Problem Solving using Stack and Queue (additional)

WIA1002/WIB1002 :
Data Structure

HTML Tag Matching

- For fully-correct HTML, each opening tag should pair with a matching closing tag. For examples, <body> pairs with </body>, <center> pairs with </center>, etc. In general, each <name> should pair with a matching </name>
- An HTML document (a) and its rendering (b) are shown below.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat
like a cheap sneaker in an old
washing machine. The three
drunken fishermen were used to
such treatment, of course, but not
the tree salesman, who even as
a stowaway now felt that he had
overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

HTML Tag Matching

- Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags.
- The checking of whether an html file has matching tags can be done using **stack**.
- Assuming that the entire HTML document is read into a string, a method isHTMLMatched can be written to do the checking:
  - make a left-to-right pass through the raw string, using index *j* to track progress.
  - indexOf method of the String class, which optionally accepts a starting index as a second parameter, locates the '<' and '>' characters that define the tags.
  - Substring method of the String class, returns the substring starting at a given index and optionally ending right before another given index.
  - Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack

HTML Tag Matching

```
public static boolean isHTMLMatched(String html) {
    Stack<String> buffer = new Stack<>( );
    int j = html.indexOf('<'); // find first ’<’ character (if any)
    while (j != -1) {
    int k = html.indexOf('>', j+1); // find next ’>’ character
    if (k == -1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
      j = html.indexOf('<', k+1); // find next ’<’ character (if any)
    }
    return buffer.isEmpty( ); // were all opening tags matched?
  }
```

HTML Tag Matching

- Question 1: Write a Java program that read an HTML file into a string that is passed into the isHTMLMatched method to check whether the file has matching tags. Test your program using another HTML file which has one of the closing tags missing.

Simulating a Waiting Line

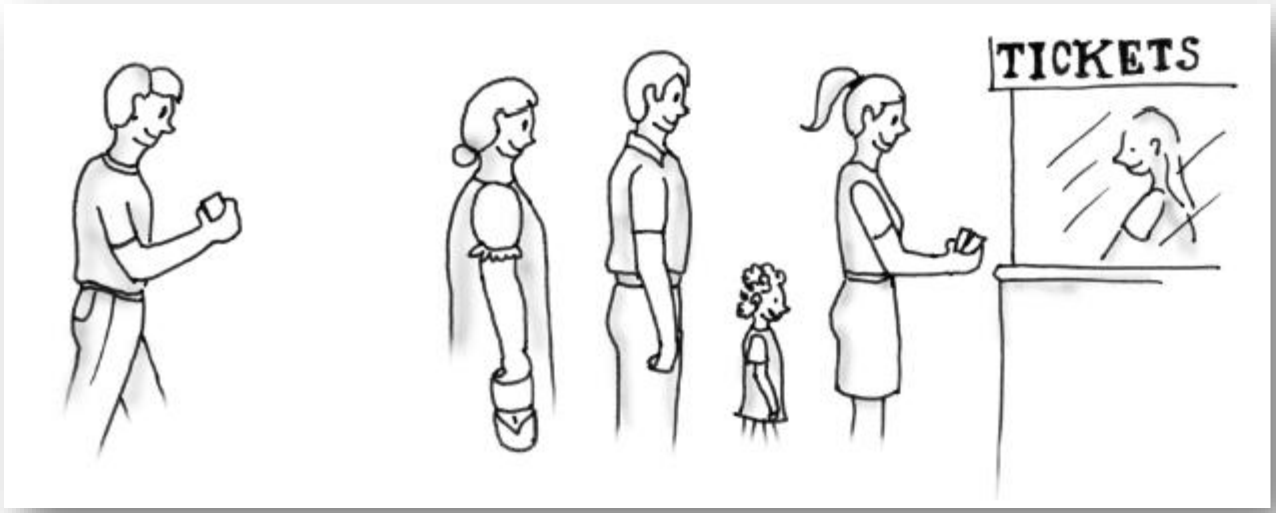
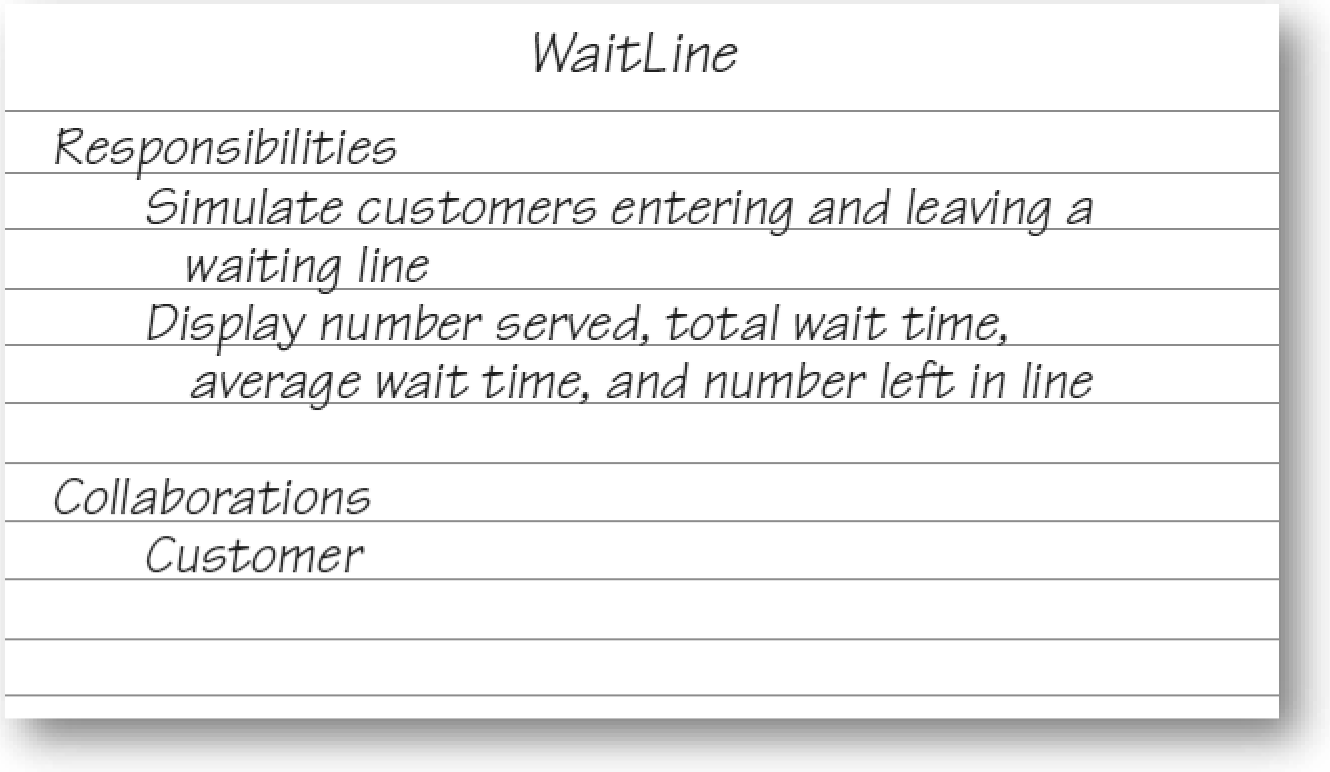


FIGURE 10-3 A line, or queue, of people

Simulating a Waiting Line



WaitLine

Responsibilities

Simulate customers entering and leaving a waiting line

Display number served, total wait time, average wait time, and number left in line

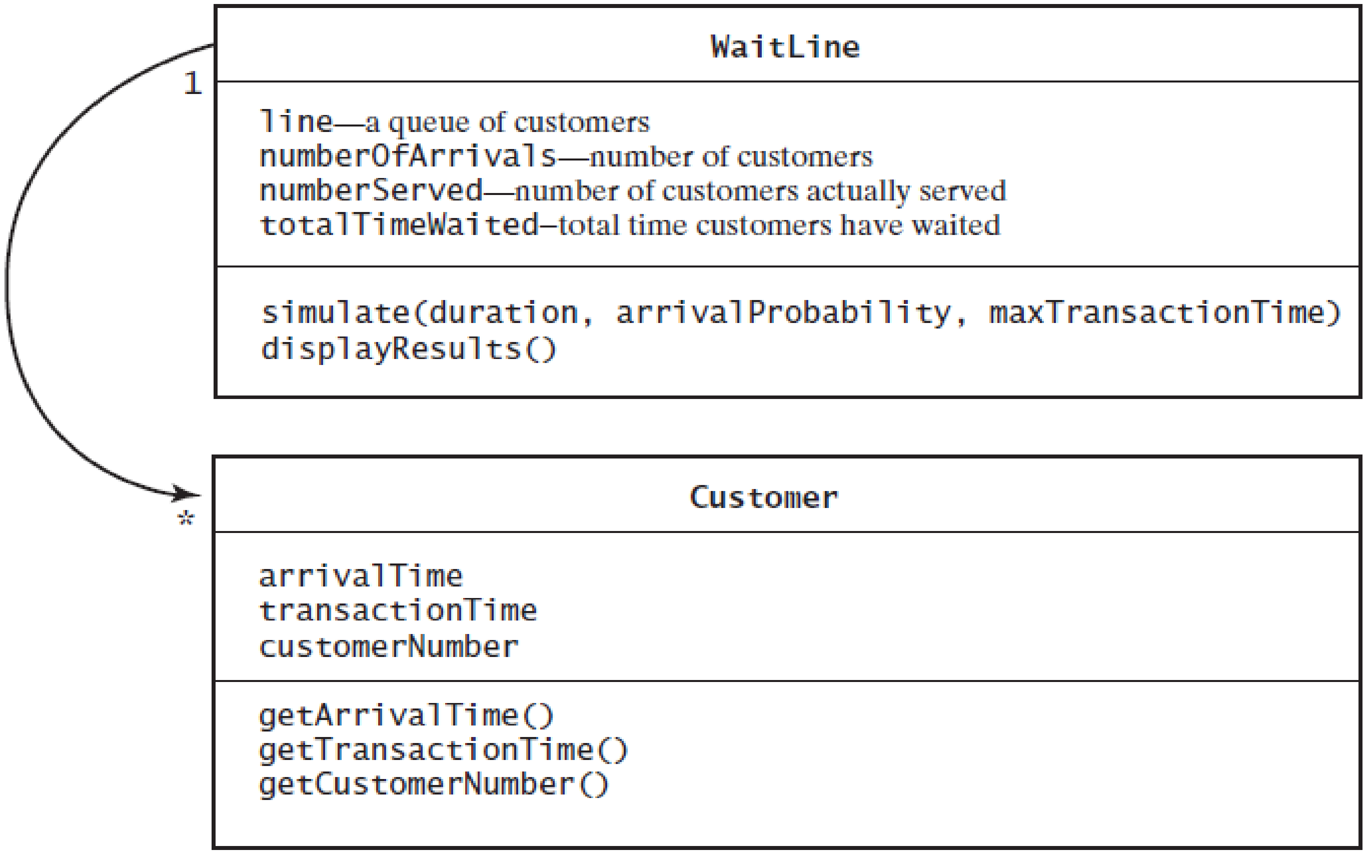
Collaborations

Customer

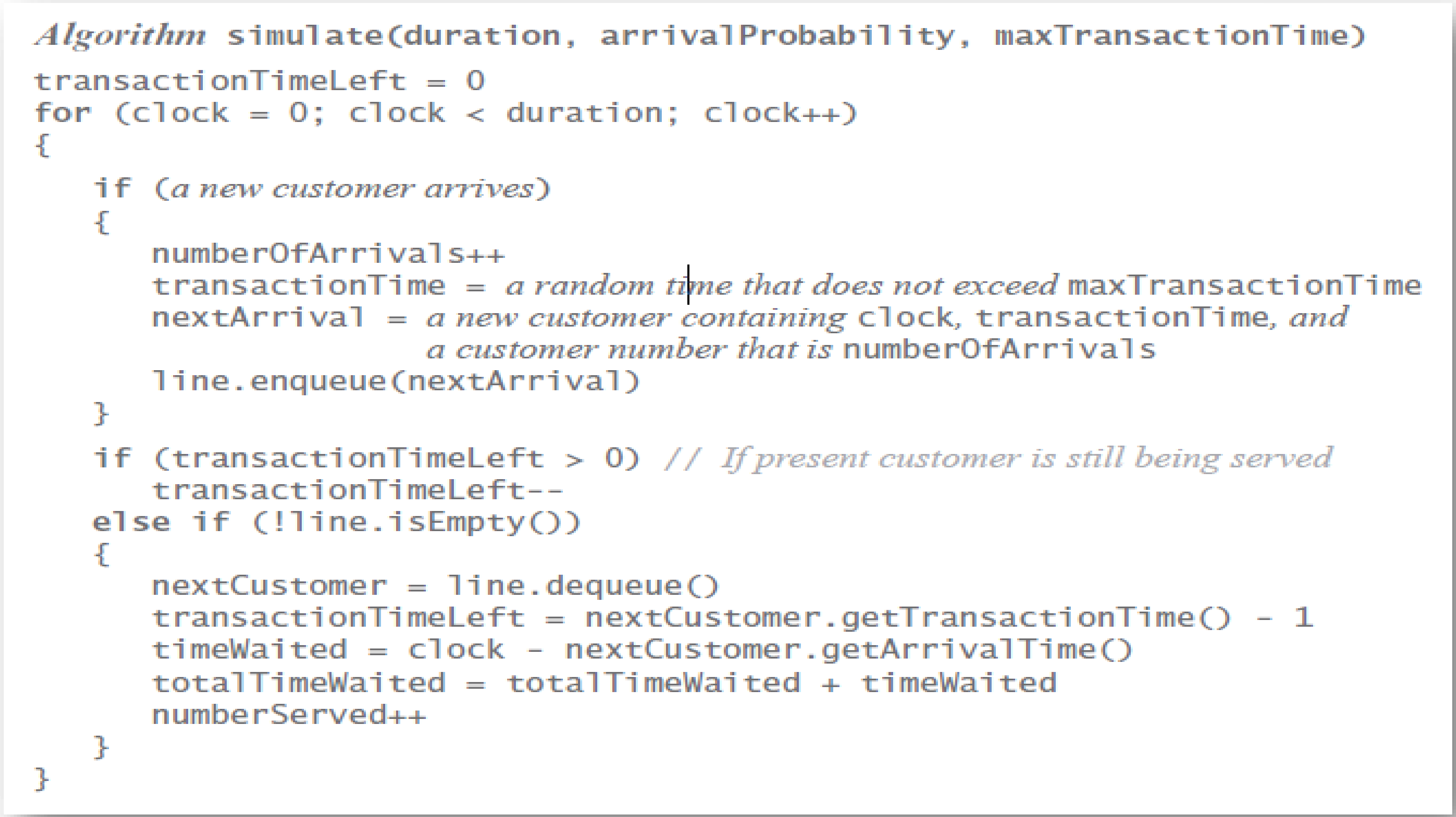
FIGURE 10-4 A CRC card for the class `WaitLine`

Simulating a Waiting Line

FIGURE 10-5 A diagram of the classes WaitLine and Customer


WaitLine
line—a queue of customers
numberOfArrivals—number of customers
numberServed—number of customers actually served
totalTimeWaited—total time customers have waited
simulate(duration, arrivalProbability, maxTransactionTime)
displayResults()
1
*
Customer
arrivalTime
transactionTime
customerNumber
getArrivalTime()
getTransactionTime()
getCustomerNumber()

Simulating a Waiting Line



```
Algorithm simulate(duration, arrivalProbability, maxTransactionTime)
transactionTimeLeft = 0
for (clock = 0; clock < duration; clock++)
{
   if (a new customer arrives)
   {
      numberOfArrivals++
      transactionTime = a random time that does not exceed maxTransactionTime
      nextArrival = a new customer containing clock, transactionTime, and
                       a customer number that is numberOfArrivals
      line.enqueue(nextArrival)
   }
   if (transactionTimeLeft > 0) // If present customer is still being served
      transactionTimeLeft--
   else if (!line.isEmpty())
   {
      nextCustomer = line.dequeue()
      transactionTimeLeft = nextCustomer.getTransactionTime() - 1
      timeWaited = clock - nextCustomer.getArrivalTime()
      totalTimeWaited = totalTimeWaited + timeWaited
      numberServed++
   }
}
```

Algorithm for **simulate**

Simulating a Waiting Line

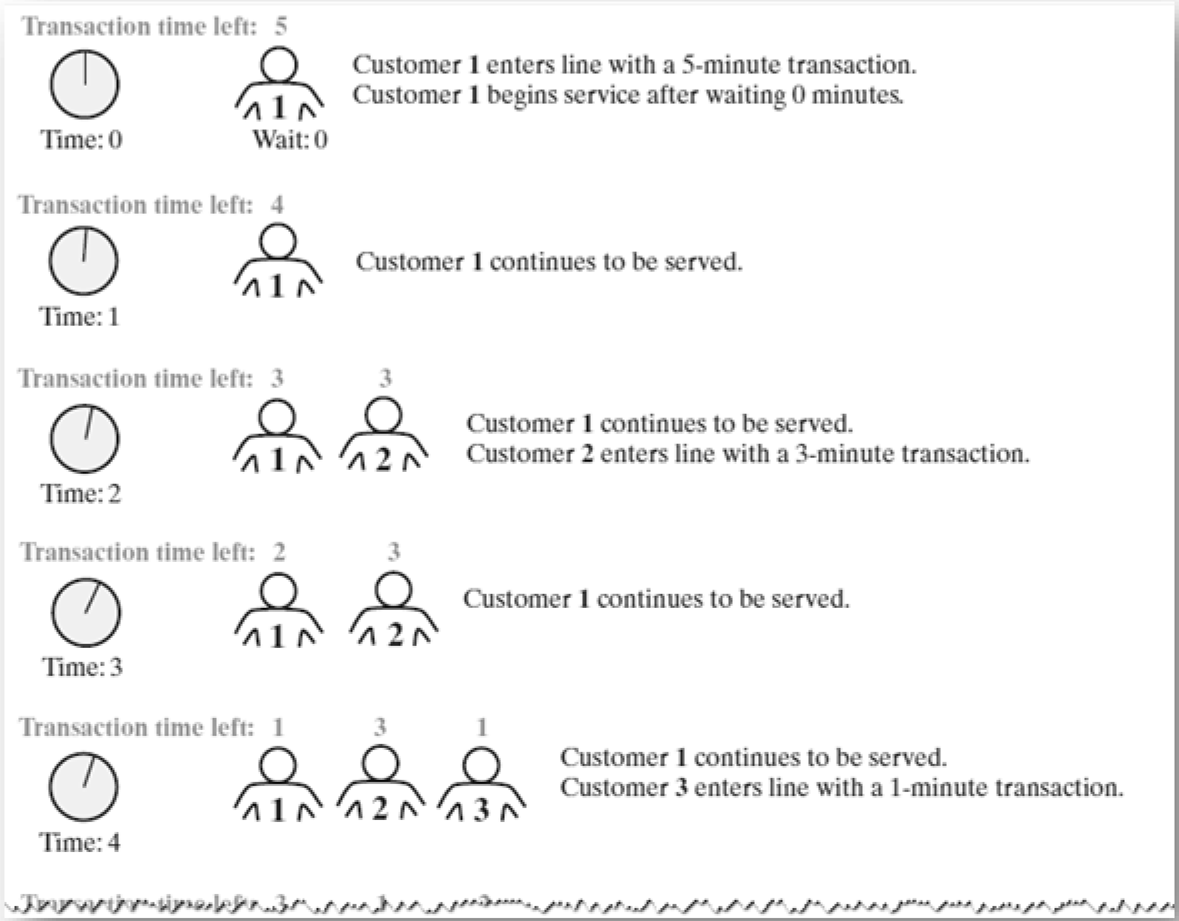

Transaction time left: 5
Time: 0
Wait: 0
Customer 1 enters line with a 5-minute transaction.
Customer 1 begins service after waiting 0 minutes.
Transaction time left: 4
Time: 1
Customer 1 continues to be served.
Transaction time left: 3 3
Time: 2
Customer 1 continues to be served.
Customer 2 enters line with a 3-minute transaction.
Transaction time left: 2 3
Time: 3
Customer 1 continues to be served.
Transaction time left: 1 3 1
Time: 4
Customer 1 continues to be served.
Customer 3 enters line with a 1-minute transaction.

FIGURE 10-6 A simulated waiting line

Simulating a Waiting Line

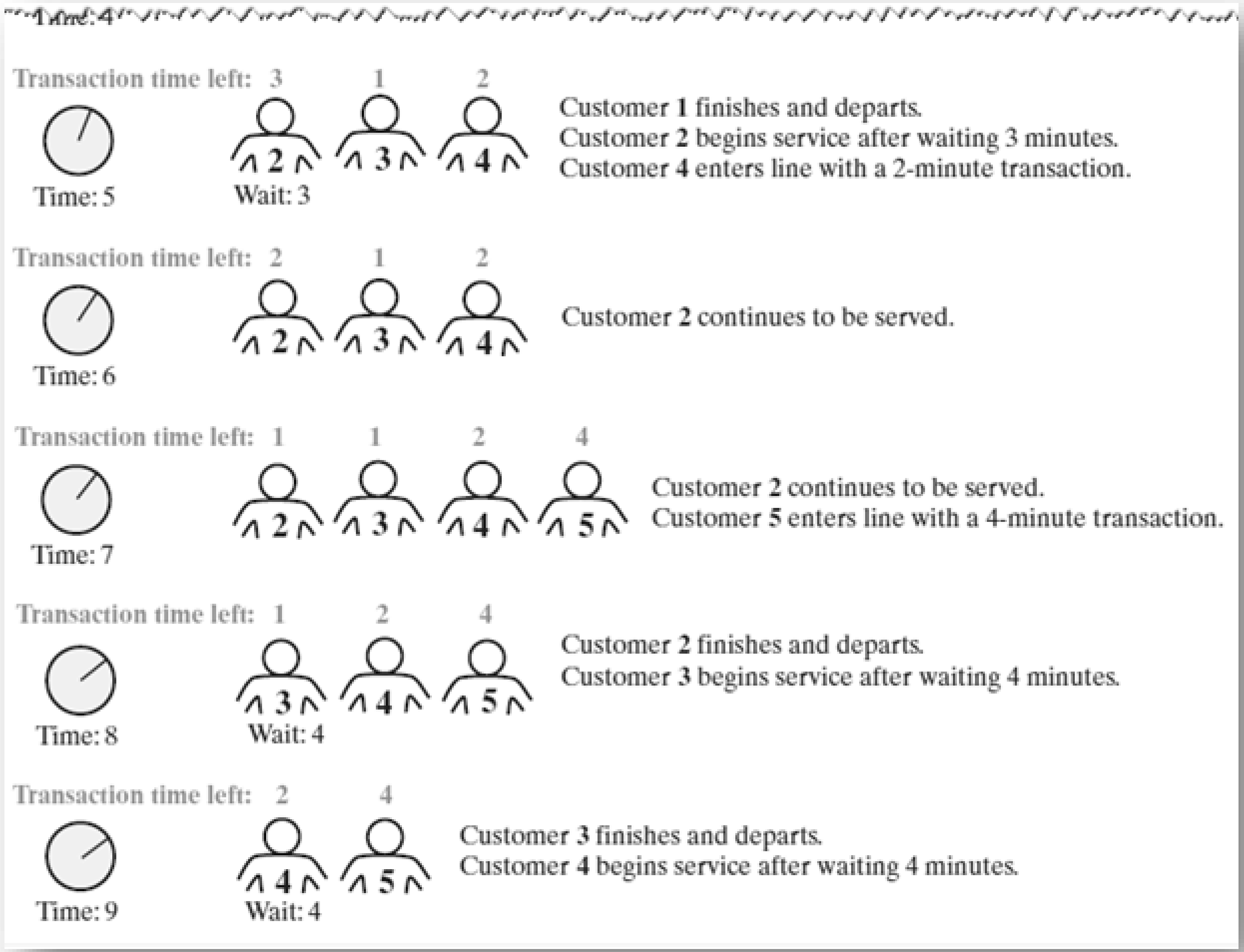
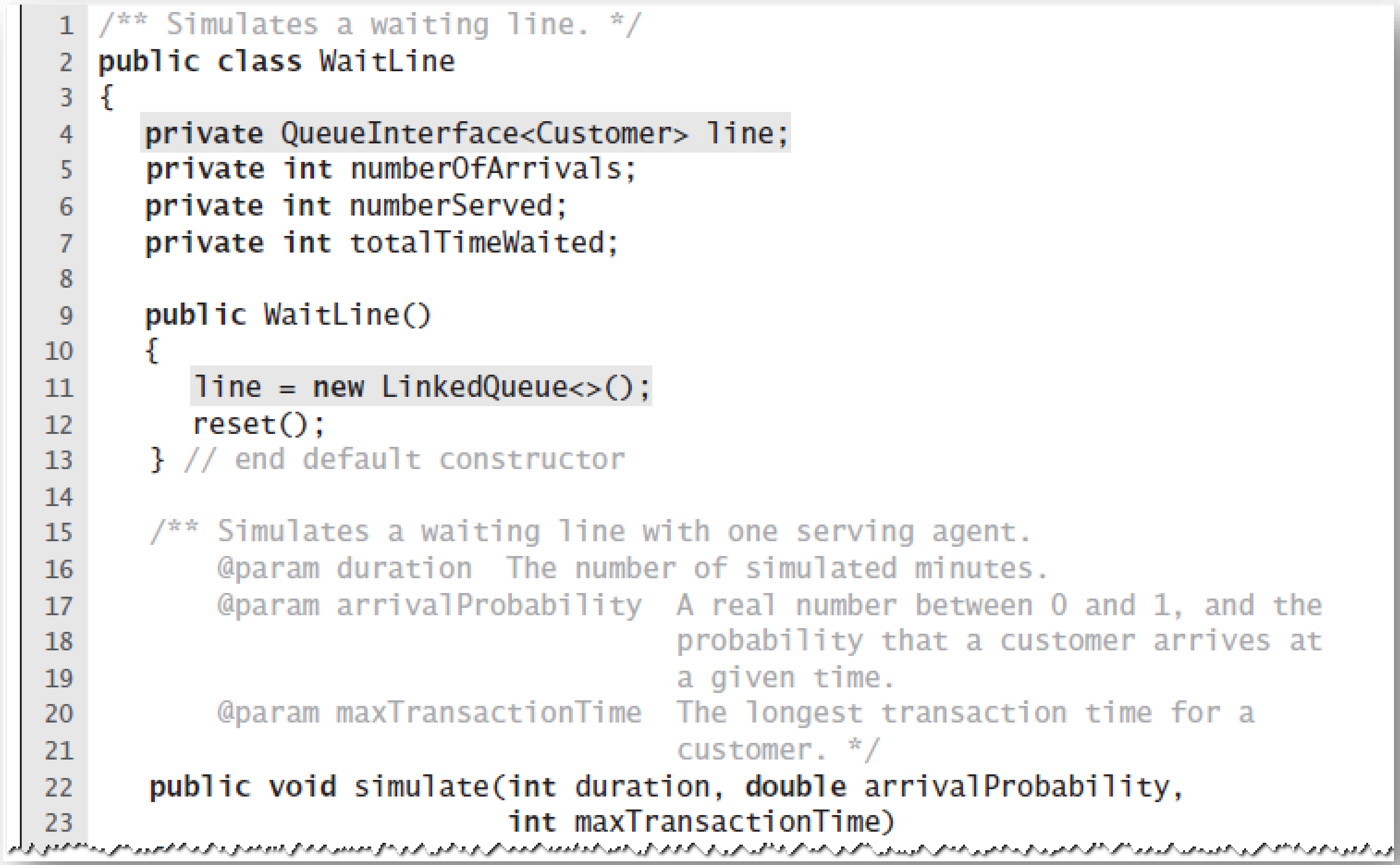

Time: 4
Transaction time left: 3 1 2
2 3 4
Time: 5
Wait: 3
Customer 1 finishes and departs.
Customer 2 begins service after waiting 3 minutes.
Customer 4 enters line with a 2-minute transaction.
Transaction time left: 2 1 2
2 3 4
Time: 6
Customer 2 continues to be served.
Transaction time left: 1 1 2 4
2 3 4 5
Time: 7
Customer 2 continues to be served.
Customer 5 enters line with a 4-minute transaction.
Transaction time left: 1 2 4
3 4 5
Time: 8
Wait: 4
Customer 2 finishes and departs.
Customer 3 begins service after waiting 4 minutes.
Transaction time left: 2 4
4 5
Time: 9
Wait: 4
Customer 3 finishes and departs.
Customer 4 begins service after waiting 4 minutes.

FIGURE 10-6 A simulated waiting line

Simulating a Waiting Line



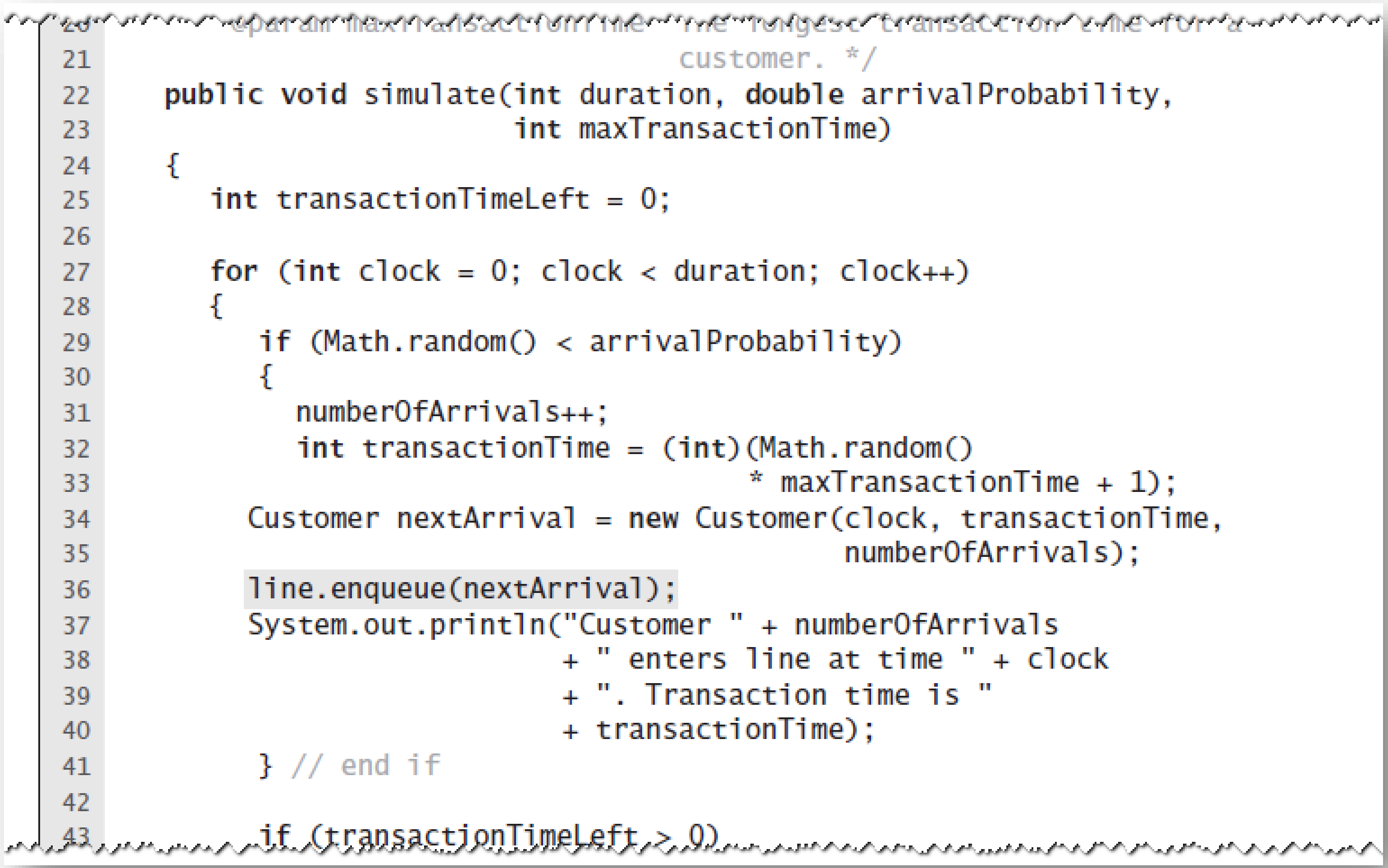
```
/** Simulates a waiting line. */
public class WaitLine
{
   private QueueInterface<Customer> line;
   private int numberOfArrivals;
   private int numberServed;
   private int totalTimeWaited;

   public WaitLine()
   {
      line = new LinkedQueue<>();
      reset();
   } // end default constructor

   /** Simulates a waiting line with one serving agent.
       @param duration  The number of simulated minutes.
       @param arrivalProbability  A real number between 0 and 1, and the
                                  probability that a customer arrives at
                                  a given time.
       @param maxTransactionTime  The longest transaction time for a
                                  customer. */
   public void simulate(int duration, double arrivalProbability,
                        int maxTransactionTime)
```

LISTING 10-2 The class `WaitLine`

Simulating a Waiting Line



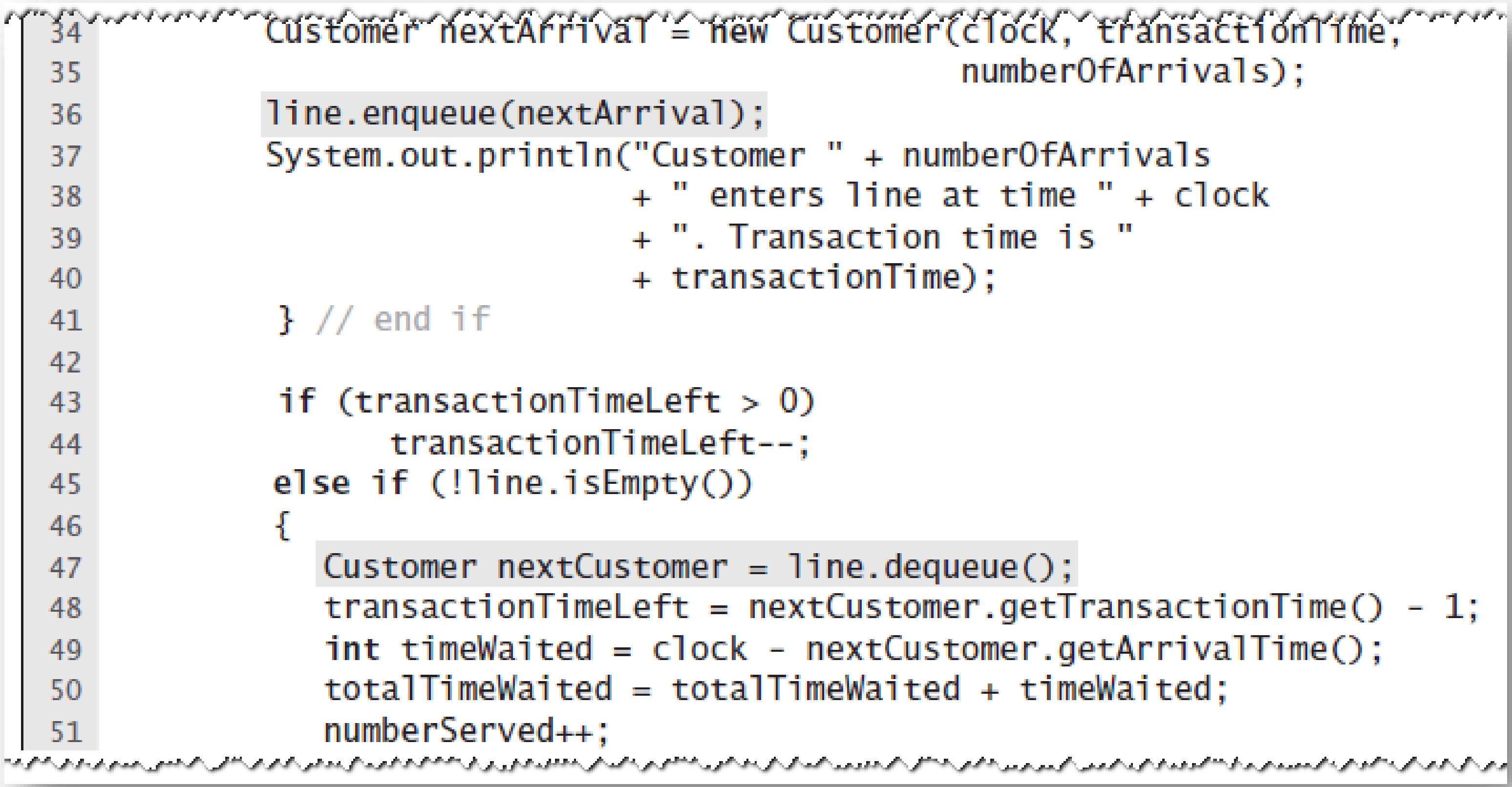
```
      @param maxTransactionTime  the longest transaction time for a
                                 customer. */
public void simulate(int duration, double arrivalProbability,
                     int maxTransactionTime)
{
   int transactionTimeLeft = 0;

   for (int clock = 0; clock < duration; clock++)
   {
      if (Math.random() < arrivalProbability)
      {
         numberOfArrivals++;
         int transactionTime = (int)(Math.random()
                                     * maxTransactionTime + 1);
     Customer nextArrival = new Customer(clock, transactionTime,
                                         numberOfArrivals);
     line.enqueue(nextArrival);
     System.out.println("Customer " + numberOfArrivals
                        + " enters line at time " + clock
                        + ". Transaction time is "
                        + transactionTime);
      } // end if

      if (transactionTimeLeft > 0)
```

LISTING 10-2 The class **`WaitLine`**

Simulating a Waiting Line

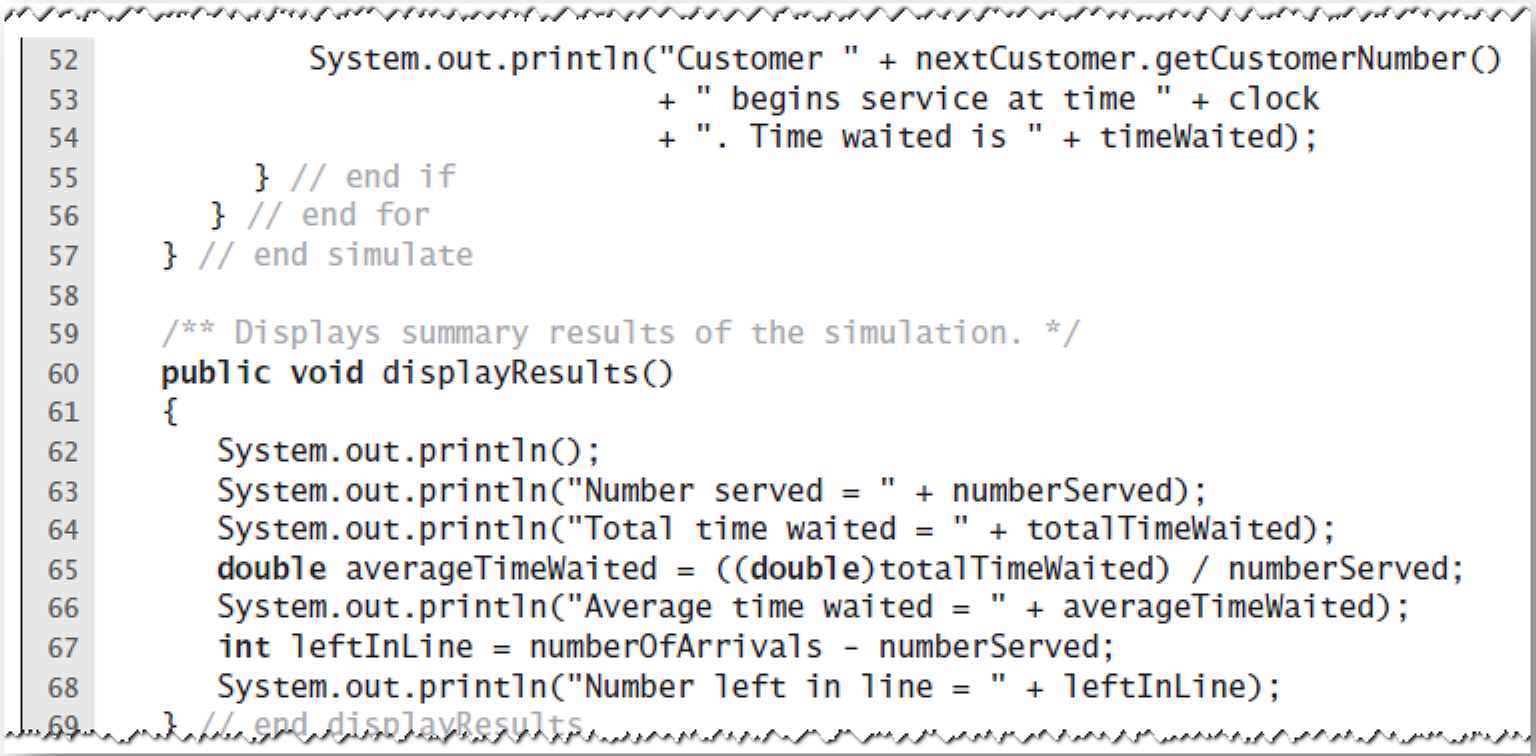


```
            Customer nextArrival = new Customer(clock, transactionTime,
                                                numberOfArrivals);
            line.enqueue(nextArrival);
            System.out.println("Customer " + numberOfArrivals
                               + " enters line at time " + clock
                               + ". Transaction time is "
                               + transactionTime);
         } // end if

         if (transactionTimeLeft > 0)
            transactionTimeLeft--;
         else if (!line.isEmpty())
         {
            Customer nextCustomer = line.dequeue();
            transactionTimeLeft = nextCustomer.getTransactionTime() - 1;
            int timeWaited = clock - nextCustomer.getArrivalTime();
            totalTimeWaited = totalTimeWaited + timeWaited;
            numberServed++;
```

LISTING 10-2 The class **WaitLine**

Simulating a Waiting Line

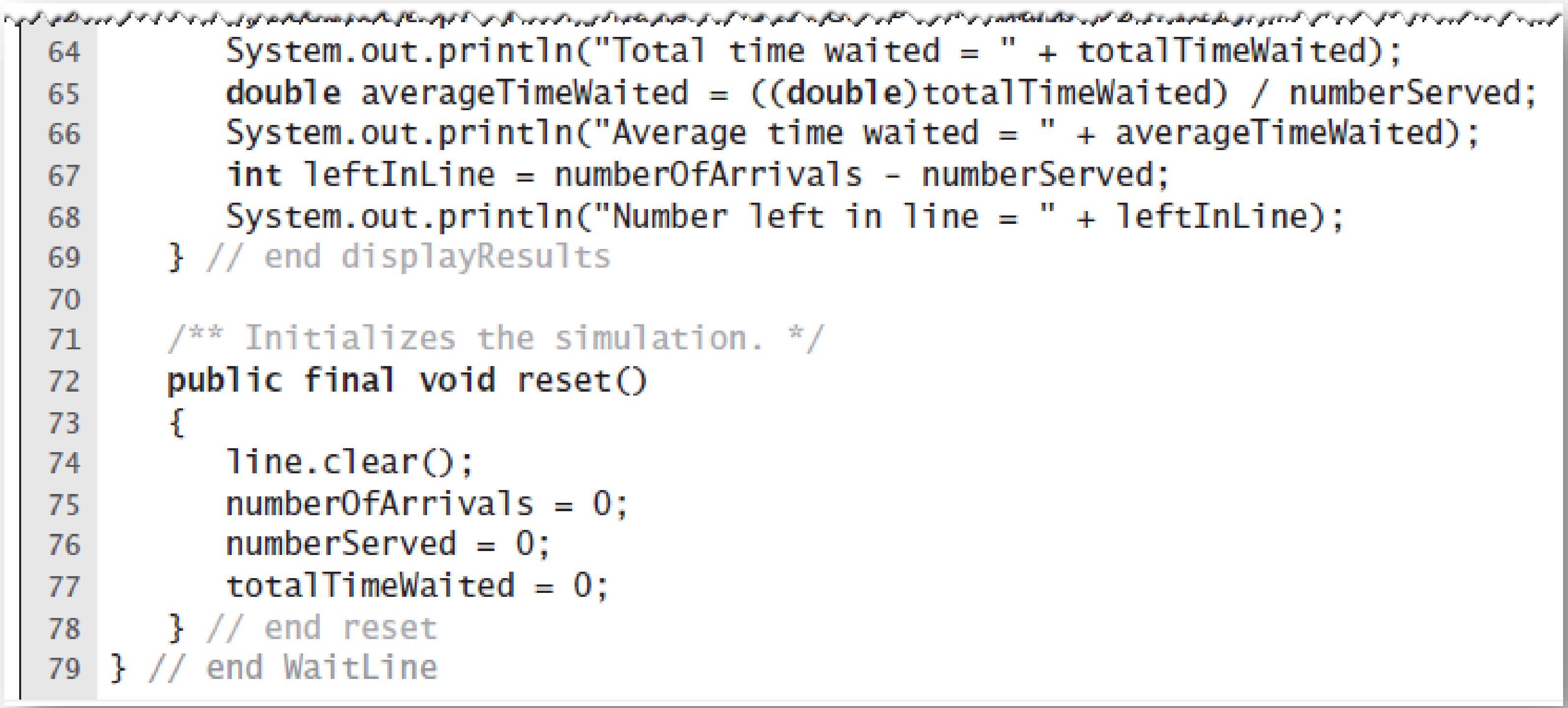


```
            System.out.println("Customer " + nextCustomer.getCustomerNumber()
                               + " begins service at time " + clock
                               + ". Time waited is " + timeWaited);
         } // end if
      } // end for
   } // end simulate

   /** Displays summary results of the simulation. */
   public void displayResults()
   {
      System.out.println();
      System.out.println("Number served = " + numberServed);
      System.out.println("Total time waited = " + totalTimeWaited);
      double averageTimeWaited = ((double)totalTimeWaited) / numberServed;
      System.out.println("Average time waited = " + averageTimeWaited);
      int leftInLine = numberOfArrivals - numberServed;
      System.out.println("Number left in line = " + leftInLine);
   } // end displayResults
```

LISTING 10-2 The class **`WaitLine`**

Simulating a Waiting Line



```
        System.out.println("Total time waited = " + totalTimeWaited);
        double averageTimeWaited = ((double)totalTimeWaited) / numberServed;
        System.out.println("Average time waited = " + averageTimeWaited);
        int leftInLine = numberOfArrivals - numberServed;
        System.out.println("Number left in line = " + leftInLine);
    } // end displayResults

    /** Initializes the simulation. */
    public final void reset()
    {
        line.clear();
        numberOfArrivals = 0;
        numberServed = 0;
        totalTimeWaited = 0;
    } // end reset
} // end WaitLine
```

LISTING 10-2 The class **`WaitLine`**

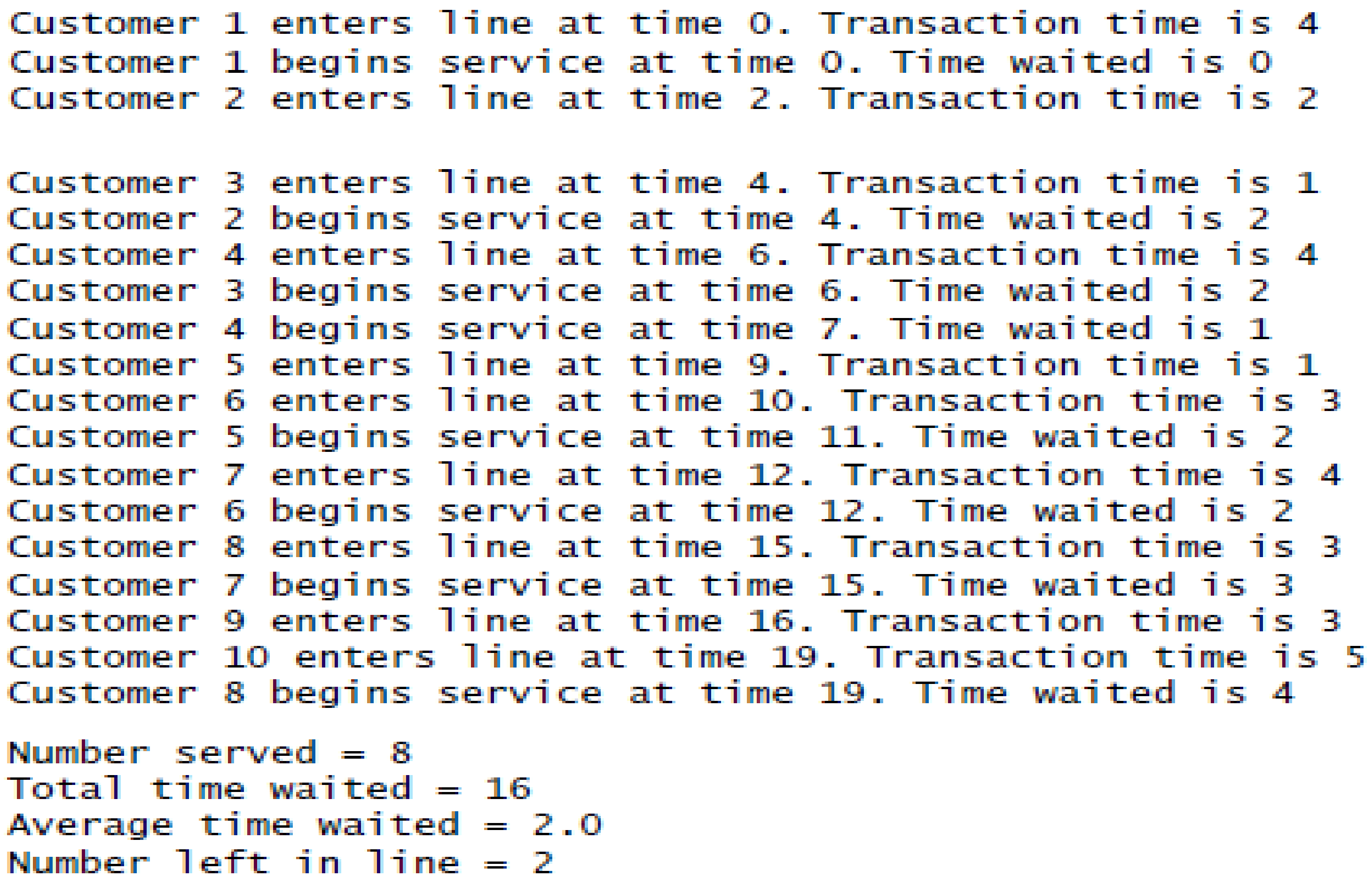
Sample output

The Java statements

```
WaitLine customerLine = new WaitLine();
customerLine.simulate(20, 0.5, 5);
customerLine.displayResults();
```

simulate the line for 20 minutes with a 50 percent arrival probability and a 5-minute maximum transaction time. They produce the following results (next slide). Since this example uses random numbers, another execution of the Java statements likely will have different results.

Sample output



```
Customer 1 enters line at time 0. Transaction time is 4
Customer 1 begins service at time 0. Time waited is 0
Customer 2 enters line at time 2. Transaction time is 2

Customer 3 enters line at time 4. Transaction time is 1
Customer 2 begins service at time 4. Time waited is 2
Customer 4 enters line at time 6. Transaction time is 4
Customer 3 begins service at time 6. Time waited is 2
Customer 4 begins service at time 7. Time waited is 1
Customer 5 enters line at time 9. Transaction time is 1
Customer 6 enters line at time 10. Transaction time is 3
Customer 5 begins service at time 11. Time waited is 2
Customer 7 enters line at time 12. Transaction time is 4
Customer 6 begins service at time 12. Time waited is 2
Customer 8 enters line at time 15. Transaction time is 3
Customer 7 begins service at time 15. Time waited is 3
Customer 9 enters line at time 16. Transaction time is 3
Customer 10 enters line at time 19. Transaction time is 5
Customer 8 begins service at time 19. Time waited is 4

Number served = 8
Total time waited = 16
Average time waited = 2.0
Number left in line = 2
```

Evaluation of expressions using stack

Watch the following video:

- Infix, Prefix and Postfix (https://youtu.be/jos1Flt21is)

- Evaluation of Prefix and Postfix expressions using stack (https://youtu.be/MeRb_1bddWg)

- Infix to Postfix (https://www.youtube.com/watch?v=vq-nUF0G4fI)

Evaluation of expressions using stack

Question 2: Write a program to convert postfix expression to infix expression. Compare it with the code for postfix evaluation from lecture Week 9.

Reference

- HTML Tag Matching - Chapter 6, Michael T. Goodrich and Roberto Tamassia. 2014. Data Structures and Algorithms in Java. 6th Edition. Wiley.
- Simulating a waiting line - Chapter 10, Frank M. Carrano and Timothy Henry. 2015. Data Structures and Abstractions with Java, 4th Edition. Pearson.
- Internet resources