public static final Coroutine currentCoroutine() {
    return current;
}
public static final Coroutine mainCoroutine() {
    return main;
}

private final class Runner extends Thread {
    Coroutine myCoroutine;
    Runner nextFree;

    Runner(Coroutine c) {
        myCoroutine = c;
        setDaemon(true);
    }

    public void run() {
        while (true) {
            myCoroutine.body();
            if (!myCoroutine.terminated) {
                myCoroutine.terminated = true;
                detach();
            }
            if (myCoroutine == Coroutine.main) {
                Coroutine.current = null;
                synchronized(Runner.class) {
                    Coroutine.main = null;
                    Runner.class.notify();
                }
                return;
            }
            nextFree = firstFree;
            firstFree = this;
            try {
                synchronized(this) {
                    wait();
                }
            } catch (InterruptedException e) {}
        }
    }

    void go() {
        if (!isAlive())
            start();
        else
        synchronized(this) {
            notify();
        }
    }
}
```

```
    private static void error(String msg) {
        throw new RuntimeException(msg);
    }

    private static Coroutine current, main;
    private Coroutine caller, callee;
    protected boolean terminated;
    private Runner myRunner;
    private static Runner firstFree;

    private void enter() {
        if (myRunner == null) {
            if (firstFree == null)
                myRunner = new Runner(this);
            else {
                myRunner = firstFree;
                firstFree = firstFree.nextFree;
                myRunner.myCoroutine = this;
            }
        }
        if (main == null) {
            main = current = this;
            myRunner.go();
            synchronized(Runner.class) {
                try {
                    while (main != null)
                        Runner.class.wait();
                } catch (InterruptedException e) {}
            }
            return;
        }
        Coroutine old_current = current;
        synchronized(old_current.myRunner) {
            current = this;
            myRunner.go();
            if (old_current.terminated)
                return;
            try {
                old_current.myRunner.wait();
            } catch(InterruptedException e) {}
        }
    }
}
```

P. Test program for `javaCoroutine`

```
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

*/
```