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

Guided Tour

Jan de Jongh

Release 5

DRAFT

Print/Release Date 20180407

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

Author's Note: At this time of writing (end of March 2018), the libraries are *not yet* available from Maven Central! **Use Option 1 below!**

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 `5.0.0`; they were 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 standalone 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>nl.jdj</groupId>
  <artifactId>jqueues</artifactId>
  <version>5.1.0</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.

[1]You may want to verify the latest stable release number from either `github` or Maven Central. This Guided Tour applies to release 5.1.0 and beyond.

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 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`.

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 following code snipplet 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`:

```
final SimEventList el = new DefaultSimEventList ();
final SimEvent e1 = new DefaultSimEvent (5.0);
final SimEvent e2 = new DefaultSimEvent (10.0);
el.add (e1);
el.add (e2);
el.print ();
```

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 instantiate a single event list for a simulation experiment, and numerous events.

The `double` argument in the `DefaultSimEvent` constructor (of which there are several) is the *schedule time* of the event on the event list. 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 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").

The output of the code snipplet is something like[1]:

```
SimEventList {X}.DefaultSimEventList@{Y}, class=DefaultSimEventList, time=-Infinity:
  t=5.0, name=No Name, object=null, action=null.
  t=10.0, name=No Name, object=null, action=null.
```

The output shows the name of the event list (as obtained from its `toString` method) and the current time ($-\infty$) in the first row, and then the events in the list in the proper order. By the way, we modified the output; the markers `{X}` and `{Y}` represents strings that most likely deviate on your system.

The output also shows the four properties of an event: its time, name, user object, and action. These will be described in more detail in the next section.

4.2 Event Properties and Event Constructors

A `SimEvent` has the following properties:

- 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").

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

- Object: A general-purpose object available for storing information associated with the event (`jsimulation` nor `jqueues` uses this field; its default value is `null`).
- Action: The action to take, a `SimEventAction` (default `null`), described in the next section.

Each property has corresponding getter and setter methods:

Properties of `SimEvent`
double getTime ()
void setTime (**double**)
String getName ()
void setName (String)
T getObject ()
void setObject (T)
SimEventAction getEventAction ()
void setEventAction (SimEventAction)

Note that `T` refers to the so-called *generic-type argument* of `SimEvent` (and also of `DefaultSimEvent`). The prototype is `SimEvent<T>`, so `T` can be any object type. The use of generic types is explained in some more details in the "Advanced Topics" section, but for now `T` can be simply read as a `Object`.

The next section describes the actions in more detail, but we first provide a list of constructors for `DefaultSimEvent`:

Constructors of `DefaultSimEvent`
DefaultSimEvent (String, **double**, T, SimEventAction)
DefaultSimEvent (**double**, T, SimEventAction)
DefaultSimEvent (**double**, SimEventAction)
DefaultSimEvent (**double**)
DefaultSimEvent ()

Any non-listed property in a constructor will obtain its default value.

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. Below we show the declaration of the 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 actions for events. The first and most often used way in our own code is to use anonymous inner classes:

```
final SimEventList el = new DefaultSimEventList ();
final SimEvent e =
  new DefaultSimEvent ("My First Real Event", 5.0, null, new SimEventAction ()
  {
    @Override
    public final void action (final SimEvent event)
    {
      System.out.println ("Event=" + event + ", time=" + event.getTime () + ".");
    }
    @Override
    public String toString ()
    {
      return "My First Action";
    }
  });
el.add (e);
el.print ();
el.run ();
el.print ();
```

Note that we are now using the full `DefaultSimEvent` constructor, passing a name, and supplying a `SimEventAction` as an anonymous inner class. In the inner class, we define the `action` method, and in the meantime override the `toString` method (to be honest, this was merely to keep the generated text within bounds). The generated output is:

```
SimEventList {X}.DefaultSimEventList@{Y}, class=DefaultSimEventList, time=-Infinity:
  t=5.0, name=My First Real Event, object=null, action=My First Action.
Event=My First Real Event, time=5.0.
SimEventList {X}.DefaultSimEventList@{Y}, class=DefaultSimEventList, time=5.0:
  EMPTY!
```

Clearly, as expected! However, rote that after "running" the event list, it turns out to be empty, and its time is now $t = 5.0$, the schedule time of our event. This is as intended, and will be explained in the next section. But first we look at an alternative way of attaching actions to events:

```
final SimEventList el = new DefaultSimEventList ()
{
  @Override
  public final String toString ()
  {
    return "My Renamed Event List";
  }
};
final SimEventAction action = new SimEventAction ()
{
  @Override
  public final void action (final SimEvent event)
  {
      System.out.println ("Event=" + event + ", time=" + event.getTime () + ".");
  }
  @Override
  public final String toString ()
  {
    return "A Shared Action";
  }
};
for (int i = 1; i <= 10; i++)
{
  final SimEvent e = new DefaultSimEvent ("Our Event", (double) i, null, action);
  el.add (e);
}
el.print ();
el.run ();
el.print ();
```

In this example, we created a single action object (again using an anonymous inner class), and reuse it among ten distinct events we schedule (we cannot reuse those). We also took the opportunity give our event list a friendlier name by overriding its `toString` method. The output is as follows:

```
SimEventList My Renamed Event List, class=, time=-Infinity:
  t=1.0, name=Our Event, object=null, action=A Shared Action.
  t=2.0, name=Our Event, object=null, action=A Shared Action.
  t=3.0, name=Our Event, object=null, action=A Shared Action.
  t=4.0, name=Our Event, object=null, action=A Shared Action.
  t=5.0, name=Our Event, object=null, action=A Shared Action.
  t=6.0, name=Our Event, object=null, action=A Shared Action.
  t=7.0, name=Our Event, object=null, action=A Shared Action.
  t=8.0, name=Our Event, object=null, action=A Shared Action.
  t=9.0, name=Our Event, object=null, action=A Shared Action.
  t=10.0, name=Our Event, object=null, action=A Shared Action.
Event=Our Event, time=1.0.
Event=Our Event, time=2.0.
Event=Our Event, time=3.0.
Event=Our Event, time=4.0.
Event=Our Event, time=5.0.
Event=Our Event, time=6.0.
Event=Our Event, time=7.0.
Event=Our Event, time=8.0.
Event=Our Event, time=9.0.
Event=Our Event, time=10.0.
SimEventList My Renamed Event List, class=, time=10.0:
  EMPTY!
```

Again note that the time on the event list after running it is the time of the last event we scheduled on it. In the output, funny enough, the `class` of the event list is now reported as empty. This is because we used an anonymous class to construct it!

So, there are different ways of attaching a `SimEventAction` to a `DefaultSimEvent`. The abundant use of anonymous inner classes as shown here is certainly not to everyone's taste, but it results in relatively compact code (even more through the use of lambda expressions, see **XXX**).

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 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.

The example below shows how to schedule new events from event actions; it also shows what happens if you schedule events in the past.

```
final SimEventList el = new DefaultSimEventList ()
{
  @Override
  public final String toString ()
  {
    return "The Event List";
  }
};

final SimEventAction schedulingAction = new SimEventAction ()
{
  private int counter = 0;
  @Override
  public final void action (final SimEvent event)
  {
      System.out.println ("Event=" + event + ", time=" + event.getTime () + ".");
      counter++;
      if (counter < 10)
        // Schedule 1 second from now.
        // Use utility method on SimEventList.
        el.schedule (event.getTime () + 1, this);
      else if (counter == 10)
      {
        // Schedule now.
        el.schedule (event.getTime (), this);
        System.out.println ("Scheduled event now.");
      }
      else
      {
        // Schedule 1 second in the past -> throws exception.
        el.schedule (event.getTime () - 1, this);
        // Never reached.
        System.out.println ("Scheduled event in the past.");
      }
  }
  @Override
  public final String toString ()
  {
    return "Scheduling Action";
  }
};

el.schedule (0, schedulingAction);
el.print ();
el.run ();
el.print ();
```

The code begins to look familiar. First, we create the event list, then a single action. The action is a bit more complicated than before; it has an internal `counter` in the anynoumous class. Using the counter, it reschedules itself ten times, the first nine times one second in the future, the tenth time at exactly the same time. As mentioned before, this is perfectly legal (and, in fact, often used in our own code). The final attempt to reschedule the action results in an exception, because the event is scheduled in the past. Note that the example also showcases a utility method in `SimEventList`, viz., `schedule (double, SimEventAction)`, which directly schedules the action on the event list at given time, creating a new `SimEvent` on the fly. In a later section we will look in more detail at more utility methods on event lists.

The output of the example is shown below[2].

```
SimEventList The Event List, class=, time=-Infinity:
  t=0.0, name=No Name, object=null, action=Scheduling Action.
Event=No Name, time=0.0.
Event=No Name, time=1.0.
Event=No Name, time=2.0.
Event=No Name, time=3.0.
Event=No Name, time=4.0.
Event=No Name, time=5.0.
Event=No Name, time=6.0.
Event=No Name, time=7.0.
Event=No Name, time=8.0.
Event=No Name, time=9.0.
Scheduled event now.
Event=No Name, time=9.0.
Exception in thread "main" java.lang.IllegalArgumentException:
Schedule time is in the past: 8.0 < 9.0!
```

Note that in this particular case, the exception thrown actually comes with an instructive message as to what caused it (you tried to schedule something on the event list at $t = 8.0$, whereas the current time is beyond that, $t = 9.0$). However, in all honesty, such messages are not present for the majority of exceptions thrown as a result of incorrect arguments from user code. We are currently working on improving this.

[2]For improved reading, we have left out the full stack-trace of the exception, and rearranged the mixed outputs from `System.out` and `System.err`. We will do that without notice in the sequel.

The output also shows the expected result from the first `el.print` statement: Only a single event is scheduled! The others are created and scheduled while the event list is being processed. It is important to realize that the contents of a `SimEventList` can always change, as long as these are changes *now or in the future*. By the way, the second invocation of `el.print` does not stand a chance; it is unreachable because of the exception thrown in `el.run`.

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 object suffices. Below we show the most common utility methods for scheduling on a `SimEventList`.

Utility methods for scheduling
void schedule (E) Schedules the event at its own time.
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 generic types is explained in some more details in the ”Advanced Topics” section, 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.

You may wonder 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)`. Currently, there is no support to remove an action from an event list. Because actions can be reused, it would require iterating over all scheduled events, and remove all events with the given action. It is not hard to implement at all, we just did not do it[3]:

```
public static void removeAction
(final SimEventList eventList, final SimAction action)
{
  if (eventList != null)
  {
    final Iterator it = eventList.iterator;
    while (it.hasNext ())
      if (it.next ().getEventAction () == action)
        it.remove ();
  }
}
```

The code fragment silently assumes the absence of `null` events in the event list, which is indeed guaranteed, and works perfectly for `null` actions. Note the somewhat unexpected method name on `SimEvent` to get its action, viz., `getEventAction`. This name was chosen in order to avoid potential name clashes. At the risk of sounding pedantic, the explicit use of the iterator looks old-fashioned, yet allows for the safe removal of elements from a collection in a loop (contrary to a much fancier `for` construction).

We conclude with an overview of non-scheduling related utility methods of `SimEventList`:

Method	**Description**
void print ()	Prints the event list to `System.out`.
void print (PrintStream)	Prints the event list to the stream.

4.6 Simultaneous Events

So far, all events scheduled in our examples had a unique schedule time. But what if two events have identical schedule times? The short answer is: The `DefaultSimEventList` will process them *in random order*. Why not respect "insertion order"? Well, the answers to this seemingly innocent question are quite involved, and are given in Section **??**.

By slightly modifying our first example in Listing 3.1, we can easily force the occurence of simultaneous events. In Listing 4.1, apart from creating and scheduling fewer jobs, we have reduced the required service time to 1.0, and it is trivial to see that for this particular schedule, the departure of job 1 coincides with the arrival of

[3]This code fragment has not been tested.

job 2, and similarly for jobs 2 and 3. In the output shown in Listing 4.2, we indeed see that $t = 2$, job 2 arrives *before* the departure of job 1, however, at $t = 3$, job 3 arrives *after* the departure of job 2. At both instants in time, the relative order of the event is, as mentioned earlied, completely coincidental.

Listing 4.1: An example simulation program with simultaneous events.

```
final SimEventList el = new DefaultSimEventList ();
final int bufferSize = 2;
final FCFS_B queue = new FCFS_B (el , bufferSize);
final SimQueueListener listener = new StdOutSimQueueListener ();
queue.registerSimEntityListener (listener);
for (int j = 1; j <= 3; j++)
{
final double jobServiceTime = 1.0;
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 ();
```

4.7 Infinite Time

In the previous section we showed that events can be scheduled simultaneously. Another noteworthy feature of a `SimEventList` is that you can schedule events at $t = -\infty$ and at $t = +\infty$[4], corresponding to the `Java`'s `Double.NEGATIVE_INFINITY` and `Double.POSITIVE_INFINITY`, respectively. There are, however, some caveats:

- All events scheduled at $t = -\infty$ ($t = +\infty$) are treated as simultaneous events.
- Even though most `SimQueue` implementations have documented support for events at infinity, they often behave differently. For instance, if a job requires a finite amount of service, it will most likely depart immediately (informally: "$x + \infty = \infty$ for real x").

In short, caution is advised when scheduling events at infinity.

In our next example shown in Listing 4.3, with corresponding output in Listing 4.4, we show how to schedule events at $-\infty$. We increase to buffer size of the `FCFS_B` queue to four because we do not want to run into job drops in this example.

In the example, we schedule two jobs, "1" and "2" at $t = -\infty$. This is only allowed because the `DefaultSimEventList` initialized itself at $t = -\infty$ (you are not allowed to schedule events "in the past"!). The jobs have unity required service

[4]In honor of Georg Cantor.

Listing 4.2: Example output of Listing 4.1.

```
StdOutSimQueueListener t=1.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=1.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[1]@FCFS_B[2]]
=> START [Start[1]@FCFS_B[2]]
=> STA_FALSE [StartArmed[false]@FCFS_B[2]]
StdOutSimQueueListener t=1.0, queue=FCFS_B[2]: ARRIVAL of job 1.
StdOutSimQueueListener t=1.0, queue=FCFS_B[2]: START of job 1.
StdOutSimQueueListener t=1.0, queue=FCFS_B[2]: START_ARMED -> false.
StdOutSimQueueListener t=2.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=2.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[2]@FCFS_B[2]]
StdOutSimQueueListener t=2.0, queue=FCFS_B[2]: ARRIVAL of job 2.
StdOutSimQueueListener t=2.0, entity=FCFS_B[2]: STATE CHANGED:
=> DEPARTURE [Dep[1]@FCFS_B[2]]
=> START [Start[2]@FCFS_B[2]]
StdOutSimQueueListener t=2.0, queue=FCFS_B[2]: DEPARTURE of job 1.
StdOutSimQueueListener t=2.0, queue=FCFS_B[2]: START of job 2.
StdOutSimQueueListener t=3.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=3.0, entity=FCFS_B[2]: STATE CHANGED:
=> DEPARTURE [Dep[2]@FCFS_B[2]]
=> STA_TRUE [StartArmed[true]@FCFS_B[2]]
StdOutSimQueueListener t=3.0, queue=FCFS_B[2]: DEPARTURE of job 2.
StdOutSimQueueListener t=3.0, queue=FCFS_B[2]: START_ARMED -> true.
StdOutSimQueueListener t=3.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[3]@FCFS_B[2]]
=> START [Start[3]@FCFS_B[2]]
=> STA_FALSE [StartArmed[false]@FCFS_B[2]]
StdOutSimQueueListener t=3.0, queue=FCFS_B[2]: ARRIVAL of job 3.
StdOutSimQueueListener t=3.0, queue=FCFS_B[2]: START of job 3.
StdOutSimQueueListener t=3.0, queue=FCFS_B[2]: START_ARMED -> false.
StdOutSimQueueListener t=4.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=4.0, entity=FCFS_B[2]: STATE CHANGED:
=> DEPARTURE [Dep[3]@FCFS_B[2]]
=> STA_TRUE [StartArmed[true]@FCFS_B[2]]
StdOutSimQueueListener t=4.0, queue=FCFS_B[2]: DEPARTURE of job 3.
StdOutSimQueueListener t=4.0, queue=FCFS_B[2]: START_ARMED -> true.
```

Listing 4.3: An example simulation program with events at $t = -\infty$ and $t = \infty$.

```java
final SimEventList el = new DefaultSimEventList ();
final int bufferSize = 4;
final FCFS_B queue = new FCFS_B (el, bufferSize);
final SimQueueListener listener = new StdOutSimQueueListener ();
queue.registerSimEntityListener (listener);
for (int j = 1; j <= 2; j++)
{
final double jobServiceTime = Double.POSITIVE_INFINITY;
final double jobArrivalTime = Double.NEGATIVE_INFINITY;
final String jobName = Integer.toString (j);
final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
final SimJob job3 = new DefaultSimJob (null, "3", Double.POSITIVE_INFINITY);
SimJQEventScheduler.scheduleJobArrival (job3, queue, 0.0);
final SimJob job4 = new DefaultSimJob (null, "4", Double.POSITIVE_INFINITY);
SimJQEventScheduler.scheduleJobArrival (job4, queue, Double.POSITIVE_INFINITY);
el.run ();
```

time, but as the output shows, this really is irrelevant: Both jobs arrive (in order of scheduling, but that is a mere coincidence!), are taken into service, and depart immediately, all at the moment ($-\infty$) of arrival, because the required service time is finite. We also schedule a third job at $t = 0$ yet with infinite required service time, which is allowed for many queue types. However, this job never leaves, not even at $t = +\infty$; it is *sticky*. The same goes for job 4, also requiring infinite service time, yet arriving at $t = +\infty$

Listing 4.4: Example output of Listing 4.3.

```
StdOutSimQueueListener t=-Infinity , entity=FCFS_B[4]: UPDATE.
StdOutSimQueueListener t=-Infinity , entity=FCFS_B[4]: STATE CHANGED:
=> ARRIVAL [Arr[1]@FCFS_B[4]]
=> START [Start[1]@FCFS_B[4]]
=> STA_FALSE [StartArmed[false]@FCFS_B[4]]
StdOutSimQueueListener t=-Infinity , queue=FCFS_B[4]: ARRIVAL of job 1.
StdOutSimQueueListener t=-Infinity , queue=FCFS_B[4]: START of job 1.
StdOutSimQueueListener t=-Infinity , queue=FCFS_B[4]: START_ARMED -> false.
StdOutSimQueueListener t=-Infinity , entity=FCFS_B[4]: UPDATE.
StdOutSimQueueListener t=-Infinity , entity=FCFS_B[4]: STATE CHANGED:
=> ARRIVAL [Arr[2]@FCFS_B[4]]
StdOutSimQueueListener t=-Infinity , queue=FCFS_B[4]: ARRIVAL of job 2.
StdOutSimQueueListener t=0.0, entity=FCFS_B[4]: UPDATE.
StdOutSimQueueListener t=0.0, entity=FCFS_B[4]: STATE CHANGED:
=> ARRIVAL [Arr[3]@FCFS_B[4]]
StdOutSimQueueListener t=0.0, queue=FCFS_B[4]: ARRIVAL of job 3.
StdOutSimQueueListener t=Infinity , entity=FCFS_B[4]: UPDATE.
StdOutSimQueueListener t=Infinity , entity=FCFS_B[4]: STATE CHANGED:
=> ARRIVAL [Arr[4]@FCFS_B[4]]
StdOutSimQueueListener t=Infinity , queue=FCFS_B[4]: ARRIVAL of job 4.
```

This is all documented behavior of `FCFS_B`: Irrespective of the arrival time, visiting jobs with infinite required service time never depart. Of course, it's just a choice, and other choices could have been made, like, in this particular case, letting job "3" depart at $t = \infty$. But what if then job "3" *arrives* at $t = -\infty$? So, even though most queueing systems will support "operation at infinity", be advised that their behavior can be totally unexpected, and consult the queue's documentation on the topic well.

4.8 When Does A Simulation End?

In the previous section, it became obvious that the event list will happily process all events on the event list until it is empty. From the perspective of `jqueues`, this is perfectly reasonable: it just processes all events on the list. However, in practical situations, one often needs more control over the end simulation. First, having processed all events often leaves the queues in an empty state, which may not be a *representable* state at all. Hence, during statistics gathering, one often does not want to wait for the completion of event processing. (This is comparable to leaving out the "transient" section starting at the start of a simulation, in order to achieve higher accuracy with statistics.) Second, one may want to end a simulation because enough accuracy has been reached.

In order to at least partially support these requirements, `SimEventList` has an alternative method `runUntil` that does exactly what you would expect: It processes the events on the event list upto a given time, and then returns. The neat thing is that you can later decide, perhaps after scheduling some extra events (*in the future!*), to resume the simulation, either with `run` or `runUntil`. Its arguments are the end time (a `double`) and two `boolean`s. The first `boolean` argument, `inclusive`, controls whether or not to include the end time, in other words, whether to run *up to* or *up to and including* the end time. The second `boolean` argument, `setTimeToEndTime`, controls whether or not to set the time to the end time *if no event is scheduled at the end time.* In Release 5, `setTimeToEndTime` only has effect if `includeEndTime == true`, but this may change in future releases.

For full control over your simulation, `SimEventList` offers the `runSingleStep` method, processing exactly one event (the one at the head of the list) before returning.

We show the use of these methods in Listings 4.5 and 4.6, this time using a plain `FCFS` queue.

4.9 Resetting the Event List

Although you can directly invoke `resetEntity ()` on an entity (a `SimEntity`), its actual intention is to be invoked from an *event-list reset*; all entities attached to an event list are required to invoke their `RESET` operation upon an event-list reset. The method `SimEventList.reset (double time)` performs the reset; the argument is the new time on the event list and on all attached entities. (Note that there is also a variant `SimEventList.reset ()` without argument, which sets the new time to the *default* time on the event list. For more details, see Chapter **??**.) *You cannot invoke an event-list reset from within the context of an event.* In other words, do not schedule it; a `SimEventList` does not allow a reset while it is "being run".

In our next example in Listing 4.8, we switch queues, and use a (egalitarian) processor-sharing (`PS`) queue. A `PS` queue shares its capacity equally among the jobs present, so if two jobs are present, each of them is served "at half rate". This queue type is thus capable of serving multiple jobs simultaneously. More details on `PS` are provided in Section **??**. We reuse the `JobSojournTimeListener`, renaming it to `JobSojournTimeListenerWithReset`, and add a proper `RESET` handler, because

Listing 4.5: Example showing different methods for running the event list.

```
final SimEventList el = new DefaultSimEventList ();
final FCFS queue = new FCFS (el);
queue.registerStdOutSimEntityListener ();
for (int j = 0; j < 10; j++)
{
final double jobServiceTime = 100.0; // Double.POSITIVE_INFINITY;
final double jobArrivalTime = (double) j;
final String jobName = Integer.toString (j);
final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
// Run the event list until t=3.0 (inclusive; set time to given time).
el.runUntil (3.0, true, true);
System.out.println ("Time␣on␣event␣list:␣" + el.getTime () + ".");
// Run the event list until t=3.5 (inclusive; set time to last event processed).
el.runUntil (3.5, true, false);
System.out.println ("Time␣on␣event␣list:␣" + el.getTime () + ".");
// Run the event list until t=3.7 (inclusive; set time to given time).
el.runUntil (3.7, true, true);
System.out.println ("Time␣on␣event␣list:␣" + el.getTime () + ".");
// Run the event list until t=5.0 (exclusive; set time to last event processed).
el.runUntil (5.0, false, false);
System.out.println ("Time␣on␣event␣list:␣" + el.getTime () + ".");
// Run the event list until t=7.0 (exclusive; set time to given time => DOES NOT WORK).
el.runUntil (7.0, false, true);
System.out.println ("Time␣on␣event␣list:␣" + el.getTime () + ".");
// Process remaining events, one at a time.
while (! el.isEmpty ())
{
el.runSingleStep ();
System.out.println ("Time␣on␣event␣list:␣" + el.getTime () + ".");
}
System.out.println ("Finished!");
```

unlike queues, listeners must take care of properly resetting themselves. It is shown in Listing 4.7. Apart from the new method `notifyResetEntity`, it is an exact copy of `JobSojournTimeListener`. Also note that we invoke the super method in `notifyResetEntity`; although not needed in this case, it is a good habit to invoke the super method, especially in reset-related methods.

In the outer loop in Listing 4.8, we pick up a reset time, -3, -2, -1, or 0, and reset the event list with that time. Subsequently, we schedule ten jobs at `resetTime`, `resetTime`+1, . . ., with respective required service times 2.2, 4.4, ... After running the event list, we provide some data and the average job sojourn time on `System.out`. The corresponding output is shown in Listing 4.9.

Perhaps somewhat surprisingly, we find that the initial times on the event list and on the queue is $-\infty$. Why? Well, so far we have not given them any clue as to what time to initialize themselves with; the `SimEventList` therefore choses the safest value, viz., `Double.NEGATIVE_INFINITY`. This is the safest value because the contract of `SimEventList` is that events (`SimEvents`) *cannot* be scheduled *at a time strictly smaller than the list's current time.* The queue, upon construction, gets attached to the event list, and simply "inherits" the list's time; it does not have a clue either. This certainly is not "wrong" in any sense, but in many practical cases,

Listing 4.6: The output of Listing 4.5.

```
StdOutSimEntityListener t=0.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=0.0, entity=FCFS: STATE CHANGED:
=> ARRIVAL [Arr[0]@FCFS]
=> START [Start[0]@FCFS]
=> STA_FALSE [StartArmed[false]@FCFS]
StdOutSimEntityListener t=1.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=1.0, entity=FCFS: STATE CHANGED:
=> ARRIVAL [Arr[1]@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]
Time on event list: 3.0.
Time on event list: 3.0.
Time on event list: 3.7.
StdOutSimEntityListener t=4.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=4.0, entity=FCFS: STATE CHANGED:
=> ARRIVAL [Arr[4]@FCFS]
Time on event list: 4.0.
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]
Time on event list: 6.0.
StdOutSimEntityListener t=7.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=7.0, entity=FCFS: STATE CHANGED:
=> ARRIVAL [Arr[7]@FCFS]
Time on event list: 7.0.
StdOutSimEntityListener t=8.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=8.0, entity=FCFS: STATE CHANGED:
=> ARRIVAL [Arr[8]@FCFS]
Time on event list: 8.0.
StdOutSimEntityListener t=9.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=9.0, entity=FCFS: STATE CHANGED:
=> ARRIVAL [Arr[9]@FCFS]
Time on event list: 9.0.
StdOutSimEntityListener t=100.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=100.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[0]@FCFS]
=> START [Start[1]@FCFS]
Time on event list: 100.0.
StdOutSimEntityListener t=200.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=200.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[1]@FCFS]
=> START [Start[2]@FCFS]
Time on event list: 200.0.
StdOutSimEntityListener t=300.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=300.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[2]@FCFS]
=> START [Start[3]@FCFS]
Time on event list: 300.0.
StdOutSimEntityListener t=400.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=400.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[3]@FCFS]
=> START [Start[4]@FCFS]
Time on event list: 400.0.
StdOutSimEntityListener t=500.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=500.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[4]@FCFS]
=> START [Start[5]@FCFS]
Time on event list: 500.0.
StdOutSimEntityListener t=600.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=600.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[5]@FCFS]
=> START [Start[6]@FCFS]
Time on event list: 600.0.
StdOutSimEntityListener t=700.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=700.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[6]@FCFS]
=> START [Start[7]@FCFS]
Time on event list: 700.0.
StdOutSimEntityListener t=800.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=800.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[7]@FCFS]
=> START [Start[8]@FCFS]
Time on event list: 800.0.
StdOutSimEntityListener t=900.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=900.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[8]@FCFS]
=> START [Start[9]@FCFS]
Time on event list: 900.0.
StdOutSimEntityListener t=1000.0, entity=FCFS: UPDATE.
StdOutSimEntityListener t=1000.0, entity=FCFS: STATE CHANGED:
=> DEPARTURE [Dep[9]@FCFS]
=> STA_TRUE [StartArmed[true]@FCFS]
Time on event list: 1000.0.
Finished!
```

Listing 4.7: The `JobSojournTimeListenerWithReset` class, showing how to reset listeners.

```java
public class JobSojournTimeListenerWithReset
extends DefaultSimQueueListener
{

private final Map<SimJob, Double> jobArrTimes = new HashMap<> ();

private int jobsPassed = 0;
private double cumJobSojournTime = 0;

@Override
public void notifyResetEntity (SimEntity entity)
{
super.notifyResetEntity (entity);
this.jobArrTimes.clear ();
this.jobsPassed = 0;
this.cumJobSojournTime = 0;
}

@Override
public void notifyArrival (double time, SimJob job, SimQueue queue)
{
if (this.jobArrTimes.containsKey (job))
throw new IllegalStateException ();
this.jobArrTimes.put (job, time);
}

@Override
public void notifyDeparture (double time, SimJob job, SimQueue queue)
{
if (! this.jobArrTimes.containsKey (job))
throw new IllegalStateException ();
final double jobSojournTime = time - this.jobArrTimes.get (job);
if (jobSojournTime < 0)
throw new IllegalStateException ();
this.jobArrTimes.remove (job);
this.jobsPassed++;
this.cumJobSojournTime += jobSojournTime;
}

@Override
public void notifyDrop (double time, SimJob job, SimQueue queue)
{
notifyDeparture (time, job, queue);
}

public double getAvgSojournTime ()
{
if (this.jobsPassed == 0)
return Double.NaN;
return this.cumJobSojournTime / this.jobsPassed;
}

}
```

Listing 4.8: Example showing resetting the event list.

```java
final SimEventList el = new DefaultSimEventList ();
final PS queue = new PS (el);
final JobSojournTimeListenerWithReset listener = new JobSojournTimeListenerWithReset ();
queue.registerSimEntityListener (listener);
System.out.println ("BEFORE RESET");
System.out.println ("  Time on event list is " + el.getTime () + ".");
System.out.println ("  Time on queue is " + queue.getLastUpdateTime () + ".");
for (int resetTime = -3; resetTime <= 0; resetTime++)
{
el.reset (resetTime);
for (int j = 1; j <= 10; j++)
{
final double jobServiceTime = (double) 2.2 * j;
final double jobArrivalTime = resetTime + (double) (j - 1);
final String jobName = Integer.toString (j);
final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
el.run ();
System.out.println ("AFTER PASS " + (resetTime + 4) + ".");
System.out.println ("  Time on event list is " + el.getTime () + ".");
System.out.println ("  Time on queue is " + queue.getLastUpdateTime () + ".");
System.out.println ("  Average job sojourn time is " + listener.getAvgSojournTime () + ".");
}
```

resetting the event list to a known, finite value (0 comes to mind...) is crucial if you want to evaluate so-called *system statistics* like ”the average number of jobs at a queue”. Obviously, measuring such a statistic from $t = -\infty$ onwards, if at all possible, is not going to yield a value other than zero. The advice is therefore to *always reset the event-list time to a finite value before scheduling events and running the event list.* Admittedly, we could have chosen set the time to zero on the event list upon a `RESET`. We could, but we didn't!

Returning to the output, it is clear that we ran four simulations that were identical, but ”shifted in time”. Because the queue is time-independent, we find the same average job-sojourn time in all cases; a result we will not attempt to analyze this time. But, apparently, we run into rounding errors of the `PS` queue and/or errors in the representation of arbitrary `Double` values in `Java`. This is nothing to worry about at the present time.

One thing we *do* want to check is the simulation *end time*, which is the time of the last job departure from `PS`, and relate this to the behavior of `PS`. Can we, in an attempt to partially verify the result found, explain this? Well, whatever the `resetTime` value, it is easy to see that from the moment the first job arrives with this particular job-arrival pattern and the respective required job service times, the queue `PS` is constantly ”providing service” to at least one job. In other words, the queue is busy from the moment the first jobs arrives until the final departure, again, whatever the initial time. We could refer to the ”work-conserving property” of `PS`, but a little thought reveals that all jobs might just as well have arrived *at the same* *time* as the first job's arrival, *if* we are merely interested in the departure time of the "last" job (ending the so-called "busy cycle"). From the moment of the first job's arrival, `PS` simply has to serve a certain amount of "work", quantified by the summation of the jobs' required service time, being

$$\sum_{j=1}^{10} 2.2j = 2.2 \sum_{j=1}^{10} j = 2.2 \times 55 = 121.$$

This implies that the end time of the simulation should be the scheduled arrival time of the first job (which is always `resetTime`) increased by 121 which is, ignoring rounding errors, indeed confirmed from the output.

Listing 4.9: The output of Listing 4.8.

```
BEFORE RESET
Time on event list is -Infinity.
Time on queue is -Infinity.
AFTER PASS 1.
Time on event list is 118.00000000000001.
Time on queue is 118.00000000000001.
Average job sojourn time is 75.7.
AFTER PASS 2.
Time on event list is 119.00000000000001.
Time on queue is 119.00000000000001.
Average job sojourn time is 75.7.
AFTER PASS 3.
Time on event list is 120.00000000000001.
Time on queue is 120.00000000000001.
Average job sojourn time is 75.7.
AFTER PASS 4.
Time on event list is 121.00000000000001.
Time on queue is 121.00000000000001.
Average job sojourn time is 75.7.
```

At this point, we leave the subject of resetting event lists and queues, and we continue on the interface of a queue, formally, the `SimQueue` interface. We are still missing a few pieces, three of them being absolutely essential: The notion of *waiting* and *service areas* of a queue, so-called *revocations*, and *multiclass* queues and jobs. In the next sections, we therefore first complete the description of the (mandatory) `SimQueue` interface, culminating into to summary in Section 5.4. Subsequently, we turn our attention to "queues of queues", so called *composite* queues in Section **??**, and finish this chapter with some important other features of `jqueues`.

4.9.1 Important Methods on a `SimEventList`

In Table 4.9.1, we list the most important methods on a `SimEventList`.

Table 4.1: Important methods on a `SimEventList`.

`void`	`reset (double)`
`double`	`getTime`
`void`	`run`
`void`	`runUntil (double, boolean, boolean)`
`void`	`runSingleStep`

Chapter 5

Entities

5.0.1 Fundamentals

Before presenting the details on the `SimEntity` interface, we take the time to summarize the fundamental concepts in `jsimulation` and `jqueues`:

- The `Java` package named `jsimulation` is a library for (single-threaded) discrete-event simulation.

- The `Java` package named `jqueues` is a library for (single-threaded) discrete-event simulation of queueing systems. The library depends on `jsimulation`.

- In order to perform discrete event simulations, an event list is needed, on which events can be scheduled. The event list maintains an ordering of the events it contains in non-decreasing simulation time. Typically, a single instance of an event list is used throughout the entire simulation study. The corresponding (abstract) types for event lists and events are defined in `jsimulation`, and named `SimEventList` and `SimEvent`, respectively. This package also provides a reasonable implementation for a `SimEventList` named `DefaultSimEventList`.

- On a `SimEventList`, all scheduled `SimEvents` are unique; you cannot schedule a `SimEvent` more than once on a single `SimEventList`. Typically, `SimEvents` are created and scheduled through various utility methods.

- The time at which a `SimEvent` is scheduled, is kept on the `SimEvent` itself, and available though the `SimEvent.getTime` method. Once scheduled, you cannot change the time of a `SimEvent`. You can, however, reschedule it at a different time through the `SimEventList.reschedule` method.

- It is perfectly legel if multiple `SimEvents` are scheduled at the same time. On a `DefaultSimEventList`, they are processed in random order.
- With each `SimEvent`, an action is associated that determines what to do when the event is processed by the event list. The generic type in `jsimulation` is `SimEventAction`. Unlike `SimEvents`, `SimAction` need not be unique on the event list, and can be shared among different events.
- Once sufficient events have been scheduled, a simulation experiment starts by running or processing the event list. In `jsimulation`, you can run the `SimEventList` until it is exhausted of events through the `SimEventList.run` method, until it has reached a specific simulation time through the `SimEventList.runUntil` method, or on an event-by-event basis through `SimEventList.runSingleStep`.
- A `SimEventList` keeps a notion of simulation time. It is available through `SimEventList.getTime`. While running, this is always the scheduled time of the current event being processed. When not, it is always smaller than or equal to the time on the first scheduled `SimEvent`.
- You cannot schedule (at the risk of an `Exception`) a `SimEvent` with time strictly smaller than the current simulation time on the `SimEventList`.
- Event may be scheduled simultaneously, in which case their order of processing is *random*.
- Events may be scheduled at $t = -\infty$ and $t = +\infty$.
- The `SimEventList.reset` and `SimEventList (double)` methods reset the event list, meaning all scheduled `SimEvents` are removed from the list, and the time on the event list is set to its default time (first method) or given time. The typical use case of these methods is running the simulation again (for instance, for variance-reduction purposes). You cannot invoke either method while the event list is being processed, at the risk of an `Exception`.
- In simulation of queueing systems, the center of attention is a simulation entity, an object with a state subject to (the actions of) simulation events.
- In `jqueues`, simulation entities are represented as `SimEntity` objects. In the present release, a `SimEntity` is either a `SimJob` (job) or `SimQueue` (or their abstract joint interface, `SimJQ`). However, other types of `SimEntity` may be added in a future release.

- A `SimEntity` has the obligation to report changes to its state (as a result of event-list processing) to registered listeners, which are of type `SimEntityListener`. Listeners to a `SimEntity` can be registered and unregistered at any time through the `SimEntity.registerSimEntityListener` and `SimEntity.unregisterSimEntityListe` methods, respectively.

- In `jqueues`, changes to the state of a `SimEntity` is always the result of the invocation of one of a set of well-defined operations on the entity. The specific type of `SimEntity` determines the operation it supports. The invocation of an operation is almost always performed by a scheduled event.

- An operation can be external or internal. Events for internal events can only be scheduled by the `SimEntity` itself.

- Upon reception of *any* operation invocation, but before doing anything, a `SimEntity` fires an `UPDATE` notification exposing the *a priori* state (i.e., the "old" state).

- After completion of an operation invocation, but *before* handing back control (to the event list), a `SimEntity` fires a `STATE CHANGED` notification exposing the *a posteriori* state (i.e., the "new" state). In some cases, no state-change notification is fired when the state did not actually change.

- The external operations on a `SimQueue` are `Arrive` and `Revoke`.

- In `jqueues`, a `SimJob` starts a visit to a `SimQueue` through the `Arrive` operation. Listeners are always notified of invocations of `Arrive`, even if no state change occurs on the entity.

- While a `SimJob` is present (visiting) a `SimQueue`, it is either in its waiting or service area. Upon arrival, a `SimJob` enters the waiting area. Moving a `SimJob` from the waiting to the service area is performed (if at all) through the internal `START` operation. Invocations of the `Start` operation are always notified to listeners, even if no state change occurs.

- The visit of a `SimJob` to a `SimQueue` can end in one of following three ways: (1) Through departure (the internal `Depart` operation), (2) through dropping (the internal `Drop` operation), or (3) through revocation (the external `Revoke` operation or the internal `AutoRevoke` operation). Whatever way the `SimJob` leaves, it can be from the waiting and service area. Invocations of the `Drop` operation are always notified to listeners, even if no state change occurs.

- A job that is present on a queue, but never leaves during a simulation run is named a sticky job.

5.1 The `UPDATE` and `STATE CHANGED` Notifications

As you may have spotted, events are actually reported twice, once as part of a `STATE CHANGED` notification, followed by a separate notification specific to the event type (`ARRIVAL`, `DEPARTURE`, etc.). As a matter of fact, these separate notifications are merely a courtesy of the queue as it is only required to issue `UPDATE`, `STATE CHANGED` and `RESET` notifications.

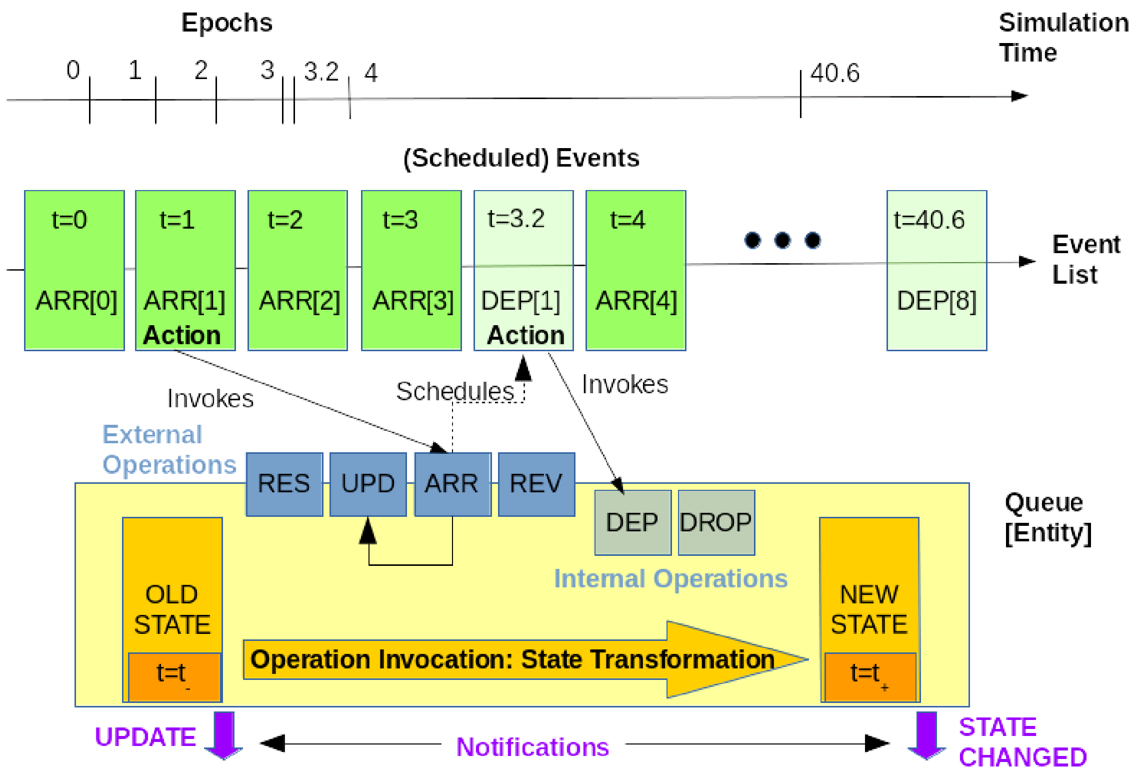
In order to understand this, we must realize that in *any* discrete-event simulation, the entities (like queues) involved possess an individual *state* that can only change at discrete moments in time (the *epochs*), and as a result of an *operation* on the entity at that time. This is explained grapically in Figure 5.1.

In the top of the figure, we show the simulation time and some of the epochs from the example. During a simulation run, the simulation time increases monotonically as a function of "real time". What this says is that the simulation time does not "strictly decrease" in real time, but at the same time, that's pretty much the only requirement on the relation between real time and simulation time. For instance, processing the epochs between $t = 0$ and $t = 9$ may take several minutes in real time, whereas epochs between $t = 9$ and $t = 100$ could be done in a mere second[1]

Below the simulation-time line, we show the event list and some of the scheduled events from the example. The apparent equal distance between the events is actually on pupose. The event list is really not concerned with the simulation-time differences between adjacent events. No matter how large the time interval between an event and its predecessor, in between (basicaly) nothing happened: There were not events, and hence, no state changes. Going back to the figure, we scheduled the arrival events ourselves before processing the event list, but we also see two "departure events" we did not schedule. In fact, these events were scheduled by the queue, in response to previous events. For instance, when job 1 arrives at $t = 1$ (simulation time), it is taken into service immediately, and after requesting the service time from job 1 (2.2), it schedules a *departure event* at $t = 3.2$, shown in a somewhat different color because we did not schedule this ourselves (we are not even allowed to do so). This shows an important aspect of the event list while processing it: in general, the actions taken

[1]This is not to say that is is impossible or even hard to let simulation time progress (roughly) at the same rate as real time, or at some scaled version of it. However, in discrete-event simulation, the relationship between real time and simulation time is generally considered unimportant.

Figure 5.1: The relations between simulation time, epochs, the event list, scheduled events, and actions, as well their impact on notifications from and operations on (the state of) an entity.



while processing the event list have full freedom to schedule new events, as long as they are not in the past. This, admittedly, is not clear from the figure. What is essential to remember is that a scheduled event await its turn while the event list is being processed, and the event list invokes the event's action when it has processed all preceding events.

Assuming the event's action involves the queue in question, we now turn our attention to the bottom part of the figure, showing our queue from the example, the `FCFS_B` queue. We assume the arrival event at $t = 1$ is processed by the event list, and the net effect of this (i.e., the action of the event) is the invokation of the arrival operation on the queue (with the job 1 as its argument). As a result of this operation, the state of the queue will transform, and registered listeners to the queue will be informed of the new state upon completion of the operation. This, effectively, is the

`STATE CHANGED` notification. In the figure this is shown with the right-pointing arrow at the bottom from `OLD STATE` to `NEW STATE`, and the `STATE CHANGED` notification below that.

However, before doing anything, the arrival operation invokes the so-called `UPDATE` operation, which exposes the *old* state to listeners and internally registered "hooks", and, subsequently, increases the most essential state property, the *last update time*. By now, you should realize that as the event list progresses in simulation time, queues (or better, *entities*) that are not affected by the events processed will not be bothered at all. Yet a queue needs to assure that time-stamped operations never lie in the past, so they need to maintain a notion of simulation time themselves. The importance of the `UPDATE` notification lies in *statistics gathering*, in which it is essential to know exactly during which (non-trivial) intervals the state of a queue did *not* change, and the lenght of such intervals. As long as you are not involved in statistics-gathering, you can safely ignore these notifications, otherwise, you can find more details in Section 14.

Once the `UPDATE` operation has been fulfilled, the `ARR` (job arrival) operation, in this particular case, checks the number of jobs waiting, and drops the arriving job if the number is 2 (=`B`) by invoking the internal operation `DROP`. However, for job 1 in the example, it finds an empty waiting area, and, in addition, no job being served at the server. This means that the job is taking into service immediately (the `START` internal operation), and since the required service time is non-zero, the queue schedules a *departure event* at $t = 3.2$. The departur event, in turn, will invoke the internal `DEP` (job departure) event because of which the job will eventually depart from the queueing system.

So, what more is there to say. Well, we seem to have so called *external* and *internal* operations, the latter of which we cannot schedule ourselves (departures, drops, ...). In a way, the external operations allow us to subject a queue to some kind of *workload* consisting of arriving jobs (as well as of, yet undescribed, other external operations on the queue). The internal operations, on the other hand, are always involved from within the queue itself, either as a response to a scheduled event, or as a response to an invocation of an external operation in combination with a state condition (e.g., the arrival of a job while 2 jobs are already waiting causes the internal `DROP` operation to be invoked).

5.2 The `Update` Operation

In the previous section we introduced the `UPDATE` and `STATE CHANGED` notifications, and explained that on a `SimEntity`, every state-change must be reported, and that the `UPDATE` notification exposes the old state of the entity for (for instance) statistics gathering. However, there also exits an `Update` operation. Its function is to set the `LastUpdateTime` state property of the entity. The example shown in Listing 5.1 with output shown in Listing 5.2 demonstrates its use, again using utility methods for scheduling. As the output shows, there seems to be little use in the `Update` operation; it just updates the time at $t = 8.5$ and $t = 1000$. The reason why you would ever want to use `Update` is a bit involved, and deferred until Chapter ??. However, we included this example to demonstrate that the end-time of the simulation can be controlled through scheduling an `Update` operation. In the next section, we will delve a little deeper into the end-time of a simulation.

Listing 5.1: The use of the `Update` operation.

```
final SimEventList el = new DefaultSimEventList ();
final int bufferSize = 2;
final FCFS_B queue = new FCFS_B (el, bufferSize);
queue.registerStdOutSimEntityListener ();
for (int j = 0; j < 4; 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);
}
SimEntityEventScheduler.scheduleUpdate (el, queue, 8.5);
SimEntityEventScheduler.scheduleUpdate (el, queue, 1000.0);
el.run ();
```

In summary:

- Queues and jobs are collectively referred to as simulation *entities*.
- Simulation entities always start in a known type-specific *default state*, also referred to their `RESET STATE` upon construction and upon performing their `RESET` operation.
- Simulation entities must always report any change to their state through `STATE CHANGED` notifications to registered *listeners*.
- Simulation entities only change their state as a result of the invocation of a well-defined *operation* on the entity. The operation can be external (like `ARRIVAL`) or internal (like `START`, `DROP`, and `DEPARTURE`).
- Any operation invocation takes zero (simulation) time to perform. Upon invocation of an operation, the new simulation time has to be supplied by the caller. With the exception of the `RESET` operation, the new time provided must not be strictly smaller than the time of the previous invocation (of *any* operation).

Listing 5.2: The output of Listing 5.1.

```
 StdOutSimEntityListener t=0.0, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=0.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[0]@FCFS_B[2]]
=> START [Start[0]@FCFS_B[2]]
=> DEPARTURE [Dep[0]@FCFS_B[2]]
 StdOutSimEntityListener t=1.0, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=1.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[1]@FCFS_B[2]]
=> START [Start[1]@FCFS_B[2]]
=> STA_FALSE [StartArmed[false]@FCFS_B[2]]
 StdOutSimEntityListener t=2.0, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=2.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[2]@FCFS_B[2]]
 StdOutSimEntityListener t=3.0, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=3.0, entity=FCFS_B[2]: STATE CHANGED:
=> ARRIVAL [Arr[3]@FCFS_B[2]]
 StdOutSimEntityListener t=3.2, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=3.2, entity=FCFS_B[2]: STATE CHANGED:
=> DEPARTURE [Dep[1]@FCFS_B[2]]
=> START [Start[2]@FCFS_B[2]]
 StdOutSimEntityListener t=7.6000000000000005, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=7.6000000000000005, entity=FCFS_B[2]: STATE CHANGED:
=> DEPARTURE [Dep[2]@FCFS_B[2]]
=> START [Start[3]@FCFS_B[2]]
 StdOutSimEntityListener t=8.5, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=14.200000000000001, entity=FCFS_B[2]: UPDATE.
 StdOutSimEntityListener t=14.200000000000001, entity=FCFS_B[2]: STATE CHANGED:
=> DEPARTURE [Dep[3]@FCFS_B[2]]
=> STA_TRUE [StartArmed[true]@FCFS_B[2]]
 StdOutSimEntityListener t=1000.0, entity=FCFS_B[2]: UPDATE.
```

- Before even starting the transformation from an old state into a new state, simulation entities must always expose their *old* state with a `UPDATE` notification.

- Depending on the registered listener, a simulation entity also fires *courtesy* notifications, revealing only a specific aspect of the state change. Courtesy notifications are *always* fired *after* the applicable `STATE CHANGED` notification.

5.3 Resetting Entities

Every simulation entity (queue or job) supports the external `RESET` operation that puts the entity into its *default* or *reset* state. It is the only operation that is allowed to set *back* the time. The new time on the entity is taken from the event list, or to $-\infty$ if no event list is available.

The `RESET` state of an entity depends on the type of entity, but it must be clearly specified. It is, however, subject to strict rules, as shown in Tables 5.1 and 5.2. For instance, in the `RESET` state, a queue is empty (no jobs visiting). The `QueueAccessVacation` and `ServerAccessCredits` properties will be decribed shortly in the next sections. In its `RESET` state, a job is not visiting any queue; its

`Queue` propery is `null`. Beware however, that queues are mandatorily attached to the event list, whereas for jobs this is not required. A queue will therefore set the `Queue` property to `null` for the jobs that it forcibly removes.

Table 5.1: Mandatory settings in the `RESET` state of a queue.

Property	Type	Default Value
`LastUpdateDate`	`double`	From event list or $-\infty$.
`Jobs`	`Set<SimJob>`	Empty set.
`QueueAccessVacation`	`boolean`	`false`.
`ServerAccessCredits`	`int`	`Integer.MAX_VALUE` ($==\infty$).
`StartArmed`	`int`	Depends on `SimQueue` type.

Table 5.2: Mandatory settings in the `RESET` state of a job.

Property	Type	Default Value
`LastUpdateDate`	`double`	From event list or $-\infty$.
`Queue`	`SimQueue`	`null`.

5.4 Summary of the `SimEntity` Interfaces

In this section, having seen almost all aspects of `SimEntitys`, `SimJobs`, and `SimQueues`, we summarize their state and behavior. We have two purposes in mind:

- To present the material presented thus far in a format that allows you to assert your understanding of the fundamental concepts in `jqueues` and `jsimulation`.
- To present the interfaces in a compact overview format that can be used as a reference. Really, at this point we have covered almost every aspect of entities, queues, and jobs, and their listeners. The remainder of this Chapter will focus at some left-overs and an important class of queues named *composite queues*. The remainder of this entire book is really about explaining rigorously the abstract interfaces and the specific concrete types of queues (mostly), jobs and listeners, and about how you can build your own. In other words, the summary

presented in this section is quite complete already, hence it is worth using as a reference.

Chapter 6

Listeners

In `jqueues`, monitoring the progress of a running simulation, or perhaps calculating statistiscs on it, starts with chosing the proper *listeners*. During the simulation, queues and jobs, from hereon collectively referred to as *entities*, are obliged to notify registered listeners of (at least) *all* changes to their states. A listener is a `Java` object implementing the required methods allowing such notifications from the entity.

Since in most practical simulation studies, the ambition level is somewhat higher that showing events on `System.out`, we will delve somewhat deeper into listeners types and how to create, modify and register them.

In the example of Listing 3.1, we used a convenience method `registerStdOutSimQueueListener` to register a listener at the `FCFS_B` queue that simply writes the details of such notifications to the standard output, `Sytem.out` in `Java`. This is extremely handy for initial testing of a simulation, but in almost all cases, a more sophisticated listener is required; one you have to create yourself. Luckily, `jqueues` comes with a large collection of listener implementations, each for a specific purpose, that you can modify to suit your needs.

Restricting ourselves for the moment to queue listeners, we can create and register our own listener that reports to `System.out` in the code in Listing 3.1:

Listing 6.1: Creating and registering a listener.

```
final SimQueueListener listener = new StdOutSimQueueListener ();
queue.registerSimEntityListener (listener);
```

Running the program of 3.1 again with this modified code fragment will yield (roughly) the same output, so we have not gained anything so far. However, a

`StdOutSimQueueListener` allows all (notification) methods to be overridden, so we can, for instance, suppress certain notifications in the output like this:

Listing 6.2: Suppressing specific notifications in a `StdOutSimQueueListener`.

```
final SimQueueListener listener = new StdOutSimQueueListener ()
{

  @Override
  public void notifyStateChanged (double time, SimEntity entity, List notifications) {}

  @Override
  public void notifyUpdate (double time, SimEntity entity) {}

  @Override
  public void notifyStartQueueAccessVacation (double time, SimQueue queue) {}

  @Override
  public void notifyStopQueueAccessVacation (double time, SimQueue queue) {}

  @Override
  public void notifyNewStartArmed (double time, SimQueue queue, boolean startArmed) {}

  @Override
  public void notifyOutOfServerAccessCredits (double time, SimQueue queue) {}

  @Override
  public void notifyRegainedServerAccessCredits (double time, SimQueue queue) {}

};
queue.registerSimEntityListener (listener);
```

In Listing 6.2, we modify the `StdOutSimQueueListener` by overriding the notification methods for server-access credits and queue-access vacations (which we do have not described yet), for the `StartArmed`-related notifications and replacing them with empty methods, effectively suppressing their respective output on `System.out`. In addition, we suppress the `UPDATE` and `STATE CHANGED` notifications. Our modified listener yields the following output:

If, on the other hand, your *only* interest is in the fundamental `RESET`, `UPDATE` and `STATE CHANGED` notifications, you can register a `StdOutSimEntityListener` as shown in Listing 6.4 or, simpler but equivalent, Listing 6.5, and their corresponding output in Listing 6.6.

In most practical cases, you will need a listener that does a bit more than reporting to `System.out`. Of course, you can override the methods in `StdOutSimQueueListener` to fit your purposes, but a better way is to use a `DefaultSimQueueListener`, or, if you just want to process the fundamental notification (`RESET`, `UPDATE` and `STATE CHANGED`), a `DefaultSimEntityListener`. Both listener type are concrete, but all required method implementations are empty. In our next example, we take a `DefaultSimQueueListener` and modify it in order to calculate the average job *sojourn* time. This time, we create a separate `class` named `JobSojournTimeListener` for the listener, shown in Listing 6.7.

Listing 6.3: Example output of Listing 3.1 with the modified listener of Listing 6.2

```
t=0.0, queue=FCFS_B[2]: ARRIVAL of job 0.
t=0.0, queue=FCFS_B[2]: START of job 0.
t=0.0, queue=FCFS_B[2]: DEPARTURE of job 0.
t=1.0, queue=FCFS_B[2]: ARRIVAL of job 1.
t=1.0, queue=FCFS_B[2]: START of job 1.
t=2.0, queue=FCFS_B[2]: ARRIVAL of job 2.
t=3.0, queue=FCFS_B[2]: ARRIVAL of job 3.
t=3.2, queue=FCFS_B[2]: DEPARTURE of job 1.
t=3.2, queue=FCFS_B[2]: START of job 2.
t=4.0, queue=FCFS_B[2]: ARRIVAL of job 4.
t=5.0, queue=FCFS_B[2]: ARRIVAL of job 5.
t=5.0, queue=FCFS_B[2]: DROP of job 5.
t=6.0, queue=FCFS_B[2]: ARRIVAL of job 6.
t=6.0, queue=FCFS_B[2]: DROP of job 6.
t=7.0, queue=FCFS_B[2]: ARRIVAL of job 7.
t=7.0, queue=FCFS_B[2]: DROP of job 7.
t=7.6000000000000005, queue=FCFS_B[2]: DEPARTURE of job 2.
t=7.6000000000000005, queue=FCFS_B[2]: START of job 3.
t=8.0, queue=FCFS_B[2]: ARRIVAL of job 8.
t=9.0, queue=FCFS_B[2]: ARRIVAL of job 9.
t=9.0, queue=FCFS_B[2]: DROP of job 9.
t=14.200000000000001, queue=FCFS_B[2]: DEPARTURE of job 3.
t=14.200000000000001, queue=FCFS_B[2]: START of job 4.
t=23.0, queue=FCFS_B[2]: DEPARTURE of job 4.
t=23.0, queue=FCFS_B[2]: START of job 8.
t=40.6, queue=FCFS_B[2]: DEPARTURE of job 8.
```

Listing 6.4: Creating and registering a `StdOutSimEntityListener`.

```java
final SimEntityListener listener = new StdOutSimEntityListener ();
queue.registerSimEntityListener (listener);
```

Listing 6.5: Using `registerStdOutSimEntityListener`.

```java
queue.registerStdOutSimEntityListener ();
```

Listing 6.6: Example output of Listing 3.1 with the modified listener of Listings 6.4 or 6.5

```
StdOutSimEntityListener t=0.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=0.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[0]@FCFS_B[2]]
  => START [Start[0]@FCFS_B[2]]
  => DEPARTURE [Dep[0]@FCFS_B[2]]
StdOutSimEntityListener t=1.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=1.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[1]@FCFS_B[2]]
  => START [Start[1]@FCFS_B[2]]
  => STA_FALSE [StartArmed[false]@FCFS_B[2]]
StdOutSimEntityListener t=2.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=2.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[2]@FCFS_B[2]]
StdOutSimEntityListener t=3.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=3.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[3]@FCFS_B[2]]
StdOutSimEntityListener t=3.2, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=3.2, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[1]@FCFS_B[2]]
  => START [Start[2]@FCFS_B[2]]
StdOutSimEntityListener t=4.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=4.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[4]@FCFS_B[2]]
StdOutSimEntityListener t=5.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=5.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[5]@FCFS_B[2]]
  => DROP [Drop[5]@FCFS_B[2]]
StdOutSimEntityListener t=6.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=6.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[6]@FCFS_B[2]]
  => DROP [Drop[6]@FCFS_B[2]]
StdOutSimEntityListener t=7.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=7.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[7]@FCFS_B[2]]
  => DROP [Drop[7]@FCFS_B[2]]
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=7.6000000000000005, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[2]@FCFS_B[2]]
  => START [Start[3]@FCFS_B[2]]
StdOutSimEntityListener t=8.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=8.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[8]@FCFS_B[2]]
StdOutSimEntityListener t=9.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=9.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[9]@FCFS_B[2]]
  => DROP [Drop[9]@FCFS_B[2]]
StdOutSimEntityListener t=14.200000000000001, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=14.200000000000001, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[3]@FCFS_B[2]]
  => START [Start[4]@FCFS_B[2]]
StdOutSimEntityListener t=23.0, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=23.0, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[4]@FCFS_B[2]]
  => START [Start[8]@FCFS_B[2]]
StdOutSimEntityListener t=40.6, entity=FCFS_B[2]: UPDATE.
StdOutSimEntityListener t=40.6, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[8]@FCFS_B[2]]
  => STA_TRUE [StartArmed[true]@FCFS_B[2]]
```

Listing 6.7: A (somewhat naive) queue listener that calculates the average job sojourn time.

```
public class JobSojournTimeListener
extends DefaultSimQueueListener
{

  private final Map<SimJob, Double> jobArrTimes = new HashMap<> ();

  private int jobsPassed = 0;
  private double cumJobSojournTime = 0;

  @Override
  public void notifyArrival (double time, SimJob job, SimQueue queue)
  {
    if (this.jobArrTimes.containsKey (job))
      throw new IllegalStateException ();
    this.jobArrTimes.put (job, time);
  }

  @Override
  public void notifyDeparture (double time, SimJob job, SimQueue queue)
  {
    if (! this.jobArrTimes.containsKey (job))
      throw new IllegalStateException ();
    final double jobSojournTime = time - this.jobArrTimes.get (job);
    if (jobSojournTime < 0)
      throw new IllegalStateException ();
    this.jobArrTimes.remove (job);
    this.jobsPassed++;
    this.cumJobSojournTime += jobSojournTime;
  }

  @Override
  public void notifyDrop (double time, SimJob job, SimQueue queue)
  {
    notifyDeparture (time, job, queue);
  }

  public double getAvgSojournTime ()
  {
    if (this.jobsPassed == 0)
      return Double.NaN;
    return this.cumJobSojournTime / this.jobsPassed;
  }

}
```

In the class, we override the (courtesy) notifications for job arrivals, departures and drops. When a job arrives, its arrival time is put into a private `Map` (`jobArrTimes`) for later reference. When a job departs or is dropped, we retrieve its arrival time, calculate its sojourn time, and add the result to the cumulative sum of sojourn times, `cumJobSojournTime`. In order to later interpret this value, we also have to maintain the number of jobs passed in a private field `jobsPassed`. At any time, the class provides the average sojourn time (over all jobs *passed*) through its `getAvgSojournTime` method. The calculation involved is trivial; the method returns `Double.NaN` when no jobs have passed.

The use of `JobSojournTimeListener` and the corresponding output are shown in Listings 6.8 and 6.9, respectively.

Listing 6.8: Our `FCFS_B` example with the custom `JobSojournTimeListener`.

```java
final SimEventList el = new DefaultSimEventList ();
final int bufferSize = 2;
final FCFS_B queue = new FCFS_B (el, bufferSize);
final JobSojournTimeListener listener = new JobSojournTimeListener ();
queue.registerSimEntityListener (listener);
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 ();
System.out.println ("Average job sojourn time is " + listener.getAvgSojournTime () + ".");
```

Listing 6.9: The output of Listing 6.8.

```
Average job sojourn time is 7.06.
```

Before going into details on the average sojourn time reported, we want to stress that our implementation of `JobSojournTimeListener` is far from complete and even erroneous, although it works correctly in this specific (use) case. For instance, it fails to consider the fact that jobs may *not* leave the queue (in whatever way). Such jobs are named *sticky* jobs, and in a true application we would have to consider them. Second, apart from `DROP` and `DEPARTURE`, there are other means by which a job can depart the queueing system (in particular, *revocations*). Third, the listener ignores `RESET` notifications from the queue; a critical error as we shall see later. We will not further discuss these and other complications here, because our primary intention is to show you the mechanisms for creating and modifying listeners. We just want to point out that the design of *robust* statistical listeners is more complicated that shown here. We provide a thorough treatment in Chapter 14.

Returning to the reported average job sojourn time. Is it correct? Well, in order to verify, we have no choice but to carefully analyze the behavior of the `FCFS_B` queue under the given workload of jobs, as given in Table 6.1. The table shows for each job its job number (Job), arival time (Arr), required service time (ReQ), jobs waiting upon its arrival (WQA), start time (Start, if applicable), exit time (either due to departure or due to dropping), sojourn time (exit time minus arrival time), and remarks if applicable. The final rows show the sum (TOT) and the average (AVG) of the required service times and the sojourn times, thus validating the result.

Table 6.1: Analysis of job sojourn times in Listing 3.1.

Job	Arr	ReqS	WQA	Start	Exit	Sojourn	Remark
0	0.0	0.0	{}	0.0	0.0	0.0	Exits immediately.
1	1.0	2.2	{}	1.0	3.2	2.2	
2	2.0	4.4	{}	3.2	7.6	5.6	
3	3.0	6.6	{2}	7.6	14.2	11.2	
4	4.0	8.8	{3}	14.2	23.0	19.0	
5	5.0	11.0	{3, 4}	-	5.0	0.0	Dropped.
6	6.0	13.2	{3, 4}	-	6.0	0.0	Dropped.
7	7.0	15.4	{3, 4}	-	7.0	0.0	Dropped.
8	8.0	17.6	{4}	23.0	40.6	32.6	
9	9.0	19.8	{4, 8}	-	9.0	0.0	Dropped.
TOT		99.0				70.6	
AVG		9.9				7.06	

Chapter 7

Generic Queue Model

7.1 Queue-Access Vacations

In `jqueues`, every queue, in other words, every `SimQueue` implementation, must support the notion of *queue-access vacations*. During a queue-access vacation, *all arriving jobs are dropped*. Butm jobs already visiting the queue are not affected. In terms of queue state, every `SimQueue` has a state property `QueueAccessVacation` of type `boolean` that determines whether or not the queue is "on vacation". The current value of this state property is available through `SimQueue.isQueueAccessVacation`. Starting and stoppping queue-access vacations is an external operation; on any `SimQueue` you can invoke this operation through `SimQueue.setQueueAccessVacation(double,boolean)`, which takes two arguments: (1) the time the operation is invoked, and (2), whether the vacation starts of ends.

It is essential to note that queue-access vacations are *always* available to you as an independent means to drop ariving jobs because you think this is the right thing to do at this time, in other words, `SimQueue` implementations are *not* allowed to use this feature in order to get their "jobs done". This turns the `QueueAccessVacation` operation into a purely *external* one. For instance, in our previous example with `FCFS_B`, the queue *could* use the mechanism of queue-access vacations in order to drop jobs upon arrival if the buffer is full. Yet, it is not allowed to do that. It simply does not touch the `QueueAccessVacation` property.

Scheduling the start and end of queue-access vacations on a queue is easily achieved through the utility method `SimQueueEventScheduler.scheduleQueueAccessVacation (SimQue` the respective arguments being the queue to which the event applies, the scheduled time, and whether or not to start/end a queue-access vacation, respectively. The following example in Listing 7.1 show how to schedule a queueu-access vacation from

$t = 1.75$ to $t = 2.25$, effectively dropping job 2 upon arrival (as its scheduled arrival time is $t = 2$). Our `SimQueue` of choice this time is `SocPS`. Just like `PS` used in the previous example, `SocPS` distributes its (full) service capacity among the jobs present, but, unlike `PS`, distributes its capacity in such a way that all jobs present depart at the same time. The `SocPS` implementation is one of our more exotic (maybe even original) ones; for more details, refer to Section **??**. Running the code yields the result on `System.out` as shown in Listing 7.2.

Listing 7.1: Setting Queue-Access-Vacations on a `SocPS` queue.

```
final SimEventList el = new DefaultSimEventList ();
final SocPS queue = new SocPS (el);
queue.registerStdOutSimEntityListener ();
el.reset (1.0);
SimQueueEventScheduler.scheduleQueueAccessVacation (queue, 1.75, true);
SimQueueEventScheduler.scheduleQueueAccessVacation (queue, 2.25, false);
for (int j = 1; j <= 4; 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 ();
```

Listing 7.2: The output of Listing 7.1.

```
StdOutSimEntityListener t=1.0, entity=SocPS: STATE CHANGED:
  => RESET [Reset@SocPS]
StdOutSimEntityListener entity=SocPS: RESET.
StdOutSimEntityListener t=1.0, entity=SocPS: STATE CHANGED:
  => ARRIVAL [Arr[1]@SocPS]
  => START [Start[1]@SocPS]
StdOutSimEntityListener t=1.75, entity=SocPS: UPDATE.
StdOutSimEntityListener t=1.75, entity=SocPS: STATE CHANGED:
  => QAV_START [QAV[true]@SocPS]
StdOutSimEntityListener t=2.0, entity=SocPS: UPDATE.
StdOutSimEntityListener t=2.0, entity=SocPS: STATE CHANGED:
  => ARRIVAL [Arr[2]@SocPS]
  => DROP [Drop[2]@SocPS]
StdOutSimEntityListener t=2.25, entity=SocPS: UPDATE.
StdOutSimEntityListener t=2.25, entity=SocPS: STATE CHANGED:
  => QAV_END [QAV[false]@SocPS]
StdOutSimEntityListener t=3.0, entity=SocPS: UPDATE.
StdOutSimEntityListener t=3.0, entity=SocPS: STATE CHANGED:
  => ARRIVAL [Arr[3]@SocPS]
  => START [Start[3]@SocPS]
StdOutSimEntityListener t=4.0, entity=SocPS: UPDATE.
StdOutSimEntityListener t=4.0, entity=SocPS: STATE CHANGED:
  => ARRIVAL [Arr[4]@SocPS]
  => START [Start[4]@SocPS]
StdOutSimEntityListener t=18.6, entity=SocPS: UPDATE.
StdOutSimEntityListener t=18.6, entity=SocPS: STATE CHANGED:
  => DEPARTURE [Dep[1]@SocPS]
  => DEPARTURE [Dep[3]@SocPS]
  => DEPARTURE [Dep[4]@SocPS]
```

Indeed, as expected, job 2 is dropped due to the scheduled queue-access vacation upon its arrival. Despite this, the arriving job 3 still finds job 1 being (exclusively) served, so `SocPS` schedules their mutual departure. However, the arrival of job 4 is ahead of this scheduled departure, so `SocPS` needs to reschedule the departure time (of all jobs present). Since the total "amount of work" of jobs 1, 3, and 4 jointly is $2.2 + 6.6 + 8.8 = 17.6$, and the arrival time of job 1 is 1.0, the joint departure time of jobs 1, 3, and 4, should be $1.0 + 17.6 = 18.6$, which is indeed confirmed in the output.

7.2 The Waiting and Service Area of a `SimQueue`

Every queue (`SimQueue`) consists of a *waiting area* and a *service area*, and visiting jobs are always present in precisely one of them, see Figure 7.2. Upon arrival, jobs enter the waiting area. If they `START`, they leave the waiting area and enter the service area. An important and perhaps non-intuitive restriction is that *jobs cannot move back from the service area into the waiting area.* Jobs may `DEPART` from, be `DROP`ped from, or be `REVOKE`d from either area. Jobs that do *not* leave the queue are named *sticky* jobs; they may reside in either area.

The model for queues in terms of waiting and service area is, admittedly, somewhat deviant from models in literature. The main point is that we make *no* assumptions whatsoever on the structure of the waiting and service areas. But for most known queueing systems, the waiting area is simply a queue holding waiting jobs, often in FIFO (First-In First-Out) order, and the service area consists of one of more servers serving jobs until completion. In (classical) processor-sharing queues, there is virtually no waiting area, as jobs enter the service area immediately.

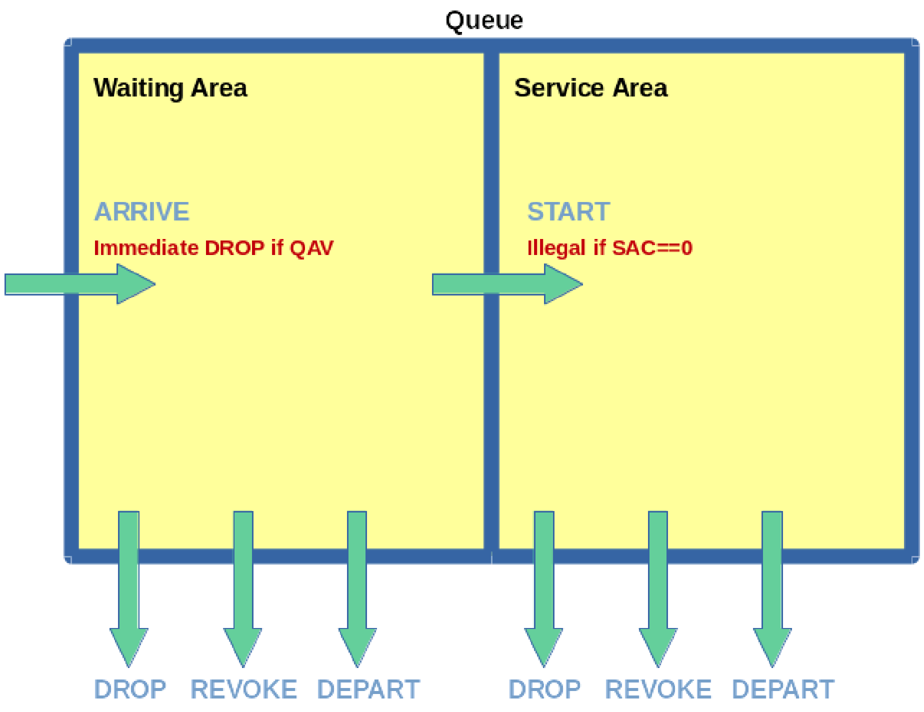
There are two more "complications" to bear in mind:

- Since jobs cannot move back from the service area into the waiting area, one has to let go of the intuitive notion that jobs in the service area area actually *being served.* Despite the fact that this is true for many queueing systems, it is false for systems like Preemptive/Resume Last-Come First-Served (`P_LCFS`), and many other preemptive queueing systems. In `P_LCFS`, whenever a job in the service area is preempted in favor of a new arrival, the preempted jobs stays in the service area, yet receives no service (at least, not for a while).

- In order for jobs to be eligible to `START`, the queue needs so-called `ServerAccessCredits`. These are described in more detail in the next section.

In Table 7.1, we list the most important methods related to waiting and service areas on a `SimQueue`.

Figure 7.1: The waiting and service areas of a `SimQueue`.



7.3 Server-Access Credits

The `ServerAccessCredits` is a state property on every `SimQueue`, and setting its value is an external operation named `SetServerAccessCredits`. Its value represents the maximum number of jobs on that particular `SimQueue` that can `START`, in other words, move from the waiting area into the service area; see also Figure 7.2. Whenever a job starts, the `ServerAccessCredits` value is decremented with one, and if it reaches zero, jobs are no longer allowed to start. The `ServerAccessCredits` value *never* affects jobs that are already in the service area.

Every `SimQueue` reports changes to *the availability of server-access credits* (i.e., not just changes to the actual value) through the `LOST_SAC` and `REGAINED SAC` notification. The former notification can be the result of starting one or more jobs *or* the invocation of `SetServerAccessCredits` with argument zero, whereas the latter notification is always the result of `SetServerAccessCredits` with argument (at least)

Table 7.1: `SimQueue` methods related to waiting and service areas.

Prototype	Symbol	Remark
`getJobs`	$J(t)$	The jobs visiting the `SimQueue`.
`getNumberOfJobs`	$\|J(t)\|$	The number of jobs currently visiting.
`isJob (SimJob)`		Checks presence of given job.
`getJobsInWaitingArea`	$J_w(t)$	The jobs in the waiting area.
`getNumberOfJobsInWaitingArea`	$\|J_w(t)\|$	The number of jobs in the waiting area.
`isJobInWaitingArea (SimJob)`		Checks presence of given job in the waiting area.
`getJobsInServiceArea`	$J_s(t)$	The jobs in the service area.
`getNumberOfJobsInServiceArea`	$\|J_s(t)\|$	The number of jobs in the service area.
`isJobInServiceArea (SimJob)`		Checks presence of given job in the service area.

non-zero.

Since the server-access credits are an integral number, it is represented by `Java`'s `int` simple type, but the value `Integer.MAX_VALUE` is interpreted as infinity. This is in fact the default value, as can be see in Table 5.1, and as long as `ServerAccessCredits` has this value, it is not affected by starting jobs (the value is not decremented), effectively turning off the mechanism of server-access credits.

In addition to the default value being ∞, `SimQueue` implementations cannot use `ServerAccessCredits` to meet their requirements. For instance, in order to implement queueing systems with multiple servers like `FCFS_c` (see Section **??**), the use of `ServerAccessCredits` could be queue handy. However, decrementing the value upon `START` of a job is the only thing queues may do (and must adhere to).

These two facts imply that if you never "touch" the `ServerAccessCredits` through the use of `SetServerAccessCredits`, you can safely forget the entire concept. On the other hand, should you have any need for it, it is always available, whatever the (concrete) queue type.

For our example showing the use of server-access credits, shown in Listing 7.3, we switch queues again, and select a `FCFS_c` queue. In the example, after creating the queue and resetting the event list, we schedule (!) setting the server-access credits on the queue to zero at $t = 0$, again using a utility method from `SimQueueEventScheduler`. We then schedule six jobs with one second inter-arrival time starting at $t = 1$, each requiring 1.05 service time. Finally we schedule setting the server-access credits to unity at $t = 8$, to three at $t = 10$, and to infinity at

$t = 15$. We show the output of the program in Listing 7.4.

Listing 7.3: Setting Server-Access Credits on a FCFS_c queue.

```
final SimEventList el = new DefaultSimEventList ();
final FCFS_c queue = new FCFS_c (el, 2);
queue.registerStdOutSimEntityListener ();
el.reset (0.0);
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 0.0, 0);
for (int j = 1; j <= 6; j++)
{
  final double jobServiceTime = 1.05;
  final double jobArrivalTime = (double) j;
  final String jobName = Integer.toString (j);
  final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
  SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
SimQueueEventScheduler.scheduleServerAccessCredits (queue,  8.0, 1);
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 10.0, 3);
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 15.0, Integer.MAX_VALUE);
el.run ();
```

The output shows indeed that at $t = 0$, the queue fires a notification OUT_OF_SAC. This makes sense, since the server-access credits were set to zero from their default value infinity. Subsequently, the arrival of the jobs at $t = 1, 2, \ldots$, is reported, but since there are no server-access credits, the jobs cannot start. The behavior of FCFS_c (and many other queueing systems) to maintain arrival-time ordering of the jobs in the waiting area, irrespective of whether these jobs are waiting for server-access credits or server availability. At $t = 8$, the queue is given a single server-access credit, and it immediately uses it to take job 1 into service. What we want to emphasize is that despite the server-access credit given to the queue, the queue does *not* issue a notification that it has regained server-access credits, because the credit is used immediately to start job 1, thus leaving zero credits available; the same as the number available before invocation of the operation SetServerAccessCredits at $t = 8$. This is a clear example of the *atomicity* of operations and notifications, which we shall explain in more detail in Section ??. At $t = 10$, the queue is granted three additional server-access credits, but it can only start two jobs, since it has only two servers available. Hence, one credit remains after starting the two jobs, and this time, the queue *does* issue a REGAIN_SAC notification. Note that in this particular case, the fact that FCFS_c only allows as many jobs in the service area as there are server available, is a policy choice of FCFS_c. It would have been legal for the queueing system to try to move as many jobs are possible from the waiting area to the service area upon having been granted new service access credits. But the choice made in FCFS_c makes a lot of sense; now the START notification is issued the moment a job actually starts its service at a server, instead of at the otherwise meaningless moment of entering the service area where is may still have to wait for a server to become available. The remainder of the notifications are quite trivial.

Listing 7.4: The output of Listing 7.3.

```
StdOutSimEntityListener t=0.0, entity=FCFS_2: STATE CHANGED:
  => RESET [Reset@FCFS_2]
StdOutSimEntityListener entity=FCFS_2: RESET.
StdOutSimEntityListener t=0.0, entity=FCFS_2: STATE CHANGED:
  => OUT_OF_SAC [SAC[0]@FCFS_2]
StdOutSimEntityListener t=1.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=1.0, entity=FCFS_2: STATE CHANGED:
  => ARRIVAL [Arr[1]@FCFS_2]
StdOutSimEntityListener t=2.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=2.0, entity=FCFS_2: STATE CHANGED:
  => ARRIVAL [Arr[2]@FCFS_2]
StdOutSimEntityListener t=3.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=3.0, entity=FCFS_2: STATE CHANGED:
  => ARRIVAL [Arr[3]@FCFS_2]
StdOutSimEntityListener t=4.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=4.0, entity=FCFS_2: STATE CHANGED:
  => ARRIVAL [Arr[4]@FCFS_2]
StdOutSimEntityListener t=5.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=5.0, entity=FCFS_2: STATE CHANGED:
  => ARRIVAL [Arr[5]@FCFS_2]
StdOutSimEntityListener t=6.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=6.0, entity=FCFS_2: STATE CHANGED:
  => ARRIVAL [Arr[6]@FCFS_2]
StdOutSimEntityListener t=8.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=8.0, entity=FCFS_2: STATE CHANGED:
  => START [Start[1]@FCFS_2]
StdOutSimEntityListener t=9.05, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=9.05, entity=FCFS_2: STATE CHANGED:
  => DEPARTURE [Dep[1]@FCFS_2]
StdOutSimEntityListener t=10.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=10.0, entity=FCFS_2: STATE CHANGED:
  => START [Start[2]@FCFS_2]
  => START [Start[3]@FCFS_2]
  => REGAIN_SAC [SAC[1]@FCFS_2]
  => STA_FALSE [StartArmed[false]@FCFS_2]
StdOutSimEntityListener t=11.05, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=11.05, entity=FCFS_2: STATE CHANGED:
  => DEPARTURE [Dep[2]@FCFS_2]
  => START [Start[4]@FCFS_2]
  => OUT_OF_SAC [SAC[0]@FCFS_2]
StdOutSimEntityListener t=11.05, entity=FCFS_2: STATE CHANGED:
  => DEPARTURE [Dep[3]@FCFS_2]
  => STA_TRUE [StartArmed[true]@FCFS_2]
StdOutSimEntityListener t=12.100000000000001, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=12.100000000000001, entity=FCFS_2: STATE CHANGED:
  => DEPARTURE [Dep[4]@FCFS_2]
StdOutSimEntityListener t=15.0, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=15.0, entity=FCFS_2: STATE CHANGED:
  => START [Start[5]@FCFS_2]
  => START [Start[6]@FCFS_2]
  => REGAIN_SAC [SAC[2147483647]@FCFS_2]
  => STA_FALSE [StartArmed[false]@FCFS_2]
StdOutSimEntityListener t=16.05, entity=FCFS_2: UPDATE.
StdOutSimEntityListener t=16.05, entity=FCFS_2: STATE CHANGED:
  => DEPARTURE [Dep[5]@FCFS_2]
  => STA_TRUE [StartArmed[true]@FCFS_2]
StdOutSimEntityListener t=16.05, entity=FCFS_2: STATE CHANGED:
  => DEPARTURE [Dep[6]@FCFS_2]
```

It should, however, be clear that queueing systems have to properly specify their behavior in the presence of limited server-access credits.

7.4 Revocations

Up to now, we have seen two means by which a `SimJob` can end its visit to a `SimQueue`: Through *departure* and through *drop*. We also found that jobs may not leave the queue at all; the sticky jobs. The final way in which a visit can end is through *revocation*; a revocation is a user-initiated exit of a job at a queue.

There are two flavors of revocations:

- Users can invoke the external `REVOKE` operation, requesting for the forced exit of a jobs. Every `SimQueue` must support the operation.
- Users can set one or more conditions on a queue. Once these conditions are met, the queue automatically revokes the job. Such revocations are named *auto-revocations*.

We shall discuss each of these flavors in turn.

7.4.1 The `Revoke` Operation

The external `Revoke` operation *requests* the removal of a job visiting a queue. If successful, a `REVOCATION` notification is fired. On every `SimQueue`, the method `revoke (double, SimJob)` revokes a job unconditionally from the queue. The first argument is (as always) the time of the request. If the job is present a priori, the revocation request cannot fail; every `SimQueue` implementation must honor it. A variant method exists: `revoke (double, SimJob, boolean)`, in which the third argument indicates whether it is allowed to revoke the job from the *service area*. If the argument is `false` and the job is indeed in the service area, the request will fail, and no notification will be fired. If, however, the job is in the waiting area, and/or the argument is set to `true` and the job is present in either area, then, again, the request cannot fail.

In our example shown in Listing 7.5 we use yet another standard queue, Shortest Job First, or `SJF`, a single-server queueing discipline that selects the jobs with the shortest required service time for service when the server becomes idle (but, it *pre-empts* the job currently being served). In `jqueues`, we have to add the additional requirement that there are service-access credits available, as pointed out in Section

7.3. In the example, we set the server-access credits to zero at $t = 0$, then we schedule four jobs (at $t = 1, 2, 3, 4$) with service time 12, 6, 4, and 3, respectively, and release all constraints on server-access credits at $t = 10$, at which time `SJF` will take job 4 into service because it has the (strictly) smallest required service time. The output is shown in Listing 7.6. Of the next two scheduled revocations requests for job 4, at $t = 11$ and $t = 12$, respectively, the first one will fail because it does not allow the revocation from the service area, which is where job4 is in at the time of the request. The second request, though, succeeds, since it allows the revocation to take place from the service area. The outcome of the remainder of the scheduled revocation events is quite trivial. One thing to bear in mind, though, is that *failed* revocation requests mostly do *not* lead to a notification; they just pass by unnoticed (see for instance the failed revocation attempt of job 4 at $t = 11$).

Listing 7.5: Revocation attempts from a `SJF` queue.

```
final SimEventList el = new DefaultSimEventList ();
final SJF queue = new SJF (el);
queue.registerStdOutSimEntityListener ();
el.reset (0.0);
final List<SimJob> jobs = new ArrayList<> ();
// Do not allow jobs to start until t=10.
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 0.0, 0);
for (int j = 1; j <= 4; j++)
{
final double jobServiceTime = 12.0 / j;
final double jobArrivalTime = (double) j;
final String jobName = Integer.toString (j);
final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
jobs.add (job);
SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
// Allow jobs to start at t=10.
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 10.0, Integer.MAX_VALUE);
// At t=10, the SJF will select job 4 for service since it has the shortest service time (3.0).
// The next revocation attempt will therefore fail, because job 4 (index 3) is in service,
// and we do not allow the revocation from the service area.
SimJQEventScheduler.scheduleJobRevocation (jobs.get (3), queue, 11.0, false);
// But this attempt will succeed, because this time we allow interruption of service.
SimJQEventScheduler.scheduleJobRevocation (jobs.get (3), queue, 12.0, true);
// Because at t=12, job 4 (index 3) is revoked, the queue will take
// job 3 (index 2) into service, with service time 4.0.
// This attempt will succeed; job 1 (index 0) is in the waiting queue.
SimJQEventScheduler.scheduleJobRevocation (jobs.get (0), queue, 13.0, false);
// However, the next attempt will fail (silently) because job 4 (index 3)
// is not longer present..
SimJQEventScheduler.scheduleJobRevocation (jobs.get (3), queue, 15.0, true);
el.run ();
```

7.4.2 Auto-Revocations

Auto-revocations are forced removals from a `SimQueue` because a user-set condition is met. The set of conditions for auto-revocation that can be set on a `SimQueue` depends on the queue's type, however, every `SimQueue` must have *any* auto-revocation

Listing 7.6: The output of Listing 7.5.

```
StdOutSimEntityListener t=0.0, entity=SJF: STATE CHANGED:
=> RESET [Reset@SJF]
StdOutSimEntityListener entity=SJF: RESET.
StdOutSimEntityListener t=0.0, entity=SJF: STATE CHANGED:
=> OUT_OF_SAC [SAC[0]@SJF]
StdOutSimEntityListener t=1.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=1.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[1]@SJF]
StdOutSimEntityListener t=2.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=2.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[2]@SJF]
StdOutSimEntityListener t=3.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=3.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[3]@SJF]
StdOutSimEntityListener t=4.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=4.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[4]@SJF]
StdOutSimEntityListener t=10.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=10.0, entity=SJF: STATE CHANGED:
=> START [Start[4]@SJF]
=> REGAIN_SAC [SAC[2147483647]@SJF]
=> STA_FALSE [StartArmed[false]@SJF]
StdOutSimEntityListener t=12.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=12.0, entity=SJF: STATE CHANGED:
=> REVOCATION [Rev[4]@SJF]
=> START [Start[3]@SJF]
StdOutSimEntityListener t=13.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=13.0, entity=SJF: STATE CHANGED:
=> REVOCATION [Rev[1]@SJF]
StdOutSimEntityListener t=16.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=16.0, entity=SJF: STATE CHANGED:
=> DEPARTURE [Dep[3]@SJF]
=> START [Start[2]@SJF]
StdOutSimEntityListener t=22.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=22.0, entity=SJF: STATE CHANGED:
=> DEPARTURE [Dep[2]@SJF]
=> STA_TRUE [StartArmed[true]@SJF]
```

condition *disabled by default*. The only auto-revocation condition every SimQueue *must* support, is the start of a job. (But, it must be disabled by default.) A simple example of this feature is given in Listings 7.7 and 7.8, again using the SJF queueing system. Note that successful auto-revocations yield a special notification, AUTO_REVOCATION.

Listing 7.7: Auto-revocations from a SJF queue.

```java
final SimEventList el = new DefaultSimEventList ();
final SJF queue = new SJF (el);
queue.registerStdOutSimEntityListener ();
// Set auto-revocation upon start.
queue.setAutoRevocationPolicy (SimQueue.AutoRevocationPolicy.UPON_START);
el.reset (0.0);
final List<SimJob> jobs = new ArrayList<> ();
// Do not allow jobs to start until t=10.
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 0.0, 0);
for (int j = 1; j <= 4; j++)
{
final double jobServiceTime = 12.0 / j;
final double jobArrivalTime = (double) j;
final String jobName = Integer.toString (j);
final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
jobs.add (job);
SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
// Allow two jobs to start at t=10.
// These will be immediately auto-revoked.
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 10.0, 2);
// Allow the other two jobs to start at t=13.
// These, again, will be immediately auto-revoked.
SimQueueEventScheduler.scheduleServerAccessCredits (queue, 13.0, Integer.MAX_VALUE);
el.run ();
```

7.4.3 Notification Types

In Table 7.4.3, we summarize the notification types supported on a SimQueue, subdivided into SimEntity, SimJQ and SimQueue notification types. The SimEntity notification types apply to any SimEntity, the SimJQ types to SimJobs and SimQueues, and the SimQueue types to SimQueues only.

The first column is the name of the notification type as it appears in (for instance) StdOutSimQueueListener output. The second column provides the arguments that are supplied with the notification type; and only if needed for clarity, the argument is named. This column, however, is merely provided so you understand the meaning of the arguments in the output and in the code; it does not provide literal lists of arguments to any method. But is should, for instance, allow you to look up the javadoc for a specific SimEntityListener, and known which methods to override, and what their arguments mean.

The STATE CHANGED notification is special, as mentioned before and discussed in more detail in Section ??. State changes are reported as a Set of sub-notifications,

Table 7.2: The notification types from a SimQueue.

SimEntity Notification Types	
RESET	double newTime
UPDATE	double oldTime
STATE CHANGED	double time, Set<SimEntityEvent> subNotifications
SimJQ Notification Types	
ARRIVAL	double time, SimJob, SimQueue
DROP	double time, SimJob, SimQueue
REVOCATION	double time, SimJob, SimQueue
AUTO_REVOCATION	double time, SimJob, SimQueue
START	double time, SimJob, SimQueue
DEPARTURE	double time, SimJob, SimQueue
SimQueue Notification Types	
QAV_START	double time, SimQueue
QAV_END	double time, SimQueue
OUT_OF_SAC	double time, SimQueue
REGAINED_SAC	double time, SimQueue
STA_FALSE	double time, SimQueue
STA_TRUE	double time, SimQueue

Listing 7.8: The output of Listing 7.7.

```
StdOutSimEntityListener t=0.0, entity=SJF: STATE CHANGED:
=> RESET [Reset@SJF]
StdOutSimEntityListener entity=SJF: RESET.
StdOutSimEntityListener t=0.0, entity=SJF: STATE CHANGED:
=> OUT_OF_SAC [SAC[0]@SJF]
StdOutSimEntityListener t=1.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=1.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[1]@SJF]
StdOutSimEntityListener t=2.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=2.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[2]@SJF]
StdOutSimEntityListener t=3.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=3.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[3]@SJF]
StdOutSimEntityListener t=4.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=4.0, entity=SJF: STATE CHANGED:
=> ARRIVAL [Arr[4]@SJF]
StdOutSimEntityListener t=10.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=10.0, entity=SJF: STATE CHANGED:
=> START [Start[4]@SJF]
=> AUTO_REVOCATION [AutoRev[4]@SJF]
=> START [Start[3]@SJF]
=> AUTO_REVOCATION [AutoRev[3]@SJF]
StdOutSimEntityListener t=13.0, entity=SJF: UPDATE.
StdOutSimEntityListener t=13.0, entity=SJF: STATE CHANGED:
=> START [Start[2]@SJF]
=> AUTO_REVOCATION [AutoRev[2]@SJF]
=> START [Start[1]@SJF]
=> AUTO_REVOCATION [AutoRev[1]@SJF]
=> REGAIN_SAC [SAC[2147483647]@SJF]
```

for which `SimEntityEvents` are reused. This avoids the more or less useless distinction in the software between an *event* and its corresponding *notification*, which, for all practical purposes, simply contains the same fields. The order of the sub-notifications in the `Set` is important, as it indicates the order of the sub-notifications. Implementations therefore typically resort to `LinkedHashSets` in order to return the sub-notifications.

7.4.4 Operations

In Table 7.4.4, we summarize the operations supported on a `SimQueue`, subdivided into `SimEntity` and `SimQueue` operations.

The first column in the table shows the name of the operation. Note that subtle change in naming between a *notification* (like `ARRIVAL`) and its corresponding *operation* (like `Arrive`). The second column indicates whether the operation is External (E) or Internal (I). Note that with the exception of `Update`, an external operation on a `SimEntity` is *never* invoked from within the entity itself. The third column provides the arguments to the operation, without going into the details of the method prototypes. The argument names are only shown when needed for clarification; for most arguments, the type is self-explanatory.

7.4.5 Important Methods on a `SimQueue`

In Table 7.4.5, we list the important methods supported on a `SimQueue`, subdivided into state (interrogation) methods, operations, and non-state properties.

The state methods allow for a complete interrogation of the state of the `SimQueue`. In short, the state of a `SimQueue` consists of

- The `LastUpdateTime` property (`double`).
- The `JobsInWaitingArea` property (`Set<SimJob>`).
- The `JobsInServiceArea` property (`Set<SimJob>`).

Table 7.4: The operations on a SimQueue.

SimEntity Operations		
Reset	E	double newTime
Update	E	double newTime
SimJQ Operations		
Arrive	E	double time, SimJob, SimQueue
Drop	I	double time, SimJob, SimQueue
Revoke	E	double time, SimJob, SimQueue, boolean interruptService
AutoRevoke	I	double time, SimJob, SimQueue
Start	I	double time, SimJob, SimQueue
Depart	I	double time, SimJob, SimQueue
SimQueue Operations		
SetQueueAccessVacation	E	double, SimQueue, boolean
SetServerAccessCredits	E	double, SimQueue, int

Table 7.6: Important methods on a `SimQueue`.

State	
`double`	`getLastUpdateTime`

`Set<SimJob>`
`int`
`boolean`
	`getJobs`
	`getNumberOfJobs`
	`isJob (SimJob)`

`Set<SimJob>`
`int`
`boolean`
	`getJobsInWaitingArea`
	`getNumberOfJobsInWaitingArea`
	`isJobInWaitingArea (SimJob)`

`Set<SimJob>`
`int`
`boolean`
	`getJobsInServiceArea`
	`getNumberOfJobsInServiceArea`
	`isJobInServiceArea (SimJob)`

`boolean`	`isQueueAccessVacation`
`int`	`getServerAccessCredits`
`boolean`	`isStartArmed`
Operations	
`void`	`resetSimEntity (double newTime)`
`void`	`update (double newTime)`
`void`	`arrive (double time, SimJob)`
`void`	`revoke (double time, SimJob)`
`boolean`	`revoke (double time, SimJob, boolean interruptService)`
`void`	`setQueueAccessVacation (double time, boolean vacation)`
`void`	`setServerAccessCredits (double time, int credits)`
Non-State Properties	
`SimEventList`	`getEventList`
`void`	`setAutoRevocationPolicy (AutoRevocationPolicy)`
`Class` `Object`	`getQoSClass` `getQoSValue`
`void`	`setName (String)`

`void`
`void`
`Set<SimEntityListener>`
	`registerSimEntityListener (SimEntityListener)`
	`unregisterSimEntityListener (SimEntityListener)`
	`getSimEntityListeners`

The methods implementing the operations on a `SimQueue` are given in the second vertical block in the table. These are self-explanatory, if not, consult Table 7.4.4.

The so-called non-state properties are given in the third vertical block in Table 7.4.5. Non-state properties come in two flavors.

First, the *essential* properties parameterize the functional behavior of the `SimQueue` and can therefore only be set upon construction. In rare cases, implementations allow setting such properties immediately *after* construction, immediately after a reset, or when the `SimQueue` is in the right (default) state. The essential properties on a `SimQueue` are

- The `EventList` property (`SimEventList`).
- The `AutoRevocationPolicy` property (`AutoRevocationPolicy`).
- The `QoSClass` (`Class`) and `QoSValue` (`Object`) properties.

On a `SimQueue`, the `EventList` property is always set upon construction. The `AutoRevocationPolicy` must be set immediately after construction (it is not in the constructor because we do not expect this feature to be used a lot), and it can only be set once! For the QoS-related properties, more relaxed requirements for setting their value are in place, but typically, these are set by passing the values to the constructor.

Second, the *non-essential* or *operational* properties do not affect the functional behavior of the `SimQueue`, and therefore, these methods can be invoked at any time. On a `SimQueue`, the two non-essential properties are

- The `Name` property (`String`).
- The `SimEntityListeners` property (`Set<SimEntityListener>`).

Chapter 8

Generic Job Model

In Table 8, we list the important methods supported on a `SimJob`, subdivided into state (interrogation) methods, operations, and non-state properties.

Table 8.1: Important methods on a `SimJob`.

State	
`double`	`getLastUpdateTime`
`SimQueue`	`getQueue`
Operations	
`void`	`resetSimEntity (double)`
`void`	`update (double)`
Non-State Properties	
`SimEventList`	`getEventList`
`Class`	`getQoSClass`
`Object`	`getQoSValue`
`void`	`setName (String)`
`void`	`registerSimEntityListener (SimEntityListener)`
`void`	`unregisterSimEntityListener (SimEntityListener)`
`Set<SimEntityListener>`	`getSimEntityListeners`

Chapter 9

Utility Methods for Scheduling on the Event List

In Table 9, we list the most important utility methods for scheduling `SimEvents`. Note that all methods are static; you cannot instantiate the various classes.

Table 9.1: Utility Methods for Scheduling.

Class	Method (Static)
`SimEntityEventScheduler`	`scheduleUpdate (SimEventList, SimEntity, double)`
`SimJQEventScheduler`	`scheduleJobArrival (SimJob, SimQueue, double)`
`SimJQEventScheduler`	`scheduleJobRevocation (SimJob, SimQueue, double, boolean)`
`SimQueueEventScheduler`	`scheduleQueueAccessVacation (SimQueue, double, boolean)`
`SimQueueEventScheduler`	`scheduleServiceAccessCredits (SimQueue, double, int)`

Chapter 10

QoS Queues

So far, the queueing systems in the examples did not have the possibility to provide service based upon job features other that its arrival or required service time. In *multiclass* (*QoS*, *discriminatory*) queueing systems, jobs are treated differently based upon their *class* (*priority*). In order to prevent name clashes with `Java` terminology, we shall consistently use the term *Qos queueing systems* to refer to them. More details on QoS queueing systems are provided in Section **??**.

In `jqueues`, the most common QoS queueing systems like Head-of-the-Line and Priority Queueing have been implemented. But their implementation is generic in the sense that you get to (*have to*) choose the set of "QoS values" for the jobs, or, in Java terminology, `class`. From hereon, we shall refer to this "set of admissible QoS values" as the `QoS class` of the queue or job, and its specific value on an entity as the `QoS value`. On an simulation entity, you can obtain the `QoS class` and the `QoS value` through the member functions `getQoSClass` and `getQoSValue`. (Clearly, simulation entities have been designed with QoS support in mind from the start.) Note that the `QoS value` of a queue is to be interpreted as the *default* value; to assign to jobs that have a `null` value (which is allowed), or a non-supported `QoS class`. There are specific rules as to when it is allowed to change the `QoS class` and/or `QoS value` of an entity, but we will not go into the details here. Suffice it to say that you should not change these settings on queues at all, but that it is safe to set them on jobs *in between queue visits*.

In our example, we choose Priority Queueing (`PQ`) to demonstrate some of the features of QoS queues and jobs[1] In `PQ`, only a single job is served at any time, and it is always the (a) job with the "highest priority". Arriving jobs with a strictly higher

[1]In the example, for simplicity, we leave out the so-called *generic-type arguments* of `PQ`. Their use is, however, highly recommended.

priority than the job in service will preempt that job, which will *resume* service when it is the highest-priority job again (in virtue of the `RESUME` preemption strategy). In `jqueues` we follow the somewhat unfortunate convention that *the smaller the* ***`QoS value`****, the higher the priority!* For QoS, we choose `Java`'s `Double`. We show the example in Listing 10.1. The most important difference with non-QoS systems, is that the `QoS class` and `QoS value` have to be provided upon construction of the queue, and upon construction of the (specific) `DefaultSimJobQoS`. We create four jobs, numbered 1 through 4, and set their (fixed) respective `QoS values` to -1 to -4. This implies that job 1 has the lowest priority, and job 4 the highest. We show the program's output in Listing 10.2. We indeed see that job 2 preempts job 1 upon its arrival, job 3 preempts job 2, and job 4 preempts job 3. Upon completion of job 4 at $t = 7$, job 3 is resumed for its remaining service time $4 - 1 = 3$, and hence leaves the system at $t = 10$. Note that the queue does *not* issue notifications as to which job is *resumed*!

Listing 10.1: Example of the use of `QoS` queues and jobs.

```java
final SimEventList el = new DefaultSimEventList ();
final PQ queue = new PQ (el, PreemptionStrategy.RESUME, Double.class, Double.NEGATIVE_INFINITY);
queue.registerStdOutSimEntityListener ();
el.reset (0.0);
for (int j = 1; j <= 4; j++)
{
final double jobServiceTime = 12.0 / j;
final double jobArrivalTime = (double) j;
final String jobName = Integer.toString (j);
final SimJob job = new DefaultSimJobQoS (null, jobName, jobServiceTime, Double.class, (double) (-j));
SimJQEventScheduler.scheduleJobArrival (job, queue, jobArrivalTime);
}
el.run ();
```

With the description of `QoS` queueing systems, we have completed our first glance at the `SimQueue` and `SimJob` interfaces. Before turning our attention to somewhat more advanced features like *composite queues*, we want to summarize the complete `SimQueue` interface in the next section.

Listing 10.2: The output of Listing 10.1.

```
StdOutSimEntityListener t=0.0, entity=PQ[RESUME]: STATE CHANGED:
=> RESET [Reset@PQ[RESUME]]
StdOutSimEntityListener entity=PQ[RESUME]: RESET.
StdOutSimEntityListener t=1.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=1.0, entity=PQ[RESUME]: STATE CHANGED:
=> ARRIVAL [Arr[1]@PQ[RESUME]]
=> START [Start[1]@PQ[RESUME]]
=> STA_FALSE [StartArmed[false]@PQ[RESUME]]
StdOutSimEntityListener t=2.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=2.0, entity=PQ[RESUME]: STATE CHANGED:
=> ARRIVAL [Arr[2]@PQ[RESUME]]
=> START [Start[2]@PQ[RESUME]]
StdOutSimEntityListener t=3.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=3.0, entity=PQ[RESUME]: STATE CHANGED:
=> ARRIVAL [Arr[3]@PQ[RESUME]]
=> START [Start[3]@PQ[RESUME]]
StdOutSimEntityListener t=4.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=4.0, entity=PQ[RESUME]: STATE CHANGED:
=> ARRIVAL [Arr[4]@PQ[RESUME]]
=> START [Start[4]@PQ[RESUME]]
StdOutSimEntityListener t=7.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=7.0, entity=PQ[RESUME]: STATE CHANGED:
=> DEPARTURE [Dep[4]@PQ[RESUME]]
StdOutSimEntityListener t=10.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=10.0, entity=PQ[RESUME]: STATE CHANGED:
=> DEPARTURE [Dep[3]@PQ[RESUME]]
StdOutSimEntityListener t=15.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=15.0, entity=PQ[RESUME]: STATE CHANGED:
=> DEPARTURE [Dep[2]@PQ[RESUME]]
StdOutSimEntityListener t=26.0, entity=PQ[RESUME]: UPDATE.
StdOutSimEntityListener t=26.0, entity=PQ[RESUME]: STATE CHANGED:
=> DEPARTURE [Dep[1]@PQ[RESUME]]
=> STA_TRUE [StartArmed[true]@PQ[RESUME]]
```

Chapter 11

Composite Queues

In the preceding sections, we have already seen some example queueing systems like `FCFS_B`, `FCFS_c` and `PS`, all of which implement the `SimQueue` interface. There are many more, all of which are described in detail in later chapters of this book. One particular "class" of `SimQueue` implementations, however, are the so-called *composite* queues, and they deserve mentioning in this guided tour.

A composite queue is a "queue of queues"; its behavior is determined by a set of other (possibly, again composite) queues, the *sub-queues*, passed at construction, and rules for routing jobs through the network of sub-queues. These rules depend on the type of composite queue.

Sounds complicated? Well, it's really not that bad. Let's just dive into it, by considering a classic example of a composite queue: the *tandem queue*. In a tandem queue, two or more queues are concatenated in the sense that an arriving job must first visit the first sub-queue, and, upon completion at the first, visit the second queue, and so on. In Listing 11.1, we show how to create a `Tandem` with two sub-queues, a `FCFS` followed by a `DLIMIT` queue.

In the example, we use a `LinkedHashSet`, instead of for instance a `HashSet`, to pass to the constructor of `Tandem`. The is essential because iteration over the sub-queues in the set must return the sets in proper order; we want `FCFS` first, then `DLIMIT`. It is recommended to always use `LinkedHashSet` for passing the set of sub-queues.

The `Tandem` queue has a third argument to its constructor, the so-called *delegate-job factory.* Because `Tandem` is in itself also a `SimQueue`, it must follow the rules of that interface, one of them being that a `SimJob` *can only visit a single `SimQueue` at a time.* Because the arriving job already visit the tandem queue, it cannot also visit any of the sub-queues at the same time; the tandem queue must therefore

Listing 11.1: A tandem of `FCFS` and `DLIMIT`.

```
final SimEventList el = new DefaultSimEventList ();
final SimQueue fcfs = new FCFS (el);
// Uncomment to print events at fcfs.
// fcfs.registerSimEntityListener (new StdOutSimEntityListener ());
final double rateLimit = 0.25;
final SimQueue dlimit = new DLIMIT (el, rateLimit);
// Uncomment to print events at dlimit.
// dlimit.registerSimEntityListener (new StdOutSimEntityListener ());
final Set<SimQueue> subQueues = new LinkedHashSet<> ();
subQueues.add (fcfs);
subQueues.add (dlimit);
final SimQueue tandem = new Tandem (el, subQueues, null);
tandem.registerSimEntityListener (new StdOutSimEntityListener ());
for (int j = 1; j <= 2; j++)
{
  final double jobServiceTime = 1.5;
  final double jobArrivalTime = (double) j;
  final String jobName = Integer.toString (j);
  final SimJob job = new DefaultSimJob (null, jobName, jobServiceTime);
  SimJQEventScheduler.scheduleJobArrival (job, tandem, jobArrivalTime);
}
el.run ();
```

create a new `SimJob` for every job visit for routing through the sub-queues. We call the jobs visiting the composite queue "real jobs", and the jobs created by the composite queue and visiting the sub-queues, "delegate jobs". The third argument to the constructor of `Tandem` (and the last argument to the constructor of most other concrete composite queues) allows passing a factory for delegate jobs. Passing `null` implies that the composite queue resorts to a default `DelegateSimJobFactory`, which should do just fine in most practical cases.

Listing 11.2: The output of Listing 11.1.

```
StdOutSimEntityListener t=1.0, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=1.0, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => ARRIVAL [Arr[1]@Tandem[FCFS,DLIMIT[0.25]]]
  => START [Start[1]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=2.0, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=2.0, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => ARRIVAL [Arr[2]@Tandem[FCFS,DLIMIT[0.25]]]
  => START [Start[2]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=2.5, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=2.5, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => DEPARTURE [Dep[1]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=4.0, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=6.5, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
StdOutSimEntityListener t=6.5, entity=Tandem[FCFS,DLIMIT[0.25]]: STATE CHANGED:
  => DEPARTURE [Dep[2]@Tandem[FCFS,DLIMIT[0.25]]]
StdOutSimEntityListener t=10.5, entity=Tandem[FCFS,DLIMIT[0.25]]: UPDATE.
```

Before looking at the output of the program in Listing 11.2, we need to clarify the operation of `DLIMIT`, the departure-rate limiter. It is described in detail in Section ??. `DLIMIT` is serverless (it never starts jobs), and lets jobs depart in order of arrival immediately, yet it guarantees a minimum time (the reciproke of the rate limit argument) between its successive departures, letting jobs wait for departure only when needed. So, indeed, DLIMIT does what it promises to do: ensuring that the rate of departures does not exceed the rate-limit argument given.

In the example code in Listing 11.1, we schedule two job arrivals, one at $t = 1$ and the other at $t = 2$, each job requiring 1.5 service time. Since our FCFS queue preceeds the DLIMIT queue, the arriving jobs (well, in fact, their delegate jobs) immediately arrive at fcfs. It is trivial to check that job "1" departs from fcfs at $t = 2.5$, and job "2" at $t = 4.0$. Hence, by virtue of Tandem, at $t = 2.5$, job "1" arrives at dlimit and since that queue has not seen any jobs yet, it lets job "1" depart immediately. As a result, job "1" (the "real" job this time) departs from tandem at $t = 2.5$ The DLIMIT queue, however, "blocks" departures for 4 seconds starting at $t = 2.5$ in order to meet its rate-limit requirement. Therefore, job 2, arriving at dlimit at $t = 4.0$, must wait until $t = 2.5+4.0 = 6.0$ before it can depart from dlimit, and, as a result, from tandem as well.

Note that tandem hides the arrival of job "2" at dlimit. This is typical for composite queues; they hide all events occuring at their sub-queues if they do not change the state of the composite queue itself. If you want to follow what happens at both sub-queues, uncomment the marked lines in Listing 11.1, thereby registering suitable listeners at them.

Our next (and final) example of a composite queue concerns a so-called *feedback queue*, in which departing jobs are "fed back" as arrivals depending on some *feedback condition*. Unlike Tandem, feedback queues only allow a single sub-queue. In jqueues Release 5, two types of tandem queues have been implemented, viz., FB_v and FB_p. In the former, jobs have to visit the sub-queue a fixed number of times before they leave the composite queue. In the latter, jobs, upon departure from the sub-queue, are fed back with a given, fixed, probability. The example in Listing 11.3 with output in Listing 11.4 demonstrates the use of feedback queues. In the example, we create both a FB_v queue and a FB_p queue, and schedule a single arrival at each of them. The final argument to the constructors of both FB_v and FB_p is, again, the optional delegate-job factory. The fourth argument to the constructor of FB_p is an optional (Java) Random object in case you want to control the random-number generation (e.g., if you want to set the *seed* of the generator). The example also shows a nice way to directly schedule actions (other than operations on queues and jobs) on a SimEventList. Refer to Chapter **??** for further reading if you are curious about this and other constructs on an event list.

Note that you cannot "reuse" sub-queues in multiple composite queues; in the example we cannot create a single FCFS queue and pass it to both fb_v and fb_p. In effect, by passing a sub-queue to a composite queue, you pass ownership over the

Listing 11.3: Two feedback queues each with a `FCFS` sub-queue.

```
// Create the event list.
final SimEventList el = new DefaultSimEventList ();
// FB_v example; scheduled at t=0.
final SimQueue fcfs1 = new FCFS (el);
final int numVisits = 7;
final SimQueue fb_v = new FB_v (el, fcfs1, numVisits, null);
fb_v.registerSimEntityListener (new StdOutSimEntityListener ());
final SimJob job1 = new DefaultSimJob (null, "1", 1.0);
SimJQEventScheduler.scheduleJobArrival (job1, fb_v, 0.0);
// Create some vertical space in the output at t=9.
el.schedule (9.0, (SimEventAction) (final SimEvent event) ->
{
  System.out.println ();
});
// FB_v example; scheduled at t=10.
final SimQueue fcfs2 = new FCFS (el);
final double pFeedback = 0.8;
final SimQueue fb_p = new FB_p (el, fcfs2, pFeedback, null, null);
fb_p.registerSimEntityListener (new StdOutSimEntityListener ());
final SimJob job2 = new DefaultSimJob (null, "2", 1.0);
SimJQEventScheduler.scheduleJobArrival (job2, fb_p, 10.0);
// Run the event list.
el.run ();
```

sub-queues to the composite queue. Also, obviously, you should not schedule events at sub-queues! Most composite-queue implementations are equipped with insane sanity checks for this, and will happily throw an exception at you if they detect unexpected events at one of their sub-queues.

Listing 11.4: The output of Listing 11.3.

```
StdOutSimEntityListener t=0.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=0.0, entity=FB_7[FCFS]: STATE CHANGED:
  => ARRIVAL [Arr[1]@FB_7[FCFS]]
  => START [Start[1]@FB_7[FCFS]]
StdOutSimEntityListener t=1.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=2.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=3.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=4.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=5.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=6.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=7.0, entity=FB_7[FCFS]: UPDATE.
StdOutSimEntityListener t=7.0, entity=FB_7[FCFS]: STATE CHANGED:
  => DEPARTURE [Dep[1]@FB_7[FCFS]]

StdOutSimEntityListener t=10.0, entity=FB_80.0%[FCFS]: UPDATE.
StdOutSimEntityListener t=10.0, entity=FB_80.0%[FCFS]: STATE CHANGED:
  => ARRIVAL [Arr[2]@FB_80.0%[FCFS]]
  => START [Start[2]@FB_80.0%[FCFS]]
StdOutSimEntityListener t=11.0, entity=FB_80.0%[FCFS]: UPDATE.
StdOutSimEntityListener t=12.0, entity=FB_80.0%[FCFS]: UPDATE.
StdOutSimEntityListener t=12.0, entity=FB_80.0%[FCFS]: STATE CHANGED:
  => DEPARTURE [Dep[2]@FB_80.0%[FCFS]]
```

This concludes our examples on composite queues. There is really a lot more to discuss, but for this initial tour, we leave it at this. We hope you are curious to the answers to question like

- When does a real job start?

- How do server-access credits work on a composite queue?

If so, please consult Chapter **??**.

Chapter 12

Queue Implementations

Chapter 13

Job Implementations

The important `SimJob` implementations discussed thus far are shown in Table 13.

Table 13.1: Important `SimJob` Implementations.

Class	Constructor Arguments
`DefaultSimJob`	`SimEventList`, `String`, `double`
`DefaultSimJobQoS`	`SimEventList`, `String`, `double`, `Class`, `Object`

Chapter 14

Statistics

Chapter 15

Use Case: The M/M/1 Queueing System

Chapter 16

Atomicity and Invariants

If you have read this tour linearly, and have reached this point, you have mastered most of the essential features of `jqueues` and `jsimulation`. In the current and next section, we cover two aspects that you may want to skip at first reading, viz., *composite* queues in the next section, and *atomicity* and *queue invariants* in this section. They are, however, signature features of `jqueues`.

As mentioned many times earlier, every `SimEntity` supports a well-defined set of operations. These operations, at least the external ones, are invoked from the event list, or otherwise from within the `SimEntity` itself while processing another operation. The state of a `SimEntity` can only change as a result of the invocation of an operation on it. Immediately after an operation invocation has been received, the `SimEntity` must fire an `UPDATE` notification to its registered listeners, exposing the entity's state immediately prior to receiving the invocation. After completion of the operation, the `SimEntity` must fire a `STATE CHANGED` notification, exposing its new state. The notification may sometimes be left out if the state did not actually change, but, many operation invocations are *always* reported to listeners, even if they do not result in state changes (for instance, invocations of `Arrive`, `Drop`, `Start` and `DEPART` on a `SimQueue`).

But this you already knew, right? So, there has to be more... Let's do this, one step at a time.

First, forget about *all* other notifications than `RESET`, `UPDATE` and `STATE CHANGED`. All other forms or notifications are essentially *courtesy* notifications derived from (mostly) `STATE CHANGED` notifications. You may have already discovered that such courtesy notifications are always fired *after* the `STATE CHANGED` notifications, if at all. It is essential to realize that a full description of the state changes is always available in the `STATE CHANGED` notification. It is often harder to process than the courtesy notifications, sure, but all courtesy notifications are in essence derived from it.

Second, `STATE CHANGED` notifications consist of a *sequence of sub-notifications.* Because, often, the invocation of an event leads to a more complicated state change than can be described in a single notification: Jobs might start upon arrival, or be dropped due to queue-access vacations, or even depart immediately upon arrival. This explains why a `STATE CHANGED` notification comes with a `Set` of `SimEntityEvent`s (in proper sequence), each describing a single step in the state transition.

Third, realize that invocations of an operation come in two flavors: An initial invocation from the event list, and, if applicable, subsequent invocations while processing that initial invocation. (These subsequent invocations are almost always for *internal* operations, but this is by no means a requirement of even the general case.) We refer to the initial revocation as the *top-level operation (invocation)*, and the others as *chained operation invocations*, or shortly (and sloppily) *chained operations.*

We have now set the stage for the big one: *Top-level operation invocations are processed atomically.* Meaning? Well, in simple terms, only the top-level operation is allowed to notify listeners, and only after the new state has been achieved completely. So, upon completion of a chain operation, its relevant notifications are deferred until the top-level notification has finished completely.

Why should we care? Well, consider the following example: We schedule at $t = 0$, a single arrival of a single job requiring 1.0 service time at an otherwise empty `FCFS` queue. There are no queue-access vacations, and the queue has unlimited server-access credits (both are default settings). When the event list processes the event, we know what happens: The job arrives, enters the waiting area, and, because the server is idle, the job starts immediately. Nothing new... But: The scheduled `Arrive` operation is top-level, the resulting `Start` operation is chained, and therefore not allowed to issue notifications by itself. This means that upon return from the `Arrive` operation, we get a combined (using an improvised notation) `STATE CHANGED[ARRIVAL, START]` notification, instead of two separate ones, `STATE CHANGED[ARRIVAL]` followed by `STATE CHANGED[START]`.

So what is the big deal? Well, in order to appreciate atomicity of top-level operation invocations, we need the concept of a *queue invariant*, being a statement about a queue (`SimQueue`) that should *always* hold. In this particular case for `FCFS`, we do not expect jobs residing in the waiting area if the service area is empty (and server-access credits are available). Hence, a properly formulated queue invariant for `FCFS` (and many other queueing systems) is: *If the service area is empty and the queue has non-zero server-access credits, then its waiting area is empty.* In other words: ”There cannot be waiting jobs if the server is idle”.

So, continue please? Well, in `jqueues`, *queue invariants are always* `true` *when notifications are fired.* This is an essential, and in many ways distinguising feature of `jqueues`. It means that `SimQueue` implementations can document their invariants, and listeners are sure that these invariants are always met. This is hard if not impossible (in general) without requiring atomicity on the invocation of operations.

In order to appreciate this, let's consider an alternative approach to implementing `SimQueue` (well, in fact, let's consider a different *interface*). The straightforward strategy we actually used in earlier versions of `jqueues` is to always *schedule* the `Start` operation on the event list on a `FCFS` queue. The implementation is really a whole lot simpler: Upon the arrival of a job, the job is added to the tail of the waiting area and if it is the only one in the waiting area, a `Start` event is scheduled on the event list, obviously with the same event time as the arrival time. The scheduled `Start` operation, once invoked, takes the job out of the waiting area, adds it to the service area, and schedules the invocation of its `Depart` operation. The `Depart` operation then removes the job from the service area, as well as taking other actions to end the visit. And, lo and behold, the approach works just fine. So simple... But: The `Arrive` and `Start` operation must obviously report their inflicted state changes to the registered listeners. This means that `Arrive` notifies its listeners with an `ARRIVAL` notification with the newly arrived job's shining presence in the waiting area. Sure, it will be gone a jiffie (not even that much) later, but the listeners are exposed to a state that is a violation of an all but reasonable queue invariant. It exposes the job in the waiting area while the service area is empty...

So, you create a much more complicated interface and implementation of a `SimQueue` (even of a `SimEntity`) just to achieve that a queue does not expose itself to listeners in a state inconsistent with the queue invariant? Are they that important? Well, yes! They are really that important...

By the way, the `FCFS` queue has an additional queue invariant: its service area does not contain jobs with zero required service time. So if the job in the example would indeed require no service, the notification in response to the invocation of the `Arrive` operation would indeed be `STATE CHANGED[ARRIVAL, START, DEPARTURE]`.

The realization that top-level events (again, sloppy) should be processed atomically was of critical importance for the implementation of *composite queues*; queues that consists of the combination of other queues, the so-called sub-queues. We will introduce them in the next section.

Chapter 17

Further Reading

Bibliography