User's guide to `javaSimulation`

1. Introduction

This guide describes `javaSimulation`, a Java package for process-based discrete event simulation. The package is a Java implementation of the simulation facilities provided by SIMULA [1][2][3]. In addition to the simulation facilities, the package also includes the facilities for list manipulation, random number drawing and coroutine sequencing as found in SIMULA.

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.

In `JavaSimulation` the processes are described in one or more subclasses of the class `Process`. Their life cycles are described by overriding the abstract method `actions`.

```
class Customer extends Process {
    public void actions() {
       // The life cycle of a customer
    }
}
```

An outline of class `Process` is shown below.

```
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);
}
```

The `time` method returns the current simulated time.

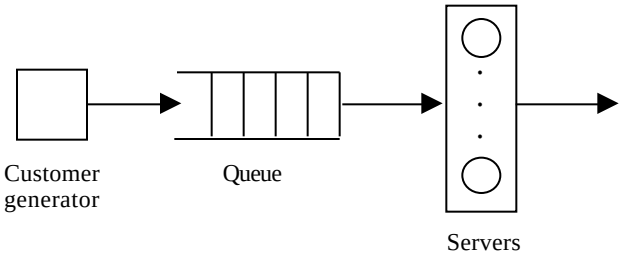
The `activate` method is used to make a specified process start executing its actions.

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.

Since class `Process` is a subclass of the class `Link`, each process has the capability of being a member of a queue. The `wait` method suspends the calling process and adds it to a queue.

The use of these facilities is best explained through an example. The figure below shows the system to be simulated.



It is an abstraction of, for example, a bank where the customers wait in a single queue for any of the tellers.

The interval between arrivals is uniformly distributed from 1 to 3 minutes. The time for serving a customer is normally distributed with a mean value of 4 minutes and a standard deviation of 1 minute. Simulation is used to find the average time a customer spends in the system.

First the processes of the system are identified.

One process is the generator of customers. It should repeatedly generate a customer and wait a period of time.

Customers and servers may also be conceived as processes. When a customer arrives, he activates an idle server (if any) and then waits in the queue. An active server repeatedly serves customers from the queue until the queue is empty. Then the server passivates.

Using the `javaSimulation` package this model may be described by the following three classes:

```
class CustomerGenerator extends Process {
    public void actions() {
        while (true) {
            activate(new Customer());
            hold(random.uniform(1, 3));
        }
    }
}
```

```
class Customer extends Process {
    public void actions() {
        double arrivalTime = time();
        into(customerQueue);
        if (!serverQueue.empty())
            activate((Process)
                     serverQueue.first());
        passivate();
        customers++;
        throughTime += time() - arrivalTime;
    }
}
```

```
class Server extends Process {
    public void actions() {
        while (true) {
            out();
            while (!customerQueue.empty()) {
                Customer served =
                    (Customer) customerQueue.first();
                served.out();
                hold(random.normal(4, 1));
                activate(served);
            }
            wait(serverQueue);
        }
     }
}
```

Customers wait in a queue called `customerQueue`. When a server activates a customer after the completion of a service, the customer just updates statistics and terminates.

Idle servers wait in a queue called `serverQueue`. When a customer activates an idle server, the server leaves this queue and serves the customer.

A complete program for simulating the system is shown below. The classes `CustomerGenerator`, `Customer` and `Server` are as defined above.

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

public class Simulation extends Process {
    int servers = 2;
    Random random = new Random(7913);

    Head customerQueue = new Head();
    Head serverQueue = new Head();
    int customers;
    double throughTime;

    public void actions() {
        for (int i = 1; i <= servers; i++)
            new Server().into(serverQueue);
        activate(new CustomerGenerator());
        hold(600);
        System.out.println(throughTime/customers);
    }

    class CustomerGenerator extends Process { ... }

    class Customer extends Process { ... }

    class Server extends Process { ... }

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

This program simulates a system with two servers over a period of 600 minutes.

The `main` method starts the simulation by activating an object of a `Process`-derived class. This object is called the *main process* and acts as main program for the simulation. The simulation stops when the main process terminates.

In this program the main process inserts a number of servers in the queue of idle servers, activates the customer generator and waits 600 units of simulated time. Before it terminates, it prints the average time spent by customers in the system.

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 simulation above is based on *active* customers and *active* servers. A model based on *active* customers and *passiv*e servers may be obtained by defining the classes `Customer` and `Server` as follows:

```
class Customer extends Process {
    public void actions() {
        double arrivalTime = time();
        if (serverQueue.empty()) {
            wait(customerQueue);
            out();
        }
        Server server = (Server) serverQueue.first();
        server.out();
        hold(random.normal(4, 1));
        server.into(serverQueue);
        if (!customerQueue.empty())
            activate((Customer) customerQueue.first());
        customers++;
        throughTime += time() - arrivalTime;
    }
}
```

```
class Server extends Link {}
```

A model based on *passive* customers and *active* serves may be obtained by defining the `CustomerGenerator`, `Customer` and `Server` classes as follows:

```
class CustomerGenerator extends Process {
    public void actions() {
        while (true) {
            new Customer().into(customerQueue);
            if (!serverQueue.empty())
              activate((Server) serverQueue.first());
            hold(random.uniform(1, 3));
        }
    }
}
```

```
class Server extends Process {
    public void actions() {
        while (true) {
            out();
            while (!customerQueue.empty())
                Customer served =
                    (Customer) customerQueue.first();
                served.out();
                hold(random.normal(4, 1));
                customers++;
                throughTime += time() - served.arrivalTime;
            }
            wait(serverQueue);
        }
    }
}
```

```
class Customer extends Link {
    double arrivalTime = time();
}
```

All three versions produce the same output (in less than a second):

```
11.79438522339415
```

However, the last version executes faster than the two other versions. This is due to the computational overhead induced by the usage of threads internally in `javaSimulation`. Each `Process` has its `actions` executed by a thread. In contrast to the two first versions, the last version has a small number of processes. In this version there are only three processes (the main process and the two server processes).

2. The simulation facilities of `javaSimulation`

The design of the `javaSimulation` package follows very closely the design of the built-in package for discrete event simulation in SIMULA, class SIMULATION. The development of `javaSimulation` has been described in [4].

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 smallest event time is resumed.

The currently active process always has the smallest 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.

3. The list handing facilities of `javaSimulation`

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.

4. The random drawing facilities of `javaSimulation`

The `javaSimulation` 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` ≥ `1`, and always `false` if `a` ≤ `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` ≤ x < `b`. If `b` ≤ `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 form 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

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

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.

References

1. O.-J. Dahl, B. Myhrhaug & K. Nygaard,
   Common Base Lanugae,
   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,
   Discrete Event Simulation in Java,
   Datalogiske skrifter, No. 89, Roskilde University (2000).