Discrete-Event Simulation of Queueing Systems in `Java`: The `jsimulation` and `jqueues` Libraries

Guided Tour

Jan de Jongh

Release 5.2, 5.3-SNAPSHOT

DRAFT

Print/Release Date 20180409

Preface

Personally, I (merely) scratched the surface of queueing theory at Twente University back in the eighties, while working on my Master's Thesis on an operating system for *transputers*. Transputers are fast RISC processors with multiple on-chip communication links; back then, they were envisioned to become the building blocks of future massively parallel computer systems. Since our main applications of interest were in robotics, I attempted basic queueing theory in an attempt to find hard real-time response-time guarantees, in order to meet physical-world, mostly safety-related, deadlines.

During the largest part of the nineties, I worked on my PhD at Delft University of Technology. This time, I got to study queueing systems modeling *distributed computing systems*, which by then had overtaken parallel systems in terms of scientific interest. The main purpose of the research was to devise and analyze scheduling strategies for dividing in space and in time the computing resources of a (closed) distributed system among groups of users, according to predefined policies (named *share scheduling* at that time). In order to gain quantitative insight, I used the classic `DEMOS` (Discrete Event Modeling On Simula) software running on the `SIMULA` programming language. I made several modifications and extensions to the software, in order for it to suit my needs. For instance, it lacked support for so-called *processor-sharing* queueing disciplines in which a server ("processor") distributes at any time its service capacity among (a subset of) jobs present. In addition, I needed a non-standard set of statistics gathered from the simulation runs. In the end, both `DEMOS` and `SIMULA` itself proved flexible enough to study the research questions.

Like `DEMOS`, the `jsimulation` and `jqueues` `Java` software packages described in this book feature discrete-event simulation of queueing systems. The libraries are, as a combo, somewhat comparable to the `DEMOS`, yet there are important differences nonetheless. For instance, the libraries focus exclusively at *algorithmic* modeling of queueing systems and job visits; they do not cover additionally required features like sophisticated random-number generation, probability distributions, gathering and analyzing statistics, and sophisticated reporting; features all integrated in `DEMOS`. In that sense, `DEMOS` is a more complete package. On the other hand, the packages feature a larger range of queueing-system types, and, for instance, a model for constructing new queueing systems through *composition* of other queues. In addition, much care has been put into the *atomicity* of certain events, which allows for a wider range of *queue invariants* supported. Despite these differences, `DEMOS` has been a major inspiration in the design and implementation of `jsimulation` and `jqueues`.

For my current employer, TNO, I have performed, over the past decade (or even decades?), many simulation studies in `Java` related to the vehicle-to-vehicle communications in (future) Intelligent Transportation Systems, studying, for instance, position dissemination over CSMA/CA wireless networks for Cooperative Adaptive Cruise Control and Platooning. At some point, I realized that it would be feasible to extract some useful and stable `Java` libraries from my ever increasing software repositories, and release them into the public domain. So, in a way, the libraries can be considered "collateral damage" from a variety of projects.

Chapter 1

Introduction

Queueing systems deal with the general notion of *waiting* for (the completion of) "something". They are ubiquitously and often annoyingly present in our everyday lives. If there is anything we do most in life, it is probably *waiting* for something to happen (finally winning a non-trivial prize in the State Lottery after paying monthly tickets over the past thirty years), arrive (the breath-taking dress we ordered from that webshop against warnings in the seller's reputation blog), change (the reception of many severely bad hands in the poker game we just happened to ran into), stop (the constant flipping into red of traffic lights while we are just within breaking distance in our urban environment), or resume (the heater that regularly happens to have a mind of its own during winter months).

Queueing systems also appear in computer systems and networks, in which they schedule available shared resources like processors, memory, and network ports among clients like computing applications. Or in wireless communications, where so-called 'listen-before-talk' access protocols (CSMA/CA) as used in wireless Local Area Networks monitor the received power level at the input stage in order to assess whether the transmission medium is idle before attempting to transmit. Or in automated production lines, where a partial product is routed to visit several service stations in sequence, each of them performing a specific task to the product.

Perhaps surprisingly given their wide variety in terms of applications, queueing systems usually share a common concept: a set of objects we will call *jobs* has to visit a set of objects we will call *queues*, in order to get something done. Depending on the complexity of the task to be performed, on the service capacity of the queue, and on the available competition among jobs, such a visit may vary in length (i.e., in sojourn time). This perhaps explains the great interest from the mathematical community in *queueing theory*: One often needs only a handful of variables and assumptions in order to model a wide range of applications. In most cases, the effects of these assumptions are modeled with suitable stochastic processes.

Despite great results in deriving closed-form analytic expressions for many queueing models, many more others are mathematically intractable. In order to gain quantitative insight into these models, one often resorts or needs to resort to *discrete-event simulation*, appropriately modeling queue scheduling behavior, and subjecting it to a workload consisting of jobs with appropriate parameters as to the amount of work each job requires, and the time between consecutive job arrivals. Even though discrete-event simulation does not provide closed-form solutions, they are often very handy and capable of, for instance, quantitative comparisons between various scheduling strategies.

This document introduces `jsimulation` and `jqueues`, open-source `java` software libraries for discrete-event simulation of queueing systems. Its main purpose is to expose you to the most important concepts in the libraries, and to get you going with your simulation studies. By no means is this document complete in its description of `jsimulation` and `jqueues`, nor is it intended to be, and for more detailed information we refer the reader to the "JQueues Reference Manual"[1] if you need precise specification of the libraries, and to the "JQueues Developer Manual"[2] if you want or need to extend either library (e.g., to add your own queueing discipline).

In Section 2 of the present document we provide installation (and build) instructions, and in Section 3 we present our "Hello World" example. In subsequent sections, in rather random order, we provide additional details and examples on the use of both libraries; attempting to allow linear reading. However, this is a living document and sections are added on demand and when time permits.

Any feedback on the clarity and/or correctness of the text is highly appreciated. Please use the *Issues* section on `github` to that purpose[3].

---

[1] The JQueues Reference Manual is currently being written, and will be available as an e-Book.
[2] The JQueues Developer Manual is currently being written, and will be available as an e-Book.
[3] See `https://github.com/jandejongh/jqueues-guided-tour`.

Chapter 2

Installation

2.1 The `jsimulation` and `jqueues` Libraries

In order to use `jsimulation` and `jqueues`, you have to install them first, which requires an Internet connection. The first public releases of `jqueues` and `jsimulation` have version number `5.2.0`; they have been released under the Apache v2.0 license. From that version number onward, both libraries are distributed as `Maven` projects available from `github.com` and the Maven Central Repository (whichever suits you).

Since both `jsimulation` and `jqueues` are libraries and hardly support stand-alone operation, we assume that you intend to install them both as dependencies to your own project. You have several options, but the two most obvious ones are:

- Install the libraries from `github`, open them as Maven *projects* in your IDE and add them as dependencies to your own project. If you use Maven yourself for the latter, you only have to add the dependency on `jqueues` in the `pom.xml`. (You do not have to add `jsimulation` because Maven does this automatically for you.)

- Create your own Maven project and add `jqueues` as a dependency, taken from the Maven Central Repository.

In both cases, you will need `maven` installed and properly configured on your system. It is also highly recommended to install `maven` support in your IDE, so that it can directly open `maven` projects.

In the first case, you need `git` as well, and you should clone both libraries from `github` as shown below:

- `$ git clone https://www.github.com/jandejongh/jsimulation`

- $ git clone https://www.github.com/jandejongh/jqueues

Note that `jsimulation` and `jqueues` can only be built against `Java 1.8` and higher.

In the second case, add the XML fragment shown in Listing 2.1 to the dependencies section in your `pom.xml`. Please make sure that you double-check the version number in the XML file[1]. The second case is safer as it uses stable, frozen, versions

Listing 2.1: The `dependency` section for `jqueues` in a `pom.xml`.

```
<dependency>
  <groupId>org.javades</groupId>
  <artifactId>jqueues</artifactId>
  <version>5.2.0</version>
  <scope>compile</scope>
  <type>jar</type>
</dependency>
```

of the libraries released to Maven Central. These releases are signed and cannot be changed without increasing the version number.

2.2 Version Numbering

For both libraries, we use three-level version numbering:

- The third, lowest, level is reserved for bug fixes, `javadoc` improvements and code (layout) "beautifications".
- The second, middle, level is reserved for functional extensions that do not break existing code (with the same major version number). Think of adding another queue, job or listener type.
- The third, major, level is reserved for changes to the core interfaces and classes that are likely to break existing code.

This implies that you can (should be able to) always "upgrade" to a later version from Maven Central as long as the major number remains the same. Upgrading from `github.com` requires a bit of care, as the latest version may not be stable yet.

Despite the fact that we take utmost efforts to *not* break existing code with upgrades of middle and minor version numbers, we cannot always avoid this. For instance, we may realize that a method should be `final` or `private` and attempt

[1] You may want to verify the latest stable release number from either `github` or Maven Central. This Guided Tour applies to release 5.2.0 and beyond.

to fix that in an apparent innocent update, but you may have overridden (or used) that particular method already in your code to suit your own purposes. Needless to say, we did not expect you to override (or just use) that particular method in your code, just as well as you did not expect that you were not supposed to do so. But in the end, your code may not be compile-able after the upgrade. In order to avoid this, we recommend that you

- Prefer interface methods rather than specific ones from classes, since the chance that we consider updates of the interface as being "minor" is virtually nil.
- Only override methods for which the `javadoc` explicitly states that they are intended to be overridden.

2.3 The `jqueues-guided-tour` Project

All example code shown in this document is available from the `jqueues-guided-tour` project on `github`. The code is organized as a Maven project. In addition to the example code, it also contains all the source files (LaTeX and other) to the present document. Bear in mind, though, that the documentation and example code in `jqueues-guided-tour` are both released under a more restrictive license than `jsimulation` and `jqueues`. In short, you are allowed to use the documentation and example code to whatever purpose. You may also redistribute both in unmodified form. However, redistributing *modified* versions of either or both of them requires the explicit permission from the legal copyright holder.

Chapter 3

Hello World: `FCFS`

In this section, we introduce our "Hello World" application for `jqueues`[1], consisting of a `FCFS` queue subject to arrivals of jobs with varying required service times.

In order to perform a simulation study in `jqueues`, the following actions need to be taken:

- The creation of an event list;
- The construction of one or more queues attached to the event list;
- The selection of the method for listening to the queue(s);
- The creation of a workload consisting of jobs and appropriately scheduling it onto the event list;
- The execution of the event list;
- The interpretation of the results, typically from the listener output.

Without much further ado, we show our "Hello World" example in Figure 3.1. We first create a single event list of type `DefaultSimEventList` and a `FCFS` queue attached to the event list (by virtue of the argument of `FCFS`'s constructor). On the queue, we register a newly created `StdOutSimEntityListener`, issuing notifications to the standard output. Note that queues and jobs are so-called *entities*; these are the relevant objects with state subject to event invocation. Subsequently, we create ten jobs named "0", "1", "2", ..., scheduled for arrival at the queue at $t = 0$, $t = 1$, $t = 2$, ..., respectively, and set their respective service times. We then schedule each

[1] In this Chapter, whenever we refer to `jqueues`, we silently assume that `jsimulation` is installed as well.

job arrival on the event list. Finally, we ”run” the event list, i.e., let it process the arrivals.

Listing 3.1: A simple simulation with a single FCFS queue and ten jobs.

```
final SimEventList el = new DefaultSimEventList (0);
final SimQueue queue = new FCFS (el);
queue.registerSimEntityListener (new StdOutSimEntityListener ());
for (int j = 0; j < 10; j++)
{
  final double jobServiceTime = (double) 2.2 * j;
  final double jobArrivalTime = (double) j;
  final String jobName = Integer.toString (j);
  final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
  SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
el.run ();
```

The event list type `DefaultSimEventList` will suffice for almost all practical cases, but it is essential to note already that a *single* event-list instance is typically used throughout *any* simulation program. Its purpose of the event list is to hold scheduled *events* in non-decreasing order of *schedule time*, and, upon request (in this case through `el.run`), starts processing the scheduled events in sequence, invoking their associated *actions*. In this case, the use of events remains hidden, because jobs are scheduled through the use of utility method `scheduleJobArrival`. The zero argument to the constructor denotes the simulation start time. If you leave it out, the start time defaults to $-\infty$.

Our queue of choice is First-Come First-Served (FCFS). The constructor takes the event list `el` as argument. The queueing system consists of a queue with infinite places to hold jobs, and a single server that ”serves” the jobs in the queue in order of their arrival. Once a queue has finished serving the (single) job, the job *departs* from the system.

So how long does it take to serve a job? Well, in `jqueues`, the default behavior is that a queue requests the job for its *required service time*. In the particular case of `DefaultSimJob` (there are many more job types), we provide a fixed service time (at *any* queue) upon creation through the third argument of the constructor.

The first argument of the `DefaultSimJob` is the event list to which it is to be attached. For jobs (well, at least the ones derived from `DefaultSimJob`), it is often safe to set this to `null`, although we could have equally well set it to `el`. However, *queues must always be attached to the event list*; a `null` value upon construction will throw an exception.

The (approximate) output of the code fragment of Listing 3.1 is shown in Listing 3.2 below. Remarkably, the listing only shows two types of notifications, viz., `UPDATE` and `STATE_CHANGED`, the latter of which can hold multiple ”sub”-notifications. Each notification outputs the name of the listener, the time on the event list, the queue (entity) that issues the notification, the notification's actual "major" type (`UPDATE` or `STATE_CHANGED`) and, if present, the sub-notifications.

Apart from the `STATE CHANGED`, `UPDATE` and `START_ARMED` lines in the output, the notifications pretty much speak for themselves. We even get notified when jobs start service (`START`). The `START_ARMED` notifications refer to state changes in a special `boolean` attribute of a queue named its `StartArmed` property. Since you will hardly need it in practical applications, we will not delve into it, but it is crucial for the implementation of certain more complex (composite) queueing systems. Suffice it to say that the `StartArmed` property *in this particular case* signals whether the queue is idle.

The two top-level notification types, `UPDATE` and `STATE CHANGED` are essential. Upon every change to a queue's state, the queue is obliged to issue the fundamental `STATE CHANGED` notification, exposing the queue's new state (including its notion of time). The `UPDATE` notification has the same function, but it is fired *before* any changes have been applied, thus revealing the queue's *old* state, including the time at which the old state was obtained. Hence, every `STATE CHANGED` notification *must* be preceded with an `UPDATE` notification. The `UPDATE` notification is crucial for the implementation of statistics (among others).

The use of `STATE_CHANGED` notifications may appear strange at first sight as many other implementations would report each of the sub-notifications individually. However, an important aspect of a queue's contract is that *it must report state changes atomically in order to meet queue invariants.* This means that listeners, when notified, will always see the queue in a consistent state, i.e., in a state that respects the invariant(s). This is one of the (we think) most distinguishing features of `jqueues`. Going back to our example: An important invariant of FCFS and many other queueing systems is that there cannot be jobs waiting in queue while the server is idle. It is easy to see that individual notifications for `ARRIVAL` and `START` would lead to violations of this invariant: Suppose that a job arrives at an idle FCFS queue. Using individual notifications, the queue has no other option than to issue a `ARRIVAL` notification immediately followed by a `START`. In between both, the queue would expose a state that is inconsistent with the invariant because the server is idle (the job has not started yet), while there is a job in its waiting queue. Note that another invariant of FCFS is that it cannot be serving jobs with zero required service time. This explains the arrival, start and departure sub-notifications for job 0 are all in a single atomic `STATE_CHANGED`.

This concludes our "Hello World" example. There is obviously a lot more to tell, but the good news is that our example has already revealed the most important

Listing 3.2: Example output of Listing 3.1.

```
StdOutSimEntityListener t=0.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[0]@FCFS]
  => START [Start[0]@FCFS]
  => DEPARTURE [Dep[0]@FCFS]
StdOutSimEntityListener t=1.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=1.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[1]@FCFS]
  => START [Start[1]@FCFS]
  => STA_FALSE [StartArmed[false]@FCFS]
StdOutSimEntityListener t=2.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=2.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[2]@FCFS]
StdOutSimEntityListener t=3.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=3.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[3]@FCFS]
StdOutSimEntityListener t=3.2, entity=FCFS: UPDATE.
StdOutSimEntityListener t=3.2, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[1]@FCFS]
  => START [Start[2]@FCFS]
StdOutSimEntityListener t=4.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=4.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[4]@FCFS]
StdOutSimEntityListener t=5.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=5.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[5]@FCFS]
StdOutSimEntityListener t=6.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=6.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[6]@FCFS]
StdOutSimEntityListener t=7.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=7.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[7]@FCFS]
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS: UPDATE.
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[2]@FCFS]
  => START [Start[3]@FCFS]
StdOutSimEntityListener t=8.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=8.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[8]@FCFS]
StdOutSimEntityListener t=9.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=9.0, entity=FCFS: STATE CHANGED:
  => ARRIVAL [Arr[9]@FCFS]
StdOutSimEntityListener t=14.200000000000001, entity=FCFS: UPDATE.
StdOutSimEntityListener t=14.200000000000001, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[3]@FCFS]
  => START [Start[4]@FCFS]
StdOutSimEntityListener t=23.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=23.0, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[4]@FCFS]
  => START [Start[5]@FCFS]
StdOutSimEntityListener t=34.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=34.0, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[5]@FCFS]
  => START [Start[6]@FCFS]
StdOutSimEntityListener t=47.2, entity=FCFS: UPDATE.
StdOutSimEntityListener t=47.2, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[6]@FCFS]
  => START [Start[7]@FCFS]
StdOutSimEntityListener t=62.60000000000001, entity=FCFS: UPDATE.
StdOutSimEntityListener t=62.60000000000001, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[7]@FCFS]
  => START [Start[8]@FCFS]
StdOutSimEntityListener t=80.20000000000002, entity=FCFS: UPDATE.
StdOutSimEntityListener t=80.20000000000002, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[8]@FCFS]
  => START [Start[9]@FCFS]
StdOutSimEntityListener t=100.00000000000001, entity=FCFS: UPDATE.
StdOutSimEntityListener t=100.00000000000001, entity=FCFS: STATE CHANGED:
  => DEPARTURE [Dep[9]@FCFS]
  => STA_TRUE [StartArmed[true]@FCFS]
```

concepts of `jqueues` like the event list, events, entities, queues, jobs, listeners and notifications. The remaining complexity is in the richness and variation of these basic concepts.

Chapter 4

Events, Actions and the Event List

This chapter describes the event and event-list features that are available from the `jsimulation` package. Note that `jsimulation` is a dependency of `jqueues`. In most usage scenarios, there is no need to directly manipulate events or the event-list; the preferred method is to use *utility* methods for that. However, in order to describe in more detail the models of entities, jobs and queues, a basic understanding of what goes on under the hood of a `DefaultSimEventList` is very helpful.

4.1 Creating the Event List and Events

At the very heart of every simulation experiment in `jqueues` is the so-called *event list*. The event list obviously holds the events, keeps them ordered, and maintains a notion of "where we are" in a simulation run. Together, an event list and the events it contains define the precise sequence of actions taken in a simulation. The code snipplet in Listing 4.1 shows how to create an event list and schedule two (empty) events, one at $t_1 = 5.0$ and one at $t_2 = 10$, and print the resulting event list on `System.out`. In `jsimulation`, the event list is of type `SimEventList`; events are of type `SimEvent`, respectively. Since both of them are Java *interfaces*, you need implementing classes to instantiate them: `DefaultSimEventList` for an event list; `DefaulSimEvent` for an event; typically you need a single event list and numerous events.

As explained in the previous chapter, the `double` argument in the `DefaultSimEventList` constructor is the initial time on the event list, its so-called *default reset time*. The `double` argument in the `DefaultSimEvent` constructor (of which there are several) is the *schedule time* of the event on the event list. Events, once created, are scheduled on the event list through the `add` method; the event list stores the events until use

Listing 4.1: Creating the event list and populating it with events.

```
final SimEventList el = new DefaultSimEventList (−5);
final SimEvent e1 = new DefaultSimEvent (5.0);
final SimEvent e2 = new DefaultSimEvent (10.0);
el.add (e1);
el.add (e2);
el.print ();
el.run ();
System.out.println ("Finished!");
```

and maintains the proper order between them. The output of the code snipplet is shown in Listing 4.2[1]:

Listing 4.2: Output of Listing 4.1.

```
SimEventList EventList[t=−5.0], class=DefaultSimEventList, time=−5.0:
t=5.0, name=No Name, object=null, action=null.
t=10.0, name=No Name, object=null, action=null.
Finished!
```

By virtue of the call to `el.print`, the output shows the name of the event list (as obtained from its `toString` method) and the current time (-5) in the first row, and then the events in the list in the proper order. Beware that the event-list is printed before the `el.run` statement; it would be empty afterwards.

Perhaps surprisingly, in `jsimulation`, the schedule time is actually held on the event, *not* on the event list. Also, a `SimEventList` is inheriting from `SortedSet` from the Java Collections Framework. These choices have the following consequences:

- Each `SimEvent` can be present *at most once* in a `SimEventList`. You cannot reuse a single event instance (like a job creation and arrival event) by scheduling it multiple times on the event list. Instead, you must either use separate event instances, or reschedule the event the moment it leaves the event list.

- You cannot (more precisely, *should not*) modify the time on the event while it is scheduled on an event list.

- You always have access to the (intended) schedule time of the event, without having to refer to an event list (if the event is scheduled at all) or use a separate variable to keep and maintain that time.

- The events must be equipped with a *total ordering* (imposed by `SortedSet`) and distinct events should not be equal (imposed by us). This means that for

[1]We may have improved the layout in the meantime.

each pair of (distinct) events scheduled on a `SimEventList`, one of them is always strictly larger than the other (in the ordering, they cannot be "equal").

- If two or more events with identical schedule times are scheduled on a single event list, their relative order needs to be determined by other means than their schedule time. The `DefaultSimEventList` uses a random-number generator to break such ties. If, for some reason, you want to maintain *insertion order*, please have a look at `DefaultSimEventList_IOEL`. Note that `IOEL` stands for Insertion Order Event List. But be warned: all (concrete) queue types in `jqueues` are specified against random ordering of simultaneous events.

Clearly, there is a lot more to say about simultaneous events, and about the reasons we chose for their random ordering while processing them, but we defer a detailed discussion for a later section. Nonetheless, it is important to realize that while an event say `e1` is being processed at some time t, any other event say `e2` scheduled at the same time on the event list is *always* processed after completion of `e1`. Even if `e1` itself actually schedules `e2`. In other words, `jsimulation` does *not* support the concept of *event preemption*, and the action of an event (see below) is always processed atomically. This implies that it will not work to use the event list (1) to get something done "immediately after" the completion of an event, (2) to do something "when all other events at t" are done", and (3) to process an event `e2` while processing an event `e1` and then returning to the original event `e1`.

4.2 Events

The output in Listing 4.2 shows four properties of a `SimEvent`:

- `Time`: The (intended) schedule time of the event (default $-\infty$).
- `Name`: The name of the event, which is only used for logging and output (default "No Name").
- `Object`: A general-purpose object available for storing information associated with the event (`jsimulation` nor `jqueues` uses this field; its default value is `null`).
- `EventAction`: The action to take, a `SimEventAction` (default `null`), described in the next section.

Each property has corresponding getter and setter methods on every `SimEvent`. In addition, `DefaultSimEvent` features multiple constructors that allow direct setting all or some of these properties upon construction.

4.3 Actions

A `SimEventAction` defined what needs to be done by the time an event is *executed* or *processed*. In Java terms, a `SimEventAction` is an interface with a single abstract method which is invoked when the event is processed, in other words, it is a `FunctionalInterface` that can be used in lambda expressions. We show its declaration in Listing 4.3.

Listing 4.3: The `SimEventAction` interface.

```
@FunctionalInterface
public interface SimEventAction<T>
{

  /** Invokes the action for supplied {@link SimEvent}.
   *
   * @param event The event.
   *
   * @throws IllegalArgumentException If <code>event</code> is <code>null</code>.
   *
   */
  public void action (SimEvent<T> event);

}
```

There are several ways to create a `SimEventAction` but nowadays, by far the easiest is to use lambda expressions, as shown in Listing 4.4. Note that we are now using the full `DefaultSimEvent` constructor, passing a name, and supplying a `SimEventAction` through a lambda expression. The generated output is shown in Listing 4.5. Note that we replaced the package and class identification of the action with `X` for formatting purposes.

Listing 4.4: Creating and using `SimEventActions`.

```
final SimEventList el = new DefaultSimEventList (0);
final SimEvent e = new DefaultSimEvent ("My First Real Event", 5.0, null, ((event) ->
  {
    System.out.println ("Event=" + event + ", time=" + event.getTime () + ".");
  }));
el.add (e);
el.print ();
el.run ();
el.print ();
```

4.4 Processing the Event List

Once the events of your liking are scheduled on the event list, you can start the simulation by *processing* or *running* the event lists. Processing the event list will cause the

Listing 4.5: Example output of Listing 4.4.

```
SimEventList EventList[t=0.0], class=DefaultSimEventList, time=0.0:
  t=5.0, name=My First Real Event, object=null, action=X$$Lambda$1/1826771953@65ab7765.
Event=My First Real Event, time=5.0.
SimEventList EventList[t=5.0], class=DefaultSimEventList, time=5.0:
  EMPTY!
```

event list to equentially invoke the actions attached to the events in increasing-time order. There are several ways to process a `SimEventList`:

- You can process the event list until it is empty with the `run` method.
- You can process the event list until some specified (simulation) time with the `runUtil` method.
- You can *single-step* through the event list with the `runSingleStep` method.

You can check whether an event list is being processed through its `isRunning` method.

While processing, the event list maintains a *clock* holding the (simulation) time of the current event. You can get the time from the event list through `getTime` nethod, although you can obtain it more easily from the event itself. You can insert new events while it is being processed, *but these events must not be in the past*. Once the event list detects insertion of events in the past, it will throw and exception.

Note that processing the event list is thread-safe in the sense that all methods involved need to obtain a *lock* before being able to process the list. Trying to process an event list that is already being processed from another thread, or from the thread that currently processes the list, will lead to an exception. Note that currently there is no safe, atomic, way to process an event list on the condition that is is not being processed already. Though you can check with `isRunning` whether the list is being processed or not, the answer from this method has zero validity lifetime.

4.5 Utility Methods for Scheduling Events

A `SimEventList` supports various methods for directly scheduling events and actions without the need to generate both the `SimEvent` *and* the `SimEventAction`. In most cases, the availability of one of the objects suffices. In Table 4.1 we show the most common utility methods for scheduling on a `SimEventList`. The use of these utility methods is highly preferred over direct manipulation of the underlying `SortedSet` interface, because we (may) intend to delete the `SortedSet` dependency in future releases altogether.

Table 4.1: Utility methods for scheduling on a `SimEventList`.

Utility methods for scheduling on `SimEventList`
void schedule (E) Schedules the event at its own time[2].
boolean cancel (E) Cancels (removes) a scheduled event, if present.
void schedule (**double**, E) Schedules the event at given time.
reschedule (**double**, E) Reschedules (if present, else schedules) the event at given new time.
E schedule (**double**, SimEventAction, String) Schedules the action at given time with given event name.
void scheduleNow (E) Schedules the event now.
E schedule (**double**, SimEventAction) Schedules the action at given time with default event name.
E scheduleNow (SimEventAction, String) Schedules the action now with given event name.
E scheduleNow (SimEventAction) Schedules the action now with default event name.

Note that `E` refers to the so-called *generic-type argument* of `SimEventList`. The prototype is `SimEventList<E extends SimEvent>`. The use of the generic type `E` allows you to restrict the use of a `SimEventList` to certain types of `SimEvents`, but for now `E` can be simply read as a `SimEvent`.

For any of the utilty methods that take a `SimEventAction` as argument, a new `SimEvent` is created on the fly, and returned from the method. Upon return from these methods, the newly created event has already been scheduled, and you *really* should not schedule it again.

So, how to *remove* events and actions from the event list? Well, since `SimEventList` implements the `Set` interface for `SimEvent` members, removing an event `e` from an event list `el` is as simple as `el.remove (e)`. However, the preferred method is `el.cancel (e)`.

Bibliography