# SSJ User's Guide

## Package `simevents`

## Simulation Clock and Event List Management

Version: December 21, 2006

This package provides the simulation clock and tools to manage the future events list. These are the basic tools for discrete-event simulation. Several different implementations of the event list are offered. Some basic tools for continuous simulation (i.e., solving differential equations with respect to time) are also provided.

# Contents

# Overview

The scheduling part of discrete-event simulations is managed by the "chief-executive" class `Sim`, which contains the simulation clock and the central monitor. The event list is taken from one of the implementations of the interface `EventList`, which provide different kinds of event list implementations. One can change the default `SplayTree` event list implementation via the method `init`. The class `Event` provides the facilities for creating and scheduling events in the simulation. Each type of event must be defined by extending the class `Event`. The class `Continuous` provides elementary tools for continuous simulation, where certain variables vary continuously in time according to ordinary differential equations.

The class `LinkedListStat` implements *doubly linked* lists, with tools for inserting, removing, and viewing objects in the list, and automatic statistical collection. These lists can contain any kind of `Object`.

# Sim

This static class contains the executive of a discrete-event simulation. It maintains the simulation clock and starts executing the events in the appropriate order. Its methods permit one to start, stop, and (re)initialize the simulation, and read the simulation clock.

---

```
package umontreal.iro.lecuyer.simevents;

public final class Sim
```

### Methods

`public static double time()`

Returns the current value of the simulation clock.

`public static void init()`

Reinitializes the simulation executive by clearing up the event list, and resetting the simulation clock to zero. This method must not be used to initialize process-driven simulation; `SimProcess.init` must be used instead.

`public static void init (EventList evlist)`

Same as `init`, but also chooses `evlist` as the event list to be used. For example, `init (new DoublyLinked())` initializes the simulation with a doubly linked linear structure for the event list. This method must not be used to initialize process-driven simulation; `SimProcess.init` must be used instead.

`public static EventList getEventList()`

Gets the currently used event list.

`public static Event removeFirstEvent()`

This method is used by the package `simprocs`; *it should not be used directly by a simulation program.* It removes the first event from the event list and sets the simulation clock to its event time.

`public static void start()`

Starts the simulation executive. There must be at least one `Event` in the event list when this method is called.

`public static void stop()`

Tells the simulation executive to stop as soon as it takes control, and to return control to the program that called `start`. This program will then continue executing from the instructions right after its call to `Sim.start`. If an `Event` is currently executing (and this event has just called `Sim.stop`), the executive will take control when the event terminates its execution.

# Event

This abstract class provides event scheduling tools. Each type of event should be defined as a subclass of the class `Event`, and should provide an implementation of the method `actions` which is executed when an event of this type occurs. The instances of these subclasses are the actual events.

When an event is constructed, it is not scheduled. It must be scheduled separately by calling one of the scheduling methods `schedule`, `scheduleNext`, `scheduleBefore`, etc. An event can also be cancelled before it occurs.

---

```
package umontreal.iro.lecuyer.simevents;

public abstract class Event
```

### Constructor

`public Event()`

Constructs a new event instance, which can be placed afterwards into the event list by calling one of the `schedule...` variants. For example, if `Bang` is an `Event` subclass, the statement "`new Bang().scheduleNext();`" creates a new `Bang` event and places it at the beginning of the event list.

### Methods

`public void schedule (double delay)`

Schedules this event to happen in `delay` time units, i.e., at time `Sim.time() + delay`, by inserting it in the event list. When two or more events are scheduled to happen at the same time, they are placed in the event list (and executed) in the same order as they have been scheduled.

`public void scheduleNext()`

Schedules this event as the *first* event in the event list, to be executed at the current time (as the next event).

`public void scheduleBefore (Event other)`

Schedules this event to happen just before, and at the same time, as the event `other`. For example, if `Bing` and `Bang` are `Event` subclasses, after the statements

```
Bang bigOne = new Bang().schedule (12.0);
new Bing().scheduleBefore (bigOne);
```

the event list contains two new events scheduled to happen in 12 units of time: a `Bing` event, followed by a `Bang` called `bigOne`.

`public void scheduleAfter (Event other)`

Schedules this event to happen just after, and at the same time, as the event `other`.

`public void reschedule (double delay)`

Cancels this event and reschedules it to happen in `delay` time units.

`public boolean cancel()`

Cancels this event before it occurs. Returns `true` if cancellation succeeds (this event was found in the event list), `false` otherwise.

`public static final boolean cancel (String type)`

Finds the first occurence of an event of class "type" in the event list, and cancels it. Returns `true` if cancellation succeeds, `false` otherwise.

`public final double time()`

Returns the (planned) time of occurence of this event.

`public final double setTime (double time)`

Sets the (planned) time of occurence of this event to `time`. This method should never be called after the event was scheduled, otherwise the events would not execute in ascending time order anymore.

`public abstract void actions();`

This is the method that is executed when this event occurs. Every subclass of `Event` that is to be instantiated must provide an implementation of this method.

# Continuous

This abstract class provides the basic structures and tools for continuous-time simulation, where certain variables evolve continuously in time, according to differential equations. Such continuous variables can be mixed with events and processes.

Each type of continuous-time variable should be defined as a subclass of `Continuous`. The instances of these subclasses are the actual continuous-time variables. Each subclass must implement the method `derivative` which returns its derivative with respect to time. The trajectory of this variable is determined by integrating this derivative. The subclass may also reimplement the method `afterEachStep`, which is executed immediately after each integration step. By default (in the class `Continuous`), this method does nothing. This method could, for example, verify if the variable has reached a given threshold, or update a graphical illustration of the variable trajectory.

One of the methods `selectEuler`, `selectRungeKutta2` or `selectRungeKutta4` must be called before starting any integration. These methods permit one to select the numerical integration method and the step size `h` (in time units) that will be used for *all* continuous-time variables. For all the methods, an integration step at time $t$ changes the values of the variables from their old values at time $t - h$ to their new values at time $t$.

Each integration step is scheduled as an event and added to the event list. When creating a continuous variable class, the `toString` method can be overriden to display information about the continuous variable. This information will be displayed when formating the event list as a string.

---

```
package umontreal.iro.lecuyer.simevents;

public abstract class Continuous
```

### Constructor

`public Continuous()`

Constructs a new continuous-time variable, *without* initializing it.

### Methods

`public void init (double val)`

Initializes or reinitializes the continuous-time variable to `val`.

`public double value()`

Returns the current value of this continuous-time variable.

`public static void selectEuler (double h)`

Selects the Euler method as the integration method, with the integration step size `h`, in time units.

`public static void selectRungeKutta2 (double h)`

Selects a Runge-Kutta method of order 2 as the integration method to be used, with step size `h`.

`public static void selectRungeKutta4 (double h)`

Selects a Runge-Kutta method of order 4 as the integration method to be used, with step size `h`.

`public void startInteg()`

Starts the integration process that will change the state of this variable at each integration step.

`public void startInteg (double val)`

Same as `startInteg`, after initializing the variable to `val`.

`public void stopInteg()`

Stops the integration process for this continuous variable. The variable keeps the value it took at the last integration step before calling `stopInteg`.

`public abstract double derivative (double t);`

This method should return the derivative of this variable with respect to time, at time $t$. Every subclass of `Continuous` that is to be instantiated must implement it. If the derivative does not depend explicitly on time, $t$ becomes a dummy parameter. Internally, the method is used with $t$ not necessarily equal to the current simulation time.

`public void afterEachStep()`

This method is executed after each integration step for this `Continuous` variable. Here, it does nothing, but every subclass of `Continuous` may reimplement it.

# LinkedListStat

This class extends `java.util.LinkedList`, with statistical probes integrated in the class to provide automatic collection of statistics on the sojourn times of objects in the list and the size of the list as a function of time. The automatic statistical collection can be enabled or disabled for each list, to reduce overhead.

The iterators returned by the `listIterator()` method are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`.

---

```
package umontreal.iro.lecuyer.simevents;

public class LinkedListStat extends LinkedList
```

## Constructors

```
public LinkedListStat()
```

Constructs a new list, initially empty.

```
public LinkedListStat (Collection c)
```

Constructs a list containing the elements of the specified collection.

```
public LinkedListStat (String name)
```

Constructs a new list with name `name`. This name can be used to identify the list in traces and reports.

```
public LinkedListStat (Collection c, String name)
```

Constructs a new list containing the elements of the specified collection `c` and with name `name`. This name can be used to identify the list in traces and reports.

## List interface methods

See the JDK documentation for more information about these methods.

```
public void clear()
```

```
public void addFirst(Object obj)
```

```
public void addLast(Object obj)
```

```
public void add(int index, Object obj)
```

```
public boolean add(Object obj)
```

```
public boolean addAll(Collection c)
```

```
public boolean addAll(int index, Collection c)
```

```
public Object getFirst()
```

```
public Object getLast()
```

```
public Object get(int index)
```

```
public boolean contains(Object obj)
```

```
public int size()
```

```
public Object removeFirst()
```

```
public Object removeLast()
```

```
public boolean remove(Object o)
```

```
public Object remove(int index)
```

```
public int indexOf(Object obj)
```

```
public int lastIndexOf(Object obj)
```

```
public Object clone()
```

```
public Object[] toArray()
```

```
public Object[] toArray(Object a[])
```

```
public ListIterator listIterator(int index)
```

```
protected ListIterator getListIterator(int index)
```

### Statistic collection methods

```
public void setStatCollecting(boolean b)
```

Starts or stops collecting statistics on this list. If the statistical collection is turned ON, the method creates two statistical probes if they do not exist yet. The first one, of the class `Accumulate`, measures the evolution of the size of the list as a function of time. It can be accessed by the method `statSize`. The second one, of the class `Tally`, and accessible via `statSojourn`, samples the sojourn times in the list of the objects removed during the observation period, i.e., between the last initialization time of this statistical probe and the current time. The method automatically calls `initStat` to initialize these two probes. When this method is used, it is normally invoked immediately after calling the constructor of the list.

`public void initStat()`

Reinitializes the two statistical probes created by `setStatCollecting (true)` and makes an update for the probe on the list size.

`public Accumulate statSize()`

Returns the statistical probe on the evolution of the size of the list as a function of the simulation time. This probe exists only if `setStatCollecting (true)` has been called for this list.

`public Tally statSojourn()`

Returns the statistical probe on the sojourn times of the objects in the list. This probe exists only if `setStatCollecting (true)` has been called for this list.

`public String report()`

Returns a string containing a statistical report on the list, provided that `setStatCollecting (true)` has been called before for this list. Even If `setStatCollecting` was called with `false` afterward, the report will be made for the collected observations. If the probes do not exist, i.e., `setStatCollecting` was never called for this object, an illegal state exception will be thrown.

`public String getName()`

Returns the name associated to this list, or `null` if no name was assigned.

# Accumulate

A subclass of `StatProbe`, for collecting statistics on a variable that evolves in simulation time, with a piecewise-constant trajectory. Each time the variable changes its value, the method `update` must be called to inform the probe of the new value. The probe can be reinitialized by `init`.

---

```
package umontreal.iro.lecuyer.simevents;

public class Accumulate extends StatProbe implements Cloneable
```

## Constructors

`public Accumulate()`

Constructs a new `Accumulate` statistical probe and initializes it by invoking `init()`.

`public Accumulate (String name)`

Construct and initializes a new `Accumulate` statistical probe with name `name` and initial time 0.

## Methods

`public void init()`

Initializes the statistical collector and puts the current value of the corresponding variable to 0. A call to `init` should normally be followed immediately by a call to `update` to give the value of the variable at the initialization time.

`public void init (double x)`

Same as `init` followed by `update(x)`.

`public void update()`

Updates the accumulator using the last value passed to `update`.

`public void update (double x)`

Gives a new observation `x` to the statistical collector. If broadcasting to observers is activated for this object, this method will also transmit the new information to the registered observers by invoking the methods `setChanged` and `notifyObservers (new Double (x))` inherited from `Observable`.

`public double getInitTime()`

Returns the initialization time for this object. This is the simulation time when `init` was called for the last time.

`public double getLastTime()`

Returns the last update time for this object. This is the simulation time of the last call to `update` or the initialization time if `update` was never called after `init`.

`public double getLastValue()`

Returns the value passed to this probe by the last call to its `update` method (or the initial value if `update` was never called after `init`).

`public Object clone()`

Clone this object.

# EventList

An interface for implementations of event lists. Different implementations are provided in SSJ: doubly linked list, splay tree, Henrickson's method, etc. The *events* in the event list are objects of the class `Event`. The method `Sim.init` permits one to select the actual implementation used in a simulation [1].

To allow the user to print the event list, the `toString` method from the `Object` class should be reimplemented in all `EventList` implementations. It will return a string in the following format: "`Contents of the event list` *event list class* `:`" for the first line and each subsequent line has format "*scheduled event time* `:` *event string*". The *event string* is obtained by calling the `toString` method of the event objects. The string should not end with the end-of-line character.

The following example is the event list of the bank example, printed at 10h30. See `examples.pdf` for more information.

```
Contents of the event list SplayTree :
   10.51 : BankEv$Arrival@cfb549
   10.54 : BankEv$Departure@8a7efd
      11 : BankEv$3@86d4c1
      14 : BankEv$4@f9f9d8
      15 : BankEv$5@820dda
```

The event time (obtained by calling the `Event.time`) must not be modified by implementations of this interface because the `Event` class already takes care of that.

---

```
package umontreal.iro.lecuyer.simevents.eventlist;

public interface EventList
```

`public boolean isEmpty();`

Returns `true` if and only if the event list is empty (no event is scheduled).

`public void clear();`

Empties the event list, i.e., cancels all events.

`public void add (Event ev);`

Adds a new event in the event list, according to the time of `ev`. If the event list contains events scheduled to happen at the same time as `ev`, `ev` must be added after all these events.

`public void addFirst (Event ev);`

Adds a new event at the beginning of the event list. The given event `ev` will occur at the current simulation time.

`public void addBefore (Event ev, Event other);`

Same as `add`, but adds the new event `ev` immediately before the event `other` in the list.

`public void addAfter (Event ev, Event other);`

Same as `add`, but adds the new event `ev` immediately after the event `other` in the list.

`public Event getFirst();`

Returns the first event in the event list. If the event list is empty, returns `null`.

`public Event getFirstOfClass (String cl);`

Returns the first event of the class `cl` (a subclass of `Event`) in the event list. If no such event are found, returns `null`.

`public ListIterator listIterator();`

Returns a list iterator over the elements of the class `Event` in this list.

`public boolean remove (Event ev);`

Removes the event `ev` from the event list (cancels this event). Returns `true` if and only if the event removal has succeeded.

`public Event removeFirst();`

Removes the first event from the event list (to cancel or execute this event). Returns the removed event. If the list is empty, then `null` is returned.

# DoublyLinked

An implementation of `EventList` using a doubly linked linear list. Each event is stored into a list node that contains a pointer to its following and preceding events. Adding an event requires a linear search to keep the event list sorted by event time. Removing the first event is done in constant time because it simply removes the first list node. List nodes are recycled for increased memory management efficiency.

---

```
package umontreal.iro.lecuyer.simevents.eventlist;
```

```
public class DoublyLinked implements EventList
```

# SplayTree

An implementation of `EventList` using a splay tree [3]. This tree is like a binary search tree except that when it is modified, the affected node is moved to the top. The rebalancing scheme is simpler than for a *red black* tree and can avoid the worst case of the linked list. This gives a $O(\log(n))$ average time for adding or removing an event, where $n$ is the size of the event list.

---

```
package umontreal.iro.lecuyer.simevents.eventlist;

public class SplayTree implements EventList
```

# BinaryTree

An implementation of `EventList` using a binary search tree. Every event is stored into a tree node which has left and right children. Using the event time as a comparator, the left child is always smaller than its parent whereas the right is greater or equal. This allows an average $O(\log(n))$ time for adding an event and searching the first event, where $n$ is the number of events in the structure. There is less overhead for adding and removing events than splay tree or red black tree. However, in the worst case, adding or removing could be done in time proportional to $n$ because the binary search tree can be turned into a linked list.

---

```
package umontreal.iro.lecuyer.simevents.eventlist;
```

```
public class BinaryTree implements EventList
```

# Henriksen

An implementation of `EventList` using the doubly-linked index list of Henriksen [2].

Events are stored in a normal doubly-linked list. An additionnal index array is added to the structure to allow quick access to the events.

---

```
package umontreal.iro.lecuyer.simevents.eventlist;

public class Henriksen implements EventList
```

# RedblackTree

An implementation of `EventList` using a *red black* tree, which is similar to a binary search tree except that every node is colored red or black. When modifying the structure, the tree is reorganized for the colors to satisfy rules that give an average $O(\log(n))$ time for removing the first event or inserting a new event, where $n$ is the number of elements in the structure. However, adding or removing events imply reorganizing the tree and requires more overhead than a binary search tree.

The present implementation uses the Java 2 `TreeMap` class which implements a red black tree for general usage. This event list implementation is not efficient.

---

```
package umontreal.iro.lecuyer.simevents.eventlist;
```

```
public class RedblackTree implements EventList
```

# References


[1] J. H. Kingston. Analysis of tree algorithms for the simulation event lists. *Acta Informatica*, 22:15–33, 1985.

[2] J. H. Kingston. Analysis of Henriksen's algorithm for the simulation event set. *SIAM Journal on Computing*, 15:887–902, 1986.

[3] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, 1985.