# Discrete-Event Simulation of Queueing Systems in Java: The JSimulation and JQueues Libraries

Jan de Jongh

Release 5

*This book is dedicated to Dick Epema, Graham Birtwistle, and Isi Mitrani.*

# Contents

# Chapter 1

# Preface

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, 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).

**XXX**

# Chapter 2

# Introduction

# Chapter 3

# Guided Tour

## 3.1 Getting Started

## 3.2 A Simple Simulation with a First-Come First-Served Queue

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

- The creation of an event list;
- The creation of one or more queues, and their attachment to the event list;
- The selection of the method for listening to the queue(s), and/or the event list;
- The creation of a workload consisting of jobs;
- Running the event list.

Without much further ado, here's an example:

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

```
final SimEventList el = new DefaultSimEventList ();
final int bufferSize = 2;
final FCFS_B queue = new FCFS_B (el, bufferSize);
queue.registerStdOutSimQueueListener ();
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 ();
```

In Listing 3.1, we first create a single event list of type `DefaultSimEventList` and a `FCFS_B` queue. On the queue, we invoke its default "listener", issuing notification to the standard output. Subsequently, we create ten jobs named "0", "1", "2", ..., scheduled for arrival at the queue at $t = 0$, $t = 1$, $t = 2$, ..., respectively. Finally, we "run" the event list, i.e., let it process the arrivals.

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`.

Our queue of choice is the First-Come First-Served queue with limited buffer size, `FCFS_B`, described in more detail in Section 6.4.2. The constructor takes two arguments, the event list `el` and the buffer size. The queueing system consists of a queue with only `B` places to hold jobs, and a single server that "serves" the jobs in the queue in order of their arrival. If a job arrives at the `FCFS_B` system while all places in the queue are occupied by other jobs, the arriving job is denied access to the system, and is *dropped*. 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*, and we set a fixed service time (at *any* queue) for each job upon creation as the third argument of the constructur. 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.

Listing 3.2: Example output of Listing 3.1.

```
StdOutSimQueueListener t=0.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener 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]]
StdOutSimQueueListener t=0.0, queue=FCFS_B[2]: ARRIVAL of job 0.
StdOutSimQueueListener t=0.0, queue=FCFS_B[2]: START of job 0.
StdOutSimQueueListener t=0.0, queue=FCFS_B[2]: DEPARTURE of job 0.
```

```
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=3.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=3.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[3]@FCFS_B[2]]
StdOutSimQueueListener t=3.0, queue=FCFS_B[2]: ARRIVAL of job 3.
StdOutSimQueueListener t=3.2, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=3.2, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[1]@FCFS_B[2]]
  => START [Start[2]@FCFS_B[2]]
StdOutSimQueueListener t=3.2, queue=FCFS_B[2]: DEPARTURE of job 1.
StdOutSimQueueListener t=3.2, queue=FCFS_B[2]: START of job 2.
StdOutSimQueueListener t=4.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=4.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[4]@FCFS_B[2]]
StdOutSimQueueListener t=4.0, queue=FCFS_B[2]: ARRIVAL of job 4.
StdOutSimQueueListener t=5.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=5.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[5]@FCFS_B[2]]
  => DROP [Drop[5]@FCFS_B[2]]
StdOutSimQueueListener t=5.0, queue=FCFS_B[2]: ARRIVAL of job 5.
StdOutSimQueueListener t=5.0, queue=FCFS_B[2]: DROP of job 5.
StdOutSimQueueListener t=6.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=6.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[6]@FCFS_B[2]]
  => DROP [Drop[6]@FCFS_B[2]]
StdOutSimQueueListener t=6.0, queue=FCFS_B[2]: ARRIVAL of job 6.
StdOutSimQueueListener t=6.0, queue=FCFS_B[2]: DROP of job 6.
StdOutSimQueueListener t=7.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=7.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[7]@FCFS_B[2]]
  => DROP [Drop[7]@FCFS_B[2]]
StdOutSimQueueListener t=7.0, queue=FCFS_B[2]: ARRIVAL of job 7.
StdOutSimQueueListener t=7.0, queue=FCFS_B[2]: DROP of job 7.
StdOutSimQueueListener t=7.6000000000000005, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=7.6000000000000005, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[2]@FCFS_B[2]]
  => START [Start[3]@FCFS_B[2]]
StdOutSimQueueListener t=7.6000000000000005, queue=FCFS_B[2]: DEPARTURE of job 2.
StdOutSimQueueListener t=7.6000000000000005, queue=FCFS_B[2]: START of job 3.
StdOutSimQueueListener t=8.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=8.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[8]@FCFS_B[2]]
StdOutSimQueueListener t=8.0, queue=FCFS_B[2]: ARRIVAL of job 8.
StdOutSimQueueListener t=9.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=9.0, entity=FCFS_B[2]: STATE CHANGED:
  => ARRIVAL [Arr[9]@FCFS_B[2]]
  => DROP [Drop[9]@FCFS_B[2]]
StdOutSimQueueListener t=9.0, queue=FCFS_B[2]: ARRIVAL of job 9.
StdOutSimQueueListener t=9.0, queue=FCFS_B[2]: DROP of job 9.
StdOutSimQueueListener t=14.200000000000001, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=14.200000000000001, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[3]@FCFS_B[2]]
  => START [Start[4]@FCFS_B[2]]
StdOutSimQueueListener t=14.200000000000001, queue=FCFS_B[2]: DEPARTURE of job 3.
StdOutSimQueueListener t=14.200000000000001, queue=FCFS_B[2]: START of job 4.
StdOutSimQueueListener t=23.0, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=23.0, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[4]@FCFS_B[2]]
  => START [Start[8]@FCFS_B[2]]
StdOutSimQueueListener t=23.0, queue=FCFS_B[2]: DEPARTURE of job 4.
StdOutSimQueueListener t=23.0, queue=FCFS_B[2]: START of job 8.
StdOutSimQueueListener t=40.6, entity=FCFS_B[2]: UPDATE.
StdOutSimQueueListener t=40.6, entity=FCFS_B[2]: STATE CHANGED:
  => DEPARTURE [Dep[8]@FCFS_B[2]]
  => STA_TRUE [StartArmed[true]@FCFS_B[2]]
StdOutSimQueueListener t=40.6, queue=FCFS_B[2]: DEPARTURE of job 8.
StdOutSimQueueListener t=40.6, queue=FCFS_B[2]: START_ARMED -> true.
```

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), and when they are dropped (jobs 5, 6, and 7) by the queue (DROP). The START_ARMED notifications refer to state changes in a special boolean attribute of a queue named its StartArmed property. It is explained in detail in Section 5.3.2 but since you will hardly need it in practical applications, we will not delve into it. Suffice it to say that StartArmed property reveals in some strange way whether the queue *would* start a hypothetical arriving job if several additional conditions are also met. It is crucial for the implementation of more complex queueing systems.

The remaining two notification types, UPDATE and STATE CHANGED are essential, however, and require special attention. Upon every change to a queue's state, the queue is obliged to issue the fundametal 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 preceeded with an UPDATE notification.

However, in the particular case of our example in Listing 3.1, the queue (state-changing) events like arrivals, drops and departures, are also reported separately. These sould be considered *courtesy* notifications. The queue fires them because it recognizes that the *listener* we registered, a so-called StdOutSimQueueListener has direct support for them. The so-called courtesy notifications are quite handy when all you are interested in are events like arrivals and departures, or for instance changes in the StartArmed property.

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).

- 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.

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.

## 3.3 Creating and Registering 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.

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 3.3: 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 3.4: 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 3.4, 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:

Listing 3.5: Example output of Listing 3.1 with the modified listener of Listing 3.4

```
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.
```

If, on the other hand, your *only* interest is in the fundamental `RESET`, `UPDATE` and `STATE CHANGED` notifications, you can register a `StdOutSimEntityListener`:

Listing 3.6: Creating and registering a `StdOutSimEntityListener`.

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

or, simpler but equivalent,

Listing 3.7: Using `registerStdOutSimEntityListener`.

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

with yields in both cases:

Listing 3.8: Example output of Listing 3.1 with the modified listener of Listings 3.6 or 3.7

```
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]]
```

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 3.9.

Listing 3.9: 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 3.10 and 3.11, respectively.

Listing 3.10: Our `FCFS_B` example with the custom `JobSojournTimeListener`.

```
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 3.11: The output of Listing 3.10.

```
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*, see Section 3.11). Third, the listener ignores `RESET` notifications from the queue; a critical error as will become clear later in Section 3.7. 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 9.

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 3.1. 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 3.1: 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 | |

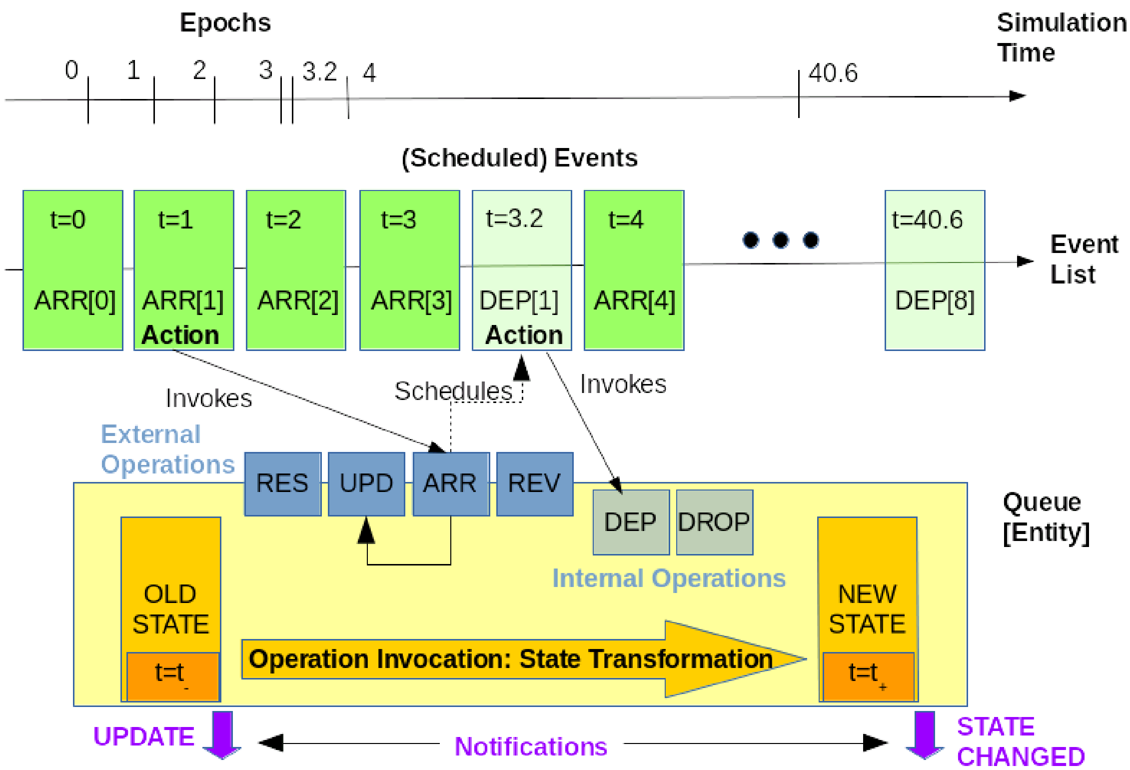
## 3.4 The `UPDATE` and `STATE CHANGED` Notifications

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 3.4.

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

Figure 3.1: 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.




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

[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.

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 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 9.

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).

## 3.5 When Does A Simulation End?

xxx

## 3.6 The `UPDATE` Operation

xxx

## 3.7 Resetting a Simulation

### 3.7.1 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 3.2 and 3.3. 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 3.2: 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 3.3: Mandatory settings in the `RESET` state of a job.

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

### 3.7.2 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 4.) *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 3.13, 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 6.6.1. We reuse the `JobSojournTimeListener`, renaming it to `JobSojournTimeListenerWithReset`, and add a proper `RESET` handler, because unlike queues, listeners must take care of properly resetting themselves. It is shown in Listing 3.12. 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.

Listing 3.12: 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 3.13: 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 () + ".");
}
```

In the outer loop in Listing 3.13, 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 3.14.

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, 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 3.14: The output of Listing 3.13.

```
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 3.13. Subsequently, we turn our attention to "queues of queues", so called *composite* queues in Section 3.14, and finish this chapter with some important other features of `jqueues`.

## 3.8 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,boo` which takes two arguments: (1) the time the operation is invoked, and (2), whether the vacation starts of ends.

It is eesential 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 (` 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 3.15 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 6.6.3. Running the code yields the result on `System.out` as shown in Listing 3.16.

Listing 3.15: 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 3.16: The output of Listing 3.15.

```
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]
```

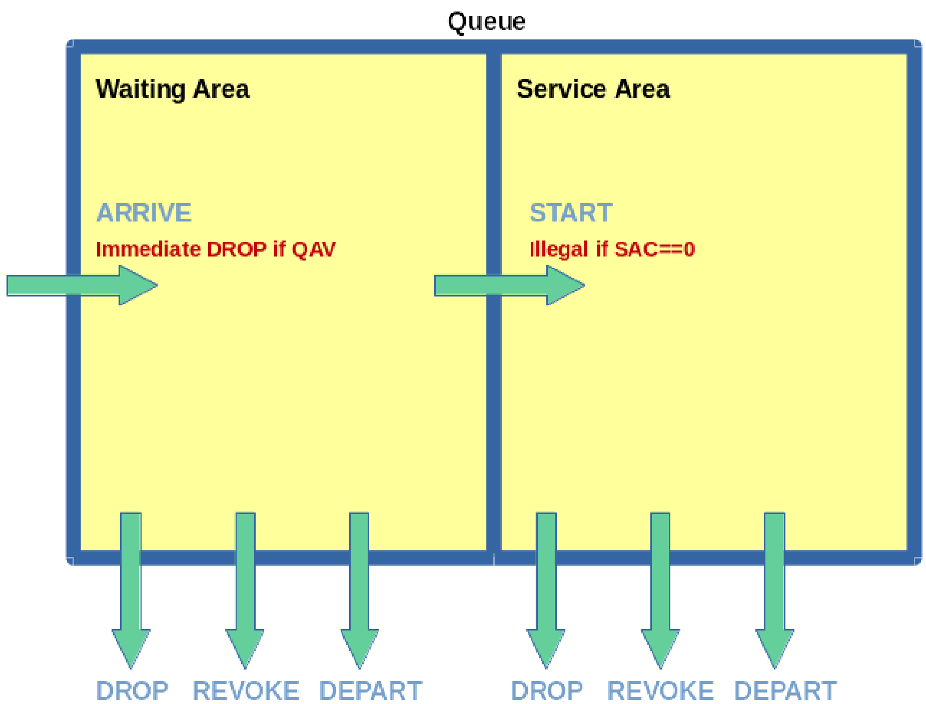
Indeed, as expected, job 2 is dropped due to the scheduled queue-access vacation upon its arrival. Despite this, the arriving job 3 still finds job 1 being (exclusively) served, so `SocPS` schedules their mutual departure. However, the arrival of job 4 is ahead of this scheduled departure, so `SocPS` 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.

## 3.9 The Waiting and Service Area of a Queue

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 3.9. 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

DROPped from, or be REVOKEd from either area. Jobs that do *not* leave the queue are named *sticky* jobs; they may reside in either area.

Figure 3.2: The waiting and service areas of a SimQueue.




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 3.4, we list the most important methods related to waiting and service areas on a SimQueue.

Table 3.4: 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. |

## 3.10 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 3.9. 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) 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 3.2, 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 $\infty$, `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 6.4.3), 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 3.17, 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 3.18.

Listing 3.17: 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 ();
```

Listing 3.18: The output of Listing 3.17.

```
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]
```

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

## 3.11 Revocations

XXX

## 3.12 Multiclass Queues

XXX

## 3.13 The Model of a Queue in `jqueues`: The `SimQueue` Interface

XXX

## 3.14 Composite Queues: The `SimQueueComposite` Interface

XXX

## 3.15 Simultaneous Events

xxx

## 3.16 Atomicity of Operations and Notifications; Queue Invariants

xxx

## 3.17 Infinite Time

xxx

## 3.18 Further Reading

xxx

# Chapter 4

# Events, Event Lists and Actions

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)<br>Schedules the event at its own time. |
| **void** schedule (**double**, E)<br>Schedules the event at given time. |
| reschedule (**double**, E)<br>Reschedules (if present, else schedules) the event at given new time. |
| E schedule (**double**, SimEventAction, String)<br>Schedules the action at given time with given event name. |
| **void** scheduleNow (E)<br>Schedules the event now. |
| E schedule (**double**, SimEventAction)<br>Schedules the action at given time with default event name. |
| E scheduleNow (SimEventAction, String)<br>Schedules the action now with given event name. |
| E scheduleNow (SimEventAction)<br>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

While reading through the previous sections, you may have wondered what would happen if two events are scheduled on exactly the same time. Well, why not just give it a try? First, we create an action class with an index number as argument; when invoked, the action merely prints its index number to `System.out`:

```
private static class IndexedSimEventAction
implements SimEventAction
{

  final int index;

  public IndexedSimEventAction (final int index)
  {
    this.index = index;
  }
```

[3]This code fragment has not been tested.

```
  @Override
  public void action (SimEvent event)
  {
    System.out.println ("Hello, I am action number " + this.index + "!");
  }

  @Override
  public String toString ()
  {
    return "Action " + index;
  }

}
```

So, let us schedule some of these at $t = 0$ in order of increasing index:

```
final SimEventList el = new DefaultSimEventList ();
for (int i = 1; i <= 10; i++)
  el.schedule (0, new IndexedSimEventAction (i), "Event " + i);
el.print ();
el.run ();
```

The potential result of this code may be a bit surprising[4]:

```
SimEventList {X}@{Y}, class=DefaultSimEventList, time=-Infinity:
  t=0.0, name=Event 3, object=null, action=Action 3.
  t=0.0, name=Event 8, object=null, action=Action 8.
  t=0.0, name=Event 9, object=null, action=Action 9.
  t=0.0, name=Event 5, object=null, action=Action 5.
  t=0.0, name=Event 1, object=null, action=Action 1.
  t=0.0, name=Event 4, object=null, action=Action 4.
  t=0.0, name=Event 7, object=null, action=Action 7.
  t=0.0, name=Event 2, object=null, action=Action 2.
  t=0.0, name=Event 6, object=null, action=Action 6.
  t=0.0, name=Event 10, object=null, action=Action 10.
Hello, I am action number 3!
Hello, I am action number 8!
Hello, I am action number 9!
Hello, I am action number 5!
Hello, I am action number 1!
Hello, I am action number 4!
Hello, I am action number 7!
Hello, I am action number 2!
Hello, I am action number 6!
Hello, I am action number 10!
```

Well, it looks like all our scheduled events were indeed processed, *but not in the order we inserted them into the list!* It was even clear *before* processing the event list that there was something "wrong" with the sequence of events. Why? Well, because we explictly instructed the `SimEventList` *not* to do process simultaneous events in so-called *insertion order*, but instead to break ties *at random* for simultaneously scheduled events. The exact reasoning for doing this is a bit involved, and deferred until the "Advanced Topics" section, but for now it is important to realize that a `DefaultSimEventList`

- processes its scheduled events in random order should they have equal schedule times;

[4] The probablity of you seeing the same result is $1/(10!)$, which equals the probability of you being *not* surprised at all about your own output.

- will *never* preempt or interrupt the current event it is processing in favor of another event that is scheduled at the same time from within the action of the current event.

Since there is absolutely nothing wrong with maintaining insertion order, you can switch to a different event-list implementation, viz., `DefaultEventList_IOEL`:

```
final SimEventList el = new DefaultSimEventList_IOEL ();
for (int i = 1; i <= 10; i++)
  el.schedule (0, new IndexedSimEventAction (i), "Event_" + i);
el.print ();
el.run ();
```

Now the output looks mores structured:

```
SimEventList {X}.DefaultSimEventList_IOEL@{Y}, class=DefaultSimEventList_IOEL,
    time=-Infinity:
  t=0.0, name=Event 1, object=null, action=Action 1.
  t=0.0, name=Event 2, object=null, action=Action 2.
  t=0.0, name=Event 3, object=null, action=Action 3.
  t=0.0, name=Event 4, object=null, action=Action 4.
  t=0.0, name=Event 5, object=null, action=Action 5.
  t=0.0, name=Event 6, object=null, action=Action 6.
  t=0.0, name=Event 7, object=null, action=Action 7.
  t=0.0, name=Event 8, object=null, action=Action 8.
  t=0.0, name=Event 9, object=null, action=Action 9.
  t=0.0, name=Event 10, object=null, action=Action 10.
Hello, I am action number 1!
Hello, I am action number 2!
Hello, I am action number 3!
Hello, I am action number 4!
Hello, I am action number 5!
Hello, I am action number 6!
Hello, I am action number 7!
Hello, I am action number 8!
Hello, I am action number 9!
Hello, I am action number 10!
```

Just in case you are curious: we use the abbreviations ROEL for Random-Order Event List and IOEL for Insertion-Order Event List. The `DefaultSimEventList` is obviously a ROEL; actually it subclasses `DefaultSimEventList_ROEL` without changes, hence you can also use `SimEventList_ROEL` as your event-list implementation (perhaps making more explicit the nature of the event list):

```
// Be very careful: this event-list does not respect insertion
// order for simultaneous events!
// ROEL = Random-Order Event List.
final SimEventList el = new DefaultSimEventList_ROEL ();
```

We want to stress that just because our default event-list implementation is a ROEL, it is by no means because ROEL "is just better" than IOEL, or the software in `jsimulation` and/or `jqueues` "works better" or even "works only" with a ROEL. Far from it, the current implementations will probably work slightly faster with an IOEL; maintaining insertion order is likely to be faster that drawing random number upon each insertion. So, by all means, use the IOEL implementation if you want to as a replacement to the default ROEL.

## 4.7 Resetting an Event List

By resetting an event list, through `reset`, we remove all the events it contains, and set the time to $t = -\infty$, or to a user-specified time through `reset (double)`. Resetting an event list is typically done before repeating a simulation experiment, for instance with a different seed value for random-number generation.

Despite the simplicitly of the concept, we cannot stress enough the importance and consequences of resetting an event list. Although not mandated by `jsimulation`, the general convention we follow is that an even-list reset puts the event list itself, but also all "entities" (like queues) that use it *in a well-known, default, often "empty", state.* Partially to that purpose, an event list informs its so-called *reset listeners* when it is reset. The concept of listeners is described in the next section.

For reasons that will become clear later, in particular when discussing obtaining statistics on queueing systems in **XXX**, it is *highly recommended* to always reset the event list to a *finite* time, even before running it for the first time.

One may wonder why we chose $-\infty$ as the default new time upon a reset, instead of a finite value (zero comes to mind here...). Well, first, we wanted a "fresh" event list to accept all events scheduled at *arbitrary* time. (Recall that even without running an event list, it will throw an exception if events are scheduled "in the past".) Second, we wanted to avoid the ubiquitous "a simulation starts at $t = 0$" assumption, at least in our own code. For instance, it would be very hard to debug the case in which a statistics-gathering object (e.g., measuring the average number of jobs in a queue) silently assumes that all simulations start at $t = 0$, and the user decides to run a simulation at (for whatever reason) $t = +100$.

On the other hand, having to remember to pass a finite time upon every reset is, admittedly, far from ideal. The `setDefaultResetTime (double)` therefore allow for changing the default event-list start time used by `reset ()`. Note though, that this still requires explicit resets of the event list, even before using it for the first time. We are still seeking to improve the reset semantics, in particular related to the new time on the event list and its effect on statistics gathering.

**ADD AN EXAMPLE OR TWO HERE!**

## 4.8 Listening to an Event List

You can listen to changes to a `SimEventList` by registering a *listener* of type `SimEventListListener` to it. At the present time, there is only support for notifications for an event list reset, and for event-list processing so there is not general "event list changed" notfication.

A listener gets notifications for:

- A *reset* of the event list. This notification is always sent while the list is *not* begin processed. In fact, if you are only interested in receiving reset notifications, you can use a `SimEventListResetListener` instead of (the full) `SimEventListListener`.
- An *update* of the event list. An update is defined as a *strictly positive jump in time during processing.*
- An *empty* event list during processing (this effectively ends processing the event list).
- A *next event* while processing the event list. These notifications, however, are only sent to listeners that implement `SimEventListListener.Fine`.

The following table summarizes the listener-related methods on a `SimEventList`.

| Method | Description |
|---|---|
| **void** addListener (SimEventListResetListener) | Adds a listener. |
| **void** removeListener (SimEventListResetListener) | Removes a listener. |

The table below list the various notification methods on a listener per listener type; the types are shown in increasing richness.

| Method | Description |
|---|---|
| **SimEventListResetListener** | |
| **void** notifyEventListReset (SimEventList) | Reset of given event list. |
| **SimEventListListener** | |
| **void** notifyEventListUpdate (SimEventList, **double**) | Event list update; *new* time. |
| **void** notifyEventListEmpty (SimEventList, **double**) | Event list empty at given time. |
| **SimEventListListener.Fine** | |
| **void** notifyNextEvent (SimEventList, **double**) | Process next event; *old* time. |

## 4.9 Advanced Topics

In this section we take a closer look at some more advanced topics `jsimulation`. The sections can be skipped at first reading.

### 4.9.1 Using Generic-Type Arguments

| **Interface** | **Type** | Description |
|---|---|---|
| `SimEvent<T>` | `T` | The type of user object. |
| `SimEventList<E extends SimEvent>` | `E` | The type of events. |
| `SimEventAction<T>` | `T` | The type of user object. |

The (partial) implementations in `jsimulation` follow the same convention as the interface they belong to:

| **Class** |
|---|
| `DefaultSimEvent<T>` |
| `AbstractSimEventList<E extends SimEvent>` |
| `DefaultSimEventList_IOEL<E extends SimEvent>` |
| `DefaultSimEventList_ROEL<E extends SimEvent>` |
| `DefaultSimEventList<E extends SimEvent>` |

### 4.9.2 Event Factories

An *event factory* has a sole purpose: generating new event instances. The use of factories is very common in Java, expecially if one needs to create a "default" instance for a given interface, in our case for `SimEvent`.

The `SimEventFactory` interface is as follows:

```
@FunctionalInterface
public interface SimEventFactory<E extends SimEvent>
{

  E newInstance
    (String name, double time, SimEventAction eventAction);

}
```

The generic type argument `E` is the (base) type of the generated `SimEvents`. The argument list for `newInstance` allows for setting all `SimEvent` properties except the user object (because we believe it is rarely of use).

One particular use for this is to set the event factory on an arbitary `SimEventList` through its `setSimEventFactory` method. This is a safe way of using an interface (or event multiple ones) as type argument to `SimEventList`:

```
interface MySimEvent
extends SimEvent
{
  BigInteger getSeqNumber ();
```

```
}

static class DefaultMySimEvent
extends DefaultSimEvent
implements MySimEvent
{

  private static BigInteger
    NEXT_SEQUENCE_NUMBER = BigInteger.ZERO;

  private final BigInteger seqNumber;

  @Override
  public final BigInteger getSeqNumber ()
  {
    return this.seqNumber;
  }

  public DefaultMySimEvent
    (final String name,
     final double time,
     final SimEventAction action)
  {
    super (name, time, null, action);
    this.seqNumber = NEXT_SEQUENCE_NUMBER;
    NEXT_SEQUENCE_NUMBER =
      NEXT_SEQUENCE_NUMBER.add (BigInteger.ONE);
  }

}

public static void main (final String[] args)
{
  final SimEventList<MySimEvent> el =
    new DefaultSimEventList (MySimEvent.class);
  el.add (new DefaultMySimEvent
    ("MySimEvent instance", 5.0, null));
  el.print ();
}
```

In the code fragment, we extended the basic `SimEvent` interface with a method to maintain a global instance counter of type `BigInteger` (ignoring the total lack of usefulness), and created a default implementation for it in `DefaultMySimEvent`. Subsequently, we created an event list, using the generic-type argument and the `class` argument in the constructor to make sure that the `DefaultSimEventList` only accepts `MyEventType` as events. Note that we use the *interface* here, not the *default implementation*. The program runs fine and prints the event scheduled on the event list.

However, if we try to schedule an action (`null` in this case):

```
el.schedule (10.0, (SimEventAction) null);
```

we are treated with an exception:

```
...IllegalStateException: Cannot instantiate MySimEvent!
```

It is more or less immediately clear what the problem is: the event list has to generate a `SimEvent` in order to schedule the action, but it tries to instantiate `MySimEvent`, which is an *interface*. (By the way, it also assumes that the event class supports a parameterless constructor!)

We need to tell the event list how to create the events for the various utility methods, and we do that by creating and registering a `SimEventFactory` for `MySimEvent`[5]:

```
final SimEventList<MySimEvent> el =
  new DefaultSimEventList (MySimEvent.class);
el.setSimEventFactory
(
  (final String name,
   final double time,
   final SimEventAction eventAction)
    -> new DefaultMySimEvent (name, time, eventAction)
);
el.add (new DefaultMySimEvent
  ("MySimEvent instance", 5.0, null));
el.schedule (10.0, (SimEventAction) null);
el.print ();
```

Now we are out of trouble:

```
...
t=5.0, name=MySimEvent instance, object=null, action=null.
t=10.0, name=null, object=null, action=null.
```

This section and the previous one on generic-type arguments hopefully showed the maturity of support the generic and runtime type arguments. However, we do not recommend their use unless for very specific use cases that require extending the `SimEvent` and/or `SimEventList` interfaces. The problem is that by restricting the allowable compile-time and runtime `SimEvent` types, the generated objects become

[5]Using a *lambda expression* this time, see **XXX**.

unusable for libraries using "bare" `SimEvents`. For instance, the `jqueues` library will not work with event-lists not supporting "plain" `SimEvents`.

### 4.9.3 Simultaneous Events: Random-Order and Insertion-Order Event Lists

In previous sections in this book, we explained that event-list implementations, at least the *non-preemptive* types, come in two natural variants: the ROEL processes simultaneous events in random order, and the IOEL does that in insertion order. We also declared that the default implementation is a ROEL, and that you can easily switch to an IOEL, possibly even gaining some performance. This section is dedicated to motivating the use of, or even the consideration of ROEL as an event list implementation, let alone making it the default! It does not introduce any new software. We already want to stress that ROEL is by no means "better" than IOEL from a user point of view. We do not intend to avocate ROEL over IOEL!

The distinction between ROEL and IOEL is all about *simultaneous* events, and we look at this phenomena from three different viewpoint:

- The *physical* viewpoint: In physics, we are rarely concerned with occurence of simultaneous events, because physical models (to our knowledge) rarely exhibit a strong behavioral dependence on "things happening at the same time".

- The *mathematical* viewpoint: In mathematical models, we *are* nearly always dealing with teh possiblity of "things being equal". For instance, it is "undone" two specify an function on two real variables (say, a "maximum indicator") without exactly specifying the result for equal inputs. However, in applications of probability theory, like queueing theory, we often deal with continuous distrubutions and the probability of simulteneous events is often zero. Although simulteneous events still require attention, one can usually get away with noting that "ties are broken at random (with equal probabilities)". Almost always, this solution approach is preferred over trying to impose an additional ordering on the events, as in "Jobs A and B both arrived at $t = 0$, but job A was first."

- The *software-engineering* viewpoint: In software-engineering, we are not that used to simultaneous events. Sure, there are cases of *concurrency* but the problems that arise are usually fixed by imposing some *order* into which statements, programs, expressions, etc. are to be processed. As a result, software engineering is very much concerned with "doing the right thing given a strict order of input". A prime example of this are Finite-State Machines supporting state transitions upon external events. Rarely, if ever, does this take into account the possiblity of simultaneous events.

Needless to say, our argument is that *the occurence of simulatenous events is very natural in mathematics, yet the concept of "insertion order" is purely relevant to software-engineering.* In other words, there is no equivalence in this context for "insertion order" in mathematics, nor is there is physics.

The reasons for choosing a Random-Order Event List are primarily motivated by its use in `jqueues`:

- We do not want to *specify* queueing systems with the notion of insertion order, and we probably cannot. Such a specification would be overly complicated, and, as mentioned before, in the mathematical context we rather break ties at random.
- We do not want to *imply* to users the conservation of insertion order in the implementations.
- We do not want to *rely* on insertion order of events in our tests.

Admittedly, for simple queueing systems like FCFS, it seems simple enough to maintain the order of arrival of jobs in the output process. But as soon as queueing systems become more complicated, especially if multiple queues are involved or feedback, the specification becomes just unnecessarily difficult. To give an example: Suppose we have a feedback FCFS queue with feedback probability 1/4 and two jobs arriving at $t = 0$; jobs A and B having required service times (for each visit) of zero and unity, respectively. Suppose with an insertion-order event list, job A's first arrival is before job B's. So, trusting the conservation of insertion order, the queue starts processing job A, which is fed back to the queue's input immediately with probability 1/4. If so, we are in trouble because we now have to specify whether to first serve job A reappearing at the input or job B. Of course we can find a solution by indeed specifying that a job that is fed back is always inserted *after* jobs already there, but the real problem is that *we have to specify all cases of simultaneus events through insertion-order arguments.* And this is just a simple example.

### 4.9.4 Event Comparators

In order to obtain the required total order on the `SimEvent`s in a `SimEventList`, the latter uses a `Comparator`, which by default is an instance of `DefaultSimEventComparator`, both for ROEL (the default) and IOEL event lists. Its core implementation is

```java
@Override
public int compare (E e1, E e2)
{
  int c = Double.compare (e1.getTime (), e2.getTime ());
  if (c == 0)
  {
    c = e1.getSimEventListDeconflictValue ()
        .compareTo
        (e2.getSimEventListDeconflictValue ());
  }
  if ((e1 == e2 && c != 0)
      || (e1 != e2 && c == 0))
    throw new RuntimeException
     ("Error attempting to order events.");
  return c;
}
```

Not surprisingly, the comparator uses the time property on the events to make a first comparison. In case of a tie, it uses the so-called *deconflict value* on the events, throwing an exception if this still yields a tie. What is intteresting to note is that the deconflict value on an event is generated when an event is *added* to an event list by overriding the `add` and `addAll` methods from the super class (i.e., `TreeSet`), e.g., in case of ROEL:

```java
@Override
public final boolean add (final E e)
{
  if (e == null)
    throw new NullPointerException
      ("Attempt to add null event to event list!");
  if (! contains (e))
  {
    e.setSimEventListDeconflictValue
      (this.rngDeconflict_ROEL.nextLong ());
    return super.add (e);
  }
  return false;
}
```

In ROEL, the deconflict value is, as expected drawn from a random-number generator (RNG) `rngDeconflict_ROEL`. Using this approach has the major advantage that even though events with equal times will be ordered at random, their ordering remains fixed as long as they are in the event list. So, the random ordering in ROEL is *not* implemented by drawing from a RNG the moment it is needed from the event list (which would, actually, be substantially more difficult), but it is already fixed upon insertion. This, for instance, has the nice feature that consecutive iterators over the (same) event set will always return the events in the same order. By the way, you can set an alternative `Comparator` by using one of `AbstractSimEventList` constructors. You cannot change it on the default implementations.

### 4.9.5 Action is a Functional Interface

Because a `SimEventAction` is an interface with exactly one abstract method, it can be used in so-called *lambda expressions* in Java 8. So, instead of

```
final SimEventAction action = new SimEventAction ()
{
  @Override
  public final void action (final SimEvent event)
  {
    // Do something with event...
  }
};
```

you can also write:

```
final SimEventAction action =
  (SimEventAction) (final SimEvent event) ->
  {
    // Do something with event...
  };
```

or even make it a one-liner.

The `SimEventAction` interface has been marked a `@FunctionalInterface`.

## 4.10 Timers

The abstract `AbstractSimTimer` class is a small utility class for scheduling a timer on a `SimEventList`. In view of the classes and methods described before it is not all that useful, but it has been kept in the library for support of legacy code. An `AbstractSimTimer` features a `schedule (double, SimEventList)` method that schedules an appropriate event after a delay (the `double` argument). When the event is processed, the (abstract) method `expireAction` is invoked, which needs to be defined in a subclass. Also, a pending timer can be (safely) cancelled through its `cancel` method.

Below is a small, naive example of its use:

```
private static class MyTimer
extends AbstractSimTimer
{
  @Override
  public final void expireAction
  (final double time)
  {
    System.out.println ("t=" + time + ": Timer expired!");
  }
}

public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  final MyTimer myTimer = new MyTimer ();
  // Progress event list until t=10.
  // Note that AbstractSimTimer does not support t=-\infty.
  el.runUntil (10.0, true, true);
  myTimer.schedule (5, el);
  el.print ();
  el.run ();
}
```

The result of which is:

```
SimEventList {X}@{Y}, class=DefaultSimEventList, time=10.0:
  t=15.0, name=_expire, object=null, action={...}.
t=15.0: Timer expired!
```

Note that you cannot easily *reschedule* the timer from within the action, because there is no access to the `SimEventList`, and that you cannot schedule the timer when the time on the event list is infinite (hence the `runUntil` in the example!).

## 4.11 Summary and Conclusions

This chapter introduced `jsimulation`, a small Java library for discrete-event simulation. An overview of the main interfaces and classes is given below:

| **Interface** | **Class** | **Description** |
|---|---|---|
| SimEvent<T> | | Event |
| | DefaultSimEvent<T> | Default Event |
| SimEventList<E> | | |
| | AbstractSimEventList<E> | Event List (partial) |
| | DefaultSimEventList_ROEL<E> | Random-Order Event List |
| | DefaultSimEventList_IOEL<E> | Insertion-Order Event List |
| | DefaultSimEventList<E> | Default Event List |
| SimEventAction<T> | | Action |
| SimEventListResetListener | | Reset Listener |
| SimEventListListener | | Normal Listener |
| SimEventListResetListener | | Detailed Listener |
| SimEventFactory<E> | | Event Factory |
| | DefaultSimEventFactory | Default Event Factory |

The remainder of this book is about `jqueues`, a library for discrete-event simulation of queueing systems. The `jqueues` library depends on `jsimulation` for scheduling and processing events and actions.

# Chapter 5

# Queueing Systems; Entities, Queues, and Jobs

This chapter introduces the concepts behind and implementation of the `jqueues` package. Together with the underlying `jsimulation` package, this package allows for discrete-event simulation of a very broad range of queueing systems.

## 5.1 Introduction and Definitions

Despite the fact that we deal with queueing, waiting, being served (or not), and being denied service on a daily basis, it is, unfortunately, not that easy at all to precisely define a system that captures these "facts of life". In our, admittedly unsatisfactory, definition, a *queue* or *queueing system* is an entity that can be visited by other entities called *jobs*; each visit being initiated by a so-called *arrival* of a specific job at that queue. A job can only visit only a single queue at a time, yet it can hop to another (or the same) queue once its visit to a particular queue has ended. A queue, on the other hand, can be visited by multiple jobs.

In *queueing theory*, a branch of mathematics, one attempts to predict the behavior of queues and jobs without being (too) concerned about the particular *reasons* of a job visit. A very common setting is that jobs visit a queue in order to get a particular *service* from that queue taking a non-trivial amount of time to complete, the *required service time* associated with the visit. In other words, visiting jobs have to *wait* for the completion of their service request. Even worse, the queue is often limited in providing the required services to multiple jobs simultaneously, so jobs have to *compete* for the *service capacity* of the server. The outcome of this competition is determined by the so-called *service discipline* or *queue discipline* of the queueing system: the way in which it distributes its limited (finite) service *capacity* among its currently visiting jobs.

Unfortunately, the view of a queueing system as consisting of a waiting area in which jobs wait before being served by one or more servers (i.e., a finite number of them) for a given required service time is too narrow in many interesting applications. For instance, in Medium-Access Control systems, the required ”service” to waiting jobs (i.e., *frames to be transmitted*) is largely determined by external events, viz., the availability of the *medium* for transmissions. In such cases, jobs are not staying in the queue because of an *intrincic* requirement of service from that queue, but merely because they are waiting for an event *external* to that queue. In other words, the notion of ”servers” in a queueing system providing ”service” works for many cases, but not for all. And from (theoretical) examples like the Processor-Sharing[1] queueing discipline, it is clear that the notion of servers serving at any time a single job exclusively, needs refinement as well. So, a single server is not constrained, in the general case, to serving only a single job. In Infinite-Server (IS), on the other hand, each job present is served by its own server (of which there are infinitely many), meaning that the ”total service capacity” is not a fixed constant, let alone be known in advance. The capacity may not even be finite.

In our conceptual model of a queueing system, we include explicitly the notions of waiting and of being served, but we do not impose *any* structure on the service. In our implementation, queues and jobs are named `SimQueue`s and `SimJob`s, respectively. Their common features are implemented in an abstract base class `SimEntity`; for simulation entity.

## 5.2 Simulation Entities

A `SimEntity` is an entity relevant to event-list scheduling in a queueing system simulation. Presently, it is either a queue (`SimQueue`) or a job (`SimJob`). A queue is an object capable of holding *visiting* jobs, providing (generic) service to these jobs, and deciding when (or if) they will leave the queue, and end the visit. A `SimEntity` is the common part of queues and jobs. What they share in common is the event list (`SimEventList`) they are attached to, the fact that they have a name, and their obligation to propagate state changes (including the currently visited queue of a job, and the jobs currently visiting a queue) to registered *listeners*. In addition, they must notify such listeners of a reset of the event list. In the table below, we summarize

[1] In Processor-Sharing (PS), a server equally distributes its service capacity among jobs present, see **XXX**.

the `SimEntity` methods.

| Method | Description |
|---|---|
| SimEventList getEventList () | Gets the non-`null` event list. |
| **void** registerSimEntityListener (SimEntityListener<J, Q> listener) | Adds a listener. |
| **void** unregisterSimEntityListener (SimEntityListener<J, Q> listener) | Removes a listener. |
| Set<SimEntityListener<J, Q>> getSimEntityListeners () | Gets a set of current listeners. |
| String toStringDefault () | Returns a default, type-specific name of the entity. |
| setName (String name) | Sets the name of the entity. |
| **void** resetEntity () | Puts the entity in its known initial state. |

## 5.3 Queues

### 5.3.1 Structure of a SimQueue

A `SimQueue` consists of two areas, and while a `SimJob` visits a queue, it is present in either one of them:

- The *waiting area*: After a successful arrival at the queue, the visiting job always enters the waiting area; in the waiting area, jobs wait before they can be "served", or until they leave the queue otherwise.

- The *service area*: In the service area, jobs receive some sort of (otherwise irrelevant) service from the queue. A job can only enter the service area *from the waiting area*; it cannot directly enter the service area.

| Method | Description |
|---|---|
| Set<J> getJobs () | The current visitor jobs. |
| **int** getNumberOfJobs () | The number of current visitor jobs. |
| Set<J> getJobsInWaitingArea () | The current visitor jobs in the waiting area. |
| **int** getNumberOfJobsInWaitingArea () | The number of current visitor jobs in the waiting area. |
| Set<J> getJobsInServiceArea () | The current visitor jobs in the service area. |
| **int** getNumberOfJobsInServiceArea () | The number of current visitor jobs in the service area. |

### 5.3.2 The SimQueue-Visit Lifecycle

#### Job Arrival

A queue visit starts with the arrival of a job through `arrive(double, J)`. The first argument of `arrive` is the arrival time; typically an arrival is scheduled as the result of processing a `SimEvent` on the event list, and the time argument is taken from the event. The second argument is the `SimJob` that arrives; note the use of the generic type `J` restricting the type of jobs (at compile time) allowed at the `SimQueue`.

| Method | Description |
|---|---|
| **void** arrive (**double**, J) | Arrival of a job at the queue. |

#### Job Drop

At the discretion of the `SimQueue`, a job can be forced to leave the queue; this is referred to as a *job drop*. Implementations of the `SimQueue` interface must specify under which conditions they decide to drop jobs. Typical examples are the unavailability of buffer space, or exceeding a maximum allowed service time. Note that job drops cannot be requested by the user of the queue. A job can be dropped from the waiting area as well as from the service area. (For completeness, we note that a job can also be dropped immediately upon arrival due to queue-access vacations; see next section.)

### Queue-Access Vacation

During a queue-access vacation, access to the `SimQueue` is prohibited and all jobs are dropped immediately upon arrival. A queue-access vacation affects the queue's behavior only upon arrivals; it has no effects whatsoever on jobs that are already present at the queue. Note that formally, a job arrival at a queue with queue-access vacation is *not* a visit, because the job is never actually present at the queue.

| Method | Description |
|---|---|
| **void** setQueueAccessVacation (**double**, **boolean**) | Starts or stops a QAV at given time. |
| **boolean** isQueueAccessVacation () | Checks for an ongoing QAV. |

### Job Start

At the discretion of the `SimQueue`, a visiting `SimJob` can be moved from the waiting area to the service area. This is referred to as a *job start*. The start of a job cannot be (directly) controlled by the user. In our interface, it is important to note that 'started jobs' do not necessarily have exclusive access to the server(s). Between arrival and start, a job is said to be *waiting*. After its start, a job is said to be *started*.

### Job Revocation

Once a job has been offered, `revoke(double, J, boolean)` tries to revoke the job, if (still) possible and if supported by the queue discipline at all. The return value indicates if the revocation succeeded. The first argument is the time of the revocation attempt, the second is the job to be revoked. The third argument indicates whether it is allowed to revoke the job from the service area. If `false`, the revocation attempt will always fail if the job is already in the service area. Note the difference between a revocation (at the caller's discretion) and a drop (at the queue's discretion).

| Method | Description |
|---|---|
| **boolean** revoke (**double**, J, **boolean**) | Revocation attempt of a job at the queue. |

### Server-Access Credits

During a *server-access vacation*, jobs are prohibited to start, i.e., there is no access to the service area for jobs in the waiting area. It does not affect jobs that have already started. Server-access vacations are actually somewhat more flexible through the notion of *server-access credits*, denoting the number of jobs still admissible to the service area, see `getServerAccessCredits()`. A server-access vacation starts when there are no more server-access credits (due to jobs starting), and ends when credits are explicitly granted to the interface through `setServerAccessCredits(double, int)`. Note that by default, each `SimQueue` has infinite server-access credits.

| Method | Description |
|---|---|
| **void** setServerAccessCredits (**double**, **int**) | Sets the remaining SACs at given time. |
| **int** getServerAccessCredits () | Gets the actual SACs. |

Note that the value `Integer.MAX_VALUE` is interpreted as $+\infty$.

### Job Departure

At the discretion of the `SimQueue`, a visiting job may leave the queue because it "got what it came for", i.e., its visit comes to an end according to the queueing discpline in place. This is referred to a *job departure*; and it cannot be enforced by the user. We want to stress that jobs do not necessarily depart from the service area, but can depart from the waiting area as well, or even immediately upon arrival (in which case we formally do not speak of a visit).

### The StartArmed Property of a SimQueue

Every `SimQueue` (implementation) supports the notion of the `StartArmed` property. The property is a but difficult to grasp at first sight and has little use in simulations, but it is essential in so-called *compressed tandem queues*, see **XXX**. Nonetheless, since it is part of the `SimQueue` interface, we state its formal definition: A `SimQueue` is in `StartArmed` state, if and only if any (hypothetical) arriving job *would* start service immediately (i.e., enter the service area upon arrival immediately), *if* the following conditions *would* hold:

- the absence of a queue access vacation,
- at least one server-access credit, and
- an empty waiting area.

The actual values of these three state properties is irrelevant, which, admittedly, makes the definition quite hard to understand. If a `SimQueue` is in `StartArmed` state, changing its state such that it meets the three conditions above, would lead to the immediate start of a hypothetical arriving job (irrespective of the type and properties of that job).

As an example, consider an instance of the (single-server) `FCFS` queueing system which has no queue-access vacation, and suppose that it is out of server-access credits, has a single job in the waiting area and no jobs in the service area. In this case, `StartArmed == true` because an arriving job would be taken into service if we apply the state transformation rules: (1) remove any queue-access vacation (check), (2) give it a single (or more) server-access credit, and (3) remove the job from the waiting area. This leaves a `FCFS` queueing system without queue-access-vacation, no jobs in the waiting area, and at least one server-access credits, and surely, such a queue would immediately start an arriving job. If initially, however, a job would be in service at the queue, we have `StartArmed == false`, because the transformed state of the queueing system has a job in service, and `FCFS` cannot guarantee at all that an arriving job would be taken into service. If on he other side, in the latter case the queue is of type `PS`, we would have `StartArmed == true`, because the presence of jobs in the service area in `PS`, does not inhibit it from immediately starting newly arrived jobs.

Informally, the `StartArmed` state of a queue reflects the fact *as far as the service area is concerned*, at least one more job can be added to it.

| Method | Description |
|---|---|
| **boolean** isStartArmed () | Gets the `StartArmed` property value. |

### Copying a SimQueue

Every `SimQueue` (implementation) supports the creation of a copy of itself.

## 5.4 The SimJob Interface

Compared to the `SimQueue` interface, the `SimJob` interface is remarkably simple. Apart from the internal maintenace of the `SimQueue` being visited, a `SimJob` only needs to provide information on the so-called *requested service time* for a queue visit, through implementation of `getServiceTime (Q)`. This method is used by a @link SimQueue to query the requested service time, and appropriately schedule a departure event for the job, but it can be called anytime. However, the returned value should not change during a visit to a `SimQueue`, and it is not manipulated by the queue being visited, in other words, it cannot be used to query the remaining service time of a job at a queue. It is safe though to change the return value in-between queue visits. However, the convention is that the method then returns the required service time at the *next* visit to the queue. For instance, many test and job-factory classes depend on this, as they often directly probe a non-visiting job for its required service time at a queue. Obviously, implementations must be prepared for invocations of this method while not visiting a queue. If `null`,s passed as agrument the service time at the current queue is used, or zero if the job is not currently visiting a queue.

| Method | Description |
|---|---|
| Q<br>getQueue () | The queue currently visiting. |
| **void**<br>setQueue (Q) | Sets the queue currently visiting. |
| **double**<br>getServiceTime (Q) | Gets the service time for a visit. |

## 5.5 Invariants and Constraints

Despite the large number of freedom degrees for `SimQueue`s, there is also a number of (obvious) restrictions on the behavior of a queue. For instance,

- a job cannot start, be dropped or be revoked before having arrived;
- a job can start at most once during a queue visit;
- a job can only leave the queueing system through departure (with or without being served), successful revocation or drop;
- a job may not leave the queue at all (a *sticky* job).

Note that with the current interface, a `SimJob` cannot visit multiple SimQueues simultaneously. The `SimQueue` currently being visited by a `SimJob` can be obtained from `SimJob.getQueue ()`; this must be maintained by implementations of `arrive (J, double)`.

## 5.6 Listening to a SimEntity

One can listen to relevant events at a `SimEntity` by registering with the entity as a `SimEntityListener`, the methods of which are summarized in the table below.

| Method | Description |
|---|---|
| **void** notifyResetEntity (SimEntity) | Notification of the reset of an entity |
| **void** notifyUpdate (**double**, SimEntity) | Notification of an update at an entity |
| **void** notifyStateChanged (**double**, SimEntity) | Notification of a state change at an entity |
| **void** notifyArrival (**double**, J, Q) | Notification of the arrival of a job at a queue |
| **void** notifyStart (**double**, J, Q) | Notification of the start of a job at a queue |
| **void** notifyDrop (**double**, J, Q) | Notification of the drop of a job at a queue |
| **void** notifyRevocation (**double**, J, Q) | Notification of the successful revocation of a job at a queue |
| **void** notifyDeparture (**double**, J, Q) | Notification of the departure of a job at a queue |

It is important to realize that most notifications are fired at the time the `SimEntity` has processed the corresponding event, and attained its new state. However, there are the following exceptions:

- Update notifications are, by their nature, fired *before* any state change;
- Arrival notifications are fired *before* the job enters the queue (or before it is dropped);
- Unsuccessful revocations are *not* reported as notification.

Notifications can have tricky semantics, and one should be careful at making assumptions of the `SimEntity` upon receiving a notification. This is because with notifications, we attempt to achieve two potentially conflicting objectives:

- Each state change is reported as notification;
- Notification listeners always find the `SimEntity` in a valid state.

So, for instance, one should not assume that during an arrival notification, the job is actually present at the queue, because a queue-access vacation may be active. In that case, the queue will report an arrival of the job, immediately followed by a notification of the job having been dropped. However, upon reception of the arrival notification, the job will not be present at the `SimQueue`, because that would exhibit an illegal state of that queue. What we would actually like here, is a notification of a *sequence* of events at the queue, all happening at the same time, yet in a particular order. In this particular example, a notification of the sequence `(ARRIVAL(t,j,q), DROP(t,j,q))` would be required, where `t` is the time, `j` is the job and `q` is the queue. And even though this may not seem important right now, realize that certain queue types depend on maintaining the validity of invariants throughout their lifetime. For instance, the `DROP` queue drops jobs immediately upon arrival, irrespective of the state of the queue-access vacation. An important sensible invariant for the `DROP` queue, is that the set of jobs visiting it *is always empty*. Analogeous invariants exist for other queue types, and we take great care to enforce these invariants to listeners as well as to independently scheduled events.

For `SimQueues`, one can ontain additional information by registering as a `SimQueueListener`, the methods of which are summarized in the table below.

| **Method** | **Description** |
|---|---|
| **void** notifyNewNoWaitArmed (**double**, Q, **boolean**) | Notification of a change of the `noWaitArmed` state |
| **void** notifyStartQueueAccessVacation (**double**, Q) | Notification of the start of a queue-access vacation |
| **void** notifyStopQueueAccessVacation (**double**, Q) | Notification of the end of a queue-access vacation |
| **void** notifyOutOfServerAccessCredits (**double**, Q) | Notification that a queue has run out of server-access credits |
| **void** notifyRegainedServerAccessCredits (**double**, Q) | Notification that a queue has regained server-access credits |

## 5.7 Formal Treatment of Simulation Entities

Whereas the previous sections introduced `SimEntity`, `SimQueue`, and `SimJob` objects from a use-case perspective, the current section tightens the rope a bit, and treats these objects more formally.

### 5.7.1 Properties of an Entity

#### Operational Properties

An *operational* property of an entity is a property that can be changed at any time by the user without effects on the entity's behavior in a simulation. In addition, these properties are never affected by a simulation (i.e., by *operations* on the entity).

Examples include:

- The *name* of an entity;
- The *listeners* to the entity.

### Essential Properties

An *essential* property of an entity is a property that can be only be changed before and after a simulation (i.e., immediately after a `reset` or construction, but before the start of the simulation). Like operational properties, however, essential properties are never affected by operations on the entity. Essential properties impact the behavior of the queueing system during simulations, but should remain constant while simulating. Examples include:

- The *buffer size* of a `FCFS_B` queue;
- The *number of servers* of a `FCFS_c` queue;
- The *waiting time* of a `DELAY` queue.

### State Properties

Unlike operational and essential properties, *state* properties cannot be directly set by the user; they can only be read. State properties are those properties that change as a result of a simulation running, and the entity being subject to scheduled operations from the event list. State properties can be further subdivided into

- **Event** properties: State properties that can only change due to an operation (either internal or external) on the entity.
- **Continuous** properties: State properties that (potentially) change continuously over time, even in the absence of operations on the entity.

Some examples of event properties:

- The *set of jobs (in the waiting/service) area)* present at a `SimQueue`;
- The queue-access vacation state of a `SimQueue`;
- The `StartArmed` state of a `SimQueue`.

Some examples of continuous properties:

- The *remaining service time* of the job in the service area in `PS` queue;
- The *total remaining work* in a `FCFS` queue;
- The *elapsed time since the start of the current busy period* for a `SimQueue`.

Note that changes to event properties are always reported to listeners of the entity; whereas changes to continuous properties are *never* reported.

Orthogonal to distinction between state and continuous properties, a state property may be

- **Exposed**: The property value is accessible to callers.
- **Hidden**: The property value is hidden from callers.

### 5.7.2 The State of a Simulation Entity

### 5.7.3 External Operations on a Simulation Entity

### 5.7.4 Internal Operations on a Simulation Entity

### 5.7.5 Notifications of a Simulation Entity

### 5.7.6 Extensions

#### State Extensions

#### Operation Extensions

#### Notification Extensions

# Chapter 6

# Fundamental Queues

In this chapter we describe implementations of `SimQueue` corresponding to well-knwon queueing disciplines.

## 6.1 Introduction

## 6.2 The Generic `SimQueue`

### 6.2.1 Introduction

### 6.2.2 Essential Properties

| SimQueue | | |
|---|---|---|
| **Essential Properties** | | |
| `EventList` | `SimEventList` | The event list; non-`null`. |
| `AutoRevocationPolicy` | `AutoRevocationPolicy` | The auto-revocation policy. |
| `RegisteredOperations` | `Set<SimEntityOperation>` | The registered operations. |
| `RegisteredNotificationTypes` | `Set<Member>` | The registered notification types. |

### 6.2.3 State Properties

| SimQueue | | | |
|---|---|---|---|
| **State Properties** | | | |
| LastUpdateTime | EE | `double` | Last update (or reset) time. |
| QueueAccessVacation | EE | `boolean` | Queue-Access Vacation state. |
| StartArmed | EE | `boolean` | The `StartArmed` state. |
| Jobs | EE | `Set<J>` | Jobs present (default: arrival order). |
| JobsInWaitingArea | EE | `Set<J>` | Jobs waiting (default: arrival order). |
| JobsInServiceArea | EE | `Set<J>` | Jobs in service (default: start order). |
| ServerAccessCredits | EE | `int` $\geq 0$ | Number of server-access credits. |

### 6.2.4 Operations

| SimQueue | | | |
|---|---|---|---|
| **Operations** | | | |
| Update | E | `double` | Update at time. |
| DoOperation | E | `double`, `Request` | Perform operation at time. |
| SetQueueAccessVacation | E | double, `boolean` | Starts/ends a QAV at time. |
| Arrive | E | `J` | Arrival of job at time. |
| Drop | I | `J` | Drop of job at time. |
| Revoke | E | `J`, `boolean` | Revocation attempt of job at time. |
| SetServerAccessCredits | E | `int` $\geq 0$ | Set the SACs at time. |
| Start | I | `J`, `double` | Starts job at time. |
| Depart | I | `J` | Lets job depart at time. |

### 6.2.5 State Invariants

| SimQueue |
| --- |
| **State Invariants** |
| $\texttt{JobsInWaitingArea} \cap \texttt{JobsInServiceArea} = \emptyset$.<br>$\texttt{Jobs} = \texttt{JobsInWaitingArea} \cup \texttt{JobsInServiceArea}$. |

### 6.2.6 Notification Types

| SimQueue | | |
| --- | --- | --- |
| **Notification Types** | | |
| RESET | double | Reset notification with new time. |
| UPDATE | double | Update notification with new time. |
| STATE_CHANGED | double, Set<Notification> | State-changed notification<br>with new time and sub-notifications. |

### 6.2.7 Internal Schedulable Events

| SimQueue | |
| --- | --- |
| **Internal Schedulable Events** | |
| DepartureEvent | The scheduled departure of a job. |

### 6.2.8 Operational Properties

| SimQueue | | |
|---|---|---|
| **Operational Properties** | | |
| `Name` | `String` | The name,<br>default depends on sub-class. |
| `SimEntityListeners` | `Set<SimEntityListener>` | The registered listeners. |

## 6.3 Serverless Queues

The *serverless* queueing systems in `jqueues` have no servers and (effectively) no service area. All state operations and state changes concentrate at the waiting area of the queue, and typically upon arrivals. Note that just because the serverless queues have no service area does not mean that visiting jobs cannot depart from it.

### 6.3.1 The `DROP` `SimQueue`

| DROP | |
|---|---|
| *Drops jobs immediately upon arrival.* | |
| **Super** | |
| `SimQueue` | See Section 6.2. |
| **State Invariants** | |
| `StartArmed` = `false`.<br>`Jobs` = $\emptyset$. | |
| **Operational Properties** | |
| `Name` | The name, default "DROP". |

### 6.3.2 The SINK SimQueue

| SINK | |
|---|---|
| *Lets jobs wait indefinitely.* | |
| **Super** | |
| SimQueue | See Section 6.2. |
| **State Invariants** | |
| StartArmed = false.<br>Jobs = JobInWaitingArea. | |
| **Operational Properties** | |
| Name | The name, default "SINK". |

### 6.3.3 The ZERO SimQueue

| ZERO | |
|---|---|
| *Lets jobs depart immediately upon arrival.* | |
| **Super** | |
| SimQueue | See Section 6.2. |
| **State Invariants** | |
| StartArmed = false.<br>Jobs = $\emptyset$. | |
| **Operational Properties** | |
| Name | The name, default "ZERO". |

### 6.3.4 The `DELAY` `SimQueue`

| DELAY | |
|---|---|
| *Lets jobs depart after a fixed wait time.* | |
| **Super** | |
| `SimQueue` | See Section 6.2. |
| **Essential Properties** | |
| `WaitTime` | The wait time, non-negative. |
| **State Invariants** | |
| `StartArmed` = `false`.<br>`Jobs` = `JobsInWaitingArea`.<br>`WaitTime` = 0 ⇒ `Jobs` = ∅. | |
| **Operational Properties** | |
| `Name` | The name, default "DELAY[`WaitTime`]". |
| **Equivalences** | |
| `DELAY`[0] = `ZERO`.<br>`DELAY`[∞] = `SINK`. | |

### 6.3.5 The `GATE` `SimQueue`

| GATE | | | |
|---|---|---|---|
| *Lets jobs depart in arrival order while its "gate" is open; lets jobs wait otherwise.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **State Properties** | | | |
| `GatePassageCredits` | EE | `int` $\geq 0$ | Remaining allowed departures. |
| **Operations** | | | |
| `SetGatePassageCredits` | E | `int` $\geq 0$ | Sets the remaining number of GPCs.<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| **State Invariants** | | | |
| `StartArmed` $=$ `false`.<br>`GatePassageCredits` $> 0 \Rightarrow$ `Jobs` $= \emptyset$. | | | |
| **Operational Properties** | | | |
| `Name` | The name, default "GATE". | | |

### 6.3.6 The `ALIMIT` `SimQueue`

<table>
<tr><td colspan="4" align="center">ALIMIT<br><br>Arrival-Rate Limiter</td></tr>
<tr><td colspan="4">Lets jobs depart immediately but drops jobs whose arrival exceeds a given arrival-rate limit.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>RateLimit</code></td><td><code>double</code></td><td colspan="2">The rate limit, >= 0; $\leq +\infty$.</td></tr>
<tr><td colspan="4" align="center">State Properties</td></tr>
<tr><td><code>RateLimited</code></td><td>HE</td><td><code>boolean</code></td><td>Whether currently (arrival) rate-limited.</td></tr>
<tr><td colspan="4" align="center">Internal Schedulable Events</td></tr>
<tr><td colspan="3"><code>RateLimitExpirationEvent</code></td><td>The expiration of the rate-limit period.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td colspan="3"><code>Name</code></td><td>The name, default "ALIMIT[RateLimit]".</td></tr>
<tr><td colspan="4" align="center">Equivalences</td></tr>
<tr><td colspan="4"><code>ALIMIT</code>[0] = <code>DROP</code>.<br><code>ALIMIT</code>[$\infty$] = <code>ZERO</code>.</td></tr>
</table>

### 6.3.7 The `DLIMIT` `SimQueue`

| | | | |
|---|---|---|---|
| DLIMIT<br>Departure-Rate Limiter | | | |
| *Lets jobs depart immediately but respects a given departure-rate limit.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `RateLimit` | `double` | The rate limit, $>= 0$; $\leq +\infty$. | |
| **State Properties** | | | |
| `RateLimited` | HE | `boolean` | Whether currently (departure) rate-limited. |
| **Internal Schedulable Events** | | | |
| `RateLimitExpirationEvent` | | | The expiration of the rate-limit period. |
| **Operational Properties** | | | |
| `Name` | | | The name, default "DLIMIT[RateLimit]". |
| **Equivalences** | | | |
| `DLIMIT`$[0] =$ `SINK`.<br>`DLIMIT`$[\infty] =$ `ZERO`. | | | |

### 6.3.8 The LeakyBucket SimQueue

| LeakyBucket | | | |
|---|---|---|---|
| *Lets jobs depart immediately but respects a given departure-rate limit, and drops arriving jobs that need to wait but find the waiting area full.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | `int` $\geq 0$ | The buffer size; `Integer.MAX_VALUE` is treated as $+\infty$. | |
| `RateLimit` | `double` | The rate limit, $>= 0$; $\leq +\infty$. | |
| **State Properties** | | | |
| `RateLimited` | HE | `boolean` | Whether currently (departure) rate-limited. |
| **Internal Schedulable Events** | | | |
| `RateLimitExpirationEvent` | The expiration of the rate-limit period. | | |
| **Operational Properties** | | | |
| `Name` | The name, default "LeakyBucket[BufferSize,RateLimit]". | | |
| **Equivalences** | | | |
| `LeakyBucket`$[\infty, L]$ = `DLIMIT`$[L]$.<br>`LeakyBucket`$[0, L]$ = `ALIMIT`$[L]$. | | | |

### 6.3.9 The `WUR` `SimQueue`

| | |
|---|---|
| **WUR** | |
| Wait Until Relieved | |
| *Lets arriving jobs wait for their departure until the next arrival.* | |
| **Super** | |
| `SimQueue` | See Section 6.2. |
| **State Invariants** | |
| $\|$`Jobs`$\|$ = 0 or 1. | |
| **Operational Properties** | |
| `Name` | The name, default "WUR". |

## 6.4 Nonpreemptive Queues

### 6.4.1 The `FCFS` `SimQueue`

<table>
<tr><td colspan="4" align="center">FCFS<br><br>First-Come First-Served</td></tr>
<tr><td colspan="4">Serves jobs in arrival order until completion with a single server<br>and infinite buffer size.<br>The server can serve at most one job at a time.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>BufferSize</code></td><td>F</td><td><code>Integer.MAX_VALUE</code></td><td>The buffer size;<br><code>Integer.MAX_VALUE</code> is treated as $+\infty$.</td></tr>
<tr><td><code>NumberOfServers</code></td><td>F</td><td>1</td><td>The number of servers.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4"><code>StartArmed</code> $=$ (<code>JobsInServiceArea</code> $= \emptyset$).<br>$|$<code>JobsInServiceArea</code>$| \leq 1$.<br><code>ServerAccessCredits</code> $> 0$, <code>JobsInWaitingArea</code> $\neq \emptyset \Rightarrow$ <code>JobsInServiceArea</code> $\neq \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td><code>Name</code></td><td colspan="3">The name, default "FCFS".</td></tr>
</table>

### 6.4.2 The `FCFS_B` `SimQueue`

<table>
<tr><td colspan="4" align="center">FCFS_B<br><br>First-Come First-Served with given buffer size</td></tr>
<tr><td colspan="4">Serves jobs in arrival order until completion with a single server<br>and given (possibly finite) buffer size (<code>BufferSize</code>).<br>The server can serve at most one job at a time.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>BufferSize</code></td><td>C</td><td><code>int</code> $\geq 0$</td><td>The buffer size;<br><code>Integer.MAX_VALUE</code> is treated as $+\infty$.</td></tr>
<tr><td><code>NumberOfServers</code></td><td>F</td><td>1</td><td>The number of servers.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4"><code>StartArmed</code> $=$ (<code>JobsInServiceArea</code> $= \emptyset$).<br>$|$<code>JobsInWaitingArea</code>$| \leq$ <code>BufferSize</code>.<br>$|$<code>JobsInServiceArea</code>$| \leq 1$.<br><code>ServerAccessCredits</code> $> 0$, <code>JobsInWaitingArea</code> $\neq \emptyset \Rightarrow$ <code>JobsInServiceArea</code> $\neq \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td><code>Name</code></td><td colspan="3">The name, default "FCFS_B[<code>BufferSize</code>]".</td></tr>
<tr><td colspan="4" align="center">Equivalences</td></tr>
<tr><td colspan="4"><code>FCFS_B</code>[<code>Integer.MAX_VALUE</code>] $=$ <code>FCFS</code>.</td></tr>
</table>

### 6.4.3 The FCFS_c SimQueue

<table>
<tr><td colspan="4" align="center">FCFS_c<br><br>First-Come First-Served with given number of servers</td></tr>
<tr><td colspan="4">Serves jobs in arrival order until completion<br>with a given number (NumberOfServers) of equal servers and infinite buffer size.<br>Each server can serve at most one job at a time.<br>Jobs do not prefer one server over another; there are no server affinities.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td>SimQueue</td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td>BufferSize</td><td>F</td><td>Integer.MAX_VALUE</td><td>The buffer size;<br>Integer.MAX_VALUE is treated as $+\infty$.</td></tr>
<tr><td>NumberOfServers</td><td>C</td><td>int $\geq 0$</td><td>The number of servers;<br>Integer.MAX_VALUE is treated as $+\infty$.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4">StartArmed $=$ ($|$JobsInServiceArea$|$ $<$ NumberOfServers).<br>$|$JobsInServiceArea$|$ $\leq$ NumberOfServers.<br>ServerAccessCredits $> 0$, JobsInWaitingArea $\neq \emptyset$<br>$\Rightarrow$ $|$JobsInServiceArea$|$ $=$ NumberOfServers.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td>Name</td><td colspan="3">The name, default "FCFS_NumberOfServers".</td></tr>
<tr><td colspan="4" align="center">Equivalences</td></tr>
<tr><td colspan="4">FCFS_0 $=$ DROP.<br>FCFS_1 $=$ FCFS.</td></tr>
</table>

### 6.4.4 The FCFS_B_c SimQueue

<table>
<tr><td colspan="4" align="center">FCFS_B_c<br><br>First-Come First-Served with given buffer size and number of servers</td></tr>
<tr><td colspan="4">Serves jobs in arrival order until completion<br>with a given number (NumberOfServers) of equal servers<br>and given (possibly finite) buffer size (BufferSize).<br>Each server can serve at most one job at a time.<br>Jobs do not prefer one server over another; there are no server affinities.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td>SimQueue</td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td>BufferSize</td><td>C</td><td>int $\geq 0$</td><td>The buffer size;<br>Integer.MAX_VALUE is treated as $+\infty$.</td></tr>
<tr><td>NumberOfServers</td><td>C</td><td>int $\geq 0$</td><td>The number of servers;<br>Integer.MAX_VALUE is treated as $+\infty$.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4">StartArmed = (|JobsInServiceArea| < NumberOfServers).<br>|JobsInWaitingArea| $\leq$ BufferSize.<br>|JobsInServiceArea| $\leq$ NumberOfServers.<br>ServerAccessCredits > 0, JobsInWaitingArea $\neq \emptyset$<br>$\Rightarrow$ |JobsInServiceArea| = NumberOfServers.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td>Name</td><td colspan="3">The name, default "FCFS_NumberOfServers[BufferSize]".</td></tr>
<tr><td colspan="4" align="center">Equivalences</td></tr>
<tr><td colspan="4">FCFS_1[Integer.MAX_VALUE] = FCFS.<br>FCFS_1[B] = FCFS_B[B].<br>FCFS_c[Integer.MAX_VALUE] = FCFS_c.<br>FCFS_0[0] = DROP.<br>FCFS_0[Integer.MAX_VALUE] = SINK.</td></tr>
</table>

### 6.4.5 The IS SimQueue

<table>
<tr><td colspan="4" align="center">IS<br><br>Infinite Servers</td></tr>
<tr><td colspan="4">Serves jobs in arrival order until completion with an infinite number of servers and infinite buffer size.<br>Each server can serve at most one job at a time.<br>Jobs do not prefer one server over another; there are no server affinities.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td>SimQueue</td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td>BufferSize</td><td>F</td><td>Integer.MAX_VALUE</td><td>The buffer size;<br>Integer.MAX_VALUE is treated as $+\infty$.</td></tr>
<tr><td>NumberOfServers</td><td>F</td><td>Integer.MAX_VALUE</td><td>The number of servers;<br>Integer.MAX_VALUE is treated as $+\infty$.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4">StartArmed $=$ true.<br>ServerAccessCredits $> 0 \Rightarrow$ JobsInWaitingArea $= \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td>Name</td><td colspan="3">The name, default "IS".</td></tr>
<tr><td colspan="4" align="center">Equivalences</td></tr>
<tr><td colspan="4">IS $=$ FCFS_Integer.MAX_VALUE.<br>IS $=$ FCFS_Integer.MAX_VALUE[Integer.MAX_VALUE].</td></tr>
</table>

### 6.4.6 The `IS_CST` `SimQueue`

| | | | |
|---|---|---|---|
| **IS_CST**<br>Infinite Servers with Constant Service Time | | | |
| *Serves jobs in arrival order until completion with an infinite number of servers and infinite buffer size.*<br>*Each job is served for a queue-determined fixed service time* (`ServiceTime`).<br>*Each server can serve at most one job at a time.*<br>*Jobs do not prefer one server over another; there are no server affinities.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | F | `Integer.MAX_VALUE` | The buffer size;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `NumberOfServers` | F | `Integer.MAX_VALUE` | The number of servers;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `ServiceTime` | C | `double` $\geq 0$ | The service time for each job. |
| **State Invariants** | | | |
| `StartArmed` $=$ `true`.<br>`ServerAccessCredits` $> 0 \Rightarrow$ `JobsInWaitingArea` $= \emptyset$.<br>`ServiceTime` $= 0 \Rightarrow$ `JobsInServiceArea` $= \emptyset$. | | | |
| **Operational Properties** | | | |
| `Name` | The name, default "IS_CST[`ServiceTime`]". | | |

### 6.4.7 The IC SimQueue

| | | | |
|---|---|---|---|
| **IC**<br>Infinite Capacity | | | |
| *Serves jobs in arrival order until completion*<br>*with an infinite number of infinite-capacity servers and infinite buffer size.*<br>*Each job is served in zero time.*<br>*Each server can serve at most one job at a time.*<br>*Jobs do not prefer one server over another; there are no server affinities.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | F | `Integer.MAX_VALUE` | The buffer size;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `NumberOfServers` | F | `Integer.MAX_VALUE` | The number of servers;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| **State Invariants** | | | |
| `StartArmed` = `true`.<br>`ServerAccessCredits` $> 0 \Rightarrow$ `Jobs` $= \emptyset$.<br>`JobsInServiceArea` $= \emptyset$. | | | |
| **Operational Properties** | | | |
| `Name` | The name, default "IC". | | |
| **Equivalences** | | | |
| `IC` = `IS_CST[0]`. | | | |

### 6.4.8 The NoBuffer_c SimQueue

| NoBuffer_c | | | |
|---|---|---|---|
| *Serves jobs in arrival order until completion with a given number* (NumberOfServers) *of equal servers and zero buffer size. Each server can serve at most one job at a time. Jobs do not prefer one server over another; there are no server affinities.* | | | |
| **Super** | | | |
| SimQueue | See Section 6.2. | | |
| **Essential Properties** | | | |
| BufferSize | F | 0 | The buffer size; |
| NumberOfServers | C | int $\geq 0$ | The number of servers; Integer.MAX_VALUE is treated as $+\infty$. |
| **State Invariants** | | | |
| StartArmed = (\|JobsInServiceArea\| < NumberOfServers). JobsInWaitingArea= $\emptyset$. \|JobsInServiceArea\| $\leq$ NumberOfServers. | | | |
| **Operational Properties** | | | |
| Name | The name, default "NoBuffer_NumberOfServers". | | |
| **Equivalences** | | | |
| NoBuffer_c = FCFS_c[0]. NoBuffer_1 = FCFS[0]. NoBuffer_0 = DROP. | | | |

### 6.4.9 The `LCFS` `SimQueue`

<table>
<tr><td colspan="4" align="center">LCFS<br><br>Last-Come First-Served</td></tr>
<tr><td colspan="4">Serves jobs in reverse arrival order until completion with a single server<br>and infinite buffer size.<br>The server can serve at most one job at a time.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>BufferSize</code></td><td>F</td><td><code>Integer.MAX_VALUE</code></td><td>The buffer size;<br><code>Integer.MAX_VALUE</code> is treated as $+\infty$.</td></tr>
<tr><td><code>NumberOfServers</code></td><td>F</td><td>1</td><td>The number of servers.</td></tr>
<tr><td colspan="4" align="center">State Properties</td></tr>
<tr><td><code>JobsInWaitingArea</code></td><td>EE</td><td><code>Set<J></code></td><td>Jobs waiting<br>in reverse-arrival order.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4"><code>StartArmed</code> $=$ (<code>JobsInServiceArea</code> $= \emptyset$).<br>$|$<code>JobsInServiceArea</code>$| \leq 1$.<br><code>ServerAccessCredits</code> $> 0$, <code>JobsInWaitingArea</code> $\neq \emptyset \Rightarrow$ <code>JobsInServiceArea</code> $\neq \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td><code>Name</code></td><td colspan="3">The name, default ”LCFS”.</td></tr>
</table>

### 6.4.10 The `SJF` `SimQueue`

| | | | |
|---|---|---|---|
| **SJF**<br><br>Shortest-Job First | | | |
| *Serves jobs in increasing service-time order until completion*<br>*with a single server and infinite buffer size.*<br>*The server can serve at most one job at a time.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | F | `Integer.MAX_VALUE` | The buffer size;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `NumberOfServers` | F | 1 | The number of servers. |
| **State Properties** | | | |
| `JobsInWaitingArea` | EE | `Set<J>` | Jobs waiting<br>in *increasing service-time* order. |
| **State Invariants** | | | |
| `StartArmed` $= ($`JobsInServiceArea` $= \emptyset)$.<br>$\vert$`JobsInServiceArea`$\vert \leq 1$.<br>`ServerAccessCredits` $> 0$, `JobsInWaitingArea` $\neq \emptyset \Rightarrow$ `JobsInServiceArea` $\neq \emptyset$. | | | |
| **Operational Properties** | | | |
| `Name` | The name, default "SJF". | | |

### 6.4.11 The `LJF` `SimQueue`

<table>
<tr><td colspan="4" align="center">LJF<br><br>Longest-Job First</td></tr>
<tr><td colspan="4">Serves jobs in decreasing service-time order until completion<br>with a single server and infinite buffer size.<br>The server can serve at most one job at a time.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>BufferSize</code></td><td>F</td><td><code>Integer.MAX_VALUE</code></td><td>The buffer size;<br><code>Integer.MAX_VALUE</code> is treated as $+\infty$.</td></tr>
<tr><td><code>NumberOfServers</code></td><td>F</td><td>1</td><td>The number of servers.</td></tr>
<tr><td colspan="4" align="center">State Properties</td></tr>
<tr><td><code>JobsInWaitingArea</code></td><td>EE</td><td><code>Set<J></code></td><td>Jobs waiting<br>in decreasing service-time order.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4"><code>StartArmed</code> $=$ (<code>JobsInServiceArea</code> $= \emptyset$).<br>$|$<code>JobsInServiceArea</code>$| \leq 1$.<br><code>ServerAccessCredits</code> $> 0$, <code>JobsInWaitingArea</code> $\neq \emptyset \Rightarrow$ <code>JobsInServiceArea</code> $\neq \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td><code>Name</code></td><td colspan="3">The name, default ”LJF”.</td></tr>
</table>

### 6.4.12 The `RANDOM` `SimQueue`

<table>
<tr><td colspan="4" align="center">RANDOM</td></tr>
<tr><td colspan="4">Serves jobs in random order until completion<br>with a single server and infinite buffer size.<br>The server can serve at most one job at a time.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>BufferSize</code></td><td>F</td><td><code>Integer.MAX_VALUE</code></td><td>The buffer size;<br><code>Integer.MAX_VALUE</code> is treated as $+\infty$.</td></tr>
<tr><td><code>NumberOfServers</code></td><td>F</td><td>1</td><td>The number of servers.</td></tr>
<tr><td colspan="4" align="center">State Properties</td></tr>
<tr><td><code>JobsInWaitingArea</code></td><td>EE</td><td><code>Set<J></code></td><td>Jobs waiting<br>in random order.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4">$\texttt{StartArmed} = (\texttt{JobsInServiceArea} = \emptyset)$.<br>$|\texttt{JobsInServiceArea}| \leq 1$.<br>$\texttt{ServerAccessCredits} > 0$, $\texttt{JobsInWaitingArea} \neq \emptyset \Rightarrow \texttt{JobsInServiceArea} \neq \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td><code>Name</code></td><td colspan="3">The name, default ”RANDOM”.</td></tr>
</table>

### 6.4.13 The SUR SimQueue

| SUR<br>Serve Until Relieved | | | |
|---|---|---|---|
| *Serves jobs in arrival order until the start of the next job with a single server and infinite buffer size. The server can serve at most one job at a time.* | | | |
| **Super** | | | |
| SimQueue | See Section 6.2. | | |
| **Essential Properties** | | | |
| BufferSize | F | Integer.MAX_VALUE | The buffer size; Integer.MAX_VALUE is treated as $+\infty$. |
| NumberOfServers | F | 1 | The number of servers. |
| **State Invariants** | | | |
| StartArmed = true.<br>$\vert$JobsInServiceArea$\vert$ $\leq 1$.<br>ServerAccessCredits $> 0 \Rightarrow$ JobsInWaitingArea $= \emptyset$. | | | |
| **Operational Properties** | | | |
| Name | The name, default "SUR". | | |

## 6.5 Preemptive Queues

In preemptive queues, the service area consists of a countable number of servers, each capable of serving a single job at a time, yet this service to a job can be *preempted* in favor of another job.

### 6.5.1 Preemption Strategies; the `PreemptionStrategy` Class

At the moment a job is preempted at a server, one of several things can happen, as determined by the *preemption strategy*. In `jqueues`, the preemption strategy is implemented in an `enum` name `PreemptionStrategy`, and it can take the following values:

- `DROP`: The preemped job is dropped.
- `RESUME`: The preempted job is put on hold, after calculating the remaining service time. Future service resumption continues at the point where the previous service was interrupted.
- `RESTART`: The preempted job is put on hold. Future service resumption requires the job to be served from scratch.
- `REDRAW`: The preempted job is put on hold. Future service resumption requires the job to be served from scratch with a new required service time.
- `DEPART`: The preempted job departs, even though it may not have finished its service requirements.
- `CUSTOM`: A different strategy that those mentioned above is applied to preempted jobs. The strategy needs to be specified in the concrete `SimQueue` implementation.

Not all preemptive-queue implementations support all preemption strategies. For instance, the `REDRAW` strategy is not supported by any implementation, because it conflicts with the contract the the service time of a `SimJob` has to remain constant during a (single) `SimQueue` visit.

The preemption strategy is a parameter that can be passed upon construction by all implemented preemptive queues in `jqueues`. Its default value is `RESUME`. If you pass an unsupported preemption strategy, the queue will throw an `UnsupportedOperationException`.

### 6.5.2 The P_LCFS SimQueue

| | | | |
|---|---|---|---|
| **P_LCFS**<br>Preemptive Last-Come First-Served | | | |
| *Serves jobs in reverse arrival order until completion or preemption with a single server and infinite buffer size.*<br>*A job in service is preempted upon the start of a new job.*<br>*The server can execute at most one job at a time, but an arbitrary number of jobs may be in service.* | | | |
| **Super** | | | |
| SimQueue | See Section 6.2. | | |
| **Essential Properties** | | | |
| BufferSize | F | Integer.MAX_VALUE | The buffer size;<br>Integer.MAX_VALUE is treated as $+\infty$. |
| NumberOfServers | F | 1 | The number of servers. |
| PreemptionStrategy | C | PreemptionStrategy | The preemption strategy. |
| **State Properties** | | | |
| JobsInWaitingArea | EE | Set<J> | Jobs waiting in *reverse-arrival* order. |
| **Operations** | | | |
| Preempt | I | double, J | Job preemption at time. |
| StartServiceChunk | I | double, J | Start service chunk for job at time. |
| **State Invariants** | | | |
| StartArmed $=$ true.<br>ServerAccessCredits $> 0 \Rightarrow$ JobsInWaitingArea $= \emptyset$. | | | |
| **Operational Properties** | | | |
| Name | The name, default "P_LCFS[PreemptionStrategy]". | | |

### 6.5.3 The SRTF SimQueue

| | | | |
|---|---|---|---|
| **SRTF**<br>Shortest Remaining-Time First | | | |
| *Serves jobs in reverse order of remaining service time until completion or preemption with a single server and infinite buffer size.*<br>*A job in service is preempted upon the start of a new job with a strictly smaller remaining service time.*<br>*Jobs are started in reverse required service-time order at the earliest opportunity (i.e., independent of the state of the service area).*<br>*The server can execute at most one job at a time, but an arbitrary number of jobs may be in service.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | F | `Integer.MAX_VALUE` | The buffer size;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `NumberOfServers` | F | 1 | The number of servers. |
| `PreemptionStrategy` | C | `PreemptionStrategy` | The preemption strategy. |
| **State Properties** | | | |
| `JobsInWaitingArea` | EE | `Set<J>` | Jobs waiting in (1) *service-time* and (2) *arrival* order. |
| **Operations** | | | |
| `Preempt` | I | `double, J` | Job preemption at time. |
| `StartServiceChunk` | I | `double, J` | Start service chunk for job at time. |
| **State Invariants** | | | |
| `StartArmed` $=$ `true`.<br>`ServerAccessCredits` $> 0 \Rightarrow$ `JobsInWaitingArea` $= \emptyset$. | | | |
| **Operational Properties** | | | |
| `Name` | The name, default "SRTF[`PreemptionStrategy`]". | | |

## 6.6 Processor-Sharing Queues

In Processor-Sharing (PS) queues, the service area consists of a countable number of servers, each capable of serving multiple jobs at a time, sharing its capacity among those jobs.

### 6.6.1 The `PS` `SimQueue`

<table>
<tr><td colspan="4" align="center">PS<br><br>Processor Sharing</td></tr>
<tr><td colspan="4">Serves all started jobs simultaneously at equal rates until completion<br>with a single server and infinite buffer size.<br>Jobs are started in arrival order at the earliest opportunity.<br>The server can execute an arbitrary number of jobs simultaneously.</td></tr>
<tr><td colspan="4" align="center">Super</td></tr>
<tr><td><code>SimQueue</code></td><td colspan="3">See Section 6.2.</td></tr>
<tr><td colspan="4" align="center">Essential Properties</td></tr>
<tr><td><code>BufferSize</code></td><td>F</td><td><code>Integer.MAX_VALUE</code></td><td>The buffer size;<br><code>Integer.MAX_VALUE</code> is treated as $+\infty$.</td></tr>
<tr><td><code>NumberOfServers</code></td><td>F</td><td>1</td><td>The number of servers.</td></tr>
<tr><td colspan="4" align="center">State Invariants</td></tr>
<tr><td colspan="4"><code>StartArmed</code> = <code>true</code>.<br><code>ServerAccessCredits</code> $> 0 \Rightarrow$ <code>JobsInWaitingArea</code> $= \emptyset$.</td></tr>
<tr><td colspan="4" align="center">Operational Properties</td></tr>
<tr><td><code>Name</code></td><td colspan="3">The name, default "PS".</td></tr>
</table>

### 6.6.2 The `CUPS` `SimQueue`

| CUPS<br>Catch-Up Processor-Sharing | | | |
|---|---|---|---|
| *Serves all started jobs with least obtained service times simultaneously at equal rates until (1) completion, or (2) another job starts or (3) the jobs in execution "catch-up" in terms of obtained service time with another set of jobs in service, with a single server and infinite buffer size.*<br>*Jobs are started in arrival order at the earliest opportunity.*<br>*The server can execute an arbitrary number of jobs simultaneously.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | F | `Integer.MAX_VALUE` | The buffer size;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `NumberOfServers` | F | 1 | The number of servers. |
| **Operations** | | | |
| `CatchUp` | I | `double` | Catch-up at time. |
| **State Invariants** | | | |
| `StartArmed` $=$ `true`.<br>`ServerAccessCredits` $> 0 \Rightarrow$ `JobsInWaitingArea` $= \emptyset$. | | | |
| **Internal Schedulable Events** | | | |
| `CatchUpEvent` | A "catch-up" event. | | |
| **Operational Properties** | | | |
| `Name` | The name, default "CUPS". | | |

### 6.6.3 The `SocPS` `SimQueue`

| SocPS<br><br>Social Processor-Sharing | | | |
|---|---|---|---|
| *Serves all started jobs simultaneously such that their departures times are equal,*<br>*distributing its service capacity among those job to that effect,*<br>*with a single server and infinite buffer size.*<br>*The service rate of a job in the service area*<br>*is linearly proportional to its remaining service time.*<br>*Jobs are started in arrival order at the earliest opportunity.*<br>*The server can execute an arbitrary number of jobs simultaneously.* | | | |
| **Super** | | | |
| `SimQueue` | See Section 6.2. | | |
| **Essential Properties** | | | |
| `BufferSize` | F | `Integer.MAX_VALUE` | The buffer size;<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| `NumberOfServers` | F | 1 | The number of servers. |
| **State Invariants** | | | |
| `StartArmed` = `true`.<br>`ServerAccessCredits` $> 0 \Rightarrow$ `JobsInWaitingArea` $= \emptyset$. | | | |
| **Operational Properties** | | | |
| `Name` | The name, default "SocPS". | | |

# Chapter 7

# Multiclass Queues and Jobs

## 7.1 Introduction

## 7.2 Nonpreemptive Multiclass Queues

### 7.2.1 The `HOL` `SimQueue`

## 7.3 Preemptive Multiclass Queues

### 7.3.1 The `PQ` `SimQueue`

## 7.4 Processor-Sharing Multiclass Queues

### 7.4.1 The `HOL-PS` `SimQueue`

# Chapter 8

# Composite Queues

## 8.1 Introduction

Composite queues consist of zero or more other queues named *subqueues* or *embedded queues*, and a visit of a job to the composite queue is equivalent to a sequence of visits to the subqueues, the order of which is determined by the composite queue. The most prominent example of a composite queue is called a *tandem queue*, consisting of a finite sequence of (distinct) queues that a job must visit in order before leaving the composite queue. If we number the subqueues $1, 2, \ldots, K$, a job visiting the tandem queue, must first complete a visit to queue 1, and upon departure from queue 1, it immediately arrives at queue 2, and so forth, until it departs from queue $K$, at which time it leaves the tandem queue. Using the previous chapters, it is actually rather easy to construct a tandem queue, e.g., a tandem queue consisting of two `P_LCFS` queues, as we have shown in Listing 8.1 below. The output of the program is shown in Listing 8.2.

Listing 8.1: A tandem queue consisting of two P_LCFS queues (using SimJob).

```
private static final class TandemJob
extends DefaultSelfListeningSimJob
{

  final List<SimQueue> queues;

  public TandemJob (final SimEventList eventList,
    final String name,
    final double requestedServiceTime,
    final List<SimQueue> queues)
  {
    super (eventList, name, requestedServiceTime);
    this.queues = queues;
    this.registerSimEntityListener (new StdOutSimEntityListener ());
  }

  @Override
  public void notifyDeparture (final double time,
    final DefaultSelfListeningSimJob job,
```

```
    final SimQueue queue)
  {
    if (this.queues.indexOf (queue) < this.queues.size () - 1)
      getEventList ().schedule (time, (SimEventAction) (SimEvent event) ->
    {
      queues.get (queues.indexOf (queue) + 1).arrive (time, job);
    });
  }

}

public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  final P_LCFS lcfs_1 = new P_LCFS (el, null);
  lcfs_1.setName ("Q1");
  final P_LCFS lcfs_2 = new P_LCFS (el, null);
  lcfs_2.setName ("Q2");
  final List<SimQueue> queueSequence = new ArrayList<> ();
  queueSequence.add (lcfs_1);
  queueSequence.add (lcfs_2);
  for (int j = 1; j <= 5; j++)
    lcfs_1.scheduleJobArrival (j, new TandemJob (el, "J" + j, 10 * j, queueSequence));
  el.run ();
}
```

Listing 8.2: Output of the program in Listing 8.1.

```
StdOutSimEntityListener t=1.0, queue=Q1: ARRIVAL of job J1.
StdOutSimEntityListener t=1.0, queue=Q1: START of job J1.
StdOutSimEntityListener t=2.0, queue=Q1: ARRIVAL of job J2.
StdOutSimEntityListener t=2.0, queue=Q1: START of job J2.
StdOutSimEntityListener t=3.0, queue=Q1: ARRIVAL of job J3.
StdOutSimEntityListener t=3.0, queue=Q1: START of job J3.
StdOutSimEntityListener t=4.0, queue=Q1: ARRIVAL of job J4.
StdOutSimEntityListener t=4.0, queue=Q1: START of job J4.
StdOutSimEntityListener t=5.0, queue=Q1: ARRIVAL of job J5.
StdOutSimEntityListener t=5.0, queue=Q1: START of job J5.
StdOutSimEntityListener t=55.0, queue=Q1: DEPARTURE of job J5.
StdOutSimEntityListener t=55.0, queue=Q2: ARRIVAL of job J5.
StdOutSimEntityListener t=55.0, queue=Q2: START of job J5.
StdOutSimEntityListener t=94.0, queue=Q1: DEPARTURE of job J4.
StdOutSimEntityListener t=94.0, queue=Q2: ARRIVAL of job J4.
StdOutSimEntityListener t=94.0, queue=Q2: START of job J4.
StdOutSimEntityListener t=123.0, queue=Q1: DEPARTURE of job J3.
StdOutSimEntityListener t=123.0, queue=Q2: ARRIVAL of job J3.
StdOutSimEntityListener t=123.0, queue=Q2: START of job J3.
StdOutSimEntityListener t=142.0, queue=Q1: DEPARTURE of job J2.
StdOutSimEntityListener t=142.0, queue=Q2: ARRIVAL of job J2.
StdOutSimEntityListener t=142.0, queue=Q2: START of job J2.
StdOutSimEntityListener t=151.0, queue=Q1: DEPARTURE of job J1.
StdOutSimEntityListener t=151.0, queue=Q2: ARRIVAL of job J1.
StdOutSimEntityListener t=151.0, queue=Q2: START of job J1.
StdOutSimEntityListener t=161.0, queue=Q2: DEPARTURE of job J1.
StdOutSimEntityListener t=172.0, queue=Q2: DEPARTURE of job J2.
StdOutSimEntityListener t=183.0, queue=Q2: DEPARTURE of job J3.
StdOutSimEntityListener t=194.0, queue=Q2: DEPARTURE of job J4.
StdOutSimEntityListener t=205.0, queue=Q2: DEPARTURE of job J5.
```

In the example, 5 jobs are scheduled to arrive at a tandem queueing system consisting of the sequence of two `P_LCFS` queues named `Q1` and `Q2`. The jobs arrive at $t = 1, 2, \ldots, 5$, and require service times at *each* of the `P_LCFS` queues of 10, 20, ..., 50, respectively. Given the high required service times compared to the interarrival times, the jobs leave `Q1`, in reverse order of arrival. However, again because of the relatively high required service times, no job can depart from `Q2` before the next arrival at that queue. As a result, `Q2` again reverses the order of arriving job in its departure process, and jobs depart from `Q2` in order of arrival at the first queue (and at the virtual tandem queue).

Despite the intriguing result that a tandem queue consisting of two `P_LCFS` queues *can* behaves like a `FCFS` queue with modified service-time requirement, and the fact that the tandem queue was so easy to realize, we want to focus at the implementation of the tandem queue, because there are some issues with it. Most important, we explained in the beginning of this section, that a composite queue behaves like the equivalent of its subqueues equipped with some *rouing* of jobs between these subqueues. However, in the example, there is no notion of a composite `SimQueue` at all! The job-arrival process "somehow know" that `Q1` is the first queue at which jobs must arrive, and the jobs *themselves* route themselves to the "next queue". In other words, all the "logic" related to routing jobs in a tandem queue is implemented in the arrival process and in the job implementation, whereas we really want these to be part of the tandem queue, the `SimQueue`. This, for instance, would allow *any* to properly visit the composite `SimQueue` without knowledge on its (internal) structure.

Well, allowing *any* `SimJob` to visit the tandem queue is not that difficult, as is shown in Listing 8.3 below. All we have to do is listen to departure events on `Q1` and then schedule an arrival event at `Q2`.

Listing 8.3: A tandem queue consisting of two P_LCFS queues (using SimQueue).

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  final P_LCFS lcfs_1 = new P_LCFS (el, null);
  lcfs_1.setName ("Q1");
  final P_LCFS lcfs_2 = new P_LCFS (el, null);
  lcfs_2.setName ("Q2");
  lcfs_1.registerSimEntityListener (new DefaultSimEntityListener ()
  {
    @Override
    public void notifyDeparture (final double time, final SimJob job, final SimQueue queue)
    {
      el.schedule (time, (SimEventAction) (SimEvent event) ->
      {
        lcfs_2.arrive (time, job);
      });
    }
  });
  for (int j = 1; j <= 5; j++)
  {
    final SimJob job = new DefaultSimJob (el, "J" + j, 10 * j);
    lcfs_1.scheduleJobArrival (j, job);
    job.registerSimEntityListener (new StdOutSimEntityListener ());
  }
  el.run ();
}
```

The result of the program is identical to that shown in 8.2, and the program itself is surprisingly short (we no longer have to create a dedicated `SimJob` for the tandem queue). This suggests (luckily) that the `jqueues` package has sufficient means to model composite queues manually. However, we are still faced with the problems that there is not really a notion of a composite queue, and we have not tackled the more generic problem of putting `SimQueue`s is tandem:

- What if any of the subqueues *drops* the job?
- How to revoke a jobs from a composite queue?
- How to obtain statistics on the composite queue?

In other words, what we really want is an implementation of a `SimQueue` that behaves as the concatenation of visits to its subqueues. In Listing 8.4 we show this approach using a `BlackTandemSimQueue` described in this chapter, with its output in 8.5.

Listing 8.4: A tandem queue consisting of two P_LCFS queues (using BlackTandem-SimQueue).

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  final P_LCFS lcfs_1 = new P_LCFS (el, null);
  lcfs_1.setName ("Q1");
  final P_LCFS lcfs_2 = new P_LCFS (el, null);
  lcfs_2.setName ("Q2");
  final Set<SimQueue> subQueues = new LinkedHashSet<> ();
  subQueues.add (lcfs_1);
  subQueues.add (lcfs_2);
  final BlackTandemSimQueue compositeQueue = new BlackTandemSimQueue (el, subQueues, null);
  for (int j = 1; j <= 5; j++)
  {
    final SimJob job = new DefaultSimJob (el, "J" + j, 10 * j);
    compositeQueue.scheduleJobArrival (j, job);
    job.registerSimEntityListener (new StdOutSimEntityListener ());
  }
  el.run ();
}
```

Listing 8.5: Output of the program in Listing 8.4.

```
StdOutSimEntityListener t=1.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J1.
StdOutSimEntityListener t=1.0, queue=Tandem[Q1,Q2]: START of job J1.
StdOutSimEntityListener t=2.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J2.
StdOutSimEntityListener t=2.0, queue=Tandem[Q1,Q2]: START of job J2.
StdOutSimEntityListener t=3.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J3.
StdOutSimEntityListener t=3.0, queue=Tandem[Q1,Q2]: START of job J3.
StdOutSimEntityListener t=4.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J4.
StdOutSimEntityListener t=4.0, queue=Tandem[Q1,Q2]: START of job J4.
StdOutSimEntityListener t=5.0, queue=Tandem[Q1,Q2]: ARRIVAL of job J5.
StdOutSimEntityListener t=5.0, queue=Tandem[Q1,Q2]: START of job J5.
StdOutSimEntityListener t=161.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J1.
StdOutSimEntityListener t=172.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J2.
StdOutSimEntityListener t=183.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J3.
StdOutSimEntityListener t=194.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J4.
StdOutSimEntityListener t=205.0, queue=Tandem[Q1,Q2]: DEPARTURE of job J5.
```

The latter example shows exactly what we want: The creation of a new `SimQueue` that behaves exactly like a tandem configuration of two other queues. Admittedly, the code in the example is not particularly smaller than that of the previous examples, and the latter examples can certainly be used as a starting point for the study of tandem (and other composite) queues. However, if, for instance, you want to create a new `SimQueue` type that is the equivalent of other queues that are somehow interconnected, or if you want to study nested composite queues, then constructing a composite queue is a much faster (and more reliable) approach.

The extensive support for composite queues is one of the most distinguishing features of `jqueues`. Next to tandem queues, there is support for various types of *feedback* queues, *encapsulated* queues and *parallel* queues. In the next chapter we describe these composite queues in detail. In the remainder of the present chapter, we introduce some important concepts that apply to *any* composite queue.

## 8.2 The SimQueueComposite Interface

In the `jqueues` implementation, composite queues implement the `SimQueueComposite` interface, which, in turn, inherits from `SimQueue`.

### 8.2.1 The 'Queues' Property

The value of the 'Queues' property of a `SimQueueComposite` is a `Set` holding the subqueues of the composite queue. The property can only be set upon construction of the composite queue; and cannot be changed thereafter, but the subqueues can be accessed through `getQueues ()`. (Note that the `Set` returned should *not* be modified.) Since the order of the subqueues is important to most composite-queue types, the implementation of `getQueues ()` *must* maintain a deterministic ordering of the subqueues.

### 8.2.2 The 'StartModel' Property

## 8.3 Tandem Queues

### 8.3.1 The `Tandem` `SimQueue`

In tandem queues, visiting jobs must visit each subqueue one in a predetermined (and fixed) sequence. The list of (distinct) subqueues is passed upon construction, and cannot be changed afterwards.

| Tandem | |
|---|---|
| Description | Serves jobs by letting their delegate jobs visit the sequence of subqueues $\mathcal{Q}$ exactly once. |
| **State** | |
| QAV | Drops arriving jobs during QAVs. |
| NoWaitArmed | Always `true`. |
| Waiting Area | Infinite waiting area, FIFO discipine. |
| SAC | The remaining number of arriving jobs that can have a delegate job arrive at the tandem queue.<br>If zero, arriving jobs wait in the waiting area.<br>`Integer.MAX_VALUE` is treated as $+\infty$. |
| Service Area | Holds the jobs whose delegate job have arrived *anywhere*. |
| **State Operations** | |
| Set QAV | Ends/starts a QAV. |
| Arrival | The queue accepts arrivals. |
| Drop | Drops arriving jobs during QAVs.<br>Drops a visiting job if its delegate job is dropped on any subqueue. |
| Revocation | Jobs can be revoked while waiting and while being served. |
| Set SAC | Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$.<br>Starts waiting jobs if SAC $> 0$. |
| Start | Starts a job if its delegate job arrives *anywhere*. |
| Departure | Jobs depart if its delegate job departs from the last subqueue. |
| **State Invariants** | |
| $\lvert J(t)\rvert = \lvert J_w(t)\rvert + \sum_{q \in \mathcal{Q}} \lvert J_q(t)\rvert$. | |
| **Properties** | |
| `EventList` | The event list; non-`null`; RO. |
| `Name` | The name, default "LJF"; non-`null`; RW. |
| `Queues` | The set (and list) of subqueues $\mathcal{Q}$; ordered; RO. |

### 8.3.2 The `CTandem2` `SimQueueComposite`

## 8.4 Parallel Queues

### 8.4.1 The `JSQ` `SimQueueComposite`

### 8.4.2 The `JRQ` `SimQueueComposite`

### 8.4.3 The `Pattern` `SimQueueComposite`

### 8.4.4 The `Parallel` `SimQueueComposite`

## 8.5 Feedback Queues

### 8.5.1 The `FB_v` `SimQueueComposite`

### 8.5.2 The `FB_p` `SimQueueComposite`

### 8.5.3 The `FB` `SimQueueComposite`

## 8.6 Jackson Queues

### 8.6.1 The `Jackson` `SimQueueComposite`

## 8.7 Encapsulator Queues

An *encapsulator* queue has exactly one subqueue.

### 8.7.1 The `Enc` `SimQueueComposite`

### 8.7.2 The `EncHS` `SimQueueComposite`

### 8.7.3 The `EncTL` `SimQueueComposite`

### 8.7.4 The `EncJL` `SimQueueComposite`

### 8.7.5 The `EncXM` `SimQueueComposite`

## 8.8 Special Composite Queues

### 8.8.1 The `DCol` `SimQueueComposite`

## 8.9 Generic Composite Queues

### 8.9.1 The `Comp` `SimQueueComposite`

# Chapter 9

# Queue and Job Statistics

## 9.1 Introduction

If you have read this book linearly up to this point, we hope you are wondering by now how to compute statistics like average job sojourn time at a queue, or average server utilization (just to name a few). Actually, this is somewhat on purpose: None of the interfaces and concrete implementations of jobs and queues provide direct support for maintaining and calculating statistics. This has three important reasons:

- It relieves the concrete `SimQueue` and `SimJob` implementations of the tedious and error-prone responsability of maintaining and calculting statistics; their basis functionality in terms of e.g. implementing the proper queueing discipline is complicated enough as it is.
- It forces the implementation of statistics in a generic way, e.g., applicable to *any* queue type.
- It allows for easy replacement of *all* statistics-gathering and maintenance code with that of a professional third-party statistics package like **XXX**.

Needless to say, `jqueues` has full support for the implementation of statistics, and actually, provides a few basic, but highly effective and extensible classes for statistics gathering itself. The starting point for gathering statistics of a `SimEntity` is to register as a suitable `SimEntityListener` to it, and update statistics upon notifications from the entity, inparticular update notifications and visit-related notifications like arrivals, drops and departures. Recall that a `SimEntity` is required to notify registered listeners *just before* it state changes through an update notification, allowing for easy maintenance of "time-average-type" statistics like the average number of jobs present at a queue. In addition (and often requiring a bit more work) the visit-related notifications allow for maintenance of "visit-related" statistics, like the average job sojourn time at a queue.

In the following section, we will first explain how to obtain statistics for a `SimQueue` or `SimJob` manually, following the approach outline above. Subsequently, we will introduce a few statistics-related class that intend to make it easier to obtain such statistics.

### 9.1.1 Example: The Average Number of Jobs at a Queue

In this statistics-related example, we create a `SimQueue` and a workload consisting of arriving `SimJob`s. Our objective is to run the simulation, and calculate the average number of jobs at the queue. To make the example more interesting, we create two queues, a FCFS queue and a Preemptive (Resume) LCFS queue. We equip each with a listener that maintains the number of jobs at the queue over time. As shown in Listing 9.1 below, we use a `DefaultSimEntityListener` as base classs, making it easier to ignore all the notifications we are not interested in. Note that we are silently assuming that the listener is registered at a `SimQueue`, and not at some other `SimEntity` type. In addition, we left out some sanity checking, e.g., on the `time` parameter (should not be in the past).

Listing 9.1: Simple listener for maintaining and calculating the average number of jobs at a SimQueue.

```
private final static class AvgJStatListener
extends DefaultSimEntityListener
{

  public AvgJStatListener (SimQueue queue)
  {
    notifyResetEntity (queue);
    queue.registerSimEntityListener (this);
  }

  private double tStart = Double.NEGATIVE_INFINITY;
  private double tLast = Double.NEGATIVE_INFINITY;
  private double cumJ = 0;

  @Override
  public void notifyResetEntity (SimEntity entity)
  {
    tStart = entity.getEventList ().getTime ();
    tLast = tStart;
    cumJ = 0;
  }

  @Override
  public void notifyUpdate (double time, SimEntity entity)
  {
    cumJ += ((SimQueue) entity).getNumberOfJobs () * (time - tLast);
    tLast = time;
  }

  public final double getStartTime ()
  {
    return this.tStart;
```

```
  }

  public final double getEndTime ()
  {
    return this.tLast;
  }

  public final double calculate ()
  {
    if (tLast > tStart)
    {
      if (! Double.isInfinite (tLast - tStart))
        return cumJ / (tLast - tStart);
      else
        return 0;
    }
    else
      return 0;
  }

}
```

Note that we are only interested in two notifications from the `SimQueue`, *reset* and *update* notifications. Because the number of jobs at a `SimQueue` is a *simple function*, we can easily integrate it over time by considering all points in time at which the function value *can* change, and cumulating the product of the interval since the last update with the current value. By definition, such points in time are the *updates* of a `SimQueue` and the constract of the latter mandates that each update results in a notification at its listeners *before* any changes have been applied to the queue. Needless to say, this requires substantial discipline for implementations of `SimQueue`, but we use it here to our advantage. In the end, all we need to do it divide the cumulated value with the total time interval, taking special care of the possibility that the interval is $+\infty$.

For the start of the interval, we use the reset notification from the queue. At each reset, we take the start time from the current time of the event list (note that we cannot take the time from the `SimQueue` directly), and reset our internal statistics. Note that we call `notifyResetEntity` directly from the constructor as well, because we cannot rely on receiving a reset between construction of the object and starting the simulation. Also note that we may assume that the number of jobs at a reset, as well as upon construction of a `SimQueue` is zero, conform the contract of a `SimQueue`. However, unless the event list or the `SimQueue` attached to it is reset to a finite time value, we must use `Double.NEGATIVE_INFINITY` as time value. This explains the default values used in `AvgJStatListener`.

We list the example program using the `AvgJStatListener` and its output in Listings 9.2 and 9.3 below. Note the very important `el.reset (0)` line in the main program. We will later show the effect of leaving out or forgetting this statement. The correctness of the calculated values in the output can be assessed easily as is shown in the comments of the main program. For `P_LCFS`, the program outputs one of two distinct values for the average, both of which are actually correct, due to an ambiguity in the event scheduling. (Which we, in all honesty, overlooked while constructing the example.) This is also explained in the comments of the program.

Listing 9.2: Example program for calculating the average number of jobs at two queues.

```java
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF " + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL " + j, j));
  }
  final AvgJStatListener avgJStatListener_fcfs = new AvgJStatListener (fcfs);
  final AvgJStatListener avgJStatListener_lcfs = new AvgJStatListener (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("  Start time: " + avgJStatListener_fcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_fcfs.getEndTime () + ".");
  System.out.println ("  Average number of jobs: " + avgJStatListener_fcfs.calculate () + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("  Start time: " + avgJStatListener_lcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_lcfs.getEndTime () + ".");
  System.out.println ("  Average number of jobs: " + avgJStatListener_lcfs.calculate () + ".");
}
```

Listing 9.3: Output from the example program in Listing 9.2.

```
FCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Average number of jobs: 1.2727272727272727.
P_LCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Average number of jobs: 1.9090909090909092 [OR 2.727272727272727, see comments].
```

As mentioned earlier, it is essential to reset the event list to a finite time value before use. Below in Listing 9.4 we show the output of the program if we leave out the `el.reset (0)` statement, essentially starting the simulation and, more importantly, gathering the statistics at $t = -\infty$. The result is zero, because we are taking the average value of a function over an infinite interval, knowing that is is only non-zero over a finite interval. Despite that fact that for this particular case, we seem to get away with the infinite interval, and are able to return a sensible value for the average, this does not hold for the general case. (For instance, suppose we would have scheduled an arrival at $t = -\infty$, the program would still return the incorrect zero value for the average.) As a rule of thumb, the use of statistics involving infinite values should be avoided at all times, again stressing the importance of resetting the event list to a finite value before its use.

Listing 9.4: Output from the example program in Listing 9.2 without resetting the event list to zero (i.e., starting at $t = -\infty$).

```
FCFS:
  Start time: -Infinity.
  End Time  : 11.0.
  Average number of jobs: 0.0.
P_LCFS:
  Start time: -Infinity.
  End Time  : 11.0.
  Average number of jobs: 0.0.
```

### 9.1.2 Example: The Total Sojourn of a Job at Visited Queues

## 9.2 The AbstractSimQueueStat Base Class

The examples in the previous sections explained how to obtain statistics on `SimQueues` and `SimJobs`. The next sections in this chapter introduce a few classes that either directly provide a few of the most basic statistics or allow you to define your own statistics more easily.

The `AbstractSimQueueStat` is the (abstract) base class for all subsequent classes. Its functionality is comparable to `AvgJStatListener` introduced in Section 9.1.1, with a few new functions and increased robustness, and delegating the actual maintenance and calulation of the statistic(s) to only a few abstract methods. To illustrate its use, we show in Listings 9.5 and 9.6 the modified listener and main program, respectively, from the example in Section 9.1.1 now using `AbstractSimQueueStat`.

Listing 9.5: Using AbstractSimQueueStat for a listener for maintaining and calculating the average number of jobs at a SimQueue.

```
private static class AvgJStatListener
extends AbstractSimQueueStat
{

  public AvgJStatListener (SimQueue queue)
  {
    super (queue);
  }

  private double cumJ = 0;

  private double avgJ = 0;

  @Override
  protected void resetStatistics ()
  {
```

```
    cumJ = 0;
    avgJ = 0;
  }

  @Override
  protected void updateStatistics (double time, double dt)
  {
    cumJ += getQueue ().getNumberOfJobs () * dt;
  }

  @Override
  protected void calculateStatistics (double startTime, double endTime)
  {
    if (startTime == endTime)
      return;
    if (! Double.isInfinite (endTime - startTime))
      avgJ = cumJ / (endTime - startTime);
    else
      avgJ = 0;
  }

  public double getAvgJ ()
  {
    calculate ();
    return this.avgJ;
  }

}
```

Listing 9.6: Example program for calculating the average number of jobs at two queues using AbstractSimQueueStat.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF_" + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL_" + j, j));
  }
  final AvgJStatListener avgJStatListener_fcfs = new AvgJStatListener (fcfs);
  final AvgJStatListener avgJStatListener_lcfs = new AvgJStatListener (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("  Start time: " + avgJStatListener_fcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_fcfs.getLastUpdateTime () + ".");
  System.out.println ("  Average number of jobs: " + avgJStatListener_fcfs.getAvgJ () + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("  Start time: " + avgJStatListener_lcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_lcfs.getLastUpdateTime () + ".");
  System.out.println ("  Average number of jobs: " + avgJStatListener_lcfs.getAvgJ () + ".");
}
```

Compared to the manual approach in Section 9.1.1, we highlight the following differences:

- In the listener, we only have to implement what to do upon a reset (`resetStatistics`) and upon an update (`updateStatistics`), and how to calculate the (internally stored) result (`calculateStatistics`); the base class takes care of registering at `SimQueue` and internally storing the times of the last reset (or the time at construction) and of the last update.

- In the concrete listener, we provide a method for access to the internally stored result, but that method automatically invokes the base class' `calculate ()` method first. The base class attempts to avoid unnecessary recalculations of the result in its subclass.

- Users of the concrete subclass have access to the methods `getStartTime` and `getLastUpdateTime` at all times. Upon calculation (without a time argument), the time of the last update determines the upper boundary of the measurement interval.

- Users may also invoke `calculate (double endTime)` which allows the extension of the measurement interval beyond the last update time. This, however, prevents subsequent update *before* the `endTime` provided, unless the object (or the queue, or the underlying event list) is reset. Nonetheless, the feature is handy because in many practical cases, one wants to take the time average over a fixed time interval, instead of a time interval of which the upper boundary is determined by the last update. Unfortunately, one cannot chose and `endTime` smaller than the last update time (at the expense of an exception thrown).

- Users can change the `SimQueue` from which the statistics are obtained through `setQueue (Q)`. Setting the queue will immediately reset the statistics object. As expected, the base class `AbstractSimQueueStat` takes care of registering at the new queue (if any) after unregistering at the old queue. The only sensible use for that feature that we can think of is delayed initialization.

## 9.3 The SimpleSimQueueStat Class

The `SimpleSimQueueStat` class implements `AbstractSimQueueStat` described in Section 9.2 for some important statistics on a `SimQueue`:

- The average number of jobs;

- The average number of jobs in the service area;
- The maximum and minimum number of jobs;
- The maximum and minimum number of jobs in the service area.

Attaching a `SimpleSimQueueStat` to a `SimQueue` is a very conventient way of obtaining a first impression of the performance of the queue; we have used it a lot ourselves during testing. Below in Listing 9.7 we illustrate its use for the example in Section 9.1.1; the output (still ambiguous!) is shown in Listing 9.8. Note that the number of updates is also available from `SimpleSimQueueStat`, which is very handy for detailed update-related debugging on `SimQueue` implementations.

Listing 9.7: Example program for obtaing basic statistics using SimpleSimQueueStat.

```java
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF " + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL " + j, j));
  }
  final SimpleSimQueueStat avgJStatListener_fcfs = new SimpleSimQueueStat (fcfs);
  final SimpleSimQueueStat avgJStatListener_lcfs = new SimpleSimQueueStat (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("  Start time: " + avgJStatListener_fcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_fcfs.getLastUpdateTime () + ".");
  System.out.println ("  Number of updates                     : "
    + avgJStatListener_fcfs.getNumberOfUpdates () + ".");
  System.out.println ("  Average number of jobs                : "
    + avgJStatListener_fcfs.getAvgNrOfJobs () + ".");
  System.out.println ("  Average number of jobs in service area: "
    + avgJStatListener_fcfs.getAvgNrOfJobsInServiceArea () + ".");
  System.out.println ("  Maximum number of jobs                : "
    + avgJStatListener_fcfs.getMaxNrOfJobs () + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("  Start time: " + avgJStatListener_lcfs.getStartTime () + ".");
  System.out.println ("  End Time  : " + avgJStatListener_lcfs.getLastUpdateTime () + ".");
  System.out.println ("  Number of updates                     : "
    + avgJStatListener_fcfs.getNumberOfUpdates () + ".");
  System.out.println ("  Average number of jobs                : "
    + avgJStatListener_lcfs.getAvgNrOfJobs () + ".");
  System.out.println ("  Average number of jobs in service area: "
    + avgJStatListener_lcfs.getAvgNrOfJobsInServiceArea ()+ ".");
  System.out.println ("  Maximum number of jobs                : "
    + avgJStatListener_lcfs.getMaxNrOfJobs () + ".");
}
```

Listing 9.8: Output from the example program in Listing 9.7.

```
FCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Number of updates                    : 6.
  Average number of jobs               : 1.2727272727272727.
  Average number of jobs in service area: 0.9090909090909091.
  Maximum number of jobs               : 2.0.
P_LCFS:
  Start time: 0.0.
  End Time  : 11.0.
  Number of updates                    : 6.
  Average number of jobs               : 1.9090909090909092 [OR 2.727272727272727, see comments].
  Average number of jobs in service area: 1.9090909090909092 [OR 2.727272727272727, see comments].
  Maximum number of jobs               : 3.0 [OR 4.0, see comments].
```

Note the remarkable difference in the average number of jobs in the service area between `FCFS` and `P_LCFS`. In the former, jobs waiting are only admitted to the service area if the previously arrived job has departed, whereas in the latter, jobs are admitted immediately to the service area upon arrival (because their service starts immediately), and preempted jobs are *not* transferred back into the waiting area. This illustrates the (surprising) distinction between the number of jobs *in service* (a concept not directly supported by `SimQueue`) and the number of jobs *in the service area* (in which service is allowed, but not guaranteed).

## 9.4 The AutoSimQueueStat Class

### 9.4.1 Introduction

The `SimpleSimQueueStat` class described in Section 9.3 provides a convenient way of obtaining some important statistics related to the number of jobs at a queue or at its service area. Although it is fairly easy to extend `SimpleSimQueueStat`, or even `AbstractSimQueueStat` for that matter, for other time-dependent performance measures, we quickly realize that the resulting code would only differ in *which* statistic we take from the queue upon updates. The `AutoSimQueueStat` class introduced in this section honors this observation, and relies on a so-called `SimQueueProbe` to obtain the momentary value of a specific performance measure at a `SimQueue`. It then automatically maintains the average, maximum and minimum values for all registered probes.

### 9.4.2 The SimQueueProbe Interface

As described in the previous section, the `SimQueueProbe` interface features obtaining the momentary value of a specific performance measure at a queue (the queue is actually a parameter, so probes can be reused). Its interface definition is shown in

Listing 9.9. Note that only double values (or values that can be cast to double) are supported.

Listing 9.9: The SimQueueProbe interface.

```
public interface SimQueueProbe<Q extends SimQueue>
{

  public double get (Q queue);

}
```

### 9.4.3 The AutoSimQueueStatEntry Class

The `AutoSimQueueStatEntry` class connects a `SimQueueProbe` with a name, and adds statistics maintenance for the values the probe provides. Unfortunately, unlike a `SimQueueProbe`, you cannot reuse a `AutoSimQueueStatEntry` among different queues.

### 9.4.4 Example

Below in Listing 9.10 we illustrate the use of `AutoSimQueueStat` for the example in Section 9.1.1; the output (still ambiguous!) is shown in Listing 9.11. Note that two probes are created, one for the number of jobs at a queue, and one for the number of jobs in the service area at a queue. For each queue, `FCFS` and `P_LCFS`, an array of entries is created, holding unique entries for the statistics required. Finally, note that `AutoSimQueueStat` features a method `report ()` (optionally, with an integer indentation parameter) for quickly reporting the calculated statistics for all registered probes.

Listing 9.10: Example program for obtaing statistics using AutoSimQueueStat.

```
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
```

```
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF_" + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL_" + j, j));
  }
  final SimQueueProbe probeJ =
    (SimQueueProbe) (SimQueue queue) -> queue.getNumberOfJobs ();
  final SimQueueProbe probeJX =
    (SimQueueProbe) (SimQueue queue) -> queue.getNumberOfJobsInServiceArea ();
  final List<AutoSimQueueStatEntry> entries_fcfs = new ArrayList<> ();
  final List<AutoSimQueueStatEntry> entries_lcfs = new ArrayList<> ();
  entries_fcfs.add (new AutoSimQueueStatEntry ("J", probeJ));
  entries_fcfs.add (new AutoSimQueueStatEntry ("JX", probeJX));
  entries_lcfs.add (new AutoSimQueueStatEntry ("J", probeJ));
  entries_lcfs.add (new AutoSimQueueStatEntry ("JX", probeJX));
  final AutoSimQueueStat stat_fcfs = new AutoSimQueueStat (fcfs, entries_fcfs);
  final AutoSimQueueStat stat_lcfs = new AutoSimQueueStat (lcfs, entries_lcfs);
  el.run ();
  System.out.println ("FCFS:");
  stat_fcfs.report (2);
  System.out.println ("P_LCFS:");
  stat_lcfs.report (2);
}
```

Listing 9.11: Output from the example program in Listing 9.7.

```
FCFS:
  Average J: 1.2727272727272727.
  Minimum J: 0.0.
  Maximum J: 2.0.
  Average JX: 0.9090909090909091.
  Minimum JX: 0.0.
  Maximum JX: 1.0.
P_LCFS:
  Average J: 1.9090909090909092 [OR 2.727272727272727, see comments].
  Minimum J: 0.0.
  Maximum J: 3.0 [OR 4.0, see comments].
  Average JX: 1.9090909090909092 [OR 2.727272727272727, see comments].
  Minimum JX: 0.0.
  Maximum JX: 3.0 [OR 4.0, see comments].
```

## 9.5 The SimpleSimQueueVisitsStat Class

The `SimpleSimQueueVisitsStat` maintains and calculates some important visits-related statistics on the queue it is registered at:

- The number of arrivals, of jobs that started and the number of departures;
- The number of dropped and (successfully) revoked jobs;
- The average waiting time (averaged over jobs that started);
- The average sojourn time (averaged over jobs that departed);
- The maximum and minimum waiting time (averaged over jobs that started);
- The maximum and minimum sojourn time (averaged over jobs that departed).

In Listing 9.12 below we provide an example of its use, again using the scheduling example from YYY; its corresponding output is shown in Listing 9.13.

Listing 9.12: Example program for obtaing basic statistics using SimpleSimQueueVisitsStat.

```java
public static void main (final String[] args)
{
  final SimEventList el = new DefaultSimEventList ();
  el.reset (0);
  // Schedule 5 jobs, 0 through 4 at t = 0, 1, 2, 3, 4
  // and service times 0, 1, 2, 3, 4, respectively.
  //
  // Departure times with FCFS: 0, 2, 4, 7, 11.
  // cumJ = 0 + 1 + 1 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 = 14.
  // avgJ = 14 / 11 = 1.2727.
  // avgWaitJ = 0 + 0 + 0 + 1 + 3 / 5 = 0.8.
  // avgSojJ  = 0 + 1 + 2 + 4 + 7 / 5 = 14 / 5 = 2.8.
  //
  // WITH P_LCFS: AMBIGUITY!
  //
  // Departure times with P_LCFS [2 arrives after departure of 1]: 0, 2, 8, 10, 11.
  // cumJ = 0 + 1 + 1 + 2 + 3 + 3 + 3 + 3 + 2 + 2 + 1 = 21.
  // avgJ = 21 / 11 = 1.9090.
  // avgWaitJ = 0.
  // avgSojJ  = 0 + 1 + 9 + 7 + 4 / 5 = 21 / 5 = 4.2.
  //
  // Departure times with P_LCFS [2 arrives before departure of 1]: 0, 8, 10, 11, 11.
  // cumJ = 0 + 1 + 2 + 3 + 4 + 4 + 4 + 4 + 3 + 3 + 2 = 30.
  // avgJ = 30 / 11 = 2.7272.
  // avgWaitJ = 0.
  // avgSojJ  = 0 + 10 + 9 + 7 + 4 / 5 = 30 / 5 = 6.0.
  //
  final FCFS fcfs = new FCFS (el);
  final P_LCFS lcfs = new P_LCFS (el, null);
  for (int j = 0; j <= 4; j++)
  {
    fcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobF " + j, j));
    lcfs.scheduleJobArrival (j, new DefaultSimJob (el, "JobL " + j, j));
  }
  final SimpleSimQueueVisitsStat visitsStatListener_fcfs = new SimpleSimQueueVisitsStat (fcfs);
  final SimpleSimQueueVisitsStat visitsStatListener_lcfs = new SimpleSimQueueVisitsStat (lcfs);
  el.run ();
  System.out.println ("FCFS:");
  System.out.println ("  Start Time            : "
    + visitsStatListener_fcfs.getStartTime ()          + ".");
  System.out.println ("  End Time              : "
    + visitsStatListener_fcfs.getLastUpdateTime ()     + ".");
  System.out.println ("  Number of Arrivals    : "
    + visitsStatListener_fcfs.getNumberOfArrivals ()   + ".");
  System.out.println ("  Number of Started Jobs: "
    + visitsStatListener_fcfs.getNumberOfStartedJobs () + ".");
  System.out.println ("  Number of Departures  : "
    + visitsStatListener_fcfs.getNumberOfDepartures () + ".");
  System.out.println ("  Minimum Waiting Time  : "
    + visitsStatListener_fcfs.getMinWaitingTime ()     + ".");
  System.out.println ("  Maximum Waiting Time  : "
    + visitsStatListener_fcfs.getMaxWaitingTime ()     + ".");
  System.out.println ("  Average Waiting Time  : "
    + visitsStatListener_fcfs.getAvgWaitingTime ()     + ".");
  System.out.println ("  Minimum Sojourn Time  : "
    + visitsStatListener_fcfs.getMinSojournTime ()     + ".");
  System.out.println ("  Maximum Sojourn Time  : "
    + visitsStatListener_fcfs.getMaxSojournTime ()     + ".");
  System.out.println ("  Average Sojourn Time  : "
    + visitsStatListener_fcfs.getAvgSojournTime ()     + ".");
  System.out.println ("P_LCFS:");
  System.out.println ("  Start Time            : "
    + visitsStatListener_lcfs.getStartTime ()          + ".");
  System.out.println ("  End Time              : "
    + visitsStatListener_lcfs.getLastUpdateTime ()     + ".");
  System.out.println ("  Number of Arrivals    : "
    + visitsStatListener_lcfs.getNumberOfArrivals ()   + ".");
  System.out.println ("  Number of Started Jobs: "
    + visitsStatListener_lcfs.getNumberOfStartedJobs () + ".");
  System.out.println ("  Number of Departures  : "
```

```
    + visitsStatListener_lcfs.getNumberOfDepartures ()  + ".");
  System.out.println ("  Minimum Waiting Time  : "
    + visitsStatListener_lcfs.getMinWaitingTime ()      + ".");
  System.out.println ("  Maximum Waiting Time  : "
    + visitsStatListener_lcfs.getMaxWaitingTime ()      + ".");
  System.out.println ("  Average Waiting Time  : "
    + visitsStatListener_lcfs.getAvgWaitingTime ()      + ".");
  System.out.println ("  Minimum Sojourn Time  : "
    + visitsStatListener_lcfs.getMinSojournTime ()      + ".");
  System.out.println ("  Maximum Sojourn Time  : "
    + visitsStatListener_lcfs.getMaxSojournTime ()      + ".");
  System.out.println ("  Average Sojourn Time  : "
    + visitsStatListener_lcfs.getAvgSojournTime ()      + ".");
}
```

Listing 9.13: Output from the example program in Listing 9.12.

```
FCFS:
  Start Time             : 0.0.
  End Time               : 11.0.
  Number of Arrivals     : 5.
  Number of Started Jobs: 5.
  Number of Departures   : 5.
  Minimum Waiting Time   : 0.0.
  Maximum Waiting Time   : 3.0.
  Average Waiting Time   : 0.8.
  Minimum Sojourn Time   : 0.0.
  Maximum Sojourn Time   : 7.0.
  Average Sojourn Time   : 2.8.
P_LCFS:
  Start Time             : 0.0.
  End Time               : 11.0.
  Number of Arrivals     : 5.
  Number of Started Jobs: 5.
  Number of Departures   : 5.
  Minimum Waiting Time   : 0.0.
  Maximum Waiting Time   : 0.0.
  Average Waiting Time   : 0.0.
  Minimum Sojourn Time   : 0.0.
  Maximum Sojourn Time   : 9.0. [OR 10.0, see comments].
  Average Sojourn Time   : 4.2. [OR 6.0, see comments].
```

## 9.6 Conclusions

# Chapter 10

# Visualization

# Chapter 11

# Building Custom Queues and Jobs

## 11.1 Introduction

## 11.2 The `AbstractSimEntity` Class

## 11.3 The `AbstractSimQueue` Class

## 11.4 The `AbstractSimQueueComposite` Class

## 11.5 The `AbstractSimJob` and `DefaultSimJob` Classes

## 11.6 Building Custom Listeners

# Chapter 12

# Use Cases

## 12.1 Introduction

## 12.2 Busy/Idle Notifications and Statistics

## 12.3 A Weird Statistic: The Fraction of Stayers

## 12.4 The M/M/1/FCFS Queue

## 12.5 Customers with Varying Impatience

## 12.6 Customers in a Party

## 12.7 Waiters with Varying Enthousiasm

## 12.8 A (Dutch) Restaurant and Its Performance

# Chapter 13

# Testing

## 13.1 Introduction

## 13.2 Test Infrastructure

## 13.3 Generating Workloads

### 13.3.1 Job Factories

### 13.3.2 Queue Workloads

### 13.3.3 Queue Events and Schedules

### 13.3.4 Load Factories

## 13.4 Standardized Workload Patterns

## 13.5 Queue Predictors

## 13.6 Confronting Queue Workloads, Predictors, and Queues

## 13.7 Building a Predictor

## 13.8 Building a Workload

# Chapter 14

# Things to Come

# Chapter 15

# Conclusions