WORKED EXAMPLE 15.2 **Simulating a Queue of Waiting Customers**

A good application of object-oriented programming is simulation. In fact, the first object-oriented language, Simula, was designed with this application in mind. One can simulate the activities of air molecules around an aircraft wing, of customers in a supermarket, or of vehicles on a road system. The goal of a simulation is to observe how changes in the design affect the behavior of a system. Modifying the shape of a wing, the location and staffing of cash registers, or the synchronization of traffic lights has an effect on turbulence in the air stream, customer satisfaction, or traffic throughput. Modeling these systems in the computer is far cheaper than running actual experiments.

Kinds of Simulation

Simulations fall into two broad categories. A *continuous simulation* constantly updates all objects in a system. A simulated clock advances in seconds or some other suitable constant time interval. At every clock tick, each object is moved or updated in some way. Consider the simulation of traffic along a road. Each car has some position, velocity, and acceleration. Its position needs to be updated with every clock tick. If the car gets too close to an obstacle, it must decelerate. The new position may be displayed on the screen.

In contrast, in a *discrete event simulation*, time advances in chunks. All interesting events are kept in a priority queue, sorted by the time in which they are to happen. As soon as one event has completed, the clock jumps to the time of the next event to be executed.

To see the contrast between these two simulation styles, consider the updating of a traffic light. Suppose the traffic light just turned red, and it will turn green again in 30 seconds. In a continuous model, the traffic light is visited every second, and a counter variable is decremented. Once the counter reaches 0, the color changes. In a discrete model, the traffic light schedules an event to be notified 30 seconds from now. For 29 seconds, the traffic light is not bothered at all, and then it receives a message to change its state. Discrete event simulation avoids "busy waiting".

In this Worked Example, you will see how to use queues and priority queues in a discrete event simulation of customers at a bank. The simulation makes use of two generic classes, `Event` and `Simulation`, that are useful for any discrete event simulation. We use inheritance to extend these classes to make classes that simulate the bank.

Events

A discrete event simulation generates, stores, and processes events. Each event has a time stamp indicating when it is to be executed. Each event has some action associated with it that must be carried out at that time. Beyond these properties, the scheduler has no concept of what an event represents. Of course, actual events must carry with them some information. For example, the event notifying a traffic light of a state change must know which traffic light to notify.

To do so, we will have all events extend a common superclass, `Event`. An `Event` object has an instance variable to indicate at which time it should be processed. When that time has arrived, the event's `process` method is called. This method may move objects around, update information, and schedule additional events.

The `Event` class also implements the `Comparable` interface. An event is considered more urgent than another if its processing time is earlier.

```
public class Event implements Comparable<Event>
{
   private double time;

   public Event(double eventTime)
   {
```

```
      time = eventTime;
   }

   public void process(Simulation sim) {}
   public double getTime() { return time; }

   public int compareTo(Event other)
   {
      if (time < other.time) { return -1; }
      else if (time > other.time) { return 1; }
      else { return 0; }
   }
}
```

The Simulation Class

In any discrete event simulation, events are kept in a priority queue. After initialization, the simulation enters an event loop in which events are retrieved from the priority queue in the order specified by their time stamp. The simulated time is advanced to the time stamp of the event, and the event is processed according to its `process` method. To simulate a specific activity, such as customer activity in a bank, extend the `Simulation` class and provide methods for displaying the current state after each event, and a summary after the completion of the simulation.

```
public class Simulation
{
   private PriorityQueue<Event> eventQueue;
   private double currentTime;
   . . .
   public void display() {}
   public void displaySummary() {}
   . . .
}
```

Here is the event loop in the `Simulation` class:

```
public void run(double startTime, double endTime)
{
   currentTime = startTime;

   while (eventQueue.size() > 0 && currentTime <= endTime)
   {
      Event event = eventQueue.remove();
      currentTime = event.getTime();
      event.process(this);
      display();
   }
   displaySummary();
}
```

In the `Simulation` class, we provide a utility method for generating reasonable random values for the time between two independent events. These random time differences can be modeled with an "exponential distribution", as follows: Let m be the mean time between arrivals. Let u be a random value that can, with equal probability, assume any floating-point value between 0 inclusive and 1 exclusive. Then inter-arrival times can be generated as

$$a = -m \log(1 - u)$$

where log is the natural logarithm. The utility method `expdist` computes these random values:

```
public static double expdist(double mean)
{
   return -mean * Math.log(1 - Math.random());
```

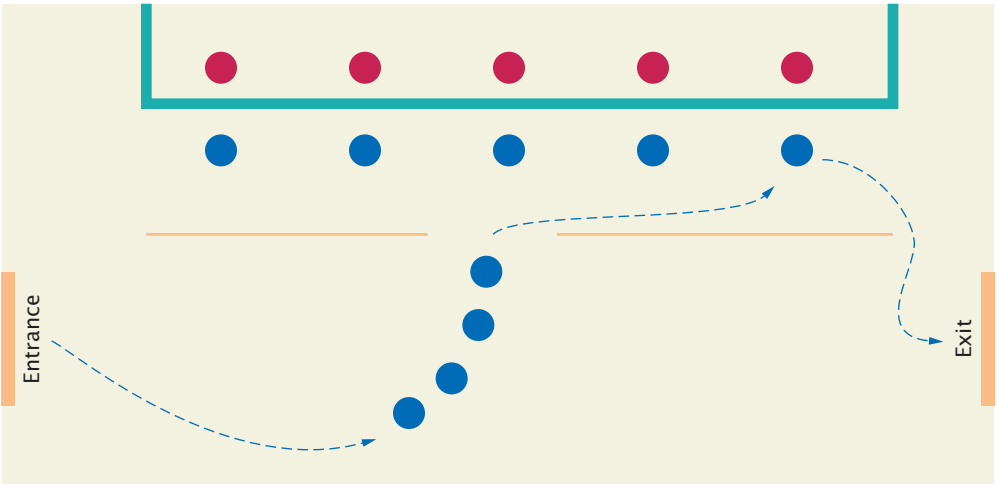
```
}
```

If a customer arrives at time t, the program can schedule the next customer arrival at t + `expdist(`m`)`.

Processing time is also exponentially distributed, with a different average. In this simulation we assume that, on average, one minute elapses between customer arrivals, and customer transactions require an average of five minutes.

The Bank

The following figure shows the layout of the bank. Customers enter the bank. If there is a queue, they join the queue; otherwise they move up to a teller. When a customer has completed a teller transaction, the time spent in the bank is logged, the customer is removed, and the next customer in the queue moves up to the teller.



The `BankSimulation` class keeps an array of tellers as well as a queue to hold waiting customers. The queue is not a priority queue but a regular FIFO (first-in, first-out) queue:

```
public class BankSimulation extends Simulation
{
   private Customer[] tellers;
   private Queue<Customer> custQueue;

   private int totalCustomers;
   private double totalTime;

   private static final double INTERARRIVAL = 1;
      // average of 1 minute between customer arrivals
   private static final double PROCESSING = 5;
      // average of 5 minutes processing time per customer
   . . .
}
```

It also keeps track of the total number of customers that have been serviced, and the total amount of time they spent in the bank (both in the waiting queue and in front of a teller.)

Teller `i` is busy if `tellers[i]` holds a reference to a `Customer` object and available if it is `null`.

When a customer is added to the bank, the program first checks whether a teller is available to handle the customer. If not, the customer is added to the waiting queue:

```
public void add(Customer c)
{
   boolean addedToTeller = false;
   for (int i = 0; !addedToTeller && i < tellers.length; i++)
   {
      if (tellers[i] == null)
      {
         addToTeller(i, c);
         addedToTeller = true;
      }
   }
   if (!addedToTeller) { custQueue.add(c); }

   addEvent(new Arrival(getCurrentTime() + expdist(INTERARRIVAL)));
}
```

In addition, the simulation must ensure that customers keep coming. We know the next customer will arrive in about one minute, but it may be a bit earlier or, occasionally, a lot later. To obtain a random time, we call `expdist(INTERARRIVAL)`. Of course, we cannot wait around for that to happen, because other events will be going on in the meantime. Therefore when a customer is added, another arrival event is scheduled to occur when this random time has elapsed.

Similarly, when a customer steps up to a teller, the average transaction will be five minutes. We need to schedule a departure event that removes the customer from the bank. This happens in the `addToTeller` method:

```
private void addToTeller(int i, Customer c)
{
   tellers[i] = c;
   addEvent(new Departure(getCurrentTime() + expdist(PROCESSING), i));
}
```

When the departure event is processed, it will notify the bank to remove the customer. The bank simulation removes the customer and keeps track of the total amount of time the customer spent in the waiting queue and with the teller. This makes the teller available to service the next customer from the waiting queue. If there is a queue, we add the first customer to this teller:

```
public void remove(int i)
{
   Customer c = tellers[i];
   tellers[i] = null;

   // Update statistics
   totalCustomers++;
   totalTime = totalTime + getCurrentTime() - c.getArrivalTime();

   if (custQueue.size() > 0)
   {
      addToTeller(i, custQueue.remove());
   }
}
```

Event Classes

The classes `Arrival` and `Departure` are subclasses of `Event`.

When a new customer is to arrive at the bank, an arrival event is processed. The processing action of that event has the responsibility of making a customer and adding it to the bank.

```
public class Arrival extends Event
{
```

```
   public Arrival(double time) { super(time); }

   public void process(Simulation sim)
   {
      double now = sim.getCurrentTime();
      BankSimulation bank = (BankSimulation) sim;
      Customer c = new Customer(now);
      bank.add(c);
   }
}
```

Departures remember not only the departure time but also the teller from whom a customer is to depart. To process a departure event, we remove the customer from the teller.

```
public class Departure extends Event
{
   private int teller;

   public Departure(double time, int teller)
   {
      super(time);
      this.teller = teller;
   }
   public void process(Simulation sim)
   {
      BankSimulation bank = (BankSimulation) sim;
      bank.remove(teller);
   }
}
```

Running the Simulation

To run the simulation, we first construct a `BankSimulation` object with five tellers. The most important task in setting up the simulation is to get the flow of events going. At the outset, the event queue is empty. We will schedule the arrival of a customer at the start time (9 A.M.). Because the processing of an arrival event schedules the arrival of each successor, the insertion of the arrival event for the first customer takes care of the generation of all arrivals. Once customers arrive at the bank, they are added to tellers, and departure events are generated.

Here is the `main` method:

```
public static void main(String[] args)
{
   final double START_TIME = 9 * 60; // 9 A.M.
   final double END_TIME = 17 * 60; // 5 P.M.

   final int NTELLERS = 5;

   Simulation sim = new BankSimulation(NTELLERS);
   sim.addEvent(new Arrival(START_TIME));
   sim.run(START_TIME, END_TIME);
}
```

Here is a typical program run. The bank starts out with empty tellers, and customers start dropping in:

```
.....<
C....<
CC...<
CCC..<
CCCC.<
C.CC.<
```

```
CCCC.<
CCCCC<
CCCCC<C
CCCCC<
C.CCC<
```

Due to the random fluctuations of customer arrival and processing, the queue can get quite long:

```
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCCC
CCCCC<CCCCCCCCCC

CCCCC<CCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCCC
CCCCC<CCCCCCCCCCC
CCCCC<CCCCCCCCCCC
```

At other times, the bank is empty again:

```
CCC.C<
CCC..<
CC...<
.C...<
.....<
C....<
```

This particular run of the simulation ends up with the following statistics:

```
457 customers. Average time 15.28 minutes.
```

If you are the bank manager, this result is quite depressing. You hired enough tellers to take care of all customers. (Every hour, you need to serve, on average, 60 customers. Their transactions take an average of 5 minutes each; that is 300 teller-minutes, or 5 teller-hours. Hence, hiring five tellers should be just right.) Yet the average customer had to wait in line more than 10 minutes, twice as long as their transaction time. This is an average, so some customers had to wait even longer. If disgruntled customers hurt your business, you may have to hire more tellers and pay them for being idle some of the time.

(See the `ch15/worked_example_2` folder in your companion code for the complete bank simulation program.)

worked_example_2/BankSimulation.java

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3
4  /**
5     Simulation of customer traffic in a bank.
6  */
7  public class BankSimulation extends Simulation
8  {
9     private Customer[] tellers;
10    private Queue<Customer> custQueue;
11
12    private int totalCustomers;
13    private double totalTime;
14
15    private static final double INTERARRIVAL = 1;
16       // average of 1 minute between customer arrivals
```

```
17    private static final double PROCESSING = 5;
18       // average of 5 minutes processing time per customer
19
20    public BankSimulation(int numberOfTellers)
21    {
22       tellers = new Customer[numberOfTellers];
23       custQueue = new LinkedList<>();
24       totalCustomers = 0;
25       totalTime = 0;
26    }
27
28    /**
29        Adds a customer to the bank.
30        @param c the customer
31    */
32    public void add(Customer c)
33    {
34       boolean addedToTeller = false;
35       for (int i = 0; !addedToTeller && i < tellers.length; i++)
36       {
37          if (tellers[i] == null)
38          {
39             addToTeller(i, c);
40             addedToTeller = true;
41          }
42       }
43       if (!addedToTeller) { custQueue.add(c); }
44
45       addEvent(new Arrival(getCurrentTime() + expdist(INTERARRIVAL)));
46    }
47
48    /**
49       Adds a customer to a teller and schedules the departure event.
50       @param i the teller number
51       @param c the customer
52    */
53    private void addToTeller(int i, Customer c)
54    {
55       tellers[i] = c;
56       addEvent(new Departure(getCurrentTime() + expdist(PROCESSING), i));
57    }
58
59    /**
60        Removes a customer from a teller.
61        @param i teller position
62    */
63    public void remove(int i)
64    {
65       Customer c = tellers[i];
66       tellers[i] = null;
67
68       // Update statistics
69       totalCustomers++;
70       totalTime = totalTime + getCurrentTime() - c.getArrivalTime();
71
72       if (custQueue.size() > 0)
73       {
74          addToTeller(i, custQueue.remove());
75       }
76    }
```

```
77
78       /**
79          Displays tellers and queue.
80       */
81       public void display()
82       {
83          for (int i = 0; i < tellers.length; i++)
84          {
85             if (tellers[i] == null)
86             {
87                System.out.print(".");
88             }
89             else
90             {
91                System.out.print("C");
92             }
93          }
94          System.out.print("<");
95          int q = custQueue.size();
96          for (int j = 1; j <= q; j++) { System.out.print("C"); }
97          System.out.println();
98       }
99
100      /**
101         Displays a summary of the gathered statistics.
102      */
103      public void displaySummary()
104      {
105         double averageTime = 0;
106         if (totalCustomers > 0)
107         {
108            averageTime = totalTime / totalCustomers;
109         }
110         System.out.println(totalCustomers + " customers. Average time "
111            + averageTime + " minutes.");
112      }
113   }
```