Queues, Deques, and Priority Queues

Chapter 10

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

The ADT Queue

- A queue is another name for a waiting line
- Used within operating systems and to simulate real-world events
  - Come into play whenever processes or events must wait
- Entries organized first-in, first-out

The ADT Queue

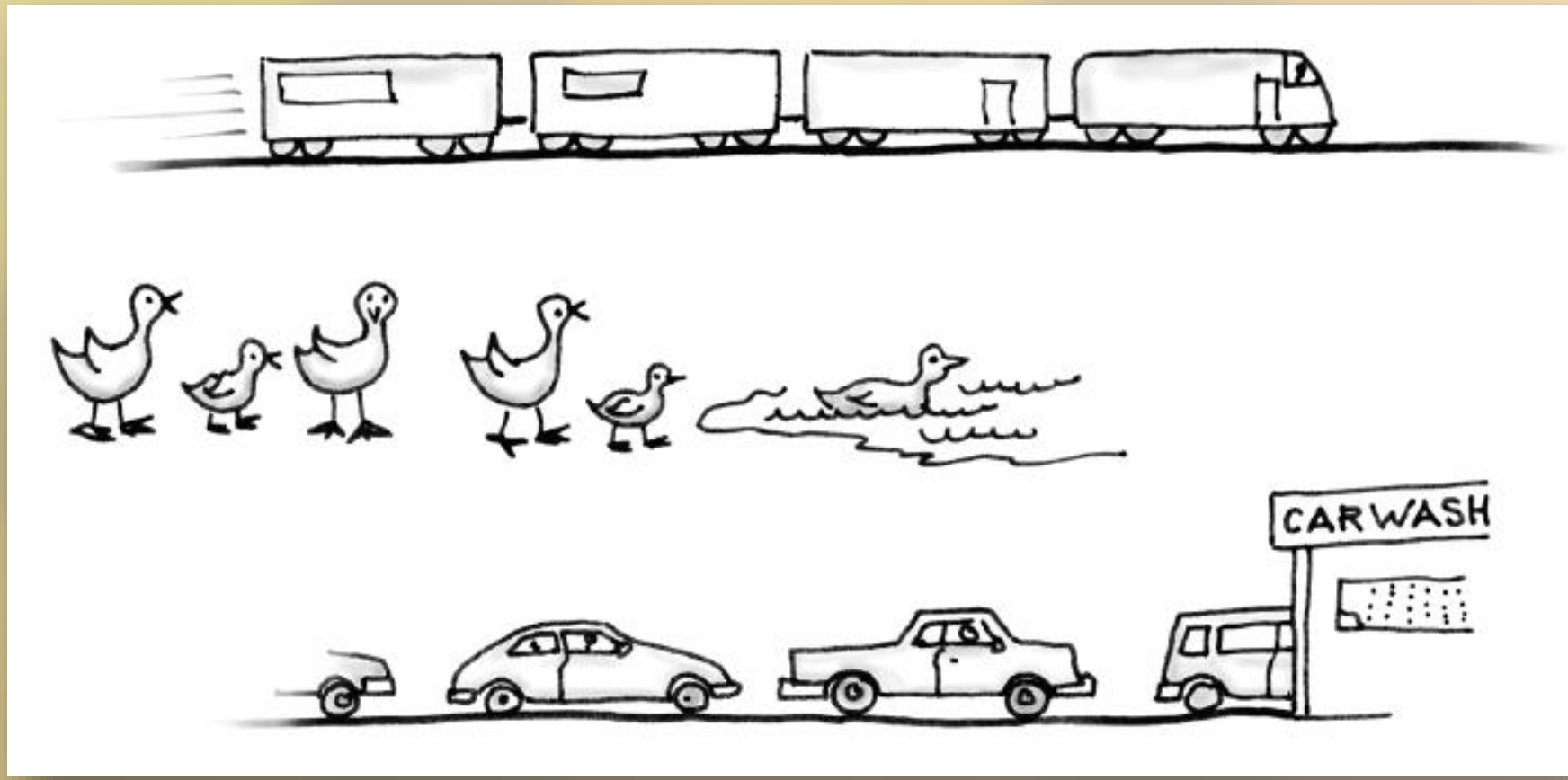


FIGURE 10-1 Some everyday queues

The ADT Queue

- Terminology
  - Item added first, or earliest, is at the front of the queue
  - Item added most recently is at the back of the queue
- Additions to a software queue must occur at its back
- Client can look at or remove only the entry at the front of the queue

The ADT Queue

Abstract Data Type: Queue

Data

- A collection of objects in chronological order and having the same data type

Operations

Pseudocode	UML	Description
`enqueue(newEntry)`	`+enqueue(newEntry: integer): void`	Task: Adds a new entry to the back of the queue. Input: `newEntry` is the new entry. Output: None.
`dequeue()`	`+dequeue(): T`	Task: Removes and returns the entry at the front of the queue. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty before the operation.

The ADT Queue

`getFront()`	`+getFront(): T`	Task: Retrieves the queue's front entry without changing the queue in any way. Input: None. Output: Returns the queue's front entry. Throws an exception if the queue is empty.
`isEmpty()`	`+isEmpty(): boolean`	Task: Detects whether the queue is empty. Input: None. Output: Returns true if the queue is empty.
`clear()`	`+clear(): void`	Task: Removes all entries from the queue. Input: None. Output: None.

The ADT Queue

```
public interface QueueInterface<T>
{
   /** Adds a new entry to the back of this queue.
       @param newEntry  An object to be added. */
   public void enqueue(T newEntry);

   /** Removes and returns the entry at the front of this queue.
       @return  The object at the front of the queue.
       @throws  EmptyQueueException if the queue is empty before the operation.
   public T dequeue();

   /** Retrieves the entry at the front of this queue.
       @return  The object at the front of the queue.
       @throws  EmptyQueueException if the queue is empty. */
   public T getFront();

   /** Detects whether this queue is empty.
       @return  True if the queue is empty, or false otherwise. */
   public boolean isEmpty();

   /** Removes all entries from this queue. */
   public void clear();
} // end QueueInterface
```


LISTING 10-1 An interface for the ADT queue

The ADT Queue

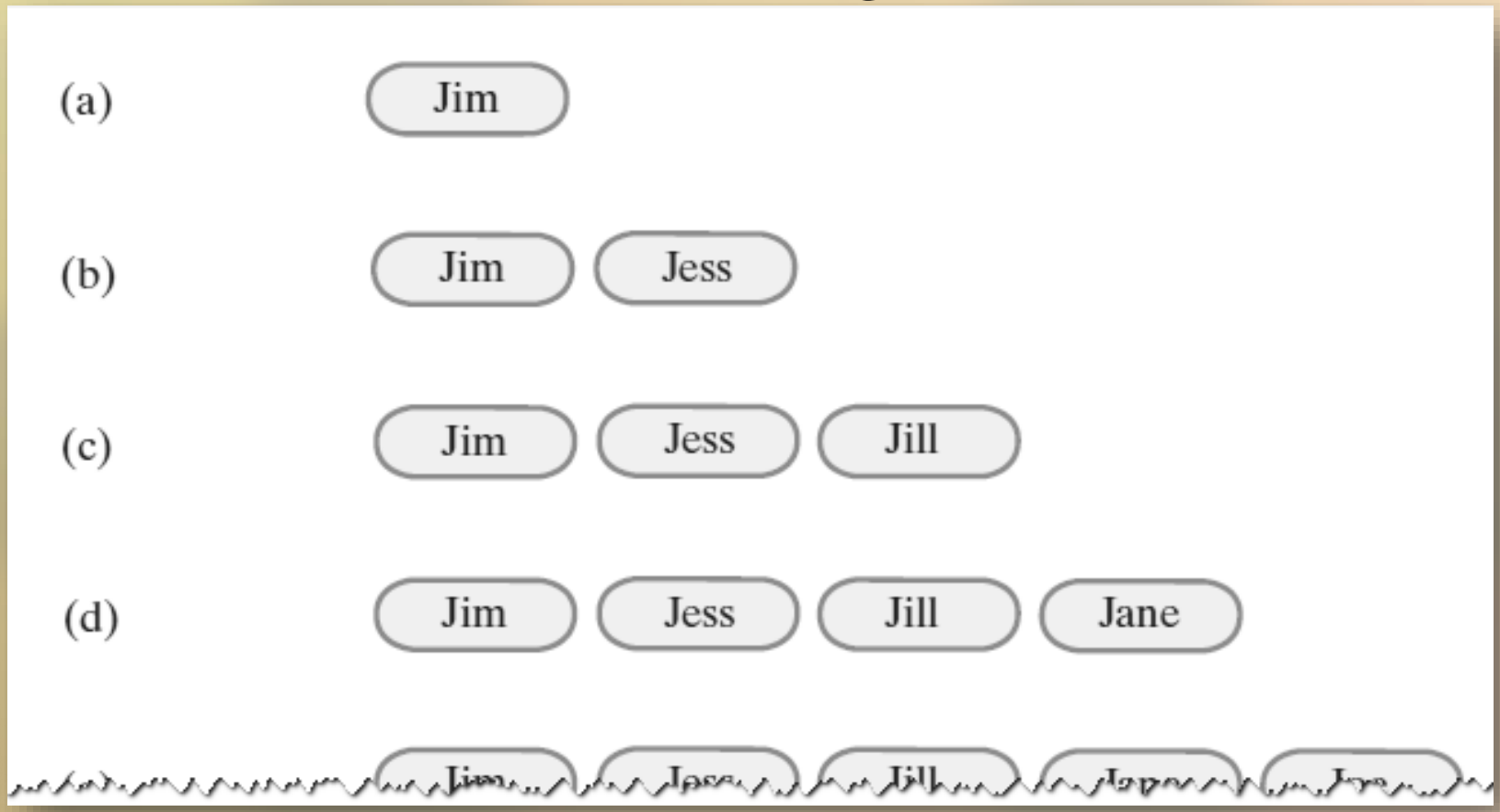


FIGURE 10-2 A queue of strings after (a) enqueue adds *Jim*; (b) enqueue adds *Jess*; (c) enqueue adds *Jill*; (d) enqueue adds *Jane*;

The ADT Queue

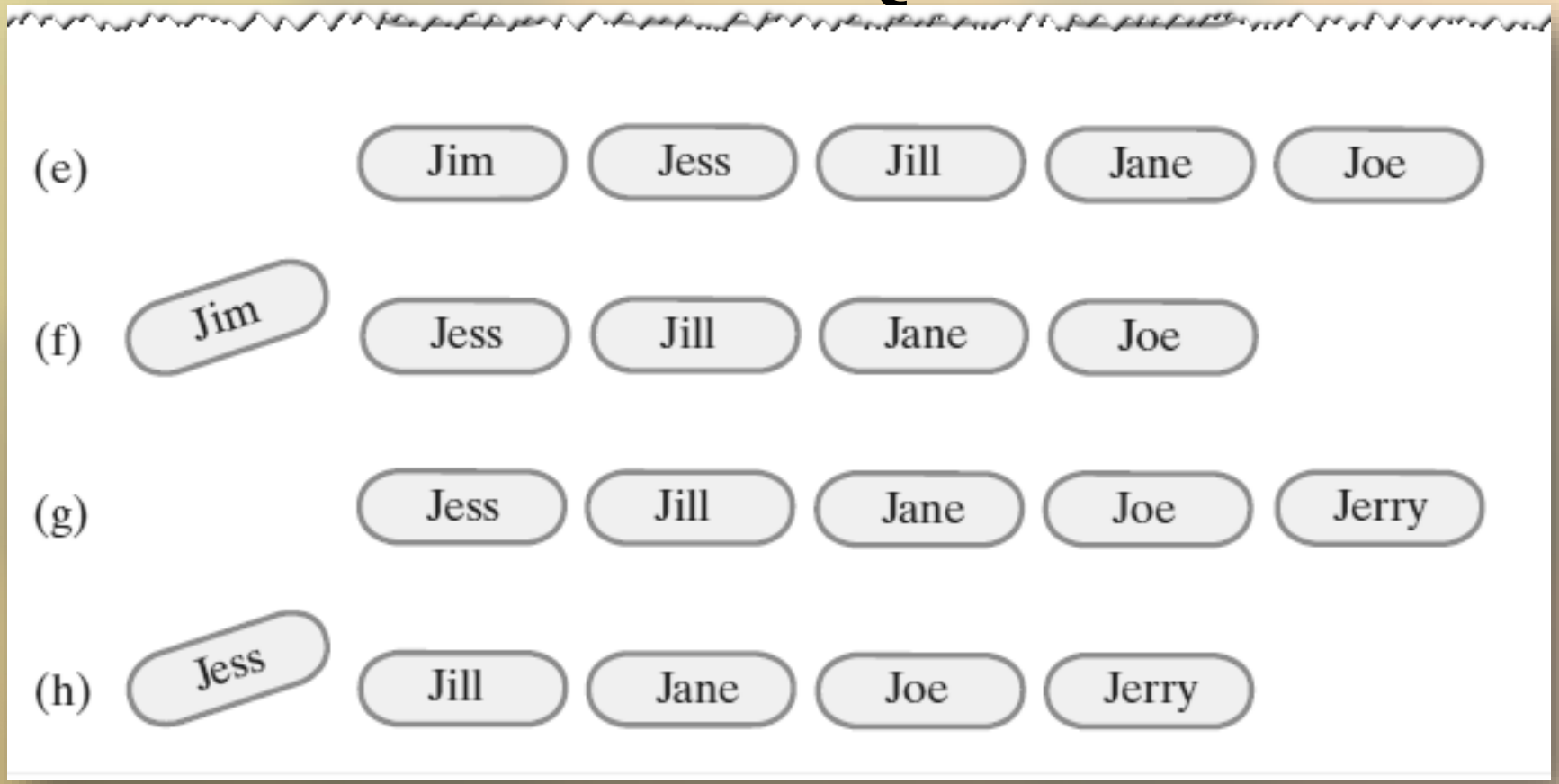


FIGURE 10-2 A queue of strings after (e) enqueue adds *Joe*; (f) dequeue retrieves and removes *Jim*; (g) enqueue adds *Jerry*; (h) dequeue retrieves and removes *Jess*

Simulating a Waiting Line

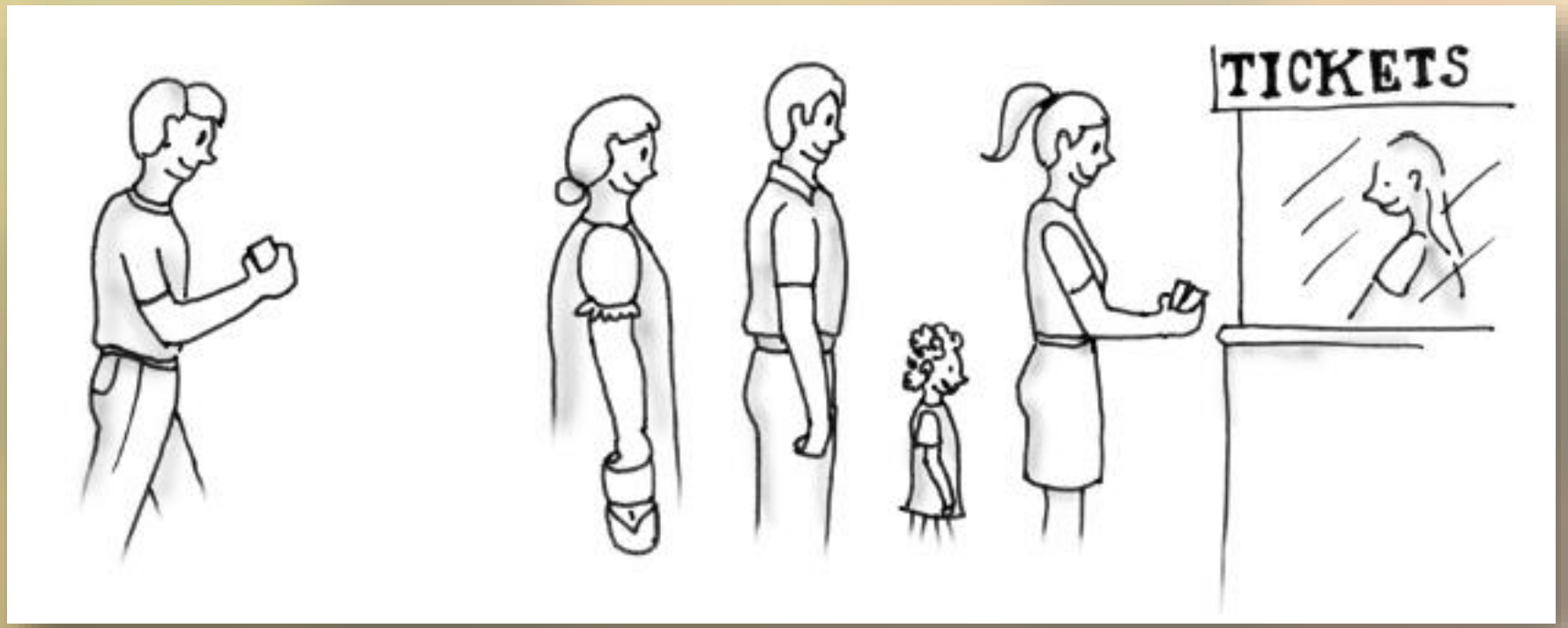


FIGURE 10-3 A line, or queue, of people

Simulating a Waiting Line

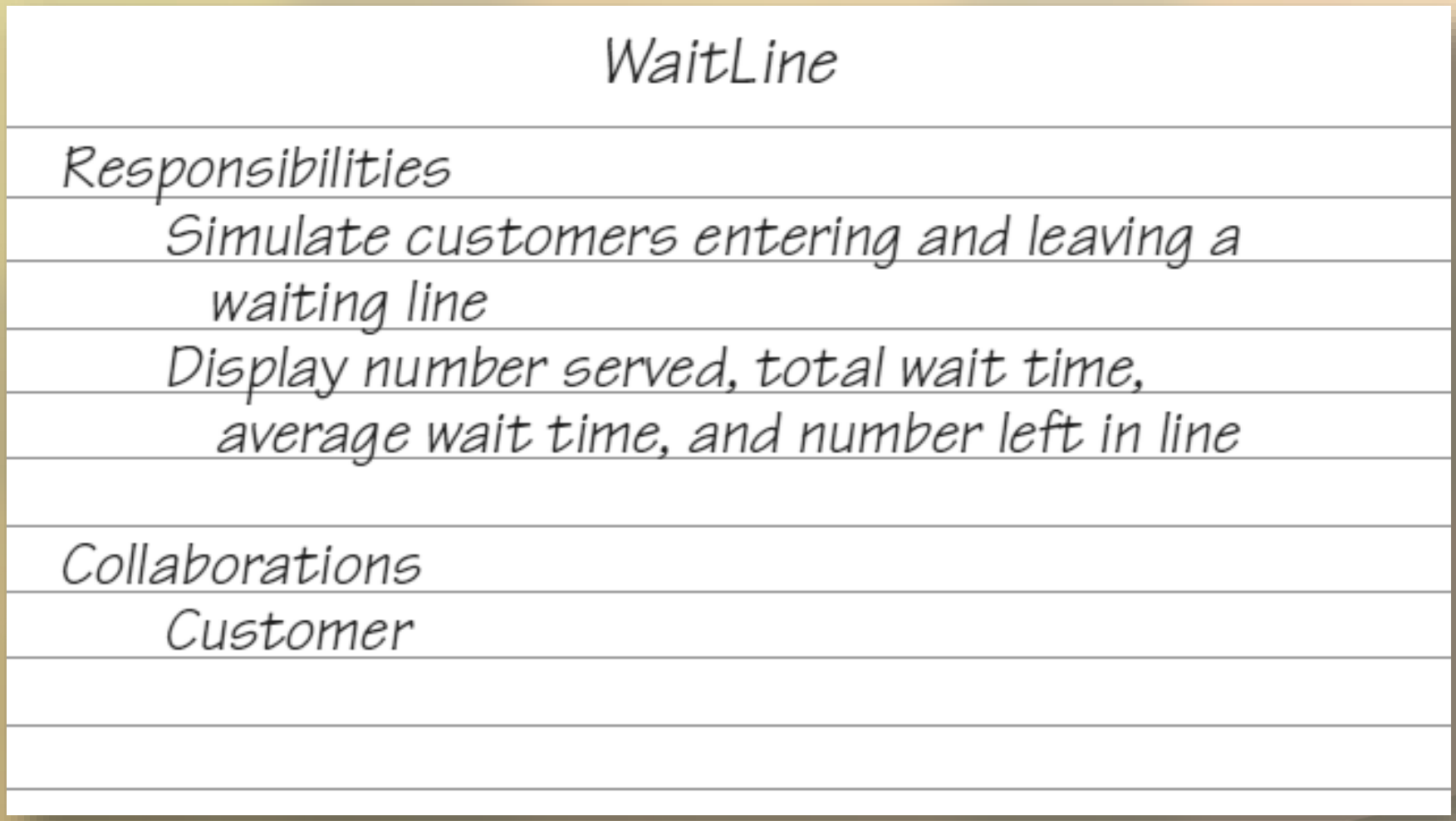


FIGURE 10-4 A CRC card for the class **`WaitLine`**

Simulating a Waiting Line

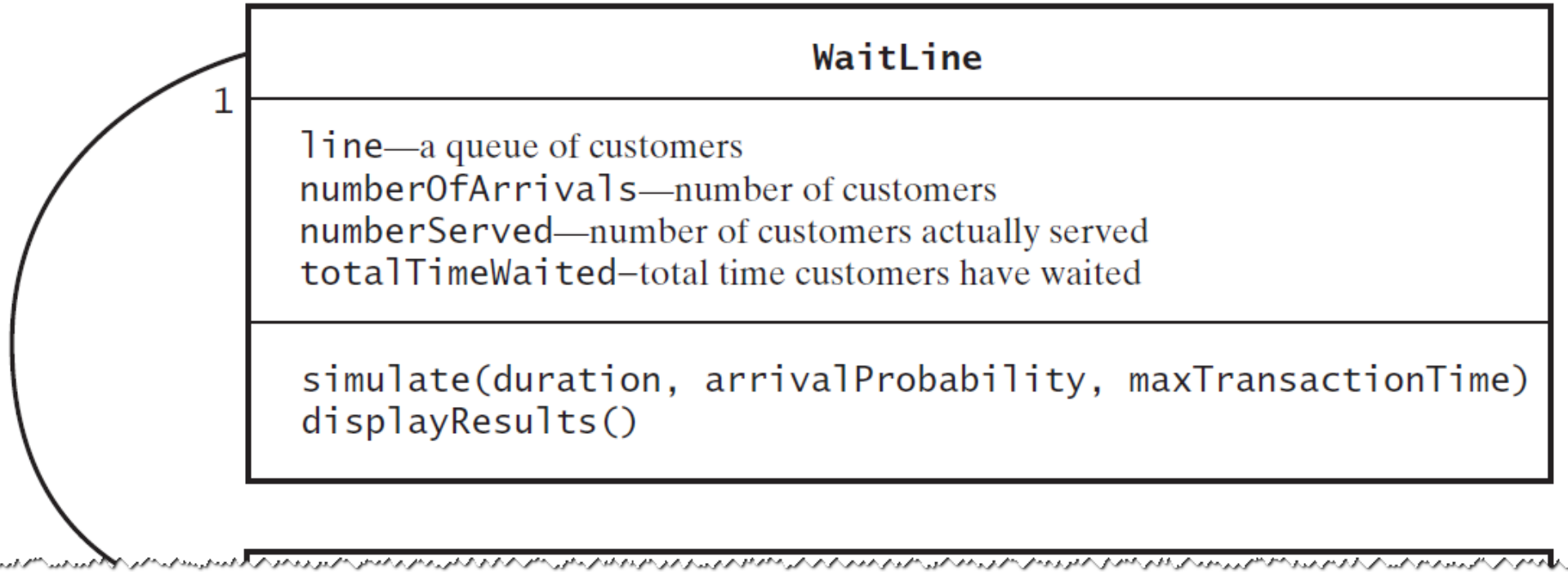


FIGURE 10-5 A diagram of the classes **WaitLine** and **Customer**

Simulating a Waiting Line

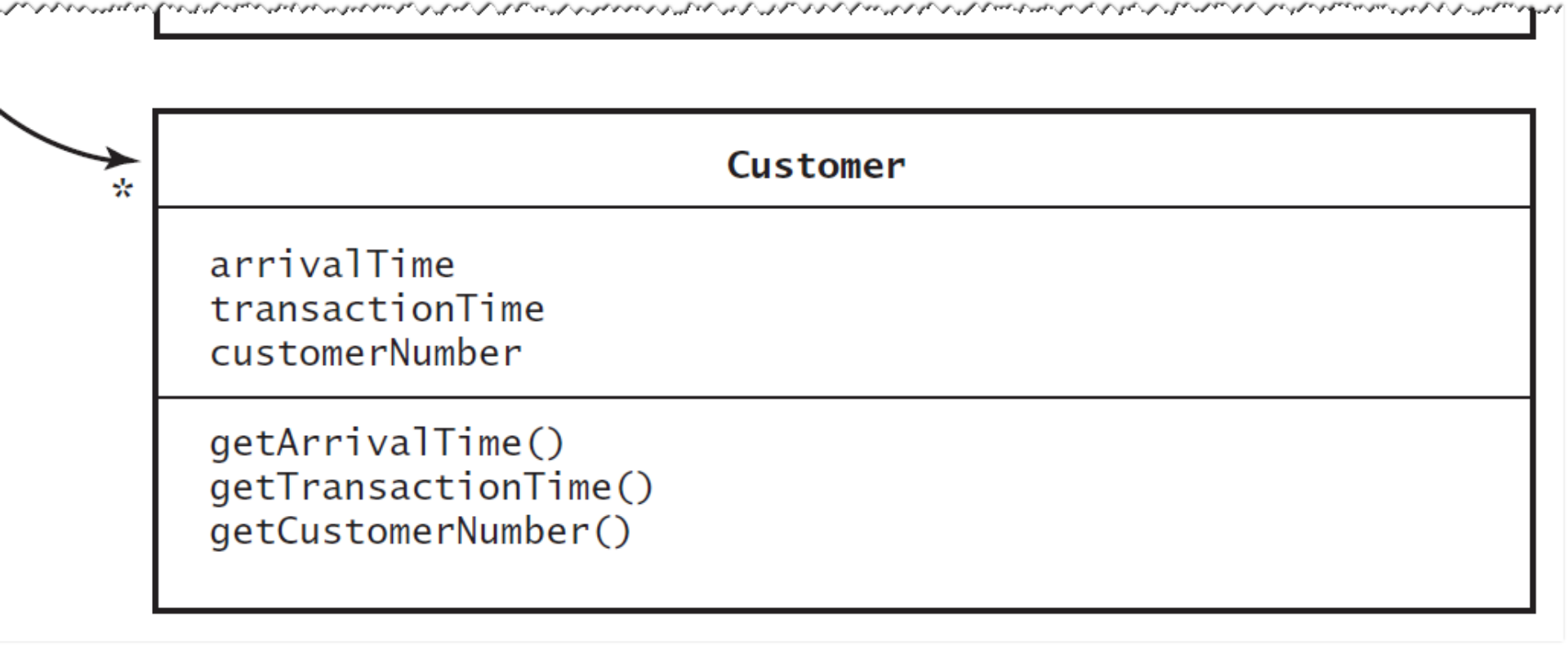
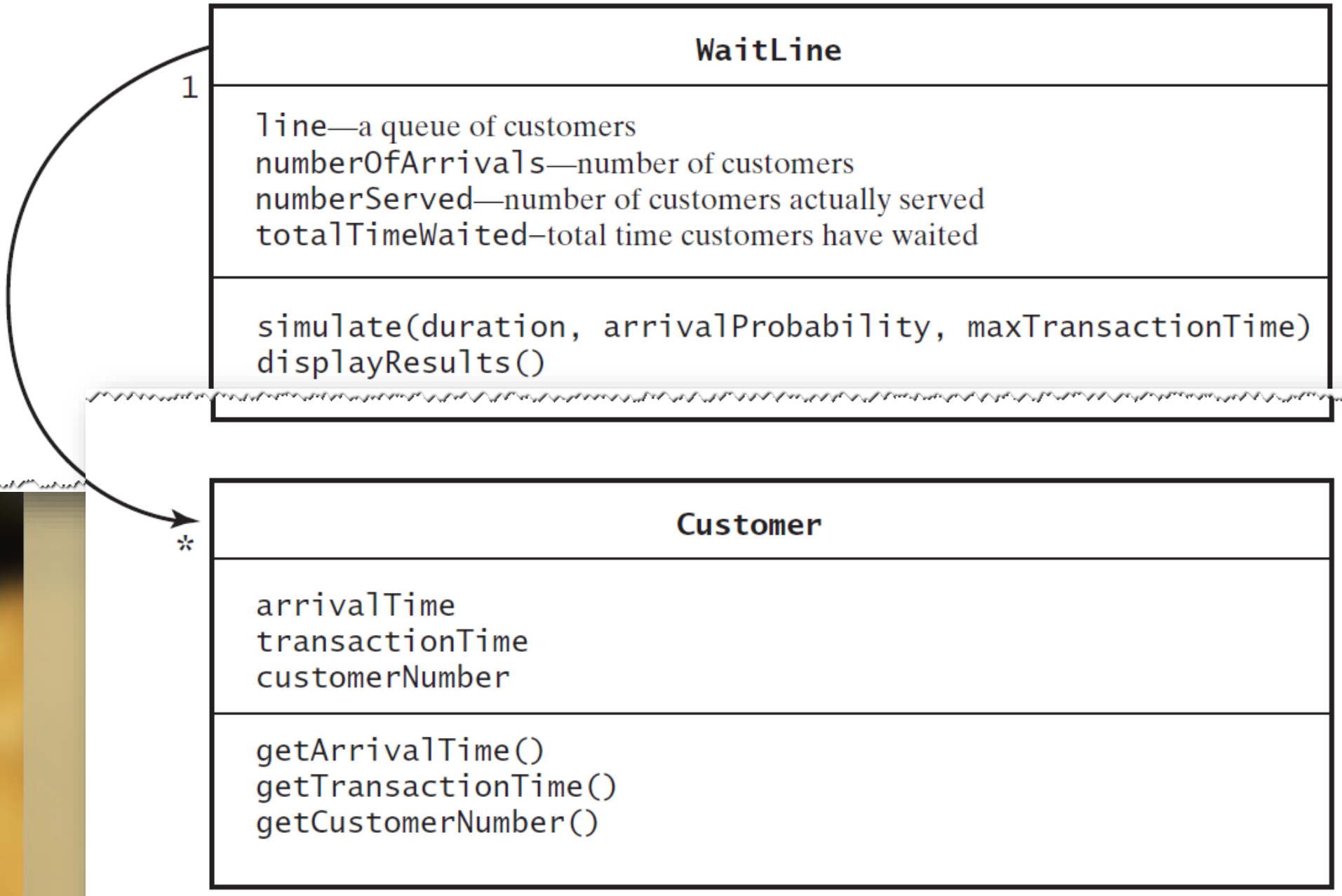
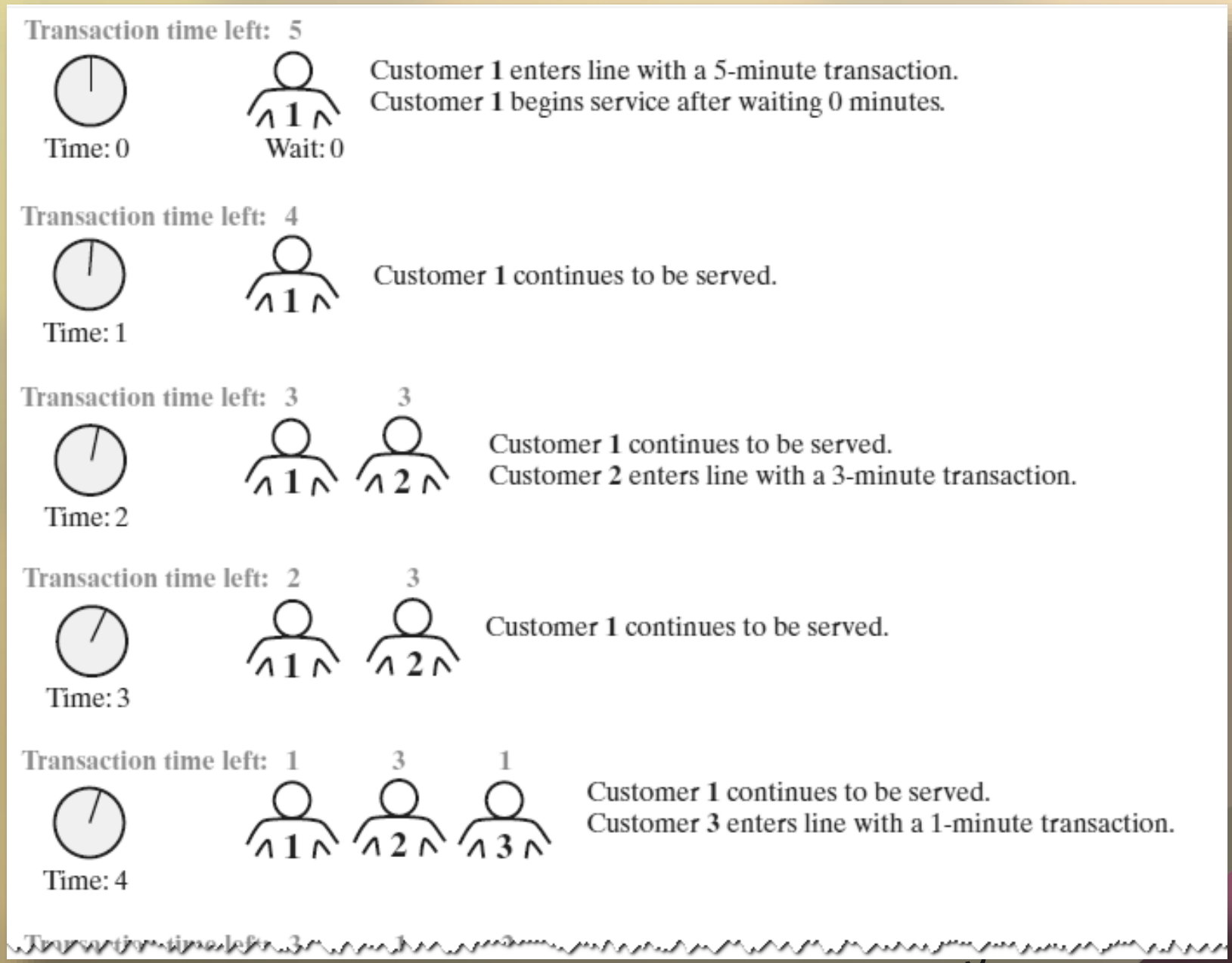


FIGURE 10-5 A diagram of the classes **`WaitLine`** and **`Customer`**



```
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++
   }
}
```

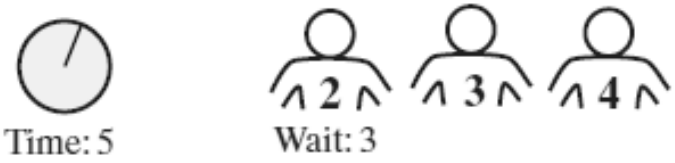
Simulating a Waiting Line



Simulating a Waiting Line

Time: 4

Transaction time left: 3 1 2



Time: 5 Wait: 3

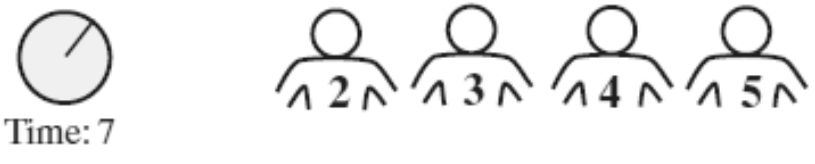
Customer **1** finishes and departs.
Customer **2** begins service after waiting 3 minutes.
Customer **4** enters line with a 2-minute transaction.

Transaction time left: 2 1 2

Time: 6

Customer **2** continues to be served.

Transaction time left: 1 1 2 4



Time: 7

Customer **2** continues to be served.
Customer **5** enters line with a 4-minute transaction.

Transaction time left: 1 2 4

Time: 8 Wait: 4

Customer **2** finishes and departs.
Customer **3** begins service after waiting 4 minutes.

Transaction time left: 2 4

Time: 9 Wait: 4

Customer **3** finishes and departs.
Customer **4** begins service after waiting 4 minutes.

Simulating a Waiting Line

```
1  /** Simulates a waiting line. */
2  public class WaitLine
3  {
4     private QueueInterface<Customer> line;
5     private int numberOfArrivals;
6     private int numberServed;
7     private int totalTimeWaited;
8
9     public WaitLine()
10    {
11       line = new LinkedQueue<>();
12       reset();
13    } // end default constructor
14
15    /** Simulates a waiting line with one serving agent.
16        @param duration  The number of simulated minutes.
17        @param arrivalProbability  A real number between 0 and 1, and the
18                                   probability that a customer arrives at
19                                   a given time.
20        @param maxTransactionTime  The longest transaction time for a
21                                   customer. */
22    public void simulate(int duration, double arrivalProbability,
23                         int maxTransactionTime)
```

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
```

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**

Simulating a Waiting Line

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.

Simulating a Waiting Line

```
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
```

Simulating a Waiting Line

```
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
```

Since this example uses random numbers, another execution of the Java statements likely will have different results.

Simulating a Waiting Line

```
64          System.out.println("Total time waited = " + totalTimeWaited);
65          double averageTimeWaited = ((double)totalTimeWaited) / numberServed;
66          System.out.println("Average time waited = " + averageTimeWaited);
67          int leftInLine = numberOfArrivals - numberServed;
68          System.out.println("Number left in line = " + leftInLine);
69       } // end displayResults
70
71       /** Initializes the simulation. */
72       public final void reset()
73       {
74          line.clear();
75          numberOfArrivals = 0;
76          numberServed = 0;
77          totalTimeWaited = 0;
78       } // end reset
79    } // end WaitLine
```

LISTING 10-2 The class **WaitLine**

Computing the Capital Gain in a Sale of Stock

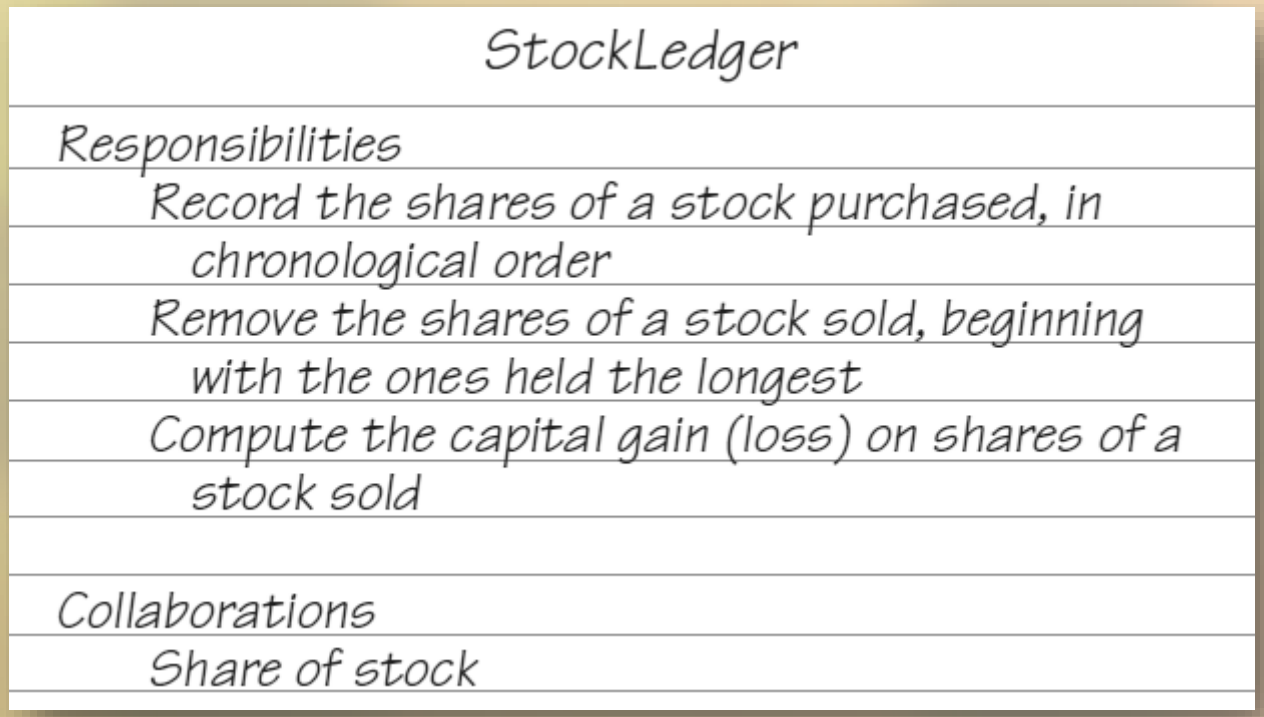


FIGURE 10-7 A CRC card for the class `StockLedger`

Computing the Capital Gain in a Sale of Stock

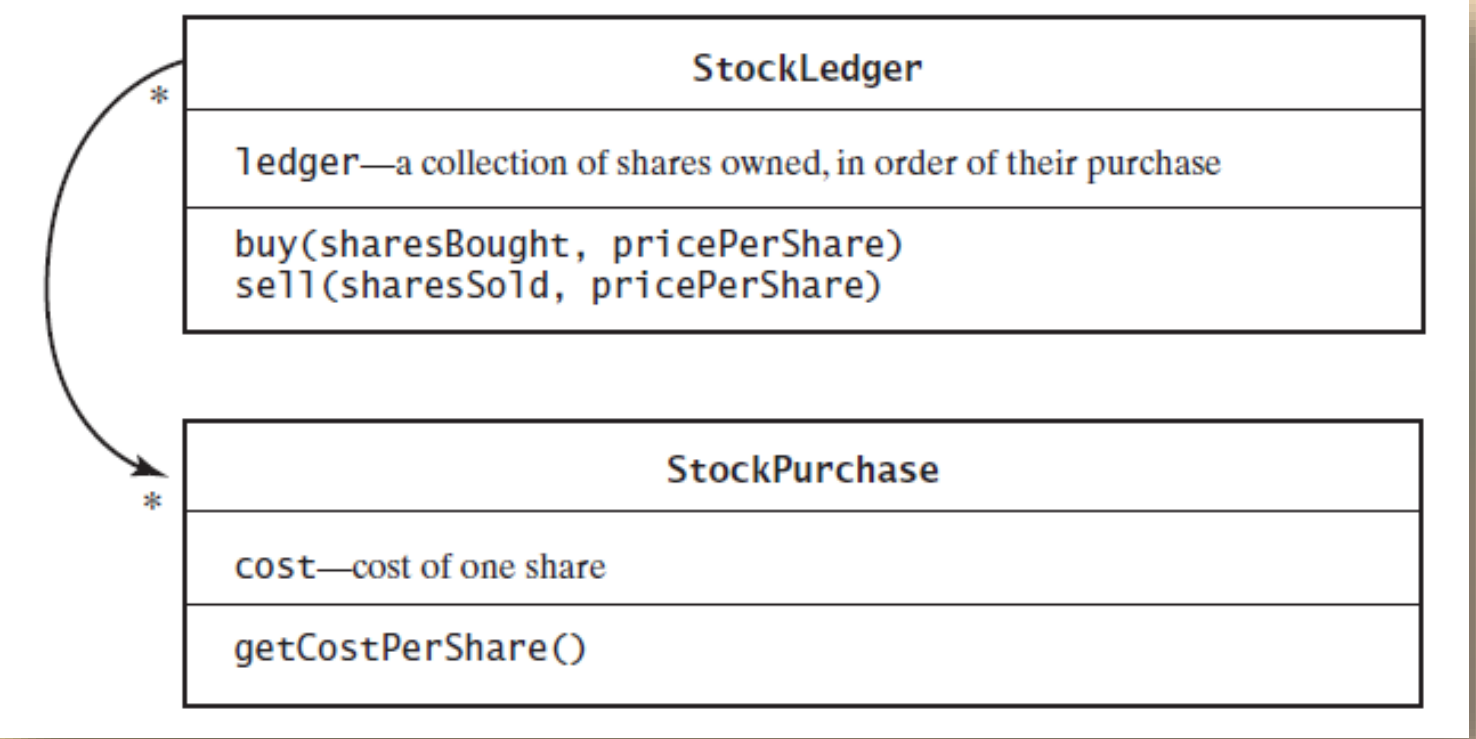


FIGURE 10-8 A diagram of the classes **StockLedger** and **StockPurchase**

Computing the Capital Gain in a Sale of Stock

```
1  /** A class that records the purchase and sale of stocks, and provides the
2      capital gain or loss. */
3  public class StockLedger
4  {
5     private QueueInterface<StockPurchase> ledger;
6
7     public StockLedger()
8     {
9        ledger = new LinkedQueue<>();
10    } // end default constructor
11
12    /** Records a stock purchase in this ledger.
13        @param sharesBought   The number of shares purchased.
14        @param pricePerShare  The price per share. */
15    public void buy (int sharesBought, double pricePerShare)
16    {
17       while (sharesBought > 0)
18       {
19          StockPurchase purchase = new StockPurchase(pricePerShare);
20          ledger.enqueue(purchase);
21          sharesBought--;
22       } // end while
23    } // end buy
24
```

LISTING 10-3 The class **StockLedger**

Computing the Capital Gain in a Sale of Stock

```
21            sharesBought--;
22         } // end while
23      } // end buy
24
25      /** Removes from this ledger any shares that were sold
26          and computes the capital gain or loss.
27          @param sharesSold     The number of shares sold.
28          @param pricePerShare  The price per share.
29          @return  The capital gain (loss). */
30      public double sell(int sharesSold, double pricePerShare)
31      {
32         double saleAmount = sharesSold * pricePerShare;
33         double totalCost = 0;
34
35         while (sharesSold > 0)
36         {
```

LISTING 10-3 The class **StockLedger**

Computing the Capital Gain in a Sale of Stock

```
28          @param pricePerShare  The price per share.
29          @return  The capital gain (loss). */
30       public double sell(int sharesSold, double pricePerShare)
31       {
32          double saleAmount = sharesSold * pricePerShare;
33          double totalCost = 0;
34
35          while (sharesSold > 0)
36          {
37             StockPurchase share = ledger.dequeue();
38             double shareCost = share.getCostPerShare();
39             totalCost = totalCost + shareCost;
40             sharesSold--;
41          } // end while
42
43          return saleAmount - totalCost; // Gain or loss
44       } // end sell
45    } // end StockLedger
```

LISTING 10-3 The class **StockLedger**

Computing the Capital Gain in a Sale of Stock

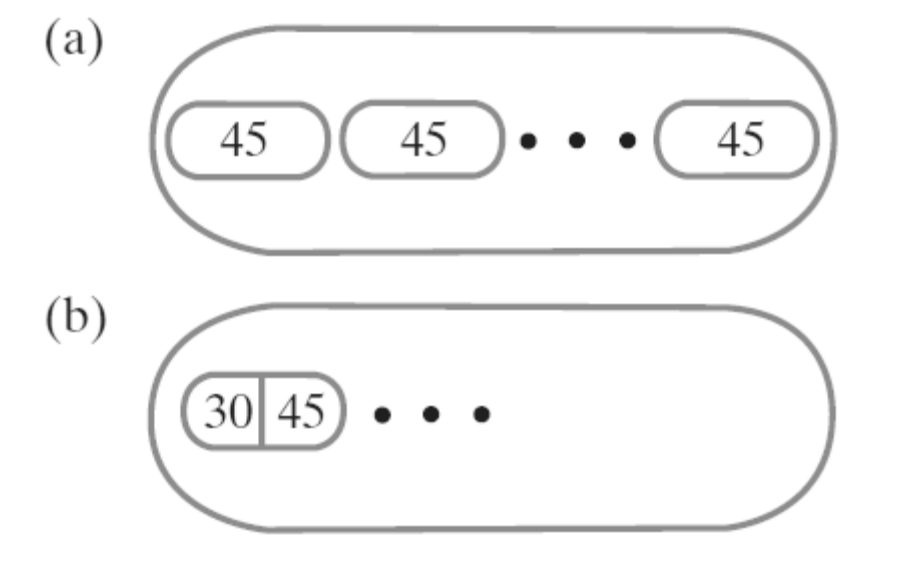


FIGURE 10-9 A queue of (a) individual shares of stock; (b) grouped shares

Java Class Library: The Interface `Queue`

Methods provided

- `add`
- `offer`
- `remove`
- `poll`
- `element`
- `peek`
- `isEmpty`
- `size`

Java Class Library: The Interface `Queue`

`public boolean add(T newEntry)`
Adds a new entry to the back of this queue, returning true if successful and throwing an exception if not.

`public boolean offer(T newEntry)`
Adds a new entry to the back of this queue, returning true or false according to the success of the operation.

`public T remove()`
Retrieves and removes the entry at the front of this queue, but throws `NoSuchElementException` if the queue is empty prior to the operation.

`public T poll()`
Retrieves and removes the entry at the front of this queue, but returns `null` if the queue is empty prior to the operation.

Java Class Library: The Interface Queue

public T element()
Retrieves the entry at the front of this queue, but throws NoSuchElementException if the queue is empty. Our method getFront throws an EmptyQueueException instead of a NoSuchElementException.

public T peek()
Retrieves the entry at the front of this queue, but returns null if the queue is empty.

public boolean isEmpty()
Detects whether this queue is empty.

public void clear()
Removes all entries from this queue.

public int size()
Gets the number of elements currently in this queue.

The ADT Deque

- A double ended queue
- *Deque* (not *dequeue*) pronounced “deck”
- Has both queue like operations and stack like operations

The ADT Deque

- Imagine that you are in a line at the post office.
- When it is finally your turn, the postal agent asks you to fill out a form.
- You step aside to do so and let the agent serve the next person in the line.
- After you complete the form, the agent will serve you next.

- Essentially, ***you go to the front of the line***, rather than waiting in line twice.

The ADT Deque

- Similarly, suppose that you join a line at its end.
- Then decide that it is too long, so you leave it.

- To simulate both of these examples, you want an ADT whose operations enable you
  - to add to the front
  - to remove from the back

  of a queue.

The ADT Deque

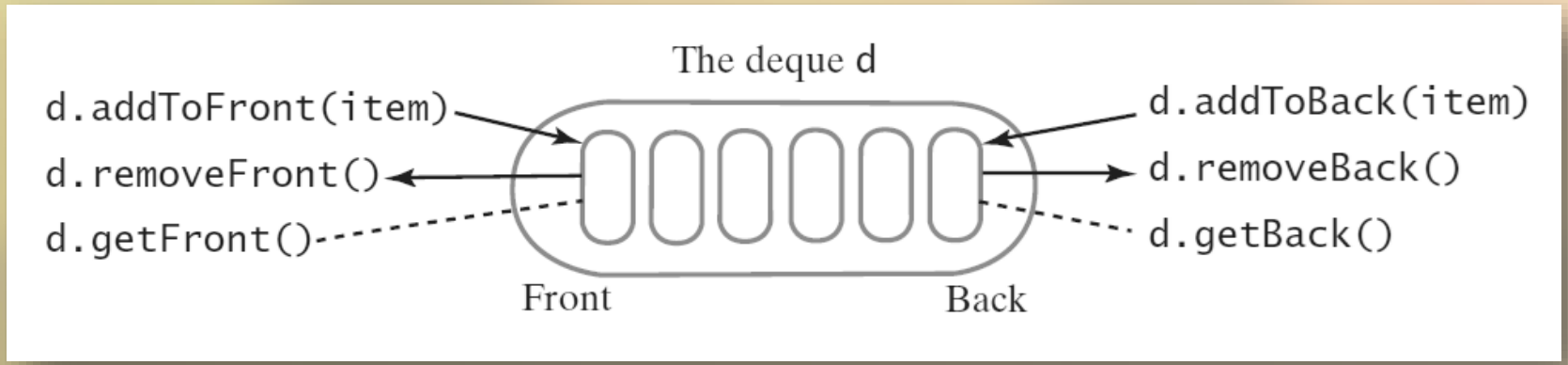


FIGURE 10-10 An instance *d* of a deque

- Although the ADT deque is called a double-ended queue, it actually behaves like a double-ended stack.
- You can push, pop, or get items at either of its ends.

The ADT Deque

```
1  /**
2     An interface for the ADT deque.
3     @author Frank M. Carrano
4  */
5  public interface DequeInterface<T>
6  {
7     /** Adds a new entry to the front/back of this deque.
8         @param newEntry  An object to be added. */
9     public void addToFront(T newEntry);
10    public void addToBack(T newEntry);
11
12    /** Removes and returns the front/back entry of this deque.
13        @return  The object at the front/back of the deque.
14        @throws  EmptyQueueException if the deque is empty before the
15                 operation. */
16    public T removeFront();
17    public T removeBack();
18
19    /** Retrieves the front/back entry of this deque.
```

LISTING 10-4 An interface for the ADT deque

The ADT Deque

```
16    public T removeFront();
17    public T removeBack();
18
19    /** Retrieves the front/back entry of this deque.
20        @return  The object at the front/back of the deque.
21        @throws  EmptyQueueException if the deque is empty. */
22    public T getFront();
23    public T getBack();
24
25    /** Detects whether this deque is empty.
26        @return  True if the deque is empty, or false otherwise. */
27    public boolean isEmpty();
28
29    /* Removes all entries from this deque. */
30    public void clear();
31 } // end DequeInterface
```

LISTING 10-4 An interface for the ADT deque

The ADT Deque

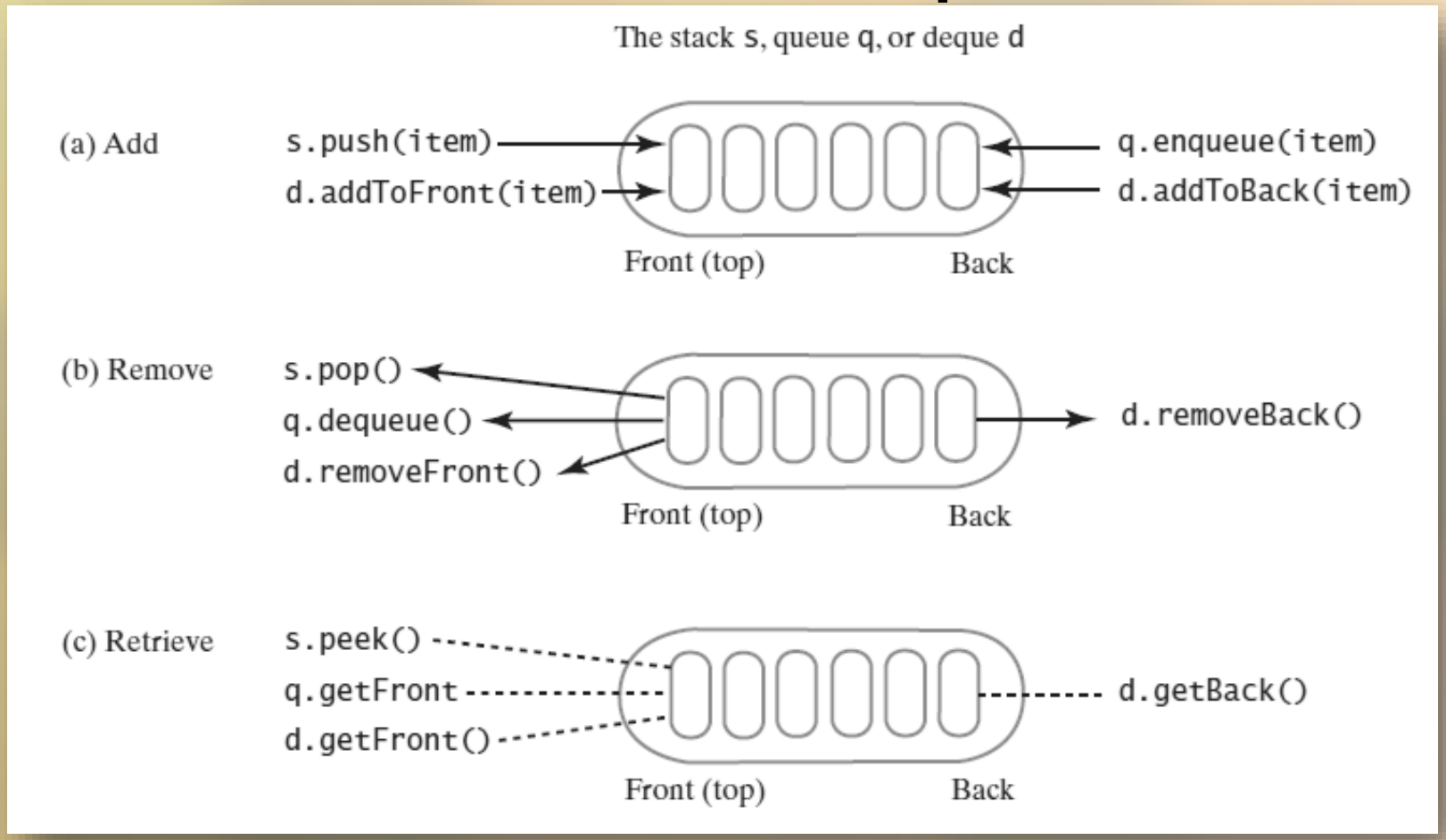


FIGURE 10-11 A comparison of operations for a stack *s*, a queue *q*, and a deque *d*: (a) add; (b) remove; (c) retrieve

The ADT Deque

```
// Read a line
d = a new empty deque
while (not end of line)
{
   character = next character read
   if (character == ←)
      d.removeBack()
   else
      d.addToBack(character)
}
// Display the corrected line
while (!d.isEmpty())
   System.out.print(d.removeFront())
System.out.println()
```

Pseudocode that uses a deque to read and display a line of keyboard input

Computing the Capital Gain in a Sale of Stock

```
public void buy(int sharesBought, double pricePerShare)
{
   StockPurchase purchase = new StockPurchase(sharesBought, pricePerShare);
   ledger.addToBack(purchase);
} // end buy
```

Method **buy** creates an instance of **StockPurchase** and places it at the back of the deque

Computing the Capital Gain in a Sale of Stock

```
public double sell(int sharesSold, double pricePerShare)
{
   double saleAmount = sharesSold * pricePerShare;
   double totalCost = 0;

   while (sharesSold > 0)
   {
      StockPurchase transaction = ledger.removeFront();
      double shareCost = transaction.getCostPerShare();
      int numberOfShares = transaction.getNumberOfShares();
      if (numberOfShares > sharesSold)
      {
         totalCost = totalCost + sharesSold * shareCost;
         int numberToPutBack = numberOfShares - sharesSold;
         StockPurchase leftOver = new StockPurchase(numberToPutBack,
```

The method **`sell`** is more involved

Computing the Capital Gain in a Sale of Stock

```
            int numberToPutBack = numberOfShares - sharesSold;
            StockPurchase leftOver = new StockPurchase(numberToPutBack,
                                                       shareCost);
            ledger.addToFront(leftOver); // Return leftover shares
            // Note: loop will exit since sharesSold will be <= 0 later
         }
         else
            totalCost = totalCost + numberOfShares * shareCost;

         sharesSold = sharesSold - numberOfShares;
      } // end while
      return saleAmount - totalCost; // Gain or loss
   } // end sell
```

The method **`sell`** is more involved

Java Class Library: The Interface `Deque`

Methods provided

- `addFirst, offerFirst`
- `addLast, offerLast`
- `removeFirst, pollFirst`
- `removeLast, pollLast`
- `getFirst, peekFirst`
- `getLast, peekLast`
- `isEmpty, clear, size`
- `push, pop`

Java Class Library: The Class `ArrayDeque`

- Implements the interface `Deque`
- Constructors provided
  - `ArrayDeque()`
  - `ArrayDeque(int initialCapacity)`

ADT Priority Queue

- Consider how a hospital assigns a priority to each patient that *overrides* time at which patient arrived.
- ADT priority queue organizes objects according to their priorities
- Definition of “priority” depends on nature of the items in the queue

ADT Priority Queue

```
1  public interface PriorityQueueInterface<T extends Comparable<? super T>>
2  {
3     /** Adds a new entry to this priority queue.
4         @param newEntry  An object to be added. */
5     public void add(T newEntry);
6
7     /** Removes and returns the entry having the highest priority.
8         @return  Either the object having the highest priority or, if the
9                  priority queue is empty before the operation, null. */
10    public T remove();
11
12    /** Retrieves the entry having the highest priority.
13        @return  Either the object having the highest priority or, if the
14                 priority queue is empty, null. */
15    public T peek();
```

LISTING 10-5 An interface for the ADT priority queue

ADT Priority Queue

```
12      /** Retrieves the entry having the highest priority.
13          @return  Either the object having the highest priority or, if the
14                   priority queue is empty, null. */
15      public T peek();
16
17      /** Detects whether this priority queue is empty.
18          @return  True if the priority queue is empty, or false otherwise. */
19      public boolean isEmpty();
20
21      /** Gets the size of this priority queue.
22          @return  The number of entries currently in the priority queue. */
23      public int getSize();
24
25      /** Removes all entries from this priority queue. */
26      public void clear();
27  }   // end PriorityQueueInterface
```

LISTING 10-5 An interface for the ADT priority queue

Tracking Your Assignments

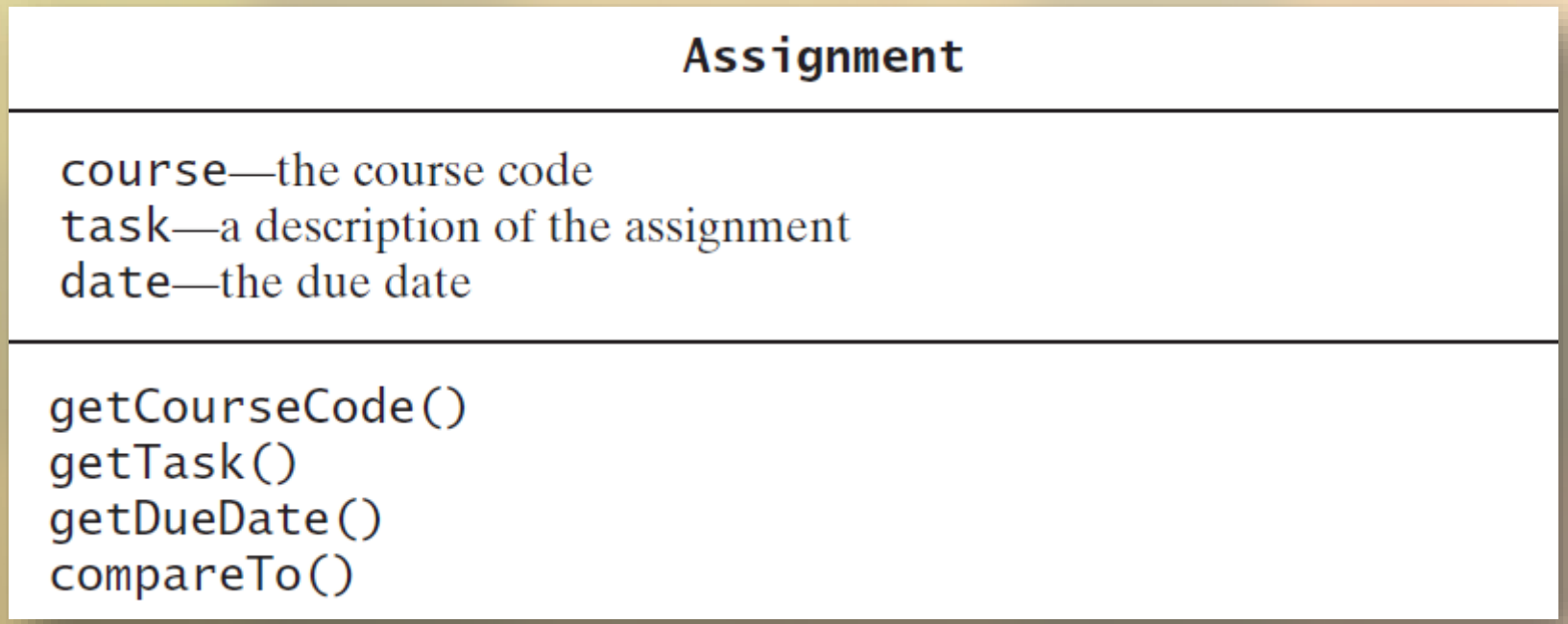


FIGURE 10-12 A diagram of the class **`Assignment`**

Tracking Your Assignments

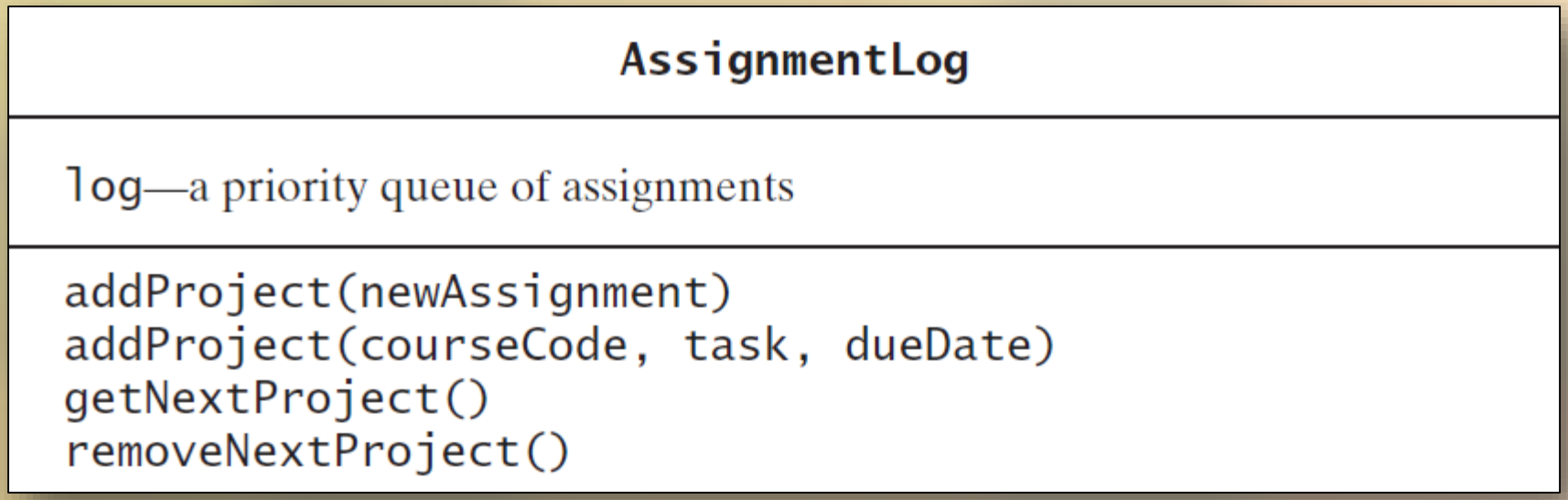


FIGURE 10-13 A diagram of the class **AssignmentLog**

Tracking Your Assignments

```
1  import java.sql.Date;
2  public class AssignmentLog
3  {
4     private PriorityQueueInterface<Assignment> log;
5
6     public AssignmentLog()
7     {
8        log = new PriorityQueue<>();
9     } // end constructor
10
11    public void addProject(Assignment newAssignment)
12    {
13       log.add(newAssignment);
14    } // end addProject
15
```

LISTING 10-6 The class **AssignmentLog**

Tracking Your Assignments

```
16    public void addProject(String courseCode, String task, Date dueDate)
17    {
18       Assignment newAssignment = new Assignment(courseCode, task, dueDate);
19       addProject(newAssignment);
20    } // end addProject
21
22    public Assignment getNextProject()
23    {
24       return log.peek();
25    } // end getNextProject
26
27    public Assignment removeNextProject()
28    {
29       return log.remove();
30    } // end removeNextProject
31 } // end AssignmentLog
```

LISTING 10-6 The class **AssignmentLog**

Java Class Library: The Class `PriorityQueue`

Basic constructors and methods

- `PriorityQueue`
- `add`
- `offer`
- `remove`
- `poll`
- `element`
- `peek`
- `isEmpty`, `clear`, `size`

End

Chapter 10

End

Chapter 10