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 and Graham Birtwistle.

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

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

Note that `E` refers to the so-called *generic-type argument* of `SimEventList`. The prototype is `SimEventList<E extends SimEvent>`. The use of generic types is explained in some more details in the "Advanced Topics" section, but for now `E` can be simply read as a `SimEvent`.

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

You may wonder how to *remove* events and actions from the event list. Well, since `SimEventList` implements the `Set` interface for `SimEvent` members, removing an event `e` from an event list `el` is as simple as `el.remove (e)`. Currently, there is no support to remove an action from an event list. Because actions can be reused, it would require iterating over all scheduled events, and remove all events with the given action. It is not hard to implement at all, we just did not do it[3]:

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

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

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

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

4.6 Simultaneous Events

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 simplicity 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" `SimEvent`s. For instance, the `jqueues` library will not work with event-lists not supporting "plain" `SimEvent`s.

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

```
@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.c., `TreeSet`), e.g., in case of ROEL:

```
@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 The `SimEntity` Interface

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 The `SimQueue` Interface

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 `NoWaitArmed` Property of a `SimQueue`

Copying a `SimQueue`

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 getQueue ()	The queue currently visiting.
void setQueue (Q)	Sets the queue currently visiting.
double 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 `SimQueue`s, 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

Chapter 6

Fundamental Queues

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

6.1 Introduction

6.2 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.

6.2.1 The `DROP` `SimQueue`

DROP		
Description	Drops jobs immediately upon arrival.	
State		
QAV	`false`/`true`	QAVs have no effect.
NoWaitArmed	`true`	Since jobs are immediately dropped.
Waiting Area	Absent	There is no waiting area.
SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Service Area	Absent	There is no service area.
State Operations		
Set QAV	`false`/`true`	Has no effect.
Arrival	Yes	The queue accepts arrivals. (But jobs are dropped immediately.)
Drop	Yes	All jobs are dropped upon arrival.
Revocation	No	Jobs cannot be revoked since there are never jobs present.
Set SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Start	No	Jobs cannot start.
Departure	No	Jobs cannot depart.
State Invariants		
The set of jobs present is empty.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "DROP"; non-`null`; RW.	

6.2.2 The `SINK` `SimQueue`

SINK		
Description	Lets jobs wait indefinitely.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	`false`	Since non-dropped jobs must wait.
Waiting Area	Present	Infinite waiting area, non-ordered.
SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Service Area	Absent	There is no service area.
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked from the waiting area.
Set SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Start	No	Jobs cannot start.
Departure	No	Jobs cannot depart.
State Invariants		
The set of jobs present in the service area is empty. All jobs present are in the waiting area.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "SINK"; non-`null`; RW.	

6.2.3 The `ZERO` `SimQueue`

ZERO		
Description	Lets jobs depart immediately upon arrival.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	`true`	Since non-dropped jobs depart immediately.
Waiting Area	Absent	There is no waiting area.
SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Service Area	Absent	There is no service area.
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals. (But jobs depart immediately.)
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	No	Jobs cannot be revoked since there are never jobs present.
Set SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Start	No	Jobs cannot start.
Departure	Yes	Non-dropped jobs depart upon arrival.
State Invariants		
The set of jobs present is empty.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "ZERO"; non-`null`; RW.	

6.2.4 The `DELAY` `SimQueue`

DELAY		
Description	Lets jobs depart after a fixed wait time.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	WaitTime == 0	Since non-dropped jobs depart immediately when the wait time is zero.
Waiting Area	Present	Infinite waiting area, non-ordered.
SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Service Area	Absent	There is no service area.
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals. (Immediate departure if WaitTime == 0.)
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting.
Set SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Start	No	Jobs cannot start.
Departure	Yes	Non-dropped/revoked jobs depart WaitTime after arrival.
State Invariants		
The set of jobs present in the service area is empty. All jobs present are in the waiting area.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "DELAY[`WaitTime`]"; non-`null`; RW.	
`WaitTime`	The wait time, zero or positive; RO.	

6.2.5 The `GATE` `SimQueue`

GATE		
Description	Depending on the state of its "gate", let jobs depart immediately or puts them into the waiting area.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	GPC > 0	Since non-dropped jobs depart immediately upon arrival provided that there are gate-passage credits (GPCs).
Waiting Area	Present	Infinite waiting area, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Service Area	Absent	There is no service area.
GPC	$\{0, 1, 2, \ldots\}$	Number of jobs to depart immediately upon arrival.
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting.
Set SAC	$\{0, 1, 2, \ldots\}$	SACs have no effect.
Start	No	Jobs cannot start.
Departure	Yes	Non-dropped/revoked jobs depart upon arrival or if (sufficient) GPCs become available.
Set GPC	$\{0, 1, 2, \ldots\}$	Overwrites the remaining number of GPCs. (Letting jobs depart if waiting and GPC > 0). `Integer.MAX_VALUE` is treated as $+\infty$.
State Invariants		
The set of jobs present in the service area is empty. All jobs present are in the waiting area. If GPC > 0, the set of jobs present is empty.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "GATE"; non-`null`; RW.	

6.3 Nonpreemptive Queues

6.3.1 The `FCFS` `SimQueue`

FCFS		
Description	Serves jobs in arrival order with a single server.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	No Jobs	Since non-dropped arriving jobs are taken into service immediately if no other jobs are present.
Waiting Area	Present	Infinite waiting area, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most one job is present in the service area. If SAC > 0, the server cannot be idle in the presence of waiting jobs.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "FCFS"; non-`null`; RW.	
`NumberOfServers`	The number of servers; always 1; RO.	

6.3.2 The FCFS_B SimQueue

FCFS_B		
Description	Serves jobs in arrival order with a single server but has limited capacity for waing jobs.	
State		
QAV	false/true	Drops arriving jobs during QAVs.
NoWaitArmed	No Jobs	Since arriving jobs are dropped or taken into service immediately if no other jobs are present.
Waiting Area	Present	Finite waiting area of size BufferSize, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. Integer.MAX_VALUE is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any).
State Operations		
Set QAV	false/true	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs and while there are BufferSize waiting jobs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; Integer.MAX_VALUE $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most one job is present in the service area. At most BufferSize jobs are present in the waiting area. If SAC > 0, the server cannot be idle in the presence of waiting jobs.		
Properties		
EventList	The event list; non-null; RO.	
Name	The name, default "FCFS_B[BufferSize]"; non-null; RW.	
NumberOfServers	The number of servers; always 1; RO.	
BufferSize	The buffer size; non-negative; RO. Integer.MAX_VALUE is treated as $+\infty$.	

6.3.3 The `FCFS_c` `SimQueue`

FCFS_c		
Description	Serves jobs in arrival order with multiple (c) servers.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	`NumberOfJobs` $< c$	Since non-dropped arriving jobs are taken into service immediately if at least one server is idle.
Waiting Area	Present	Infinite waiting area, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the jobs (upto c) currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job(s) if SAC > 0 and it least one server is idle.
Start	Yes	Starts jobs if there are SACs and at least one server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most c jobs are present in the service area. If SAC > 0, a server cannot be idle in the presence of waiting jobs.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "FCFS_`NumberOfServers`"; non-`null`; RW.	
`NumberOfServers`	The number c of servers; non-negative; RO. `Integer.MAX_VALUE` is treated as $+\infty$.	

6.3.4 The `IS` `SimQueue`

IS		
Description	Serves jobs in arrival order with an infinite number of servers.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	`true`	Since non-dropped arriving jobs are taken into service immediately.
Waiting Area	Present	Infinite waiting area, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the jobs currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs in arrival order while SAC > 0.
Start	Yes	Starts jobs if there are SACs.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
If SAC > 0, there are no jobs in the waiting area.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "IS"; non-`null`; RW.	

6.3.5 The `IS_CST` SimQueue

IS_CST		
Description	Serves jobs in arrival order with an infinite number of servers. Each job is served for a queue-determined fixed service time.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	`true`	Since non-dropped arriving jobs are taken into service immediately.
Waiting Area	Present	Infinite waiting area, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the jobs currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs in arrival order while SAC > 0.
Start	Yes	Starts jobs if there are SACs.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
If SAC > 0, there are no jobs in the waiting area. If `ServiceTime` $== 0$, there are no jobs in the service area.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "IS_CST[`ServiceTime`]"; non-`null`; RW.	
`ServiceTime`	The service time for each job; non-negative; RO.	

6.3.6 The `IC` `SimQueue`

IC		
Description	Serves jobs in arrival order with an infinite number of servers, each of which is of infinite capacity and serves any job in zero time.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	`true`	Since non-dropped arriving jobs are taken into service immediately.
Waiting Area	Present	Infinite waiting area, arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the jobs currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting for SACs.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting jobs in arrival order while SAC > 0.
Start	Yes	Starts jobs if there are SACs.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
If SAC > 0, there are no jobs in the waiting area. There are no jobs in the service area.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "IC"; non-`null`; RW.	

6.3.7 The `LCFS SimQueue`

LCFS		
Description	Serves jobs in reverse arrival order with a single server.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	No Jobs	Since non-dropped arriving jobs are taken into service immediately if no other jobs are present.
Waiting Area	Present	Infinite waiting area, reverse arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most one job is present in the service area. If SAC > 0, the server cannot be idle in the presence of waiting jobs.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "LCFS"; non-`null`; RW.	
`NumberOfServers`	The number of servers; always 1; RO.	

6.3.8 The `SJF` `SimQueue`

SJF		
Description	Serves jobs in increasing service-time order with a single server.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	No Jobs	Since non-dropped arriving jobs are taken into service immediately if no other jobs are present.
Waiting Area	Present	Infinite waiting area, ordered increasing in service time.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most one job is present in the service area. If SAC > 0, the server cannot be idle in the presence of waiting jobs.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "SJF"; non-`null`; RW.	
`NumberOfServers`	The number of servers; always 1; RO.	

6.3.9 The `LJF` `SimQueue`

LJF		
Description	Serves jobs in decreasing service-time order with a single server.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	No Jobs	Since non-dropped arriving jobs are taken into service immediately if no other jobs are present.
Waiting Area	Present	Infinite waiting area, ordered decreasing in service time.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most one job is present in the service area. If SAC > 0, the server cannot be idle in the presence of waiting jobs.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "LJF"; non-`null`; RW.	
`NumberOfServers`	The number of servers; always 1; RO.	

6.3.10 The `RANDOM` `SimQueue`

RANDOM		
Description	Serves jobs in random order with a single server.	
State		
QAV	`false`/`true`	Drops arriving jobs during QAVs.
NoWaitArmed	No Jobs	Since non-dropped arriving jobs are taken into service immediately if no other jobs are present.
Waiting Area	Present	Infinite waiting area, random order with equal probabilities.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any).
State Operations		
Set QAV	`false`/`true`	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals.
Drop	Yes	Drops arriving jobs during QAVs.
Revocation	Yes	Jobs can be revoked while waiting and while being served.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Departure	Yes	Started and non-revoked jobs depart.
State Invariants		
At most one job is present in the service area. If SAC > 0, the server cannot be idle in the presence of waiting jobs.		
Properties		
`EventList`	The event list; non-`null`; RO.	
`Name`	The name, default "RANDOM"; non-`null`; RW.	
`NumberOfServers`	The number of servers; always 1; RO.	

6.4 Preemptive Queues

6.4.1 The P_LCFS SimQueue

P_LCFS		
Description	Serves jobs in reverse arrival order with a single server, preempting the current job in service, if any, upon an arrival.	
State		
QAV	false/true	Drops arriving jobs during QAVs.
NoWaitArmed	true	Since non-dropped arriving jobs are taken into service immediately.
Waiting Area	Present	Infinite waiting area, reverse arrival order.
SAC	$\{0, 1, 2, \ldots\}$	The remaining number of jobs that can start. `Integer.MAX_VALUE` is treated as $+\infty$.
Service Area	Present	Holds the job currently in service (if any), and all *preempted* jobs (if any).
State Operations		
Set QAV	false/true	Ends/starts a QAV.
Arrival	Yes	The queue accepts arrivals, preempting the job currently in service (if any).
Drop	Yes	Drops arriving jobs during QAVs and preempted jobs with `DROP` preemption strategy.
Revocation	Yes	Jobs can be revoked while waiting, while being served, and if having been preempted in the service area.
Set SAC	$\{0, 1, 2, \ldots\}$	Sets/overwrites SAC; `Integer.MAX_VALUE` $== +\infty$. Starts waiting job if SAC > 0 and server idle.
Start	Yes	Starts jobs if there are SACs and the server is idle.
Preemption	Yes	Job in service is preempted upon a new (non-dropped) arrival.
Departure	Yes	Started and non-revoked jobs depart. Jobs with `DROP` preemption strategy only depart if not preempted after start.

P_LCFS (continued)	
State Invariants	
If SAC > 0, the server cannot be idle in the presence of waiting jobs. The server cannot be idle in the presence of preempted jobs in the service area.	
Properties	
`EventList`	The event list; non-`null`; RO.
`Name`	The name, default "P_LCFS[`PreemptionStrategy`]"; non-`null`; RW.
`PreemptionStrategy`	The preemption strategy, default `RESUME`; non-`null`; RO.
`NumberOfServers`	The number of servers; always 1; RO.

6.4.2 The SRTF SimQueue

6.5 Processor-Sharing Queues

6.5.1 The PS SimQueue

6.5.2 The CATCH-UP-PS SimQueue

6.6 Multiclass Queues

6.6.1 The HOL SimQueue

6.6.2 The PQ SimQueue

6.6.3 The WPS SimQueue

6.6.4 The G-WPS SimQueue

6.6.5 The HOL-PS SimQueue

6.7 SimQueue Equalities

DELAY[0]	ZERO
DELAY[Double.POSITIVE_INFINITY]	SINK
FCFS_B[0]	DROP
FCFS_B[Integer.MAX_VALUE]	FCFS
FCFS_0	SINK
FCFS_1	FCFS
FCFS_Integer.MAX_VALUE	IS
IS_CST[0]	IC

Chapter 7

Multiclass Queues and Jobs

Chapter 8

Fundamental Multiclass Queues

Chapter 9

Composite Queues

9.1 Introduction

Composite queues consist of zero or more other queues named *subqueues* or *embedded queues* through which visiting jobs must pass. The sequence of visits to the embedded queues is determined by the composite queue.

XXX

9.2 Types (Colors) of Composite Queues

Chapter 10

Fundamental Composite Queues

10.1 Introduction

10.2 Tandem Queues

10.3 Compressed Tandem Queues

10.4 Parallel Queues

10.5 Feedback Queues

10.6 Jackson Networks

10.7 Special Composite Queues

10.7.1 Encapsulator Queues

10.7.2 Drop-Collector Queues

Chapter 11

Queue and Job Statistics

Chapter 12

Visualization of Queues and Jobs with Swing Components

Chapter 13

Advanced Topics

13.1 Job Factories

13.2 Queue Events and Schedules

13.3 Load Factories

Chapter 14

Building Custom Queues and Jobs

14.1 Introduction

14.2 Building Custom Entities

14.2.1 Introduction

14.2.2 The `AbstractSimEntity` Class

14.3 Building Custom Queues

14.3.1 Introduction

14.3.2 The `AbstractSimQueueBase` Class

14.3.3 The `AbstractSimQueue` Class

14.4 Building Custom Jobs

14.4.1 Introduction

14.4.2 The `AbstractSimJob` Class

14.4.3 The `DefaultSimJob` Class

Chapter 15

Other Topics

15.1 Introduction

15.2 Building Custom Listeners

Chapter 16

Test Infrastructure

16.1 Introduction

16.2 Test-Infrastructure Overview

16.2.1 Introduction

16.2.2 Queue Workloads

16.2.3 Queue Predictors

16.2.4 Confronting Queue Workloads, Predictors, and Queues

16.3 Workloads

16.3.1 Creating Workloads

16.3.2 Workload Ambiguities

16.3.3 Standardized Workload Patterns

16.3.4 Building a Workload

16.4 Predictors

16.4.1 Using a Predictor

16.4.2 Building a Predictor

Chapter 17

Conclusions