CHAPTER 21
Stacks and Queues

TOPICS

21.1 Stacks and Their Applications

CONCEPT: A stack is a collection of items that allows addition and removal of items in a last-in-first-out manner.

In Chapter 18 you learned about linked lists, array lists, vectors, and other types of collection classes that are part of the Java Collection Framework. The `List` classes you learned about in that chapter are very general, and allow elements to be added and removed at any position in the list. There are many applications that use lists of objects in more restricted ways. For example, an application may need to maintain a list of objects that requires additions and removals only at the ends, or even at only one end of the list. In a list that requires additions and removals to take place at only one end, the only item available for removal is the item that was last added. Such a list is called a *stack*, and the end of the list at which additions and removals take place is called the *top of the stack.*

Examples and Applications of Stacks

There are many examples of stacks in real life. For example, when a number of cars are parked single file in a narrow driveway, the last car parked must be the first one to leave. Another example is a stack of plates in a cafeteria. Workers add plates to the top of the stack, and each patron removes a plate from the top of the stack.

There are many problems, both in computer science and out of it, that can be solved with the use of stacks. For example, all programming languages allow programmers to define methods and call them

from different places within the program. The computer must keep track of the *return address* for each method call, that is, the place in the program from which the method was called. The computer then uses this return address when the method executes a return statement. Whenever a method is called, the computer adds its return address to a stack. Later, when the method executes a return, the computer removes an address from the stack and uses it as the target address for the return statement. The stack is the right data structure to use because the last method called is the first to return.

Stack Operations

A stack can be viewed as an abstract data type that supports three main operations, traditionally called push, peek, and pop. The *push* operation takes a single item and adds it to the top of the stack. The *peek* operation returns the item currently at the top of the stack, but does not remove it. The *pop* operation removes (and returns) the item currently stored at the top of the stack.

The Java Collection Framework provides a `Stack` class that supports generic types. The class is in the `java.util` package. Table 21-1 lists most of its methods.

Table 21-1 Stack class methods

`Stack<E>()`	Constructor.
`E push(E item)`	Adds `item` to the top of the stack and returns the item added.
`E pop()`	Removes and returns the item at the top of the stack. If the stack is empty, `pop` throws `EmptyStackException`.
`E peek()`	Returns the item that is currently at the top of the stack, but does not remove it from the stack. If the stack is empty, `peek` throws `EmptyStackException`.
`boolean empty()`	Returns `true` if this stack is empty and `false` otherwise.

Code Listing 21-1 is a simple demonstration of the use of the `Stack` class. The program creates a stack and pushes `String` objects onto it. Then it keeps popping the stack and printing the values popped until the stack is empty. Notice that the last-in-first-out nature of the stack causes the strings to be popped in the reverse of the order in which they were pushed.

Code Listing 21-1 (`StackDemo1.java`)

```
1  import java.util.*;
2
3  /**
4     This program demonstrates the java.util.Stack class.
5  */
6
7  public class StackDemo1
8  {
9     public static void main(String [] args)
10    {
11       // Create a stack of strings and add some names
12       Stack<String> stack = new Stack<String>();
13       String [ ] names = {"Al", "Bob", "Carol"};
14       System.out.println("Pushing onto the stack the names:");
15       System.out.println("Al Bob Carol");
16       for (String s : names)
17          stack.push(s);
18
19       // Now pop and print everything on the stack
20       String message = "Popping and printing all stack values:";
21       System.out.println(message);
22       while (!stack.empty())
23          System.out.print(stack.pop() + " ");
24    }
25 }
```

Program Output

```
Pushing onto the stack the names:
Al Bob Carol
Popping and printing all stack values:
Carol Bob Al
```

Stacks of Primitive Types

The `Stack` class provided by the Java Collections Framework does not directly accept values of primitive type. To use it with a primitive type such as `int` or `double`, you must use the corresponding wrapper class. For example, to have a stack of `int`, you must create a stack of `Integer`:

```
Stack<Integer> intStack = new Stack<Integer>();
```

Then you can pass values of the appropriate primitive type to the `push` method, and the value will automatically be boxed as explained in Chapter 17. Likewise, popping a value from the stack and assigning to a variable of the appropriate primitive type will cause it to be unboxed. The use of stacks of primitive types is illustrated in the program of Code Listing 21-2. The program pushes some numbers onto a stack, then pops and prints them.

Code Listing 21-2 (`StackDemo2.java`)

```
1  import java.util.*;
2
3  /**
4     This program demonstrates the use
5     of stacks with primitive types.
6  */
7
8  public class StackDemo2
9  {
10    public static void main(String [] args)
11    {
12       Stack<Integer> intStack = new Stack<Integer>();
13
14       // Push some numbers onto the stack
15       for (int k = 1; k < 10; k++)
16          intStack.push(k*k);
17
18       // Pop and print all numbers
19       while (!intStack.empty())
20       {
21          int x = intStack.pop();
22          System.out.print( x + "   ");
23       }
24    }
25 }
```

Program Output

```
81 64 49 36 25 16 9 4 1
```

21.2 Array Implementation of Stacks

CONCEPT: A stack can be implemented by using an array to hold the stack items and an integer to keep track of the top of the stack.

A stack is just a list that enforces last-in-first-out access. Its implementation can be based on any of the types of list classes studied in Chapter 17: `ArrayList`, `LinkedList`, or `Vector`. In fact, implementation of a stack can be based on simple arrays. In this section and the next, we look at how arrays and linked lists can be used to implement stacks.

The idea is simple. You create an array large enough to hold the largest number of items you will need to have in the stack at any given time, say, four (the capacity of a stack in a real application would likely be much larger), and you use an integer index, `top`, to point to the next slot in the array that is available to receive an item. Thus, for a stack of integers, you would have:

```
int [ ] s = new int [4];     // Array holds stack elements
int top = 0;                 // Pointer to top of stack
```

The variable `top`, when used in this manner, is traditionally called the *stack top pointer*, or simply, the *stack pointer*.

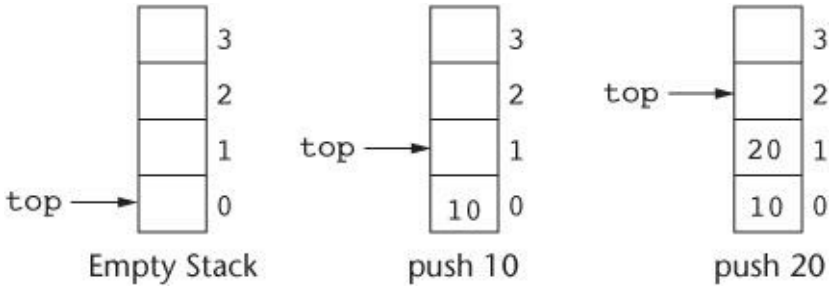
The Stack Push Operation

To push a new item `x`, first we check to see if there is room in the stack. If so, we place it in the slot pointed to by `top` and then increment `top` to point to the next slot. If the stack has no room, we throw an exception as follows:

```
if (top == s.length)
   throw new StackOverFlowException();
else
{
   // Add the new item to the stack and update top
   s[top] = x;
   top ++;
}
```

`StackOverFlowException` is not a Java exception: it is one that we have to define. Figure 21-1 illustrates a sequence of events that starts with an empty stack and adds two items.

Figure 21-1 Pushing items onto a stack



The Stack `empty` Method

The boolean expression `top == 0` is true when the stack is empty and false when it is not, so the `empty` method simply returns the value of that expression:

```
return top == 0;
```

The Stack `peek` and `pop` Methods

The two stack methods `peek` and `pop` are similar: both check to see if the stack is empty before attempting to access the item at the top of the stack, and throw `EmptyStackException` if the stack is empty. The `peek` method returns the item at the top of the stack as follows:

```
return s[top-1];
```

The `pop` method needs to return the same value as `peek`, but in addition, it needs to decrement the stack pointer to indicate that the stack slot currently holding the item at the top of the stack is becoming available. This is accomplished as follows:

```
top --;
return s[top];
```

The code we have outlined here will correctly implement a stack of integers. The stack class makes use of a separate file, *StackExceptions.java*, containing definitions of the following subclasses of the `RuntimeException` class. These two exception classes will be thrown by our stack methods.

```
class StackOverFlowException extends RuntimeException
{
}
class EmptyStackException extends RuntimeException
{
}
```

The code for the stack class is given next, as Code Listing 21-3.

Code Listing 21-3 (`ArrayStack.java`)

```
1  /**
2    Array Implementation of a stack.
3  */
4
5  public class ArrayStack
6  {
7     private int [] s; // Holds stack elements
8     private int top;  // Stack top pointer
9
10    /**
11       Constructor.
12       @param capacity The capacity of the stack.
13    */
14
15    public ArrayStack (int capacity)
16    {
17       s = new int[capacity];
18       top = 0;
19    }
20
21    /**
22       The empty method checks for an empty stack.
23       @return true if stack is empty.
24    */
25
```

```
26     public boolean empty()
27     {
28        return top == 0;
29     }
30
31     /**
32       The push method pushes a value onto the stack.
33       @param x The value to push onto the stack.
34       @exception StackOverflowException When the
35       stack is full.
36    */
37
38    public void push(int x)
39    {
40       if (top == s.length)
41          throw new StackOverFlowException();
42       else
43       {
44          s[top] = x;
45          top ++;
46       }
47    }
48
49    /**
50       The pop method pops a value off the stack.
```

```
51        @return The value popped.
52        @exception EmptyStackException When the
53        stack is empty.
54     */
55
56     public int pop()
57     {
58        if (empty())
59           throw new EmptyStackException();
60        else
61        {
62           top--;
63           return s[top];
64        }
65     }
66
67     /**
68        The peek method returns the value at the
69        top of the stack.
70        @return value at top of the stack.
71        @exception EmptyStackException When the
72        stack is empty.
73     */
74
75     int peek()
76     {


77        if (empty())
78           throw new EmptyStackException();
79        else
80         {
81            return s[top-1];
82         }
83      }
84  }
```

The `ArrayStackDemo` class, shown in Code Listing 21-4, demonstrates the use of this stack. It has a main method that creates a stack with capacity 5, stores some integers on it, and then pops and prints them.

Code Listing 21-4 (`ArrayStackDemo.java`)

```
1  /**
2     This class demonstrates the
3     use of the ArrayStack class.
4  */
5
6  public class ArrayStackDemo
7  {
8     public static void main(String [] arg)
9     {
10       String str;  // Use for output
11       ArrayStack  st = new ArrayStack(5);
12       str = "Pushing 10 20 onto the stack.";
13       System.out.println(str);
14       st.push(10);
15       st.push(20);
16       str = "Value at top of the stack is ";
17       System.out.println(str + st.peek());
18       str = "Popping and printing all values:";
19       System.out.println(str);
20       while (!st.empty())
21          System.out.print(st.pop() + " ");
22    }
23 }
```

Program Output

```
Pushing 10 20 onto the stack.
Value at top of the stack is 20
Popping and printing all values:
20 10
```

Stacks of Objects

The stack in Code Listing 21-3 is a stack of integers. Writing a stack that works with other types is similar. For example, to write a stack that works with `String` objects, we would modify the `ArrayStack` class by changing all occurrences of `int` that refer to the type of the items being stored in the stack to `String`. In particular, the parameter and return types of `peek`, `pop`, and `push` would change as follows:

```
int peek()            becomes      String peek()
int pop()             becomes      String pop()
void push(int x)      becomes       void push(String x)
```

The type of array that holds the stack items also needs to change as follows:

```
int [ ] s;              becomes      String [ ] s;
s = new int[capacity]   becomes      s = new String[capacity];
```

One other change must be made to the `pop` method when the stack is being used to store object types:

the array entry referring to the stack item being removed must be set to `null` to facilitate garbage collection of the item being removed from the stack as follows:

```
s[top-1] = null;
```

For example, the `pop` method modified to work with string objects would look like this:

```
public String pop()
{
   if (empty())
      throw new EmptyStackException();
   else
   {
      String retVal = s[top-1];
      s[top-1] = null; // Facilitate garbage collection
      top--;
      return retVal;
   }
}
```

Later in this chapter we will look at how we can use generic types to write a single stack class that works with values of all types.

A disadvantage of using an array implementation is the need to fix the size of the array at the time the stack is created. This causes a problem if the application using the stack needs to store more items than the size of the stack will allow. Basing the implementation of an array on a linked list avoids this problem.

21.3 Linked Implementation of Stacks

CONCEPT: A stack can be implemented by using a singly-linked list to hold the stack items, and having the head of the list serve as the top of the stack.

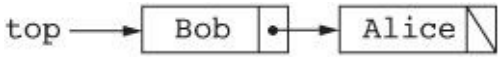
A stack can be implemented by using any type of list object, including the `LinkedList` class in the `java.util` package, to hold the stack items. The `LinkedList` class, however, supports many operations that are not needed when all we need is a stack: for example, most applications of stacks do not need iterators, or the ability to add and remove items from the middle of the list. We can avoid this additional overhead by basing a stack on a simple linked list that we code ourselves.

The linked list will be based on the following `Node` class, declared private inside a `LinkedStack` class:

```
private class Node
{
   String value;
   Node next;
   Node(String val, Node n)
   {
      value = val;
      next = n;
   }
}
```

Inside the `LinkedStack` class, we will have a reference to a linked list of items. A reference `top` will point to the node at the front of the list: this node will be the one removed by the next call to `pop`. For example, starting with an empty stack and pushing the strings `"Alice"` and `"Bob"` in succession will result in the scenario shown in Figure 21-2.

Figure 21-2 A linked stack after pushing two items



A further operation of pushing `"Carol"` on the stack would give the result shown in Figure 21-3:

Figure 21-3 The linked stack of Figure 21-2 after push of the string `"Carol"`

At this point, executing a pop operation would remove `"Carol"` from the stack and return us to the scenario shown in Figure 21-2.

Implementation of Stack Methods

The `LinkedStack` class will have a field `top` that points to the item at the top of the stack and simultaneously serves as the head of the linked list. The stack will be empty whenever the expression

```
top == null
```

is true, so the `empty` method simply returns the value of that expression:

```
public boolean empty()
{
   return top == null;
}
```

The `push` operation adds a new item `s` to the front of the list:

```
top = new Node(s, top);
```

The `peek` method returns the value field of the node at the front of the list:

```
return top.value;
```

The `pop` method saves the value in the node at the top, removes the top node from the list by moving the head pointer past it, and returns the saved value:

```
String retValue = top.value;
top = top.next;
return retValue;
```

Both `peek` and `pop` throw an exception if they are called on an empty stack. Finally, the `LinkedStack` class is equipped with a `toString` method that uses a `StringBuilder` object to create a string representation of the class. Having this method allows the contents of the entire stack `st` to be displayed by passing it to the `System.out.print` method:

```
System.out.println(st);
```

The `LinkedStack` class can be seen in Code Listing 21-5.

Code Listing 21-5 (`LinkedStack.java`)

```
1  /*
2     This program demonstrates how to write and
3     use a stack class based on linked lists.
4  */
5
6  class LinkedStack
7  {
8     /**
9        The Node class is used to implement the
10       linked list.
11    */
12
13    private class Node
14    {
15       String value;
16       Node next;
17       Node(String val, Node n)
18       {
19          value = val;
20          next = n;
21       }
22    }
23
24    private Node top = null;  // Top of the stack
```

```
25
26      /**
27         The empty method checks for an empty stack.
28         @return true if stack is empty, false otherwise.
29      */
30
31      public boolean empty()
32      {
33         return top == null;
34      }
35
36      /**
37         The push method adds a new item to the stack.
38         @param s The item to be pushed onto the stack.
39      */
40
41      public void push(String s)
42      {
43         top = new Node(s, top);
44      }
45
46      /**
47         The Pop method removes the value at the
48         top of the stack.
49         @return The value at the top of the stack.
```

```
50          @exception EmptyStackException When the
51          stack is empty.
52       */
53
54       public String pop()
55       {
56          if (empty())
57             throw new EmptyStackException();
58          else
59          {
60             String retValue = top.value;
61             top = top.next;
62             return retValue;
63          }
64       }
65
66       /**
67          The peek method returns the top value
68          on the stack.
69          @return The value at the top of the stack.
70          @exception EmptyStackException When the
71          stack is empty.
72       */
73
74       public String peek()
75       {


76          if (empty())
77             throw new EmptyStackException();
78          else
79             return top.value;
80       }
81
82       /**
83          The toString method computes a string
84          representation of the contents of the stack.
85          @return The string representation of the
86          stack contents.
87       */
88
89       public String toString()
90       {
91          StringBuilder sBuilder = new StringBuilder();
92          Node p = top;
```

```
93          while (p != null)
94          {
95             sBuilder.append(p.value);
96             p = p.next;
97             if (p != null)
98                sBuilder.append("\n");
99          }
100         return sBuilder.toString();
101      }
102   }
```

Code Listing 21-6 gives the `LinkedStackDemo` class, which demonstrates the use of the `LinkedStack` class.

Code Listing 21-6 (`LinkedStackDemo.java`)

```
1   /**
2     This class demonstrates the use of the
3     LinkedStack class.
4   */
5
6   public class LinkedStackDemo
7   {
8      public static void main(String [ ] args)
9      {
10        LinkedStack st = new LinkedStack();
11        System.out.println("Pushing: Amy Bob Chuck");
12        System.out.println("Contents of Stack:");
13        st.push("Amy");
14        st.push("Bob");
15        st.push("Chuck");
16        System.out.println(st);
17        String name = st.pop();
18        System.out.println("Popped: " + name);
19        System.out.println("Contents of Stack:");
20        System.out.println(st);
21     }
22  }
```

Program Output

```
Pushing: Amy Bob Chuck
Contents of Stack:
Chuck
Bob
Amy
Popped: Chuck
Contents of Stack:
```

```
Bob
Amy
```

Checkpoint

21.1 What is the common name for an operation that adds an element to a stack?
21.2 What is the common name for an operation that removes an element from a stack?
21.3 What should a stack method for removing an item do when the stack is empty?
21.4 In what order are elements added and removed from a stack?
21.5 Cite an example from everyday life where a collection of items behaves like a stack.

21.4 Queues and Their Applications

CONCEPT: A queue is a collection of items that is accessed in a first-in-first-out fashion.
You saw in the last section that a stack is a collection of items that is accessed in a last-in-first-out fashion. A queue is like a stack, except its items are added and removed in a first-in-first-out fashion. Queues, like stacks, have many applications in the real world. For example, at a grocery store checkout line, the first customer in the line is the first to be served.

Queues find many applications in computer science. Consider, for example, a network of computers that must share one printer. The printer server associates a queue with the printer, and print jobs that arrive from the various computers are added to this queue. Print jobs are removed from this queue and are serviced in a first-come-first-served fashion.

A queue can be viewed as a list where one end is designated as the *front* of the queue, and the other end is designated as the *rear*. We add items at the rear of the queue and remove items from the front. A queue is required to support the following operations:

- *enqueue (x)*: add a new item *x* to the rear of the queue
- *dequeue*( ): remove and return the item at the front of the queue
- *empty*( ): check if the queue is empty
- *peek*( ): return, but do not remove, the item at the front of the queue

21.5 Array Implementation of Queues

CONCEPT: Fixed size arrays can be used to implement queues.
To implement a queue using an array, we need an array `q`, and two integer indices `front` and `rear`. Assuming we want to implement a queue that will hold strings, we declare these variables as follows:

Array Implementation of Queues

```
String [ ] q;
int front, rear;
```

It is useful to establish a condition that specifies how the variables `front` and `rear` are being used, and require that all queue methods that manipulate `front` and `rear` preserve this condition. This condition, which we will refer to as the *queue invariant*, is as follows:

- `front` is the index of the slot in `q` that holds the next item that will be dequeued. This slot will normally be *filled.*
- `rear` is the index of the slot in `q` that will hold the next item that will be enqueued. This slot will normally be *empty.*

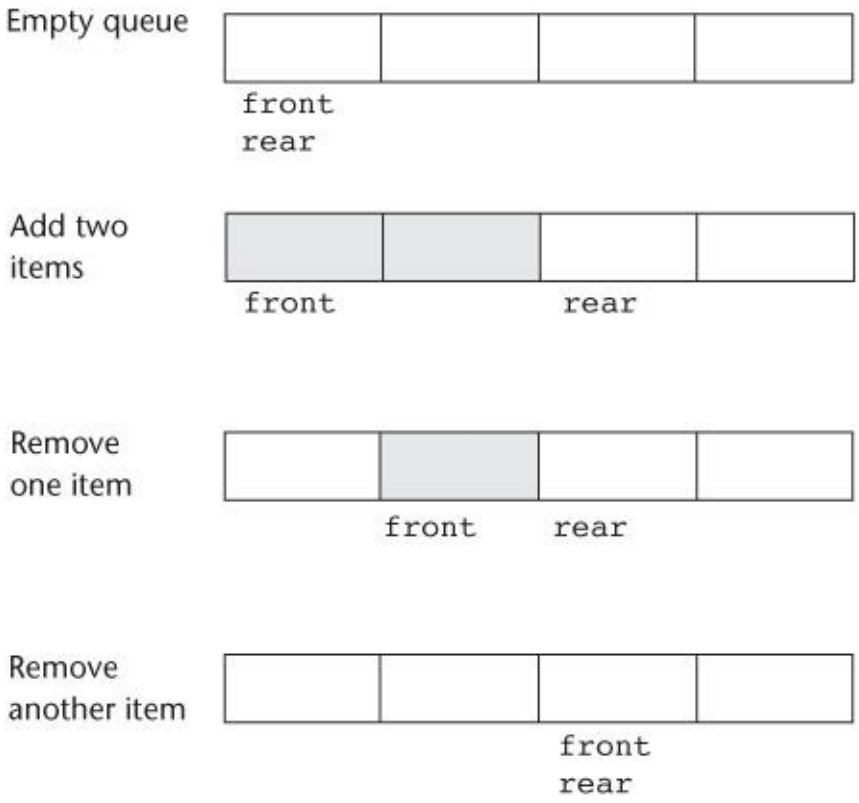
Initially, the queue is empty, and both `front` and `rear` are initialized to 0 as required by the queue invariant (the first element stored will be placed at 0, and the first element removed will come from 0). The queue invariant is illustrated in Figure 21-4.

Now suppose we want to add an item `s` to the queue. The queue invariant says that we should store `s` in the slot `q[rear]` and then increment `rear` so it points to the next empty slot as follows:

```
q[rear] = s;
rear ++;
```

Thus, after adding one item, `rear` will be 1, and after adding a second, `rear` will be 2. At this point `front` will still be 0. This is illustrated in the first two rows of Figure 21-4.

Figure 21-4 Use of an array-based queue



Now consider the operation of removing an item. The item to be removed is `q[front]`. We save the value at that position so it can be returned to the caller, set the queue entry to `null`, and increment `front` so that it points to the next item that will be dequeued.

```
String value = q[front];
q[front] = null;
front ++;
```

This is illustrated in the third and fourth rows of Figure 21-4. Notice that as long as we have not reached the end of the array, `rear` will increase as items are added, and `front` will increase as items are removed, and `rear` and `front` will be equal precisely when the queue is empty.

Now suppose we start with the queue shown in the last row of Figure 21-4 and add two more items. At that point `front` will still be 2, and `rear` will be 0 because that is the next available empty slot in the array. That situation is shown in the top row of Figure 21-5. We see that to make the best use of available space in the array, the indices `front` and `rear` need to wrap around to 0 whenever they get to the end of the array. Accordingly, the correct code for adding an item to the queue is

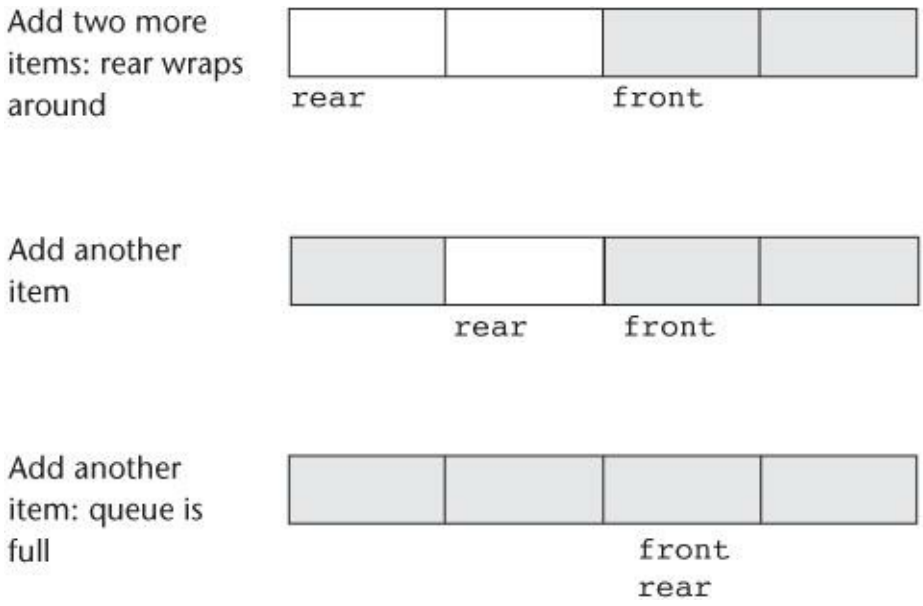
```
q[rear]= s;
rear ++;
if (rear == q.length) rear = 0;
```

and the correct code for removing an item is

```
String value = q[front];
q[front] = null;
front ++;
if (front == q.length) front = 0;
```

When array indices wrap around to the beginning of the array in this fashion, we say that we are using the array as a *circular buffer*.

Figure 21-5 Use of an array as a circular buffer



Notice in Figure 21-5 that `front` is also equal to `rear` when the array is completely full. This makes it impossible to distinguish a queue that is full from one that is empty just by looking at `front` and `rear`. One way to work around this is to have the queue keep a count of the number of items it contains: an enqueue operation increments this counter, and a dequeue operation decrements it. Then we can tell that the queue is empty when this count is 0, and that the queue is full when the count is equal to the length of the array.

A complete implementation of a queue is given in Code Listing 21-7. It uses a variable named `size` to keep track of the number of items in the queue, and has a method named `capacity` that returns the length of the array used inside the queue. Notice that the `enqueue` and `dequeue` methods check for an empty or full queue, and throw appropriately named exceptions when they cannot accomplish their designated task. The exceptions thrown are in a separate file named *QueueExceptions.java:*

```
class QueueOverFlowException extends RuntimeException
{
}
class EmptyQueueException extends RuntimeException
{
}
```

The `ArrayQueue` class in Code Listing 21-7 overrides the `toString` method to return a readable string representation of the contents of the queue. The string returned from this method encodes the current value of the `front` and `rear` indices, together with the value and position of each item in the queue. You can see the format of the string returned by the `toString` method, and how it encodes the state of the queue, in the Program Output following Code Listing 21-8.

Code Listing 21-7 (`ArrayQueue.java`)

```
1  import java.util.*;
2
3  /**
4    The ArrayQueue class uses an
5    array to implement a queue.
6  */
7
8  class ArrayQueue
9  {
10    private String [ ] q;  // Holds queue elements
11    private int front;     // Next item to be removed
12    private int rear;      // Next slot to fill
13    private int size;      // Number of items in queue
14
15    /**
16       Constructor.
17       @param capacity  The capacity of the queue.
18    */
19
20    ArrayQueue(int capacity)
21    {
22       q = new String[capacity];
23       front = 0;
24       rear = 0;
```

```
25          size = 0;
26       }
27
28       /**
29          The capacity method returns the length of
30          the array used to implement the queue.
31          @return The capacity of the queue.
32       */
33
34       public int capacity()
35       {
36          return q.length;
37       }
38
39       /**
40          The enqueue method adds an element to the queue.
41          @param s The element to be added to the queue.
42          @exception QueueOverFlowException When there
43          is no more room in the queue.
44       */
45
46       public void enqueue(String s)
47       {
48          if (size == q.length)
```

```
49            throw new QueueOverFlowException();
50         else
51         {
52            // Add to rear
53            size ++;
54            q[rear] = s;
55            rear ++;
56            if (rear == q.length) rear = 0;
57         }
58      }
59
60      /**
61         The peek method returns the item
62         at the front of the queue.
63         @return element at front of queue.
64         @exception EmptyQueueException When
65         the queue is empty.
66      */
67
68      public String peek()
69      {
70         if (empty())
71            throw new EmptyQueueException();
72         else
73            return q[front];
```

```
74      }
75
76      /**
77         The dequeue method removes and returns
78         the element at the front of the queue.
79         @return The element at the front of the queue.
80         @exception EmptyQueueException When
81         the queue is empty.
82      */
83
84      public String dequeue()
85      {
86         if (empty())
87            throw new EmptyQueueException();
88         else
89          {
90            size --;
91            // Remove from front
92            String value = q[front];
93
94            // Facilitate garbage collection
95            q[front] = null;
96
97            // Update front
```

```
98            front++;
99            if (front == q.length) front = 0;
100
101           return value;
102        }
103     }
104
105     /**
106        The empty method checks to see if
107        this queue is empty.
108        @return true if the queue is empty and
109        false otherwise.
110     */
111
112     public boolean empty()
113     {
114        return size == 0;
115     }
116
117     /**
118        The toString method returns a
119        readable representation of the
120        contents of the queue.
121        @return  The string representation


122        of the contents of the queue.
123     */
124
125     public String toString()
126     {
127        StringBuilder sBuilder = new StringBuilder();
128        sBuilder.append("front = " + front + "; ");
129        sBuilder.append("rear = " + rear + "\n");
130        for (int k = 0; k < q.length; k++)
131        {
132           if (q[k] != null)
133              sBuilder.append(k + " " + q[k]);
134           else
135              sBuilder.append(k + " ?");
136           if (k != q.length - 1)
137              sBuilder.append("\n");
138        }
139        return sBuilder.toString();
140     }
141  }
```

The program shown in Code Listing 21-8 demonstrates the use of the `ArrayQueue` class.

Code Listing 21-8 (`ArrayQueueDemo.java`)

```
/**
  The ArrayQueueDemo demonstrates the use
  of the ArrayQueue class.
*/

public class ArrayQueueDemo
{
   public static void main(String [] args)
   {
      String str;  // Holds various string values

      ArrayQueue queue = new ArrayQueue(4);
      str = "Queue has capacity ";
      System.out.println(str + queue.capacity());

      // Add 4 names
      String [ ] names
         = {"Alfonso", "Bob", "Carol", "Deborah"};
      System.out.println("Adding names: ");
      for (String s : names)
      {
         System.out.print(s + " ");
         queue.enqueue(s);
      }
      System.out.println("\nState of queue is: ");
      System.out.println(queue);

      // Remove 2 names
      System.out.println("Removing 2 names.");
      queue.dequeue(); queue.dequeue();
      System.out.println("State of queue is: ");
      System.out.println(queue);

      // Add a name
      System.out.println("Adding the name Elaine:");
      queue.enqueue("Elaine");
      System.out.println("State of queue is: ");
      System.out.println(queue);
   }
}
```

Program Output

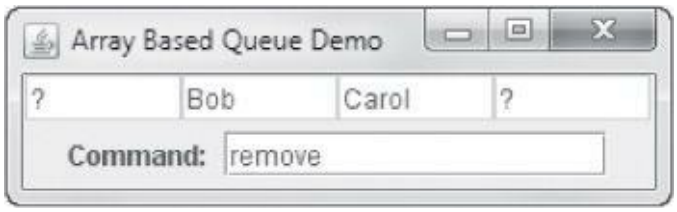
```
Queue has capacity 4
Adding names:
Alfonso Bob Carol Deborah
State of queue is:
front = 0; rear = 0
0 Alfonso
1 Bob
2 Carol
3 Deborah
Removing 2 names.
State of queue is:
front = 2; rear = 0
0 ?
1 ?
2 Carol
3 Deborah
Adding the name Elaine:
State of queue is:
front = 2; rear = 1
0 Elaine
```

```
1 ?
2 Carol
3 Deborah
```

A GUI Front End to the Queue Demo Program

Now we describe a graphical user interface to the queue implemented in Code Listing 21-7. A screenshot of the user interface after the user starts with a queue of size four, adds three names, and then removes a name, is shown in Figure 21-6.

Figure 21-6 Screenshot of GUI front end for the queue program



The user interface consists of a frame with a panel at the top that has an array of text fields. At the bottom of the frame, there is text field used by the user to enter commands. The program recognizes the following commands:

- `enqueue x`: add `x` to the queue
- `add x`: add `x` to the queue (same as `enqueue x`)
- `dequeue`: remove an item from the queue
- `remove`: remove an item from the queue (same as `dequeue`)

The program attaches an action listener to the command entry text field. This action listener is called whenever the user types a command. The listener retrieves the text of the command, parses it by using a `Scanner` object, identifies the command and its argument, and calls the appropriate queue method. Then the listener calls the `refresh` method, passing it a string that represents the current state of the queue. The `refresh` method examines its argument to determine the position and value of each queue element, and then updates the queue view text fields at the top of the frame. Code Listing 21-9 gives the rest of the details.

Code Listing 21-9 (`GUIQueueDemo.java`)

```
1  import java.awt.event.*;
2  import java.awt.*;
3  import javax.swing.*;
4  import java.util.*;
5
6  /**
7     This program is a graphical user interface
8     to the ArrQueue class.
9  */
10
11 public class GUIQueueDemo extends JFrame
12 {
13    private JTextField [ ] qViewTextField;
14    private ArrayQueue queue;
15
16    private JTextField commandEntryTextField;
17
18    /**
19       Constructor.
20    */
21
22    GUIQueueDemo()
23    {


24       setTitle("Array Based Queue Demo");
25
26       // Create queue
27       queue = new ArrayQueue(4);
28       int qSize = queue.capacity();
29
30       // Create view for queue and put
31       // it at top of frame.
32       qViewTextField = new JTextField[qSize];
33       LayoutManager layout = new GridLayout(1, qSize);
34       JPanel qViewPanel = new JPanel(layout);
35       for (int k = 0; k < qViewTextField.length; k++)
36       {
37          qViewTextField[k] = new JTextField();
38          JTextField t = qViewTextField[k];
39          qViewPanel.add(t);
40          t.setEditable(false);
41          t.setBackground(Color.WHITE);
42       }
43       add(qViewPanel, BorderLayout.NORTH);
44
```

```
45          // Create commandEntryTextField and put it
46          // in a panel at the bottom of the frame.
47          commandEntryTextField = new JTextField(15);
48          ActionListener lis = new CmdTextListener();
49          commandEntryTextField.addActionListener(lis);
50          JPanel commandEntryPanel = new JPanel();
51          commandEntryPanel.add(new JLabel("Command: "));
52          commandEntryPanel.add(commandEntryTextField);
53          add(commandEntryPanel, BorderLayout.SOUTH);
54
55          // Finish setting up frame
56          pack();
57          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
58          setVisible(true);
59       }
60
61       /**
62          This private inner class responds to the
63          commands typed into command entry text field.
64       */
65       private class CmdTextListener
66          implements ActionListener
67       {
68          public void actionPerformed(ActionEvent evt)
69          {
70             String cmdText = commandEntryTextField.getText();
```

```
71          Scanner sc = new Scanner(cmdText);
72          if (!sc.hasNext()) return;
73          String cmd = sc.next();
74          if (cmd.equals("add") || cmd.equals("enqueue"))
75          {
76             String item = sc.next();
77             queue.enqueue(item);
78             refresh(queue.toString());
79             return;
80          }
81          if (cmd.equals("remove") || cmd.equals("dequeue"))
82          {
83             queue.dequeue();
84             refresh(queue.toString());
85             return;
86          }
87       }
88    }
89
90    /**
91       The refresh  method stores the current
92       queue entries in the corresponding text
93       fields of the queue view.
94       @param The string encoding the current
```

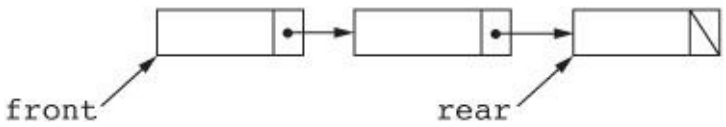
```
95          contents of the queue.
96       */
97
98       private void refresh(String qStr)
99       {
100         Scanner sc = new Scanner(qStr);
101         sc.nextLine();    // Skip first, rear info
102         while (sc.hasNext())
103         {
104            int k = sc.nextInt();
105            String qEntry = sc.next();
106            qViewTextField[k].setText(qEntry);
107         }
108      }
109
110      /**
111         The main method creates the frame so the user
112         can start interacting with the program.
113      */
114
115      public static void main(String [] arg)
116      {
117         new GUIQueueDemo();
118      }
119   }
```

21.6 Linked List Implementation of Queues

CONCEPT: Singly-linked lists can be used to implement queues.

A simple linked list can be used to implement a queue. A reference *front* is used as the head of the linked list, and marks the end of the list from which items will be removed. A second reference, *rear*, is used to mark the end at which additions will take place. Figure 21-7 shows a queue with three elements.

Figure 21-7 A linked list representing a queue with three elements



Notice that *front* and *rear* will be pointing to the same node whenever there is only one node in the list, and will both be set to `null` whenever the list is empty.

Queue Initialization and Enqueuing of Items

Using the same `Node` class as in the `LinkedStack` program, we can define a class `LinkedQueue` that has two fields, `front` and `rear`, to keep track of the list of queue items:

```
Node front = null;
Node rear = null;
```

To add an item `s` to an empty queue, we create a node containing `s` and set both `rear` and `front` to point to it as follows:

```
rear = new Node(s, null);
front = rear;
```

To add an item `s` to a nonempty queue, we create a node containing `s`, set `rear.next` to point to that node, and then move `rear` so that it points to the newly added node as follows:

```
rear.next = new Node(s, null);
rear = rear.next;
```

Dequeuing Items

To remove and return a value from the queue, first we store the value to be returned. Then we move `front` down past the node being removed and return the stored value as follows:

```
String value = front.value;
front = front.next;
return value;
```

This code, however, will leave `rear` pointing to an item that has been removed if the item being removed is the last one in the list. In this case, `rear` must be set to `null`. The correct code for removing is therefore

```
String value = front.value;
front = front.next;
if (front == null)
    rear = null;
return value;
```

The code that implements the `LinkedQueue` class, with appropriate driver code is given in Code Listing 21-10.

Code Listing 21-10 (`LinkedQueue.java`)

```
1  /**
2     This class implements a queue based
3     on linked lists.
4  */
5
6  public class LinkedQueue
7  {
8     private class Node
9     {
10       String value;
11       Node next;
12       Node(String val, Node n)
13       {
14          value = val;
15          next = n;
16       }
17    }
18
19    private Node front = null;
20    private Node rear = null;
21
22    /**
23       The method enqueue adds a value
24       to the queue.
25       @param s The value to be added
```

```
26          to the queue.
27       */
28
29       public void enqueue(String s)
30       {
31          if (rear != null)
32          {
33             rear.next = new Node(s, null);
34             rear = rear.next;
35          }
36          else
37          {
38             rear = new Node(s, null);
39             front = rear;
40          }
41       }
42
43       /**
44          The empty method checks to see if
45          the queue is empty.
46          @return true if and only if queue
47          is empty.
48       */
49
50       public boolean empty()
51       {
```

```
52          return front == null;
53       }
54
55       /**
56          The method peek returns value at the
57          front of the queue.
58          @return item at front of queue.
59          @exception EmptyQueueException When the
60          queue is empty.
61       */
62
63       public String peek()
64       {
65          if (empty())
66             throw new EmptyQueueException();
67          else
68             return front.value;
69       }
70
71       /**
72          The dequeue method removes and returns
73          the item at the front of the queue.
74          @return item at front of queue.
75          @exception EmptyQueueException When
76          the queue is empty.
```

```
77      */
78
79      public String dequeue()
80      {
81         if (empty())
82            throw new EmptyQueueException();
83         else
84         {
85            String value = front.value;
86            front = front.next;
87            if (front == null) rear = null;
88            return value;
89         }
90      }
91
92      /**
93         The toString method concatenates all strings
94         in the queue to give a string representation
95         of the contents of the queue.
96         @return string representation of this queue.
97      */
98
99      public String toString()
100     {


101        StringBuilder sBuilder = new StringBuilder();
102
103        // Walk down the list and append all values
104        Node p = front;
105        while (p != null)
106        {
107           sBuilder.append(p.value + " ");
108           p = p.next;
109        }
110        return sBuilder.toString();
111     }
112  }
```

Code Listing 21-11 demonstrates the use of the `LinkedQueue` class.

Code Listing 21-11 (`LinkedQueueDemo.java`)

```
1  /**
2    The LinkedQueueDemo class demonstrates
3    the use of the LinkedQueue class.
4  */
5
6  public class LinkedQueueDemo
7  {
8     public static void main(String [] args)
9     {
10       LinkedQueue queue = new LinkedQueue();
11
12       // Add 4 names
13       String [ ] names =
14          {"Alfonso", "Bob", "Carol", "Deborah"};
15       System.out.println("Adding names: ");
16       for (String s : names)
17       {
18          System.out.print(s + " ");
19          queue.enqueue(s);
20       }
21
22       System.out.println("\nState of queue is: ");
23       System.out.println(queue);
24
25       // Remove 2 names
26       System.out.println("Removing 2 names.");
27       queue.dequeue(); queue.dequeue();
28       System.out.println("State of queue is: ");
29       System.out.println(queue);
30
31       // Add another name
32       System.out.println("Adding the name Elaine:");
33       queue.enqueue("Elaine");
34       System.out.println("State of queue is: ");
35       System.out.println(queue);
36    }
37 }
```

Program Output

```
Adding names:
Alfonso Bob Carol Deborah
State of queue is:
Alfonso Bob Carol Deborah
Removing 2 names.
State of queue is:
Carol Deborah
Adding the name Elaine:
State of queue is:
Carol Deborah Elaine
```

Checkpoint

21.6 What is the common name for an operation that adds an element to a queue?
21.7 In what order are elements added and removed from a queue?
21.8 Cite an example of a queue from everyday life.
21.9 In an array implementation of a queue, why is it necessary to treat the array as a circular

buffer?

21.7 Generic Implementation of Stacks and Queues

CONCEPT: Classes that implement collections should be written to use generic types.
A generic collection class, whether a stack or a queue, has a great advantage: it can be used to hold items of any type. Implementing a generic collection is not much different from implementing a type-specific one: you simply parameterize the name of the class with a type parameter, say `T`, and then use `T` wherever you would need to use the type of the items being stored in the collection. For example, a skeleton for an array implementation of a generic stack class looks like this:

```
class GenStack<T>
{
   private T [] s;      // Holds stack elements
   private int top;     // Stack top pointer
   public GenStack (int capacity)
   {

   }
   public boolean empty() { return top == 0; }
   public T push(T x)
   {

   }
   public T pop()
   {

   }
}
```

There is a complication, however, because Java does not allow arrays of generic type to be instantiated. For example, the following statement to instantiate an array to hold the stack items will not compile:

```
T[ ] s = new T[capacity];
```

As explained in Chapter 19, there is a workaround: first instantiate an array of `Object`, and then cast it to the desired generic type as follows:

```
T[ ] s = (T[ ]) new Object[capacity];
```

This compiles, albeit with a compiler warning. Code Listing 21-12 shows a complete implementation of an array-based generic stack. The code uses the exception classes declared in the `StackExceptions.java` encountered earlier.

Code Listing 21-12 (`GenStack.java`)

```
1  /**
2     This class implements a generic array
3     based stack.
4  */
5
6  public class GenStack<T>
7  {
8     private T [] s;    // Body of stack
9     private int top;   // Stack top pointer
10
11    /**
12       Constructor.
13       @param capacity The capacity of the stack.
14    */
15
16    public GenStack (int capacity)
17    {
18       s = (T[ ]) new Object [capacity];
19       top = 0;
20    }
21    /**
22       The empty method checks to see if
23       the stack is empty.
24       @return true if and only if the
25       stack is empty.
```

```
26      */
27
28      public boolean empty() { return top == 0; }
29
30      /**
31         The push  method adds x to the stack.
32         @param x the value to be pushed onto
33         the stack.
34         @return the value that was pushed
35         onto the stack.
36         @exception StackOverFlowException When
37         the stack is full.
38      */
39
40      public T push(T x)
41      {
42         if (top == s.length)
43            throw new StackOverFlowException();
44         else
45         {
46            s[top] = x;
47            top ++;
48            return x;
49         }
50      }
```

```
51
52      /**
53         The pop method removes and returns the
54         item at the top of the stack.
55         @return item at the top of the stack.
56         @exception EmptyStackException When the
57         stack is empty.
58      */
59
60      public T pop()
61      {
62         if (empty())
63            throw new EmptyStackException();
64         else
65         {
66            T retVal = s[top-1];
67            s[top-1] = null;
68            top--;
69            return retVal;
70         }
71      }
72   }
```

The program in Code Listing 21-13 shows how to use the `GenericStack` class.

Code Listing 21-13 (`GenericStackDemo.java`)

```
1   /**
2      This class demonstrates the use of
3      the generic stack class GenStack
4   */
5
6   public class GenericStackDemo
7   {
8      public static void main(String [] arg)
9      {
10        GenStack <String> st = new GenStack<String>(5);
11        st.push("George");
12        st.push("Washington");
13        System.out.println(st.pop());
14        System.out.println(st.pop());
15     }
16  }
```

Program Output

```
Washington
```

21.8 Queues and Breadth-First Search

You do not need to implement your own queue if you need to use one in your program. The Java Collection Framework provides many different classes that implement the `Queue` interface. One such class is the familiar `LinkedList` class that we encountered in Chapter 20. The `Queue` interface declares the methods shown in Table 21-2.

Table 21-2 Some methods in the `Queue` interface

`boolean add(E, e)`	Adds an element to the rear of the queue (*enqueue*)
`E peek()`	Returns, but does not remove, the element at the front of the queue
`E remove()`	Removes and returns the element at the front of the queue (*dequeue*)
`boolean isEmpty()`	Checks to see if the queue is empty

In the Spotlight: Directory Searching Using Breadth-First Search

Queues are very useful in solving problems that involve *breadth-first search*, a type of search through the nodes of a digraph. (Digraphs were introduced in the Spotlight section of Chapter 20.) Breadth-first search starts searching at one node and then continues the search through other nodes in order of increasing distance from the start node. Here we show how breadth-first search can be used to search a directory and all contained subdirectories for a specified file or subdirectory. It is not necessary to understand what a digraph is to follow this presentation.

Files and directories in a computer system are stored in a hierarchical fashion, in which a directory may contain files and other directories. When a directory *D* is contained in another directory *E*, we say that *D* is a child directory of *E*. This creates a situation in which directories can have children, children of children, and, in general, *descendants*. The problem here is to search through some initial directory and all its descendants to find a specified file *F* and print the path to *F* if it is found.

Breadth-first search is based on using a queue to store directories that are descendants of the initial directory. A descendant directory is placed on the queue if it has been encountered, but has not yet been searched. Initially, the queue contains only one directory: the initial directory. We proceed in stages. At each stage, we remove the directory *X* at the front of the queue and search it. If we find the desired file, we stop the search. Otherwise, we add all the child directories of *X* to the (rear of the) queue.

Suppose that the initial directory is *X*. In the beginning, the queue contains only *X*, and *X* will be removed from the queue and searched. If the process does not stop there, then all the children of *X* will be added to the queue. The children of *X* are the first-generation descendants. At the second stage, one of the first-generation descendants, say *Y*, will be removed and searched. If the desired file is not found, then all the children of *Y* (which are second-generation descendants of *X*) will be added to the rear of the queue for later processing. Because nodes will be removed from the queue and searched in the order in which they are added, all directories will be searched in increasing order of their generation. You can see from this that breadth-first search searches nodes in order of increasing distance from the start node.

We need several methods from the `java.util.File` class, as shown in Table 21-3.

Table 21-3 Some methods of the `File` class

`String [] list()`	Returns an array of strings representing names of the files or directories stored in this directory. Returns `null` if the `File` object is not a directory or if there is an IO error in trying to determine the directory listing.
`File [] listFiles()`	Returns an array of `File` objects representing the files or subdirectories stored in this directory. Returns `null` if the `File` object is not a directory or if there is an IO error in trying to determine the directory listing.
`boolean isDirectory()`	Returns true if this `File` object is a directory and not a regular file.
`String getAbsolutePath()`	Returns the full pathname of the file or directory.

The program in Code Listing 21-14 follows closely the preceding description of breadth-first search. For brevity, the test data is hard coded into the main method, and the program searches for a file named "`Menu.tex`" starting in the directory "`C:/Users/gcm`". As shown by the program output, this file is found in a descendant subdirectory named "`Backup\CSC531`".

Code Listing 21-14 (`DirSearch.java`)

```
1  /**
2   * This program shows how to use breadth-first search
3   * based on a queue to do a directory search.
4   */
5  import java.io.File;
6  import java.util.*;
7
8  public class DirSearch
9  {
10     public static void main(String[] args)
11     {
12        File initDir = new File("C:/Users/gcm");
13        String filePath = search(initDir, "Menus.tex");
14        if (filePath == null)
15            System.out.println("Not found");
16        else
17            System.out.println(filePath);
18     }
19
20     /**
21      * This method searches a given directory and all its
22      * subdirectories looking for specified file or subdirectory.
23      * @param initDir : the initial directory to search.
24      * @param searchFileName : the name of the file or subdirectory
```

```
25       *                              to search for.
26       * @return the full path name of the searched for file or directory.
27       */
28      static String search(File initDir, String searchFileName)
29      {
30        Queue<File> directoriesToSearch = new LinkedList<File>();
31        directoriesToSearch.add(initDir);
32
33        while (!directoriesToSearch.isEmpty())
34        {
35             // Get next directory to search
36             File currDir = directoriesToSearch.remove();
37             // Get contents of current directory
38             String[] dirContents = currDir.list();
39             // Directory contents will be null if there is
40             // a problem listing the directory contents
41             if (dirContents == null) continue;
42             // Do the directory contents contain the desired file?
43             if (Arrays.asList(dirContents).contains(searchFileName))
44             {
45                 return currDir.getAbsolutePath();
46             }
47             // Desired file not in current directory


48             // Add all the children of this directory to the queue
49             // of directories to be searched
50             File [] childDirectories = currDir.listFiles();
51             for (File f : childDirectories)
52             {
53                 if (f.isDirectory())
54                     directoriesToSearch.add(f);
55             }
56        }
57        // No more directories left to search, so not found
58        return null;
59      }
60 }
```

Program Output

```
C:\Users\gcm\Backup\CSC531
```

21.9 Common Errors to Avoid

- **Forgetting to check for an empty stack or queue**. You should always check to see if a stack is

empty before calling `peek` or `pop`, and you should always check to see if a queue is empty before attempting to access or dequeue an element from the queue. If you forget to do this, the exception thrown will terminate your program.

- **Handling an empty stack or queue by catching the exception thrown**. Exception handling should be reserved for conditions your program cannot anticipate and prevent. Use the methods for checking if a collection is empty instead.
- **Not maintaining the stack or queue invariant**. If you are writing a stack or queue, establish a convention (invariant) for how you use the fields of your stack or queue class, and make sure all methods preserve the invariant.

Review Questions and Exercises

Multiple Choice and True/False

1. A collection that is accessed in *first-in-first-out* fashion is called _____.
a. a stack
b. a queue
c. a linked list
d. an array-based collection
2. A collection that is accessed in *last-in-first-out* fashion is called _____.
a. a stack
b. a queue
c. a linked list
d. none of the above
3. The order in which cars go through a toll booth is best described as _____.
a. a stack
b. a queue
c. a linked list
d. none of the above
4. The concept of seniority, which some employers use to hire and fire workers is _____.
a. a stack
b. a queue
c. a linked list
d. none of the above
5. The stack method that returns an element from the stack without removing it is _____.
a. `pop`
b. `push`
c. `peek`
d. `spy`
6. If the stack method `push` is called on an empty stack, _____.
a. it throws an `EmptyStackException`
b. it adds its argument to the stack
c. it calls the stack method `empty`
d. none of the above
7. **True or False:** You can use the JCF stack to directly create a stack of `int`.

8. **True or False:** If `pop` is called on an empty stack, it will not return until the user puts something on the stack.
9. **True or False:** When using an array to implement a stack, the `push` method will wrap around to the beginning of the stack when it reaches the end.
10. **True or False:** In a linked implementation of a queue, the references `front` and `rear` can only be equal if the queue is empty.

Find the Error

Find the error in each of the following code segments:

1.
```
// An array implementation of a stack
int pop()
{
    if (top == 0)
        throw new EmptyStackException( );
    else
    {
        return s[top-1];
        top --;
    }
}
```
2.
```
// A linked implementation of a queue
void enqueue(int x)
{
    if (empty())
    {
        rear = new Node(x);
        front = rear;
    }
    else
    {
        rear = new Node(x);
        rear = rear.next;
    }
}
```
3.
```
//A linked implementation of a stack
int pop()
{
    if(empty())
        throw new EmptyStackException();
    return top.value;
}
```
4.

```
// A linked implementation of a queue
int dequeue()
{
    if(empty())
        throw new EmptyQueueException();
    int value = front.value;
    front ++;
    return value;
}
```

5.

```
// An array implementation of a queue
int dequeue()
{
    if (empty())
        throw new EmptyQueueException();
        int val = q[rear];
        rear ++;
        return val;
}
```

Algorithm Workbench

1. A palindrome is a word that reads the same backward as forward. For example, the words *madam, radar, dad*, and *kayak* are all palindromes. Write a method that takes a parameter `s` of type `String` and uses a stack to see if `s` is a palindrome.
2. Write a class that uses an array to implement a stack of integers. The stack should have the following fields:

```
int [ ]s;         // Holds stack elements
int top = -1;     // Points to last item pushed
```

   The field `top` will point to an item that is actually stored on the stack, instead of pointing to the next available slot. The stack will set `top` to −1 when it is empty. Write the constructor for this class, and all the usual stack methods.
3. Suppose that you have two stacks but no queues. You have an application that needs to use a queue. Explain how to use the two stacks to simulate a single queue.

Short Answer

1. What is the name for a last-in-first-out collection?
2. What is the name for a first-in-first-out collection?
3. What is the name for a first-in-last-out collection?
4. What does the `pop` method do when it is called on an empty stack?
5. Why does the array entry containing an item that is being popped have to be set to `null`?
6. What problem would you encounter if you tried to use the head of a linked list for the rear of a queue, and use the other end of the list for the front?

Programming Challenges

1. Double-Ended Queue

A *deque* (pronounced "deck") is a list-based collection that allows additions and removals to take place at both ends. A deque supports the operations *addFront (x)* , *removeFront*( ), *addRear (x)*, *removeRear*( ), *size*( ), and *empty*( ). Write a class that implements a deque that stores strings using a doubly-linked list that you code yourself. Demonstrate your class with a graphical user interface that allows users to manipulate the deque by typing appropriate commands in a `JTextField` component, and see the current state of the deque displayed in a `JTextArea` component. Consult the documentation for the `JTextArea` class for methods you can use to display each item in the deque on its own line.

2. Array-Based Deque

Array-Based Deque

Implement a deque as described in Programming Challenge 1, except base your implementation on an array. The constructor for the class should accept an integer parameter for the capacity of the deque and create an array of that size. Use a graphical user interface based on an array of text fields similar to what is shown in Figure 21-6. Test your deque class by constructing a deque with capacity 10.

3. Prefix Expressions

An expression is in *prefix form* when operators are written before their operands. Here are some examples of prefix expressions and the values they evaluate to:

Expression	Value
12	12
+ 2 51	53
* 5 7	35
* + 16 4 + 3 1	80

An expression (such as 12) that begins with an integer is a prefix expression that evaluates to itself. Otherwise, an expression is a prefix expression if it begins with an operator and is followed by two prefix expressions. In this latter case, the value of the expression is recursively computed from the values of its constituent prefix sub-expressions.

Write a program that allows the user to enter prefix expressions in a text field. The program reads the expression, evaluates it, and displays the value in a suitable GUI component. Assume that the user enters expressions that use only positive integers and the two operators + and *. Your program should

use a stack to store values of sub-expressions as they are computed, and another stack to store operators that have not yet been applied.

4. Properly Nested Delimiters

A Java program can have the following type of delimiters: {, }, (,), [, and]. In a correct Java program, these delimiters must be properly nested. Think of each left delimiter {, (, and [as opening a scope, and think of each right delimiter },), and] as closing a scope opened by a corresponding left delimiter. A string of characters containing these delimiters *has proper nesting of delimiters* if each scope that is opened is eventually closed, and the scopes are opened and closed in a last-opened-first-closed fashion.

Write a program that reads a file containing Java source code and checks it for proper nesting of delimiters. Your program should read the source code from the file and print it to the screen. If the file is properly nested, all of it is printed to the screen and a message is printed that the file is properly nested. If the file is not properly nested, then copying of the file to the screen stops as soon as improper nesting is detected, and your program prints a message that the file has errors. To simplify your task, you may assume these delimiters do not appear inside of comments and string literals, and that they do not appear in the program as character literals.

5. Tracing Genealogies

A file has genealogy data for a collection of *N* people. The first line of the file contains the integer *N* followed by *N* additional lines of data. Each of these additional lines specifies a list of children for a single person. The line starts with the name of the person, followed by the number of that person's children, followed by the names of the children. Here is an example of a file specifying genealogy information for ten people.

```
10
Al      3   Beth Carol Dino
Beth    1   Pablo
Carol   3   Ben Alex Mary
Dino    0
Pablo   1   Angela Miguel
Ben     0
Alex    0
Mary    0
Angela  0
Miguel  0
```

For example, Al has three children named Beth, Carol, and Dino; Beth has one child named Pablo; and Dino has no children. You may assume that all names are unique.

Write a program which reads a file of genealogy information and then allows the user to enter pairs of names *X* and *Y*. The program then determines whether *Y* is a descendant of *X*, and if so, prints a list of names beginning with *X* and ending with *Y*, such that each person in the chain is a child of person

preceding them on the list. Otherwise, the program states that *Y* is not a descendant of *X*.