CHAPTER

20

Lists, Stacks, Queues, and Priority Queues

Objectives

- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy (§20.2).
- To use the common methods defined in the `Collection` interface for operating collections (§20.2).
- To use the `Iterator` interface to traverse the elements in a collection (§20.3).
- To use a foreach loop to traverse the elements in a collection (§20.3).
- To use a `forEach` method to perform an action on each element in a collection (§20.4).
- To explore how and when to use `ArrayList` or `LinkedList` to store a list of elements (§20.5).
- To compare elements using the `Comparable` interface and the `Comparator` interface (§20.6).
- To use the static utility methods in the `Collections` class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections (§20.7).
- To develop a multiple bouncing balls application using `ArrayList` (§20.8).
- To distinguish between `Vector` and `ArrayList` and to use the `Stack` class for creating stacks (§20.9).
- To explore the relationships among `Collection`, `Queue`, `LinkedList`, and `PriorityQueue` and to create priority queues using the `PriorityQueue` class (§20.10).
- To use stacks to write a program to evaluate expressions (§20.11).

20.1 Introduction

Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software.

Key Point

data structure

Chapters 18–29 are typically taught in a data structures course. A *data structure* is a collection of data organized in some fashion. The structure not only stores data, but also supports operations for accessing and manipulating the data. Without knowing data structures, you can still write programs, but your program may not be efficient. With a good knowledge of data structures, you can build efficient programs, which are important for practical applications.

why learning data structure

container

In object-oriented thinking, a data structure, also known as a *container* or *container object*, is an object that stores other objects, referred to as data or elements. To define a data structure is essentially to define a class. The class for a data structure should use data fields to store data and provide methods to support such operations as search, insertion, and deletion. To create a data structure is therefore to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into or deleting an element from the data structure.

Section 11.11 introduced the `ArrayList` class, which is a data structure to store elements in a list. Java provides several more data structures (lists, vectors, stacks, queues, priority queues, sets, and maps) that can be used to organize and manipulate data efficiently. These are commonly known as *Java Collections Framework*. We will introduce the applications of lists, vectors, stacks, queues, and priority queues in this chapter and sets and maps in the next chapter. The implementation of these data structures will be discussed in Chapters 24–27. Through implementation, students gain a deep understanding on the efficiency of data structures and on how and when to use certain data structures. Finally, we will introduce design and implement data structures and algorithms for graphs in Chapters 28 and 29.

Java Collections Framework

20.2 Collections

The `Collection` interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

Key Point

The Java Collections Framework supports two types of containers:

collection

- One for storing a collection of elements is simply called a *collection*.

map

- The other, for storing key/value pairs, is called a *map*.

Maps are efficient data structures for quickly searching an element using a key. We will introduce maps in the next chapter. Now we turn our attention to the following collections:

`Set`

- `Set`s store a group of nonduplicate elements.

`List`

- `List`s store an ordered collection of elements.

`Stack`

- `Stack`s store objects that are processed in a last-in, first-out fashion.

`Queue`

- `Queue`s store objects that are processed in a first-in, first-out fashion.

`PrioriryQueue`

- `PriorityQueue`s store objects that are processed in the order of their priorities.

The common operations of these collections are defined in the interfaces, and implementations are provided in concrete classes, as shown in Figure 20.1.

Note

All the interfaces and classes defined in the Java Collections Framework are grouped in the `java.util` package.

Design Guide

The design of the Java Collections Framework is an excellent example of using interfaces, abstract classes, and concrete classes. The interfaces define the common operations.

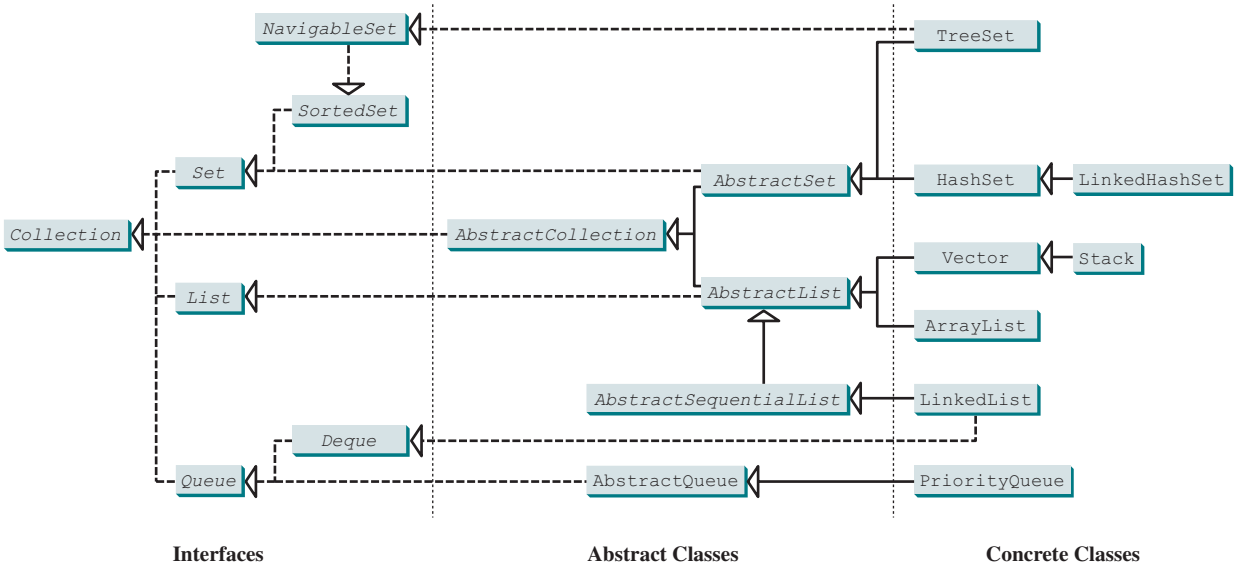


FIGURE 20.1 A collection is a container that stores objects.

The abstract classes provide partial implementation. The concrete classes implement the interfaces with concrete data structures. Providing an abstract class that partially implements an interface makes it convenient for the user to write the code. The user can simply define a concrete class that extends the abstract class rather than implementing all the methods in the interface. The abstract classes such as `AbstractCollection` are provided for convenience. For this reason, they are called *convenience abstract classes*.

convenience abstract class

The `Collection` interface is the root interface for manipulating a collection of objects. Its public methods are listed in Figure 20.2. The `AbstractCollection` class provides partial implementation for the `Collection` interface. It implements all the methods in `Collection` except the `add`, `size`, and `iterator` methods. These are implemented in the concrete subclasses.

The `Collection` interface provides the basic operations for adding and removing elements in a collection. The `add` method adds an element to the collection. The `addAll` method adds all the elements in the specified collection to this collection. The `remove` method removes an element from the collection. The `removeAll` method removes the elements from this collection that are present in the specified collection. The `retainAll` method retains the elements in this collection that are also present in the specified collection. All these methods return `boolean`. The return value is `true` if the collection is changed as a result of the method execution. The `clear()` method simply removes all the elements from the collection.

basic operations

Note

The methods `addAll`, `removeAll`, and `retainAll` are similar to the set union, difference, and intersection operations.

set operations

The `Collection` interface provides various query operations. The `size` method returns the number of elements in the collection. The `contains` method checks whether the collection contains the specified element. The `containsAll` method checks whether the collection contains all the elements in the specified collection. The `isEmpty` method returns `true` if the collection is empty.

query operations

The `Collection` interface provides the `toArray()` method, which returns an array of `Object` for the collection. It also provides the `toArray(T[])` method, which returns an array of the `T[]` type.

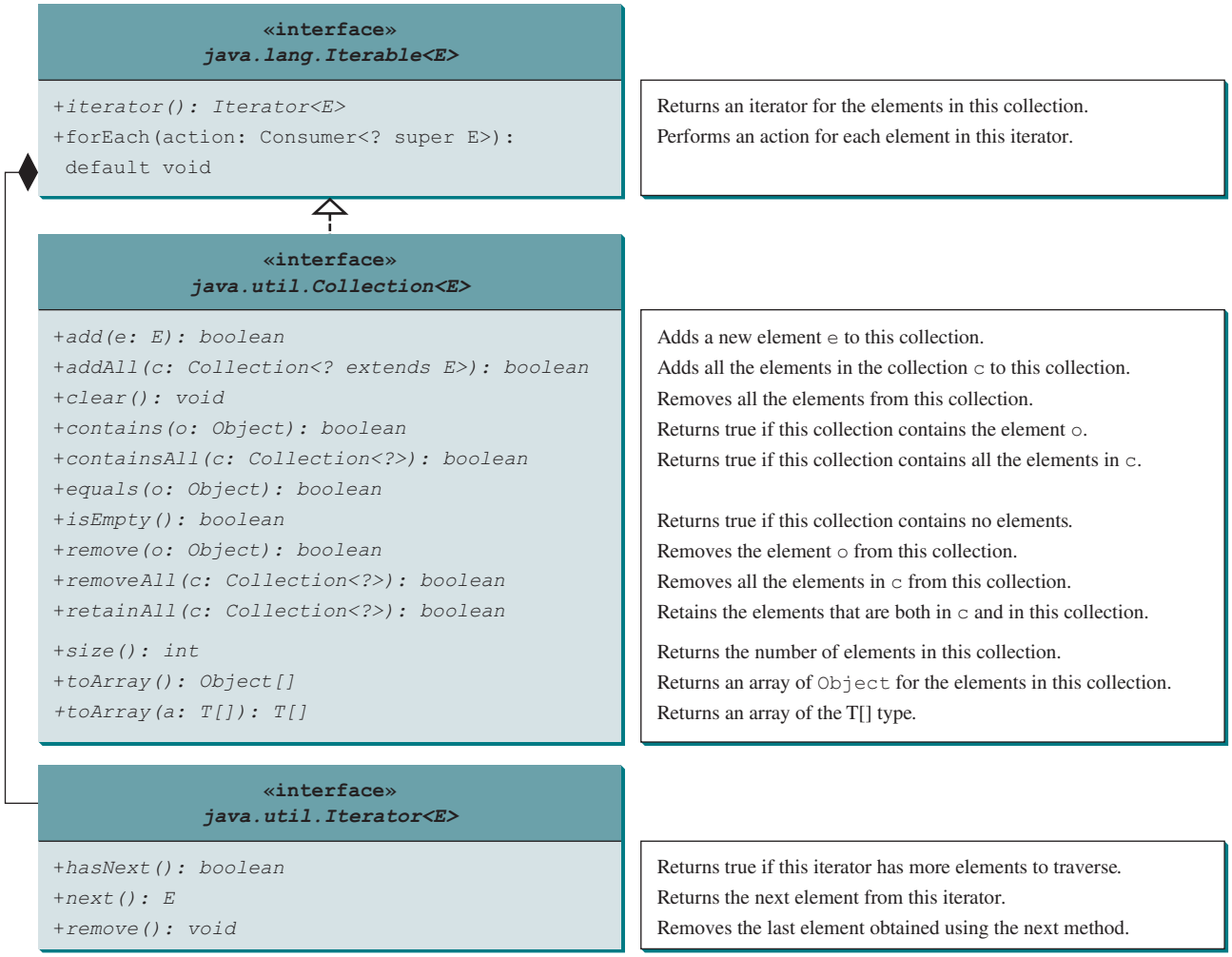


FIGURE 20.2 The `Collection` interface contains the methods for manipulating the elements in a collection, and you can obtain an iterator object for traversing elements in the collection.

unsupported operations

Design Guide

Some of the methods in the `Collection` interface cannot be implemented in the concrete subclass. In this case, the method would throw `java.lang.UnsupportedOperationException`, a subclass of `RuntimeException`. This is a good design you can use in your project. If a method has no meaning in the subclass, you can implement it as follows:

```
public void someMethod() {
  throw new UnsupportedOperationException
    ("Method not supported");
}
```

Listing 20.1 gives an example to use the methods defined in the `Collection` interface.

LISTING 20.1 TestCollection.java

```
1  import java.util.*;
2
3  public class TestCollection {
4    public static void main(String[] args) {
5      ArrayList<String> collection1 = new ArrayList<>();
```

create an array list

```
 6      collection1.add("New York");                                    add elements
 7      collection1.add("Atlanta");
 8      collection1.add("Dallas");
 9      collection1.add("Madison");
10
11      System.out.println("A list of cities in collection1:");
12      System.out.println(collection1);
13
14      System.out.println("\nIs Dallas in collection1? "
15        + collection1.contains("Dallas"));                            contains?
16
17      collection1.remove("Dallas");
18      System.out.println("\n" + collection1.size() +                  size?
19         " cities are in collection1 now");
20
21      Collection<String> collection2 = new ArrayList<>();
22      collection2.add("Seattle");
23      collection2.add("Portland");
24      collection2.add("Los Angeles");
25      collection2.add("Atlanta");
26
27      System.out.println("\nA list of cities in collection2:");
28      System.out.println(collection2);
29
30      ArrayList<String> c1 = (ArrayList<String>)(collection1.clone());  clone
31      c1.addAll(collection2);                                           addAll
32      System.out.println("\nCities in collection1 or collection2: ");
33      System.out.println(c1);
34
35      c1 = (ArrayList<String>)(collection1.clone());
36      c1.retainAll(collection2);                                        retainAll
37      System.out.print("\nCities in collection1 and collection2: ");
38      System.out.println(c1);
39
40      c1 = (ArrayList<String>)(collection1.clone());
41      c1.removeAll(collection2);                                        removeAll
42      System.out.print("\nCities in collection1, but not in 2: ");
43      System.out.println(c1);
44    }
45  }
```

```
A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]
Is Dallas in collection1? true
3 cities are in collection1 now
A list of cities in collection2:
[Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland, Los Angeles, Atlanta]
Cities in collection1 and collection2: [Atlanta]
Cities in collection1, but not in 2: [New York, Madison]
```

The program creates a concrete collection object using **`ArrayList`** (line 5) and invokes the **`Collection`** interface's **`contains`** method (line 15), **`remove`** method (line 17), **`size`** method (line 18), **`addAll`** method (line 31), **`retainAll`** method (line 36), and **`removeAll`** method (line 41).

For this example, we use `ArrayList`. You can use any concrete class of `Collection` such as `HashSet` and `LinkedList` to replace `ArrayList` to test these methods defined in the `Collection` interface.

The program creates a copy of an array list (lines 30, 35, and 40). The purpose of this is to keep the original array list intact and use its copy to perform `addAll`, `retainAll`, and `removeAll` operations.

Note

All the concrete classes in the Java Collections Framework implement the `java.lang.Cloneable` and `java.io.Serializable` interfaces except that `java.util.PriorityQueue` does not implement the `Cloneable` interface. Thus, all instances of `Collection` except priority queues can be cloned and all instances of `Collection` can be serialized.

`Cloneable`
`Serializable`

20.2.1 What is a data structure?

20.2.2 Describe the Java Collections Framework. List the interfaces, convenience abstract classes, and concrete classes under the `Collection` interface.

20.2.3 Can a collection object be cloned and serialized?

20.2.4 What method do you use to add all the elements from one collection to another collection?

20.2.5 When should a method throw an `UnsupportedOperationException`?

20.3 Iterators

Each collection is `Iterable`. You can obtain its `Iterator` object to traverse all the elements in the collection.

`Iterator` is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure.

The `Collection` interface extends the `Iterable` interface. The `Iterable` interface defines the `iterator` method, which returns an iterator. The `Iterator` interface provides a uniform way for traversing elements in various types of collections. The `iterator()` method in the `Iterable` interface returns an instance of `Iterator`, as shown in Figure 20.2, which provides sequential access to the elements in the collection using the `next()` method. You can also use the `hasNext()` method to check whether there are more elements in the iterator, and the `remove()` method to remove the last element returned by the iterator.

Listing 20.2 gives an example that uses the iterator to traverse all the elements in an array list.

LISTING 20.2 TestIterator.java

```
1  import java.util.*;
2
3  public class TestIterator {
4    public static void main(String[] args) {
5      Collection<String> collection = new ArrayList<>();
6      collection.add("New York");
7      collection.add("Atlanta");
8      collection.add("Dallas");
9      collection.add("Madison");
10
11     Iterator<String> iterator = collection.iterator();
12     while (iterator.hasNext()) {
13       System.out.print(iterator.next().toUpperCase() + " ");
14     }
15     System.out.println();
16   }
17 }
```

create an array list
add elements

iterator
`hasNext()`
`next()`

```
NEW YORK ATLANTA DALLAS MADISON
```

The program creates a concrete collection object using `ArrayList` (line 5) and adds four strings into the list (lines 6–9). The program then obtains an iterator for the collection (line 11) and uses the iterator to traverse all the strings in the list and displays the strings in uppercase (lines 12–14).

Tip

You can simplify the code in lines 11–14 using a foreach loop without using an iterator, as follows:

```
for (String element: collection)
  System.out.print(element.toUpperCase() + " ");
```

foreach loop

This loop is read as "for each element in the collection, do the following." The foreach loop can be used for arrays (see Section 7.2.7) as well as any instance of `Iterable`.

In Java 10, you can declare a variable using `var`. The compiler automatically figures out the type of the variable based on the context. For example,

```
var x = 3;
var y = x;
```

The compiler automatically determines that variable `x` is of the type `int` from the integer value `3` assigned to `x` and determines that variable `y` is of the type `int` since `int` variable `x` is assigned to `y`. This is not a good example of using the `var` type. The real good use of the `var` type is to replace long types. For example,

```
Iterator<String> iterator = new ArrayList<String>().iterator();
```

can be replaced by

```
var iterator = new ArrayList<String>().iterator();
```

This is simple and convenient and can help avoid mistakes, because sometimes the programmer may forget what the correct type for iterator.

Caution

We purposely delay the introduction of `var` until now to avoid this useful feature to be improperly used. Don't use it for primitive data types.

Note

`var` is introduced in Java 10. Ideally, it should be a keyword. However, for backward compatibility, `var` is not a keyword and not a reserved word, but rather an identifier with special meaning as the type of a local variable declaration.

20.3.1 How do you obtain an iterator from a collection object?

20.3.2 What method do you use to obtain an element in the collection from an iterator?

20.3.3 Can you use a foreach loop to traverse the elements in any instance of `Collection`?

20.3.4 When using a foreach loop to traverse all elements in a collection, do you need to use the `next()` or `hasNext()` methods in an iterator?

20.4 Using the `forEach` Method

You can use the **`forEach`** *method to perform an action for each element in a collection.*

Java 8 added a new default method **`forEach`** in the **`Iterable`** interface. The method takes an argument for specifying the action, which is an instance of a functional interface **`Consumer<? super E>`**. The **`Consumer`** interface defines the **`accept(E e)`** method for performing an action on the element **`e`**. You can rewrite the preceding example using a **`forEach`** method in Listing 20.3.

LISTING 20.3 TestForEach.java

```
1  import java.util.*;
2
3  public class TestForEach {
4    public static void main(String[] args) {
5      Collection<String> collection = new ArrayList<>();
6      collection.add("New York");
7      collection.add("Atlanta");
8      collection.add("Dallas");
9      collection.add("Madison");
10
11     collection.forEach(e -> System.out.print(e.toUpperCase() + " "));
12   }
13 }
```

create an array list
add elements
forEach method

```
NEW YORK ATLANTA DALLAS MADISON
```

The statement in line 11 uses a lambda expression in (a), which is equivalent to using an anonymous inner class as shown in (b). Using a lambda expression not only simplifies the syntax but also simplifies the semantics.

```
forEach(e ->
  System.out.print(e.toUppserCase() + " "))
```

(a) Use a lambda expression

```
forEach(
  new java.util.function.Consumer<String>() {
    public void accept(String e) {
      System.out.print(e.toUpperCase() + " ");
    }
  }
)
```

(b) Use an anonymous inner class

You can write the code using a foreach loop or using a **`forEach`** method. Using a **`forEach`** is simpler in most cases.

20.4.1 Can you use the **`forEach`** method on any instance of **`Collection`**? Where is the **`forEach`** method defined?

20.4.2 Suppose each element in **`list`** is a **`StringBuilder`**, write a statement using a **`forEach`** method to change the first character to uppercase for each element in **`list`**.

20.5 Lists

The `List` *interface extends the* `Collection` *interface and defines a collection for storing elements in a sequential order. To create a list, use one of its two concrete classes:* `ArrayList` *or* `LinkedList`.

We used `ArrayList` to test the methods in the `Collection` interface in the preceding sections. Now, we will examine `ArrayList` in more depth. We will also introduce another useful list, `LinkedList`, in this section.

20.5.1 The Common Methods in the `List` Interface

`ArrayList` and `LinkedList` are defined under the `List` interface. The `List` interface extends `Collection` to define an ordered collection with duplicates allowed. The `List` interface adds position-oriented operations as well as a new list iterator that enables a list to be traversed bidirectionally. The methods introduced in the `List` interface are shown in Figure 20.3.

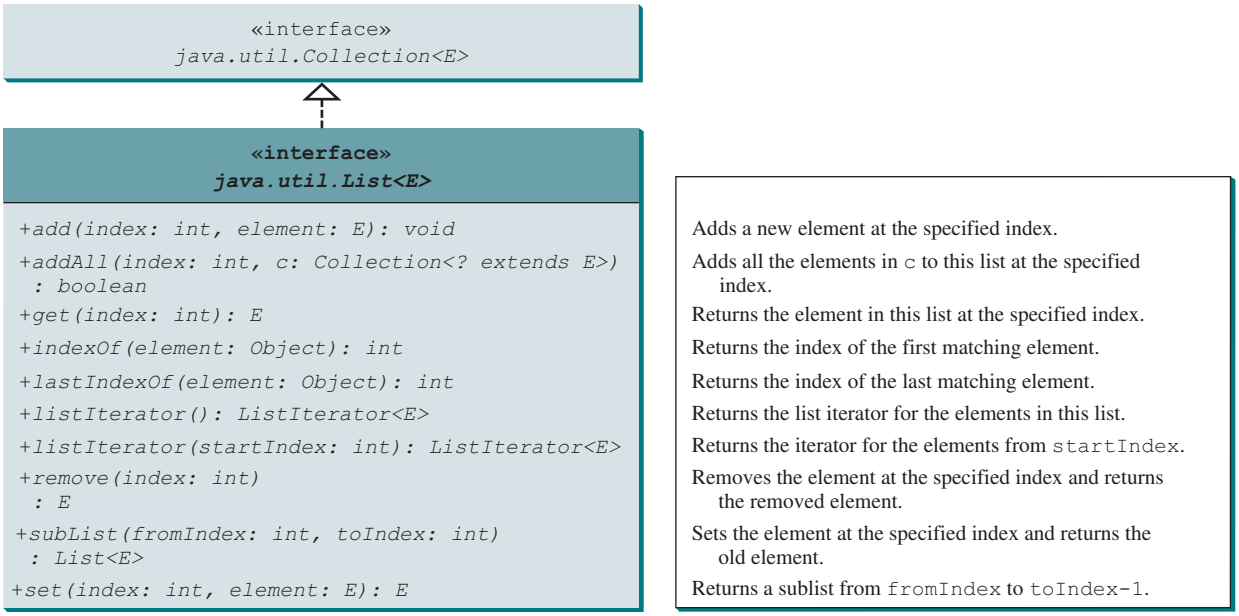


FIGURE 20.3 The `List` interface stores elements in sequence and permits duplicates.

The `add(index, element)` method is used to insert an element at a specified index and the `addAll(index, collection)` method to insert a collection of elements at a specified index. The `remove(index)` method is used to remove an element at the specified index from the list. A new element can be set at the specified index using the `set(index, element)` method.

The `indexOf(element)` method is used to obtain the index of the specified element's first occurrence in the list and the `lastIndexOf(element)` method to obtain the index of its last occurrence. A sublist can be obtained by using the `subList(fromIndex, toIndex)` method.

The `listIterator()` or `listIterator(startIndex)` method returns an instance of `ListIterator`. The `ListIterator` interface extends the `Iterator` interface to add bidirectional traversal of the list. The methods in `ListIterator` are listed in Figure 20.4.

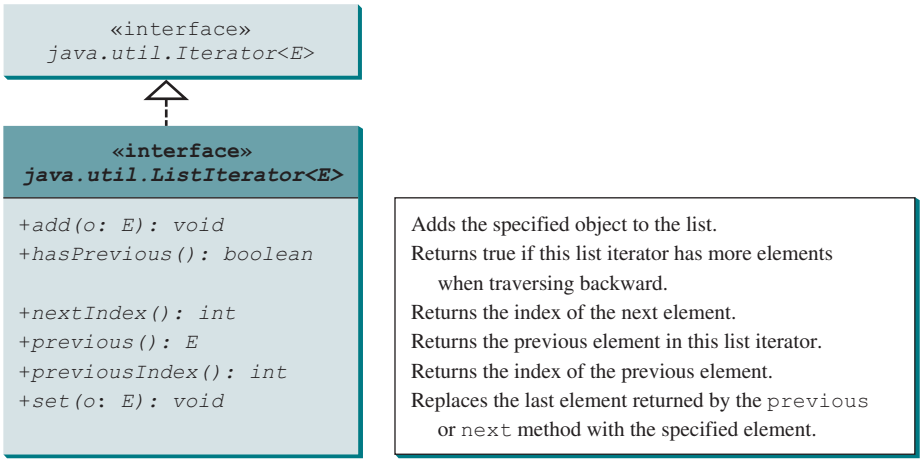


FIGURE 20.4 `ListIterator` enables traversal of a list bidirectionally.

The `add(element)` method inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by the `next()` method defined in the `Iterator` interface, if any, and after the element that would be returned by the `previous()` method, if any. If the list doesn't contain any elements, the new element becomes the sole element in the list. The `set(element)` method can be used to replace the last element returned by the `next` method, or the `previous` method with the specified element.

The `hasNext()` method defined in the `Iterator` interface is used to check whether the iterator has more elements when traversed in the forward direction, and the `hasPrevious()` method to check whether the iterator has more elements when traversed in the backward direction.

The `next()` method defined in the `Iterator` interface returns the next element in the iterator, and the `previous()` method returns the previous element in the iterator. The `nextIndex()` method returns the index of the next element in the iterator, and the `previousIndex()` returns the index of the previous element in the iterator.

The `AbstractList` class provides a partial implementation for the `List` interface. The `AbstractSequentialList` class extends `AbstractList` to provide support for linked lists.

20.5.2 The `ArrayList` and `LinkedList` Classes

ArrayList vs. LinkedList

The `ArrayList` class and the `LinkedList` class are two concrete implementations of the `List` interface. `ArrayList` stores elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array. `LinkedList` stores elements in a *linked list*. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements at the beginning of the list, `ArrayList` is the most efficient. If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose `LinkedList`. A list can grow or shrink dynamically. Once it is created, an array is fixed. If your application does not require the insertion or deletion of elements, an array is the most efficient data structure.

linked list

`ArrayList` is a resizable-array implementation of the `List` interface. It also provides methods for manipulating the size of the array used internally to store the list, as shown in Figure 20.5. Each `ArrayList` instance has a capacity, which is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an `ArrayList`, its capacity grows automatically. An `ArrayList` does not automatically shrink. You can use the `trimToSize()` method to reduce the array capacity to the size of the list. An `ArrayList` can be constructed using its no-arg constructor, `ArrayList(Collection)`, or `ArrayList(initialCapacity)`.

trimToSize()

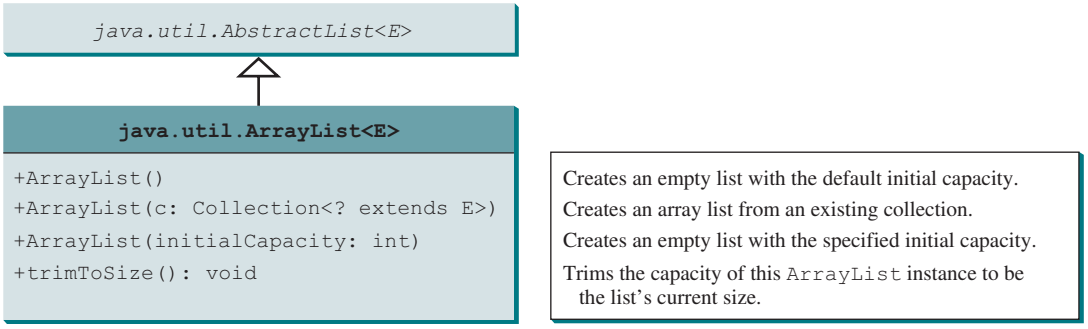


FIGURE 20.5 `ArrayList` implements `List` using an array.

`LinkedList` is a linked list implementation of the `List` interface. In addition to implementing the `List` interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list, as shown in Figure 20.6. A `LinkedList` can be constructed using its no-arg constructor or `LinkedList(Collection)`.

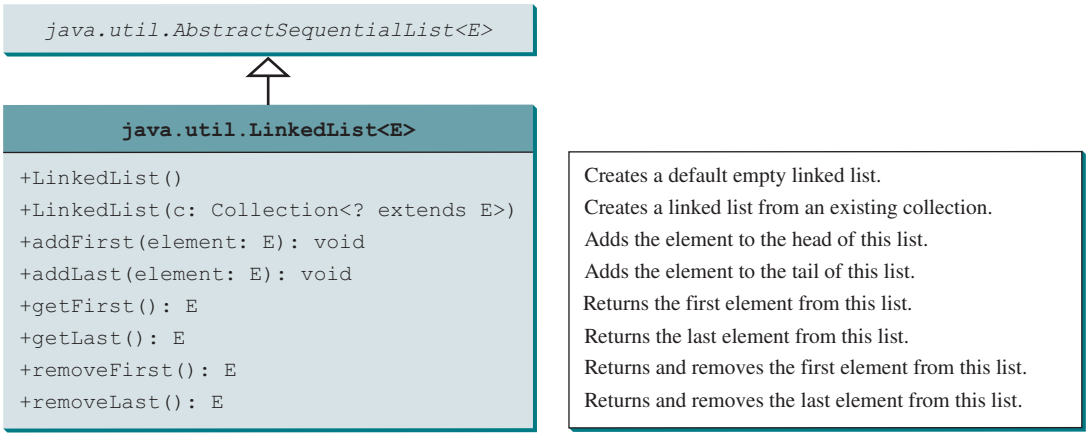


FIGURE 20.6 `LinkedList` provides methods for adding and inserting elements at both ends of the list.

Listing 20.4 gives a program that creates an array list filled with numbers and inserts new elements into specified locations in the list. The example also creates a linked list from the array list and inserts and removes elements from the list. Finally, the example traverses the list forward and backward.

LISTING 20.4 TestArrayAndLinkedList.java

```
1  import java.util.*;
2
3  public class TestArrayAndLinkedList {
4    public static void main(String[] args) {
5      List<Integer> arrayList = new ArrayList<>();                    array list
6      arrayList.add(1); // 1 is autoboxed to an Integer object
7      arrayList.add(2);
8      arrayList.add(3);
9      arrayList.add(1);
10     arrayList.add(4);
11     arrayList.add(0, 10);
12     arrayList.add(3, 30);
13
14     System.out.println("A list of integers in the array list:");
15     System.out.println(arrayList);
```

linked list

list iterator

list iterator

```
16
17      LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
18      linkedList.add(1, "red");
19      linkedList.removeLast();
20      linkedList.addFirst("green");
21
22      System.out.println("Display the linked list forward:");
23      ListIterator<Object> listIterator = linkedList.listIterator();
24      while (listIterator.hasNext()) {
25        System.out.print(listIterator.next() + " ");
26      }
27      System.out.println();
28
29      System.out.println("Display the linked list backward:");
30      listIterator = linkedList.listIterator(linkedList.size());
31      while (listIterator.hasPrevious()) {
32        System.out.print(listIterator.previous() + " ");
33      }
34    }
35  }
```

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
```

A list can hold identical elements. Integer **1** is stored twice in the list (lines 6 and 9). **ArrayList** and **LinkedList** operate similarly. The critical difference between them pertains to internal implementation, which affects their performance. **LinkedList** is efficient for inserting and removing elements at the beginning of the list, and **ArrayList** is more efficient for all other operations. For examples of demonstrating the performance differences between **ArrayList** and **LinkedList**, see Listing.pearsoncmg.com/supplement/ArrayListvsLinkedList.pdf.

The **get(i)** method is available for a linked list, but it is a time-consuming operation. Do not use it to traverse all the elements in a list as shown in (a). Instead, you should use a foreach loop as shown in (b) or a **forEach** method as shown in (c). Note (b) and (c) use an iterator implicitly. You will know the reason when you learn how to implement a linked list in Chapter 24.

```
for (int i = 0; i < list.size(); i++)
  process list.get(i);
}
```

(a) Very inefficient

```
for (listElementType e: list) {
  process e;
}
```

(b) Efficient

```
list.forEach(e ->
  process e
)
```

(c) Efficient

`Arrays.asList(T... a)` method

Tip

Java provides the static **asList** method for creating a list from a variable-length list of arguments. Thus, you can use the following code to create a list of strings and a list of integers:

```
List<String> list1 = Arrays.asList("red", "green", "blue");
List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);
```

In Java 9, you can use the static **List.of** method to replace `Arrays.asList` for creating a list from a variable-length list of arguments. The name **List.of** is more intuitive and easier to memorize than **Arrays.asList**.

20.5.1 How do you add and remove elements from a list? How do you traverse a list in both directions?

20.5.2 Suppose `list1` is a list that contains the strings `red`, `yellow`, and `green`, and `list2` is another list that contains the strings `red`, `yellow`, and `blue`. Answer the following questions:

a. What are `list1` and `list2` after executing `list1.addAll(list2)`?

b. What are `list1` and `list2` after executing `list1.add(list2)`?

c. What are `list1` and `list2` after executing `list1.removeAll(list2)`?

d. What are `list1` and `list2` after executing `list1.remove(list2)`?

e. What are `list1` and `list2` after executing `list1.retainAll(list2)`?

f. What is `list1` after executing `list1.clear()`?

20.5.3 What are the differences between `ArrayList` and `LinkedList`? Which list should you use to insert and delete elements at the beginning of a list?

20.5.4 Are all the methods in `ArrayList` also in `LinkedList`? What methods are in `LinkedList` but not in `ArrayList`?

20.5.5 How do you create a list from an array of objects?

20.6 The `Comparator` Interface

`Comparator` can be used to compare the objects of a class that doesn't implement `Comparable` or define a new criteria for comparing objects.

You have learned how to compare elements using the `Comparable` interface (introduced in Section 13.6). Several classes in the Java API, such as `String`, `Date`, `Calendar`, `BigInteger`, `BigDecimal`, and all the numeric wrapper classes for the primitive types, implement the `Comparable` interface. The `Comparable` interface defines the `compareTo` method, which is used to compare two elements of the same class that implements the `Comparable` interface.

What if the elements' classes do not implement the `Comparable` interface? Can these elements be compared? You can define a *comparator* to compare the elements of different classes. To do so, define a class that implements the `java.util.Comparator<T>` interface and overrides its `compare(a, b)` method.

comparator

```
public int compare(T a, T b)
```

Returns a negative value if `a` is less than `b`, a positive value if `a` is greater than `b`, and zero if they are equal.

The `GeometricObject` class was introduced in Section 13.2, Abstract Classes. The `GeometricObject` class does not implement the `Comparable` interface. To compare the objects of the `GeometricObject` class, you can define a comparator class, as given in Listing 20.5.

LISTING 20.5 GeometricObjectComparator.java

```
1  import java.util.Comparator;
2
3  public class GeometricObjectComparator
4      implements Comparator<GeometricObject>, java.io.Serializable {
5    public int compare(GeometricObject o1, GeometricObject o2) {
6      double area1 = o1.getArea();
7      double area2 = o2.getArea();
8
9      if (area1 < area2)
10       return -1;
```

implements `Comparator`
implements `compare`

```
11      else if (area1 == area2)
12        return 0;
13      else
14        return 1;
15    }
16  }
```

Line 4 implements `Comparator<GeometricObject>`. Line 5 overrides the `compare` method to compare two geometric objects. The class also implements `Serializable`. It is generally a good idea for comparators to implement `Serializable` so they can be serialized. The `compare(o1,o2)` method returns `1` if `o1.getArea() > o2.getArea()`, `0` if `o1.getArea() == o2.getArea()`, and `-1` otherwise.

Listing 20.6 gives a method that returns a larger object between two geometric objects. The objects are compared using the `GeometricObjectComparator`.

LISTING 20.6 TestComparator.java

```
1  import java.util.Comparator;
2
3  public class TestComparator {
4    public static void main(String[] args) {
5      GeometricObject g1 = new Rectangle(5, 5);
6      GeometricObject g2 = new Circle(5);
7
8      GeometricObject g =
9        max(g1, g2, new GeometricObjectComparator());
10
11     System.out.println("The area of the larger object is " +
12       g.getArea());
13   }
14
15   public static GeometricObject max(GeometricObject g1,
16       GeometricObject g2, Comparator<GeometricObject> c) {
17     if (c.compare(g1, g2) > 0)
18       return g1;
19     else
20       return g2;
21   }
22 }
```

invoke max

the max method

invoke compare

```
The area of the larger object is 78.53981633974483
```

The program creates a `Rectangle` and a `Circle` object in lines 5 and 6 (the `Rectangle` and `Circle` classes were defined in Section 13.2, Abstract Classes). They are all subclasses of `GeometricObject`. The program invokes the `max` method to obtain the geometric object with the larger area (lines 8 and 9).

The comparator is an object of the `Comparator` type whose `compare(a,b)` method is used to compare the two elements. The `GeometricObjectComparator` is created and passed to the `max` method (line 9) and this comparator is used in the `max` method to compare the geometric objects in line 17.

Since the `Comparator` interface is a single abstract method interface, you can use a lambda expression to simplify the code by replacing line 9 with the following code

```
max(g1, g2, (o1, o2) -> o1.getArea() > o2.getArea() ?
  1 : o1.getArea() == o2.getArea() ? 0 : -1);
```

Here, **o1** and **o2** are two parameters in the **compare** method in the **Comparator** interface. The method returns **1** if **o1.getArea() > o2.getArea()**, **0** if **o1.getArea() == o2.getArea()**, and **-1** otherwise.

Note

Comparing elements using the **Comparable** interface is referred to as comparing using *natural order*, and comparing elements using the **Comparator** interface is referred to as comparing using *comparator*.

`Comparable vs. Comparator`
natural order
using comparator

The preceding example defines a comparator for comparing two geometric objects since the **GeometricObject** class does not implement the **Comparable** interface. Sometimes a class implements the **Comparable** interface, but if you would like to compare their objects using a different criteria, you can define a custom comparator. Listing 20.7 gives an example that compares string by their length.

LISTING 20.7 SortStringByLength.java

```
1  public class SortStringByLength {
2    public static void main(String[] args) {
3      String[] cities = {"Atlanta", "Savannah", "New York", "Dallas"};
4      java.util.Arrays.sort(cities, new MyComparator());
5
6      for (String s : cities) {
7        System.out.print(s + " ");
8      }
9    }
10
11   public static class MyComparator implements
12       java.util.Comparator<String> {
13     @Override
14     public int compare(String s1, String s2) {
15       return s1.length() - s2.length();
16     }
17   }
18 }
```

sort using comparator

define custom comparator

override compare method

```
Dallas Atlanta Savannah New York
```

The program defines a comparator class by implementing the **Comparator** interface (lines 11 and 12). The **compare** method is implemented to compare two strings by their lengths (lines 14–16). The program invokes the **sort** method to sort an array of strings using a comparator (line 4).

Since **Comparator** is a functional interface, the code can be simplified using a lambda expression as follows:

```
java.util.Arrays.sort(cities,
  (s1, s2) -> {return s1.length() - s2.length();});
```

or simply

```
java.util.Arrays.sort(cities,
  (s1, s2) -> s1.length() - s2.length());
```

The **List** interface defines the **sort(comparator)** method that can be used to sort the elements in a list using a specified comparator. Listing 20.8 gives an example of using a comparator to sort strings in a list by ignoring cases.

LISTING 20.8 SortStringIgnoreCase.java

```
1  public class SortStringIgnoreCase {
2    public static void main(String[] args) {
3      java.util.List<String> cities = java.util.Arrays.asList
4        ("Atlanta", "Savannah", "New York", "Dallas");
5      cities.sort((s1, s2) -> s1.compareToIgnoreCase(s2));
6
7      for (String s: cities) {
8        System.out.print(s + " ");
9      }
10   }
11 }
```

lambda comparator

```
Atlanta dallas new York Savannah
```

The program sorts a list of strings using a comparator that compares strings ignoring case (line 5). If you invoke `list.sort(null)`, the list will be sorted using its natural order.

The comparator is created using a lambda expression. Note the lambda expression here does nothing but simply invokes the `compareToIgnoreCase` method. In the case like this, you can use a simpler and clearer syntax to replace the lambda expression as follows:

```
cities.sort(String::compareToIgnoreCase);
```

method reference

Here `String::compareToIgnoreCase` is known as *method reference*, which is equivalent to a lambda expression. The compiler automatically translates a method reference to an equivalent lambda expression.

Comparator.comparing method

The `Comparator` interface also contains several useful static methods and default methods. You can use the static `comparing(Function<? sup T, ? sup R> keyExtracter)` method to create a `Comparator<T>` that compares the elements using the key extracted from a `Function` object. The `Function` object's `apply(T)` method returns the key of type `R` for the object `T`. For example, the following code in (a) creates a `Comparator` that compares strings by their length using a lambda expression, which is equivalent to the code using an anonymous inner class in (b) and a method reference in (c).

```
Comparator.comparing(e -> e.length())
```

(a) Use a lambda expression

```
Comparator.comparing(String::length)
```

(c) Use a method reference

```
Comparator.comparing(
  new java.util.function.Function<String, Integer>() {
    public Integer apply(String s) {
      return s.length();
    }
  })
```

(b) Use an anonymous inner class

The `comparing` method in the `Comparator` interface is implemented essentially as follows for the preceding example:

```
// comparing returns a Comparator
public static Comparator<String> comparing(Function<String, Integer> f) {
  return (s1, s2) -> f.apply(s1).compareTo(f.apply(s2));
}
```

You can replace the comparator in Listing 20.7 using the following code:

```
java.util.Arrays.sort(cities, Comparator.comparing(String::length));
```

The Comparator.comparing method is particularly useful to create a Comparator using a property from an object. For example, the following code sorts a list of Loan objects (see Listing 10.2) based on their loanAmount property.

```
Loan[] list = {new Loan(5.5, 10, 2323), new Loan(5, 10, 1000)};
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount));
```

You can sort using a primary criteria, second, third, and so on using the Comparator's default thenComparing method. For example, the following code sorts a list of Loan objects first on their loanAmount then on annualInterestRate.

thenComparing method

```
Loan[] list = {new Loan(5.5, 10, 100), new Loan(5, 10, 1000)};
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount)
  .thenComparing(Loan::getAnnualInterestRate));
```

The default reversed() method can be used to reverse the order for a comparator. For example, the following code sorts a list of Loan objects on their loanAmount property in a decreasing order.

```
Arrays.sort(list, Comparator.comparing(Loan::getLoanAmount).
  reversed());
```

20.6.1 What are the differences between the Comparable interface and the Comparator interface? In which package is Comparable, and in which package is Comparator?

20.6.2 How do you define a class A that implements the Comparable interface? Are two instances of class A comparable? How do you define a class B that implements the Comparator interface, and override the compare method to compare two objects of type B1? How do you invoke the sort method to sort a list of objects of the type B1 using a comparator?

20.6.3 Write a lambda expression to create a comparator that compares two Loan objects by their annualInterestRate. Create a comparator using the Comparator.comparing method to compare Loan objects on annualInterestRate. Create a comparator to compare Loan objects first on annualInterestRate then on loanAmount.

20.6.4 Create a comparator using a lambda expression and the Comparator.comparing method, respectively, to compare Collection objects on their size.

20.6.5 Write a statement that sorts an array of Point2D objects on their *y* values and then on their *x* values.

20.6.6 Write a statement that sorts an ArrayList of strings named list in increasing order of their last character.

20.6.7 Write a statement that sorts a two-dimensional array of double[][] in increasing order of their second column. For example, if the array is double[][] x = {{3, 1}, {2, -1}, {2, 0}}, the sorted array will be {{2, -1}, {2, 0}, {3, 1}}.

20.6.8 Write a statement that sorts a two-dimensional array of double[][] in increasing order of their second column as the primary order and the first column as the secondary order. For example, if the array is double[][] x = {{3, 1}, {2, -1}, {2, 0}, {1, -1}}, the sorted array will be {{1, -1}, {2, -1}, {2, 0}, {3, 1}}.

20.7 Static Methods for Lists and Collections

The **`Collections`** *class contains static methods to perform common operations in a collection and a list.*

Section 11.12 introduced several static methods in the **`Collections`** class for array lists. The **`Collections`** class contains the **`sort`**, **`binarySearch`**, **`reverse`**, **`shuffle`**, **`copy`**, and **`fill`** methods for lists and **`max`**, **`min`**, **`disjoint`**, and **`frequency`** methods for collections, as shown in Figure 20.7.

java.util.Collections

List	`+sort(list: List): void`	Sorts the specified list.
	`+sort(list: List, c: Comparator): void`	Sorts the specified list with the comparator.
	`+binarySearch(list: List, key: Object): int`	Searches the key in the sorted list using binary search.
	`+binarySearch(list: List, key: Object, c: Comparator): int`	Searches the key in the sorted list using binary search with the comparator.
	`+reverse(list: List): void`	Reverses the specified list.
	`+reverseOrder(): Comparator`	Returns a comparator with the reverse ordering.
	`+shuffle(list: List): void`	Shuffles the specified list randomly.
	`+shuffle(list: List, rmd: Random): void`	Shuffles the specified list with a random object.
	`+copy(des: List, src: List): void`	Copies from the source list to the destination list.
	`+nCopies(n: int, o: Object): List`	Returns a list consisting of *n* copies of the object.
	`+fill(list: List, o: Object): void`	Fills the list with the object.
Collection	`+max(c: Collection): Object`	Returns the `max` object in the collection.
	`+max(c: Collection, c: Comparator): Object`	Returns the `max` object using the comparator.
	`+min(c: Collection): Object`	Returns the `min` object in the collection.
	`+min(c: Collection, c: Comparator): Object`	Returns the `min` object using the comparator.
	`+disjoint(c1: Collection, c2: Collection): boolean`	Returns true if `c1` and `c2` have no elements in common.
	`+frequency(c: Collection, o: Object): int`	Returns the number of occurrences of the specified element in the collection.

FIGURE 20.7 The **`Collections`** class contains static methods for manipulating lists and collections.

sort list

You can sort the comparable elements in a list in its natural order with the **`compareTo`** method in the **`Comparable`** interface. You may also specify a comparator to sort elements. For example, the following code sorts strings in a list:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.sort(list);
System.out.println(list);
```

The output is **`[blue, green, red]`**.

ascending order
descending order

The preceding code sorts a list in ascending order. To sort it in descending order, you can simply use the **`Collections.reverseOrder()`** method to return a **`Comparator`** object that orders the elements in reverse of natural order. For example, the following code sorts a list of strings in descending order:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.sort(list, Collections.reverseOrder());
System.out.println(list);
```

The output is **`[yellow, red, green, blue]`**.

`binarySearch`

You can use the **`binarySearch`** method to search for a key in a list. To use this method, the list must be sorted in increasing order. If the key is not in the list, the method returns $-$ (*insertion point* $+$ 1). Recall that the insertion point is where the item would fall in the list

if it were present. For example, the following code searches the keys in a list of integers and a sorted list of strings:

```
List<Integer> list1 =
  Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));

List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
  Collections.binarySearch(list2, "red"));
System.out.println("(4) Index: " +
  Collections.binarySearch(list2, "cyan"));
```

The output of the preceding code is:

```
(1) Index: 2
(2) Index: -4
(3) Index: 2
(4) Index: -2
```

reverse

You can use the **reverse** method to reverse the elements in a list. For example, the following code displays **[blue, green, red, yellow]**:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.reverse(list);
System.out.println(list);
```

shuffle

You can use the **shuffle(List)** method to randomly reorder the elements in a list. For example, the following code shuffles the elements in **list**:

```
List<String> list = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list);
System.out.println(list);
```

You can also use the **shuffle(List, Random)** method to randomly reorder the elements in a list with a specified **Random** object. Using a specified **Random** object is useful to generate a list with identical sequences of elements for the same original list. For example, the following code shuffles the elements in **list**:

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list1, new Random(20));
Collections.shuffle(list2, new Random(20));
System.out.println(list1);
System.out.println(list2);
```

You will see that **list1** and **list2** have the same sequence of elements before and after the shuffling.

copy

You can use the **copy(dest, src)** method to copy all the elements from a source list to a destination list on the same index. The destination list must be as long as the source list. If it is longer, the remaining elements in the destination list are not affected. For example, the following code copies **list2** to **list1**:

```
List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
List<String> list2 = Arrays.asList("white", "black");
Collections.copy(list1, list2);
System.out.println(list1);
```

The output for **list1** is **[white, black, green, blue]**. The **copy** method performs a shallow copy: only the references of the elements from the source list are copied.

nCopies

You can use the `nCopies(int n, Object o)` method to create an immutable list that consists of `n` copies of the specified object. For example, the following code creates a list with five `Calendar` objects:

```
List<GregorianCalendar> list1 = Collections.nCopies
    (5, new GregorianCalendar(2005, 0, 1));
```

The list created from the `nCopies` method is immutable, so you cannot add, remove, or update elements in the list. All the elements have the same references.

fill

You can use the `fill(List list, Object o)` method to replace all the elements in the list with the specified element. For example, the following code displays `[black, black, black]`:

```
List<String> list = Arrays.asList("red", "green", "blue");
Collections.fill(list, "black");
System.out.println(list);
```

max and min methods

You can use the `max` and `min` methods for finding the maximum and minimum elements in a collection. The elements must be comparable using the `Comparable` interface or the `Comparator` interface. See the following code for examples:

```
Collection<String> collection = Arrays.asList("red", "green", "blue");
System.out.println(Collections.max(collection));     // Use Comparable
System.out.println(Collections.min(collection,
          Comparator.comparing(String::length)));  // Use Comparator
```

disjoint method

The `disjoint(collection1, collection2)` method returns `true` if the two collections have no elements in common. For example, in the following code, `disjoint(collection1, collection2)` returns `false`, but `disjoint(collection1, collection3)` returns `true`:

```
Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1, collection2));
System.out.println(Collections.disjoint(collection1, collection3));
```

frequency method

The `frequency(collection, element)` method finds the number of occurrences of the element in the collection. For example, `frequency(collection, "red")` returns `2` in the following code:

```
Collection<String> collection = Arrays.asList("red", "cyan", "red");
System.out.println(Collections.frequency(collection, "red"));
```

20.7.1 Are all the methods in the `Collections` class static?

20.7.2 Which of the following static methods in the `Collections` class are for lists and which are for collections?

`sort, binarySearch, reverse, shuffle, max, min, disjoint, frequency`

20.7.3 Show the output of the following code:

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
    List<String> list =
      Arrays.asList("yellow", "red", "green", "blue");
    Collections.reverse(list);
    System.out.println(list);

    List<String> list1 =
```

```
      Arrays.asList("yellow", "red", "green", "blue");
    List<String> list2 = Arrays.asList("white", "black");
    Collections.copy(list1, list2);
    System.out.println(list1);

    Collection<String> c1 = Arrays.asList("red", "cyan");
    Collection<String> c2 = Arrays.asList("red", "blue");
    Collection<String> c3 = Arrays.asList("pink", "tan");
    System.out.println(Collections.disjoint(c1, c2));
    System.out.println(Collections.disjoint(c1, c3));

    Collection<String> collection =
      Arrays.asList("red", "cyan", "red");
    System.out.println(Collections.frequency(collection, "red"));
  }
}
```

20.7.4 Which method can you use to sort the elements in an `ArrayList` or a `LinkedList`? Which method can you use to sort an array of strings?

20.7.5 Which method can you use to perform binary search for elements in an `ArrayList` or a `LinkedList`? Which method can you use to perform binary search for an array of strings?

20.7.6 Write a statement to find the largest element in an array of comparable objects.

20.8 Case Study: Bouncing Balls

This section presents a program that displays bouncing balls and enables the user to add and remove balls.

Section 15.12 presents a program that displays one bouncing ball. This section presents a program that displays multiple bouncing balls. You can use two buttons to suspend and resume the movement of the balls, a scroll bar to control the ball speed, and the + or − button to add or remove a ball, as shown in Figure 20.8.

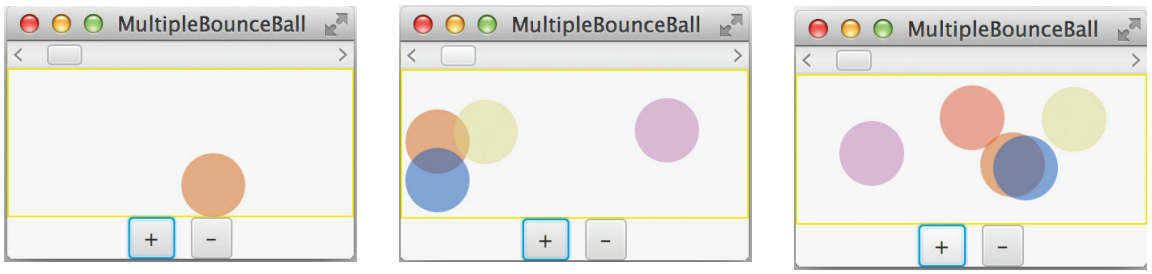


FIGURE 20.8 Pressing the + or − button adds or removes a ball. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

The example in Section 15.12 only had to store one ball. How do you store multiple balls in this example? The `Pane`'s `getChildren()` method returns an `ObservableList<Node>`, a subtype of `List<Node>`, for storing the nodes in the pane. Initially, the list is empty. When a new ball is created, add it to the end of the list. To remove a ball, simply remove the last one in the list.

Each ball has its state: the *x*-, *y*-coordinates, color, and direction to move. You can define a class named `Ball` that extends `javafx.scene.shape.Circle`. The *x*-, *y*-coordinates and the color are already defined in `Circle`. When a ball is created, it starts from the upper-left corner and moves downward to the right. A random color is assigned to a new ball.

The `MultiplBallPane` class is responsible for displaying the ball and the `MultipleBounceBall` class places the control components and implements the control. The relationship of these classes is shown in Figure 20.9. Listing 20.9 gives the program.

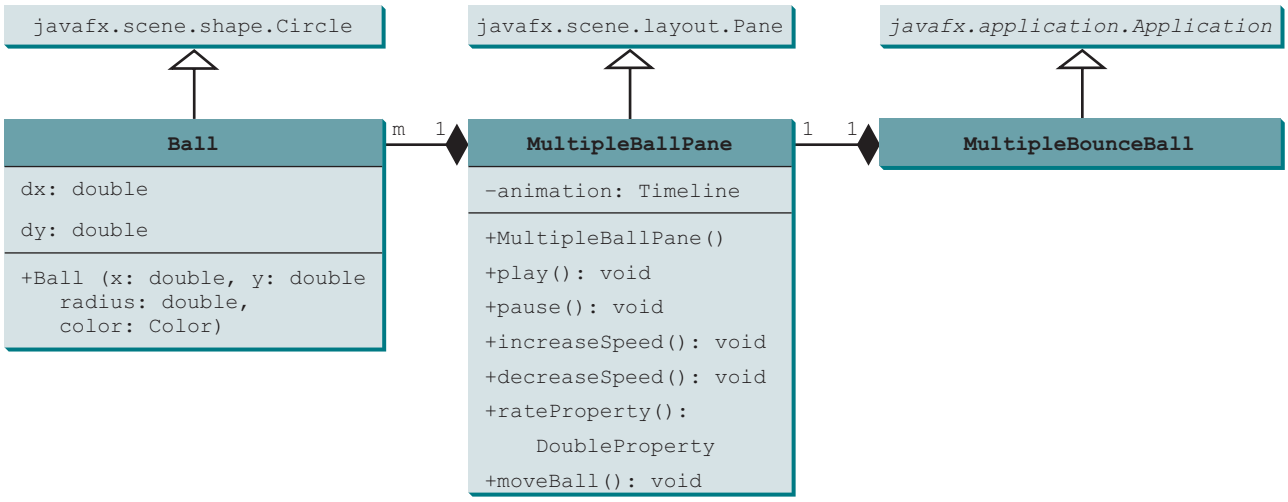


FIGURE 20.9 `MultipleBounceBall` contains `MultipleBallPane` and `MultipleBallPane` contains `Ball`.

LISTING 20.9 MultipleBounceBall.java

```
1  import javafx.animation.KeyFrame;
2  import javafx.animation.Timeline;
3  import javafx.application.Application;
4  import javafx.beans.property.DoubleProperty;
5  import javafx.geometry.Pos;
6  import javafx.scene.Node;
7  import javafx.stage.Stage;
8  import javafx.scene.Scene;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.ScrollBar;
11 import javafx.scene.layout.BorderPane;
12 import javafx.scene.layout.HBox;
13 import javafx.scene.layout.Pane;
14 import javafx.scene.paint.Color;
15 import javafx.scene.shape.Circle;
16 import javafx.util.Duration;
17
18 public class MultipleBounceBall extends Application {
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     MultipleBallPane ballPane = new MultipleBallPane();
22     ballPane.setStyle("-fx-border-color: yellow");
23
24     Button btAdd = new Button("+");
25     Button btSubtract = new Button("-");
26     HBox hBox = new HBox(10);
27     hBox.getChildren().addAll(btAdd, btSubtract);
28     hBox.setAlignment(Pos.CENTER);
29
30     // Add or remove a ball
31     btAdd.setOnAction(e -> ballPane.add());
32     btSubtract.setOnAction(e -> ballPane.subtract());
```

create a ball pane
set ball pane border
create buttons
add buttons to HBox
add a ball
remove a ball

```
33
34      // Pause and resume animation
35      ballPane.setOnMousePressed(e -> ballPane.pause());                  pause animation
36      ballPane.setOnMouseReleased(e -> ballPane.play());                  resume animation
37
38      // Use a scroll bar to control animation speed
39      ScrollBar sbSpeed = new ScrollBar();                                create a scroll bar
40      sbSpeed.setMax(20);
41      sbSpeed.setValue(10);
42      ballPane.rateProperty().bind(sbSpeed.valueProperty());              bind animation rate
43
44      BorderPane pane = new BorderPane();
45      pane.setCenter(ballPane);
46      pane.setTop(sbSpeed);
47      pane.setBottom(hBox);
48
49      // Create a scene and place the pane in the stage
50      Scene scene = new Scene(pane, 250, 150);
51      primaryStage.setTitle("MultipleBounceBall"); // Set the stage title
52      primaryStage.setScene(scene); // Place the scene in the stage
53      primaryStage.show(); // Display the stage
54    }
55
56    private class MultipleBallPane extends Pane {
57      private Timeline animation;
58
59      public MultipleBallPane() {
60        // Create an animation for moving the ball
61        animation = new Timeline(
62          new KeyFrame(Duration.millis(50), e -> moveBall()));
63        animation.setCycleCount(Timeline.INDEFINITE);
64        animation.play(); // Start animation
65      }
66
67      public void add() {
68        Color color = new Color(Math.random(),
69          Math.random(), Math.random(), 0.5);
70        getChildren().add(new Ball(30, 30, 20,color));                    add a ball to pane
71      }
72
73      public void subtract() {
74        if (getChildren().size() > 0) {
75          getChildren().remove(getChildren().size() - 1);                 remove a ball
76        }
77      }
78
79      public void play() {
80        animation.play();
81      }
82
83      public void pause() {
84        animation.pause();
85      }
86
87      public void increaseSpeed() {
88        animation.setRate(animation.getRate() + 0.1);
89      }
90
91      public void decreaseSpeed() {
92        animation.setRate(
```

```
93            animation.getRate() > 0 ? animation.getRate() -  0.1 :  0);
94        }
95
96        public DoubleProperty rateProperty() {
97          return animation.rateProperty();
98        }
99
100       protected void moveBall() {
101         for (Node node:  this.getChildren()) {
102           Ball ball = (Ball)node;
103           // Check boundaries
104           if (ball.getCenterX() < ball.getRadius() ||
105               ball.getCenterX() > getWidth() - ball.getRadius()) {
106             ball.dx *= -1; // Change ball move direction
107           }
108           if (ball.getCenterY() < ball.getRadius() ||
109               ball.getCenterY() > getHeight() - ball.getRadius()) {
110             ball.dy *= -1; // Change ball move direction
111           }
112
113           // Adjust ball position
114           ball.setCenterX(ball.dx + ball.getCenterX());
115           ball.setCenterY(ball.dy + ball.getCenterY());
116         }
117       }
118     }
119
120     class Ball extends Circle {
121       private double dx = 1, dy = 1;
122
123       Ball(double x, double y, double radius, Color color) {
124         super(x, y, radius);
125         setFill(color); // Set ball color
126       }
127     }
128   }
```

move all balls

change *x*-direction

change *y*-direction

adjust ball positions

declare d*x* and d*y*

create a ball

The `add()` method creates a new ball with a random color and adds it to the pane (line 70). The pane stores all the balls in a list. The `subtract()` method removes the last ball in the list (line 75).

When the user clicks the + button, a new ball is added to the pane (line 31). When the user clicks the − button, the last ball in the array list is removed (line 32).

The `moveBall()` method in the `MultipleBallPane` class gets every ball in the pane's list and adjusts the balls' positions (lines 114 and 115).

20.8.1 What is the return value from invoking `pane.getChildren()` for a pane?

20.8.2 How do you modify the code in the `MutilpleBallApp` program to remove the first ball in the list when the − button is clicked?

20.8.3 How do you modify the code in the `MutilpleBallApp` program so each ball will get a random radius between 10 and 20?

20.9 `Vector` and `Stack` Classes

`Vector` is a subclass of `AbstractList` and `Stack` is a subclass of `Vector` in the Java API.

The Java Collections Framework was introduced in Java 2. Several data structures were supported earlier, among them the `Vector` and `Stack` classes. These classes were redesigned to fit into the Java Collections Framework, but all their old-style methods are retained for compatibility.

Vector is the same as **ArrayList**, except that it contains synchronized methods for accessing and modifying the vector. Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently. We will discuss synchronization in Chapter 32, Multithreading and Parallel Programming. For the many applications that do not require synchronization, using **ArrayList** is more efficient than using **Vector**.

The **Vector** class extends the **AbstractList** class. It also has the methods contained in the original **Vector** class defined prior to Java 2, as shown in Figure 20.10.

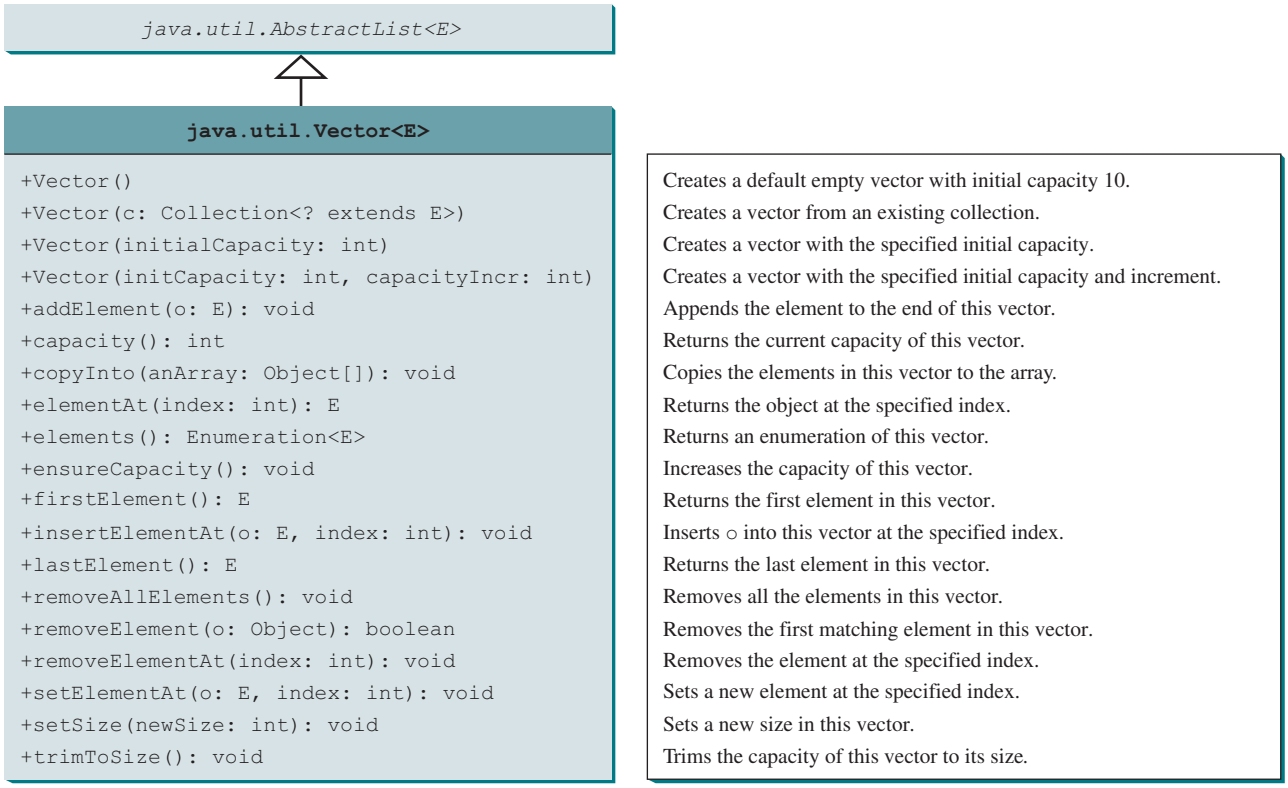


FIGURE 20.10 Starting in Java 2, the **Vector** class extends **AbstractList** and also retains all the methods in the original **Vector** class.

Most of the methods in the **Vector** class listed in the UML diagram in Figure 20.10 are similar to the methods in the **List** interface. These methods were introduced before the Java Collections Framework. For example, **addElement(Object element)** is the same as the **add(Object element)** method, except that the **addElement** method is synchronized. Use the **ArrayList** class if you don't need synchronization. It works much faster than **Vector**.

Note

The **elements()** method returns an **Enumeration**. The **Enumeration** interface was introduced prior to Java 2 and was superseded by the **Iterator** interface.

Note

Vector is widely used in Java legacy code because it was the Java resizable-array implementation before Java 2.

In the Java Collections Framework, **Stack** is implemented as an extension of **Vector**, as illustrated in Figure 20.11.

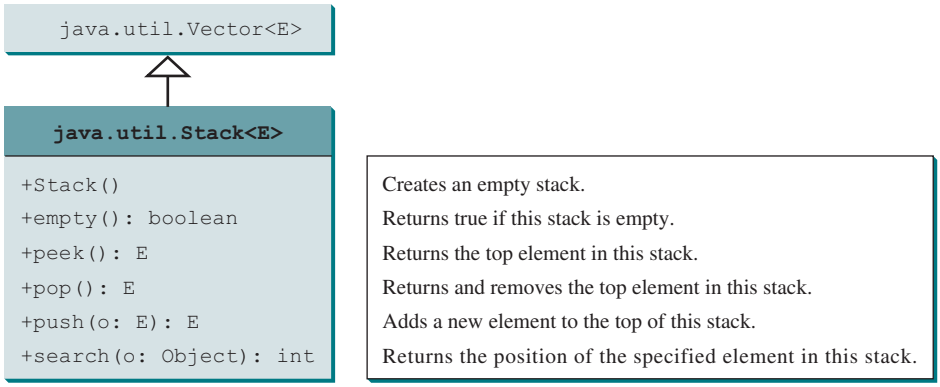


FIGURE 20.11 The `Stack` class extends `Vector` to provide a last-in, first-out data structure.

The `Stack` class was introduced prior to Java 2. The methods shown in Figure 20.11 were used before Java 2. The `empty()` method is the same as `isEmpty()`. The `peek()` method looks at the element at the top of the stack without removing it. The `pop()` method removes the top element from the stack and returns it. The `push(Object element)` method adds the specified element to the stack. The `search(Object element)` method checks whether the specified element is in the stack.

Check Point

20.9.1 How do you create an instance of `Vector`? How do you add or insert a new element into a vector? How do you remove an element from a vector? How do you find the size of a vector?

20.9.2 How do you create an instance of `Stack`? How do you add a new element to a stack? How do you remove an element from a stack? How do you find the size of a stack?

20.9.3 Does Listing 20.1, TestCollection.java, compile and run if all the occurrences of `ArrayList` are replaced by `LinkedList`, `Vector`, or `Stack`?

Key Point

20.10 Queues and Priority Queues

In a priority queue, the element with the highest priority is removed first.

queue
priority queue

A *queue* is a first-in, first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a *priority queue*, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first. This section introduces queues and priority queues in the Java API.

20.10.1 The `Queue` Interface

Queue interface

The `Queue` interface extends `java.util.Collection` with additional insertion, extraction, and inspection operations, as shown in Figure 20.12.

queue operations

The `offer` method is used to add an element to the queue. This method is similar to the `add` method in the `Collection` interface, but the `offer` method is preferred for queues. The `poll` and `remove` methods are similar, except that `poll()` returns `null` if the queue is empty, whereas `remove()` throws an exception. The `peek` and `element` methods are similar, except that `peek()` returns `null` if the queue is empty, whereas `element()` throws an exception.

20.10.2 `Deque` and `LinkedList`

The `LinkedList` class implements the `Deque` interface, which extends the `Queue` interface, as shown in Figure 20.13. Therefore, you can use `LinkedList` to create a queue. `LinkedList` is ideal for queue operations because it is efficient for inserting and removing elements from both ends of a list.

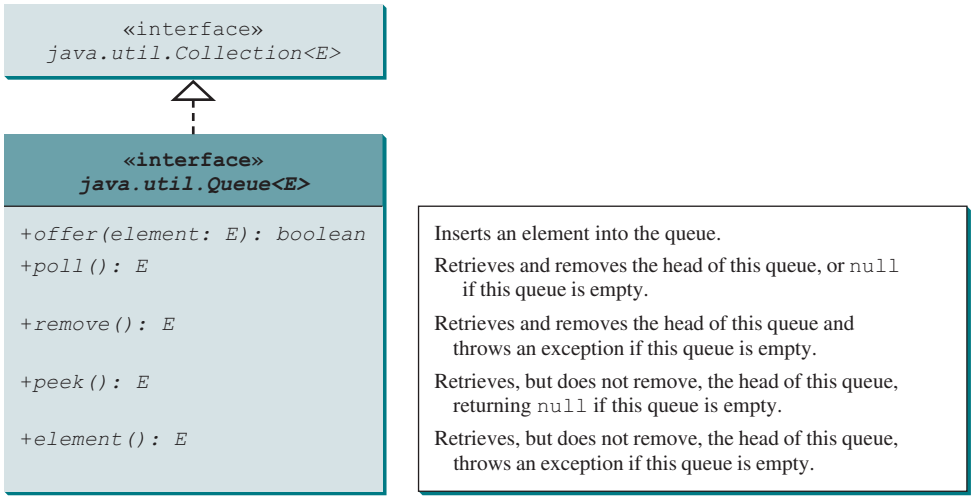


FIGURE 20.12 The `Queue` interface extends `Collection` to provide additional insertion, extraction, and inspection operations.

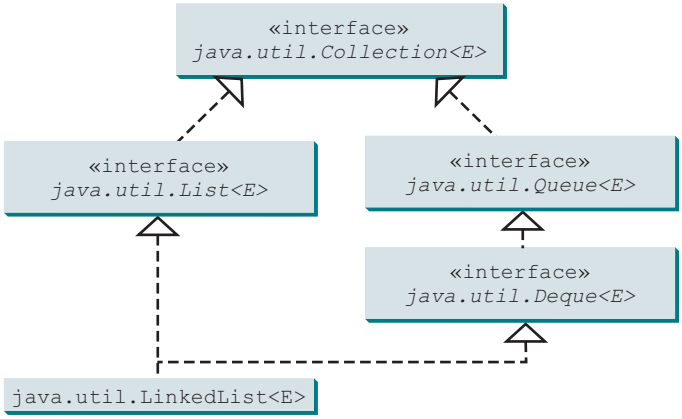


FIGURE 20.13 `LinkedList` implements `List` and `Deque`.

`Deque` supports element insertion and removal at both ends. The name *deque* is short for "double-ended queue" and is usually pronounced "deck." The `Deque` interface extends `Queue` with additional methods for inserting and removing elements from both ends of the queue. The methods `addFirst(e)`, `removeFirst()`, `addLast(e)`, `removeLast()`, `getFirst()`, and `getLast()` are defined in the `Deque` interface.

Listing 20.10 shows an example of using a queue to store strings. Line 3 creates a queue using `LinkedList`. Four strings are added to the queue in lines 4–7. The `size()` method defined in the `Collection` interface returns the number of elements in the queue (line 9). The `remove()` method retrieves and removes the element at the head of the queue (line 10).

LISTING 20.10 TestQueue.java

```
1  public class TestQueue {
2    public static void main(String[] args) {
3      java.util.Queue<String> queue = new java.util.LinkedList<>();     creates a queue
4      queue.offer("Oklahoma");                                          inserts an element
5      queue.offer("Indiana");
6      queue.offer("Georgia");
7      queue.offer("Texas");
8                                                                        queue size
```

queue size
remove element

```
 9      while (queue.size() > 0)
10        System.out.print(queue.remove() + " ");
11    }
12  }
```

```
Oklahoma Indiana Georgia Texas
```

PriorityQueue class

The `PriorityQueue` class implements a priority queue, as shown in Figure 20.14. By default, the priority queue orders its elements according to their natural ordering using `Comparable`. The element with the least value is assigned the highest priority, and thus is removed from the queue first. If there are several elements with the same highest priority, the tie is broken arbitrarily. You can also specify an ordering using `Comparator` in the constructor `PriorityQueue(initialCapacity, comparator)`.

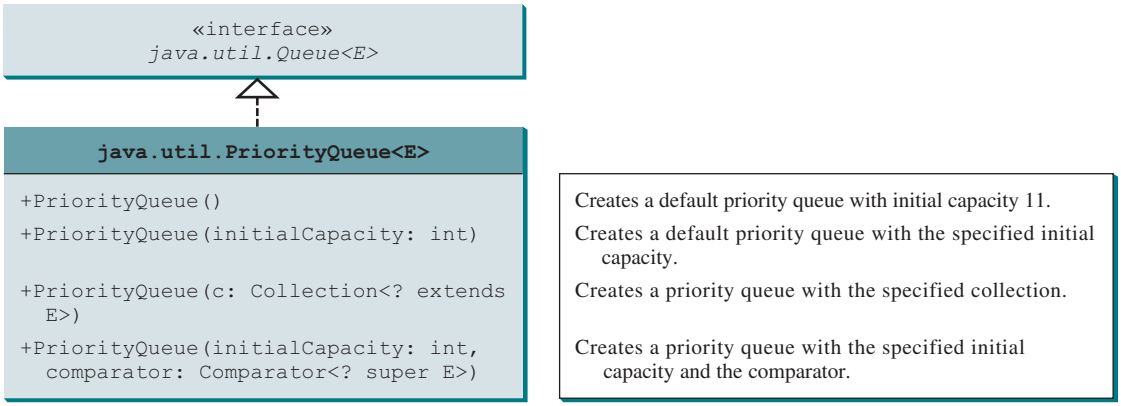


FIGURE 20.14 The `PriorityQueue` class implements a priority queue.

Listing 20.11 shows an example of using a priority queue to store strings. Line 5 creates a priority queue for strings using its no-arg constructor. This priority queue orders the strings using their natural order, so the strings are removed from the queue in increasing order. Lines 16 and 17 create a priority queue using the comparator obtained from `Collections.reverseOrder()`, which orders the elements in reverse order, so the strings are removed from the queue in decreasing order.

LISTING 20.11 PriorityQueueDemo.java

a default queue
inserts an element

```
 1  import java.util.*;
 2
 3  public class PriorityQueueDemo {
 4    public static void main(String[] args) {
 5      PriorityQueue<String> queue1 = new PriorityQueue<>();
 6      queue1.offer("Oklahoma");
 7      queue1.offer("Indiana");
 8      queue1.offer("Georgia");
 9      queue1.offer("Texas");
10
11      System.out.println("Priority queue using Comparable:");
12      while (queue1.size() > 0) {
13        System.out.print(queue1.remove() + " ");
14      }
15
16      PriorityQueue<String> queue2 = new PriorityQueue<>(
17        4, Collections.reverseOrder());
```

```
18      queue2.offer("Oklahoma");
19      queue2.offer("Indiana");
20      queue2.offer("Georgia");
21      queue2.offer("Texas");
22
23      System.out.println("\nPriority queue using Comparator:");
24      while (queue2.size() > 0) {
25        System.out.print(queue2.remove() + " ");
26      }
27    }
28  }
```

a queue with comparator

```
Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
```

20.10.1 Is `java.util.Queue` a subinterface of `java.util.Collection`, `java.util.Set`, or `java.util.List`? Does `LinkedList` implement `Queue`?

20.10.2 How do you create a priority queue for integers? By default, how are elements ordered in a priority queue? Is the element with the least value assigned the highest priority in a priority queue?

20.10.3 How do you create a priority queue that reverses the natural order of the elements?

20.11 Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.

Stacks and queues have many applications. This section gives an application that uses stacks to evaluate expressions. You can enter an arithmetic expression from Google to evaluate the expression, as shown in Figure 20.15.

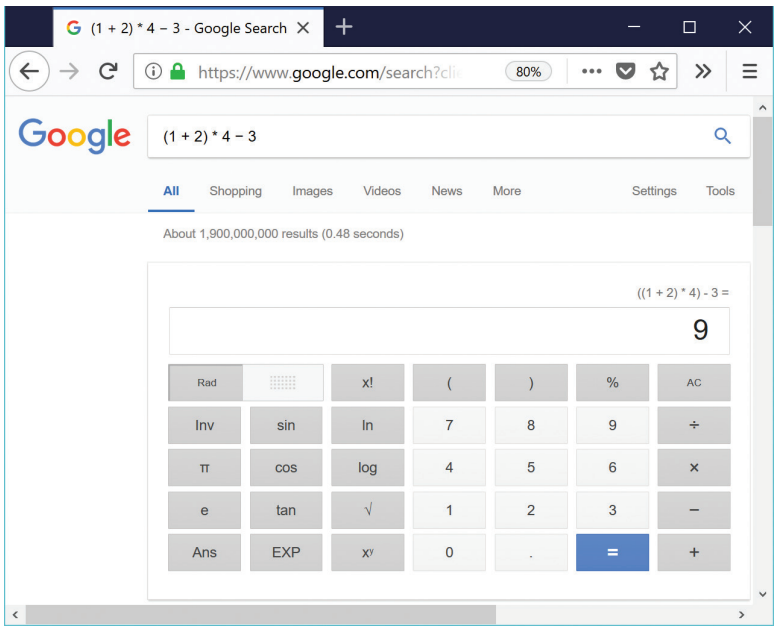


FIGURE 20.15 You can evaluate an arithmetic expression using a Google search engine.
Source: Google and the Google logo are registered trademarks of Google Inc., used with permission.

compound expression

How does Google evaluate an expression? This section presents a program that evaluates a *compound expression* with multiple operators and parentheses (e.g., `(1 + 2) * 4 - 3`). For simplicity, assume the operands are integers, and the operators are of four types: `+`, `-`, `*`, and `/`.

process an operator

The problem can be solved using two stacks, named `operandStack` and `operatorStack`, for storing operands and operators, respectively. Operands and operators are pushed into the stacks before they are processed. When an *operator is processed*, it is popped from `operatorStack` and applied to the first two operands from `operandStack` (the two operands are popped from `operandStack`). The resultant value is pushed back to `operandStack`.

The algorithm proceeds in two phases:

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to `operandStack`.

1.2. If the extracted item is a `+` or `-` operator, process all the operators at the top of `operatorStack` and push the extracted operator to `operatorStack`.

1.3. If the extracted item is a `*` or `/` operator, process the `*` or `/` operators at the top of `operatorStack` and push the extracted operator to `operatorStack`.

1.4. If the extracted item is a `(` symbol, push it to `operatorStack`.

1.5. If the extracted item is a `)` symbol, repeatedly process the operators from the top of `operatorStack` until seeing the `(` symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of `operatorStack` until `operatorStack` is empty.

Table 20.1 shows how the algorithm is applied to evaluate the expression `(1 + 2) * 4 - 3`.

TABLE 20.1 Evaluating an Expression

Expression	*Scan*	*Action*	`operandStack`	`operatorStack`
(1 + 2) * 4 − 3 ↑	(	Phase 1.4		(
(1 + 2) * 4 − 3 ↑	1	Phase 1.1	1	(
(1 + 2) * 4 − 3 ↑	+	Phase 1.2	1	+ (
(1 + 2) * 4 − 3 ↑	2	Phase 1.1	2 1	(
(1 + 2) * 4 − 3 ↑	)	Phase 1.5	3	
(1 + 2) * 4 − 3 ↑	*	Phase 1.3	3	*
(1 + 2) * 4 − 3 ↑	4	Phase 1.1	4 3	*
(1 + 2) * 4 − 3 ↑	−	Phase 1.2	12	-
(1 + 2) * 4 − 3 ↑	3	Phase 1.1	3 12	-
(1 + 2) * 4 − 3 ↑	none	Phase 2	9	

Listing 20.12 gives the program, and Figure 20.16 shows some sample output.

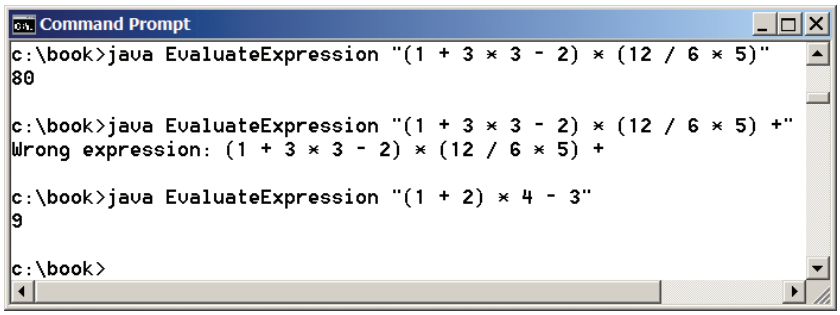


FIGURE 20.16 The program takes an expression as command-line arguments. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

LISTING 20.12 EvaluateExpression.java

```
1  import java.util.Stack;
2
3  public class EvaluateExpression {
4    public static void main(String[] args) {
5      // Check number of arguments passed
6      if (args.length != 1) {                                    check usage
7        System.out.println(
8          "Usage: java EvaluateExpression \"expression\"");
9        System.exit(1);
10     }
11
12     try {
13       System.out.println(evaluateExpression(args[0]));         evaluate expression
14     }
15     catch (Exception ex) {
16       System.out.println("Wrong expression: " + args[0]);     exception
17     }
18   }
19
20   /** Evaluate an expression */
21   public static int evaluateExpression(String expression) {
22     // Create operandStack to store operands
23     Stack<Integer> operandStack = new Stack<>();               operandStack
24
25     // Create operatorStack to store operators
26     Stack<Character> operatorStack = new Stack<>();            operatorStack
27
28     // Insert blanks around (, ), +, -, /, and *
29     expression = insertBlanks(expression);                     prepare for extraction
30
31     // Extract operands and operators
32     String[] tokens = expression.split(" ");                   extract tokens
33
34     // Phase 1: Scan tokens
35     for (String token: tokens) {                               process tokens
36       if (token.length() == 0) // Blank space
37         continue; // Back to the while loop to extract the next token
38       else if (token.charAt(0) == '+' || token.charAt(0) == '-') {   + or − scanned
39         // Process all +, -, *, / in the top of the operator stack
40         while (!operatorStack.isEmpty() &&
41           (operatorStack.peek() == '+' ||
```

```
42              operatorStack.peek() == '-' ||
43              operatorStack.peek() == '*' ||
44              operatorStack.peek() == '/')) {
45              processAnOperator(operandStack, operatorStack);
46            }
47
48            // Push the + or - operator into the operator stack
49            operatorStack.push(token.charAt(0));
50          }
* or / scanned  51          else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
52            // Process all *, / in the top of the operator stack
53            while (!operatorStack.isEmpty() &&
54              (operatorStack.peek() == '*' ||
55              operatorStack.peek() == '/')) {
56              processAnOperator(operandStack, operatorStack);
57            }
58
59            // Push the * or / operator into the operator stack
60            operatorStack.push(token.charAt(0));
61          }
( scanned  62          else if(token.trim().charAt(0) =='(') {
63            operatorStack.push('('); // Push '(' to stack
64          }
) scanned  65          else if (token.trim().charAt(0) ==')') {
66            // Process all the operators in the stack until seeing '('
67            while (operatorStack.peek() != '(') {
68              processAnOperator(operandStack, operatorStack);
69            }
an operand scanned  70
71            operatorStack.pop(); // Pop the '(' symbol from the stack
72          }
73          else {  // An operand scanned
74            // Push an operand to the stack
75            operandStack.push(Integer.valueOf(token));
76          }
77        }
78
79        // Phase 2: Process all the remaining operators in the stack
clear operatorStack  80        while (!operatorStack.isEmpty()) {
81          processAnOperator(operandStack, operatorStack);
82        }
83
84        // Return the result
return result  85        return operandStack.pop();
86      }
87
88      /** Process one operator: Take an operator from operatorStack and
89       *  apply it on the operands in the operandStack */
90      public static void processAnOperator(
91          Stack<Integer> operandStack, Stack<Character> operatorStack) {
92        char op = operatorStack.pop();
93        int op1 = operandStack.pop();
94        int op2 = operandStack.pop();
process +  95        if (op == '+')
96          operandStack.push(op2 + op1);
process −  97        else if (op == '-')
98          operandStack.push(op2 - op1);
process *  99        else if (op == '*')
100         operandStack.push(op2 * op1);
process /  101       else if (op == '/')
```

```
102         operandStack.push(op2 / op1);
103       }
104
105       public static String insertBlanks(String s) {
106         String result = "";
107
108         for (int i = 0; i < s.length(); i++) {
109           if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
110               s.charAt(i) == '+' || s.charAt(i) == '-' ||
111               s.charAt(i) == '*' || s.charAt(i) == '/')
112             result += " " + s.charAt(i) + " ";
113           else
114             result += s.charAt(i);
115         }
116
117         return result;
118       }
119     }
```

insert blanks

You can use the `GenericStack` class provided by the book, or the `java.util.Stack` class defined in the Java API for creating stacks. This example uses the `java.util.Stack` class. The program will work if it is replaced by `GenericStack`.

The program takes an expression as a command-line argument in one string.

The `evaluateExpression` method creates two stacks, `operandStack` and `operatorStack` (lines 23 and 26), and extracts operands, operators, and parentheses delimited by space (lines 29–32). The `insertBlanks` method is used to ensure that operands, operators, and parentheses are separated by at least one blank (line 29).

The program scans each token in the `for` loop (lines 35–77). If a token is empty, skip it (line 37). If a token is an operand, push it to `operandStack` (line 75). If a token is a `+` or `-` operator (line 38), process all the operators from the top of `operatorStack`, if any (lines 40–46), and push the newly scanned operator into the stack (line 49). If a token is a `*` or `/` operator (line 51), process all the `*` and `/` operators from the top of `operatorStack`, if any (lines 53–57), and push the newly scanned operator to the stack (line 60). If a token is a `(` symbol (line 62), push it into `operatorStack`. If a token is a `)` symbol (line 65), process all the operators from the top of `operatorStack` until seeing the `)` symbol (lines 67–69) and pop the `)` symbol from the stack.

After all tokens are considered, the program processes the remaining operators in `operatorStack` (lines 80–82).

The `processAnOperator` method (lines 90–103) processes an operator. The method pops the operator from `operatorStack` (line 92) and pops two operands from `operandStack` (lines 93 and 94). Depending on the operator, the method performs an operation and pushes the result of the operation back to `operandStack` (lines 96, 98, 100, and 102).

20.11.1 Can the `EvaluateExpression` program evaluate the following expressions `"1 + 2"`, `"1 + 2"`, `"(1) + 2"`, `"((1)) + 2"`, and `"(1 + 2)"`?

20.11.2 Show the change of the contents in the stacks when evaluating `"3 + (4 + 5) * (3 + 5) + 4 * 5"` using the `EvaluateExpression` program.

20.11.3 If you enter an expression `"4 + 5 5 5"`, the program will display 10. How do you fix this problem?

Key Terms

collection 798
comparator 809
convenience abstract class 799
data structure 798
linked list 806
list 798
priority queue 798
queue 798

Chapter Summary

1. The `Collection` interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets.

2. Each collection is `Iterable`. You can obtain its `Iterator` object to traverse all the elements in the collection.

3. All the concrete classes except `PriorityQueue` in the Java Collections Framework implement the `Cloneable` and `Serializable` interfaces. Thus, their instances can be cloned and serialized.

4. A list stores an ordered collection of elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list not only can store duplicate elements but also allows the user to specify where they are stored. The user can access elements by an index.

5. Two types of lists are supported: `ArrayList` and `LinkedList`. `ArrayList` is a resizable-array implementation of the `List` interface. All the methods in `ArrayList` are defined in `List`. `LinkedList` is a *linked-list* implementation of the `List` interface. In addition to implementing the `List` interface, this class provides the methods for retrieving, inserting, and removing elements from both ends of the list.

6. `Comparator` can be used to compare the objects of a class that doesn't implement `Comparable`.

7. The `Vector` class extends the `AbstractList` class. Starting with Java 2, `Vector` has been the same as `ArrayList`, except that the methods for accessing and modifying the vector are synchronized. The `Stack` class extends the `Vector` class and provides several methods for manipulating the stack.

8. The `Queue` interface represents a queue. The `PriorityQueue` class implements `Queue` for a *priority queue*.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

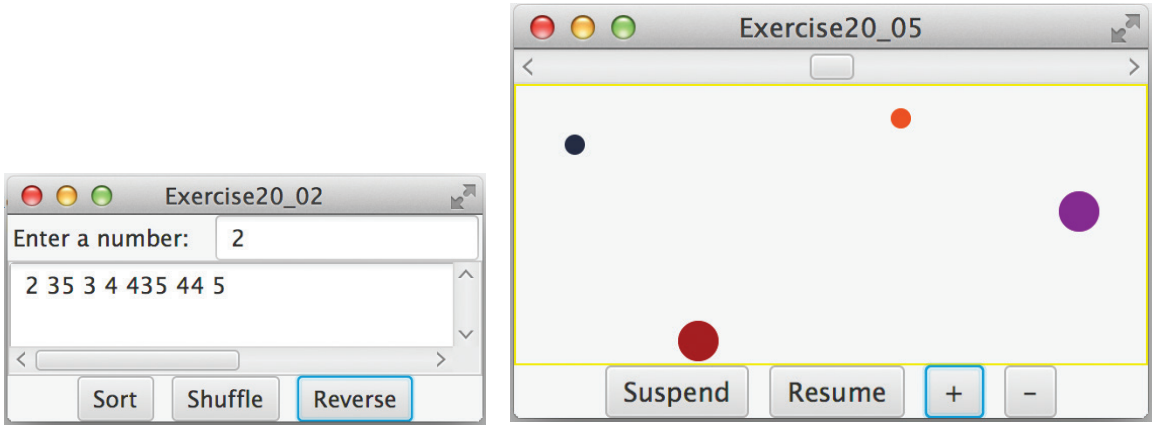
MyProgrammingLab™

Programming Exercises

Sections 20.2–20.7

***20.1** (*Display words in descending alphabetical order*) Write a program that reads words from a text file and displays all the words (duplicates allowed) in descending alphabetical order. The words must start with a letter. The text file is passed as a command-line argument.

***20.2** (*Store numbers in a linked list*) Write a program that lets the user enter numbers from a graphical user interface and displays them in a text area, as shown in Figure 20.17a. Use a linked list to store the numbers. Do not store duplicate numbers. Add the buttons *Sort*, *Shuffle*, and *Reverse* to sort, shuffle, and reverse the list.



(a) (b)

FIGURE 20.17 (a) The numbers are stored in a list and displayed in the text area. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission. (b) The colliding balls are combined.

***20.3** (*Guessing the capitals*) Rewrite Programming Exercise 8.37 to store the pairs of states and capitals so that the questions are displayed randomly.

***20.4** (*Implement Comparable*) Implement a new class that implements `Comparator` of `GregorianCalendar` class to be able sort the calendar in increasing order based on day, month, and year, in that order. Write a method to display the `GregorianCalendar` instance in "dd-MMM-yyyy" format using `SimpleDateFormat` class. Write a test program with 10 `GregorianCalendar` instances and display the results after the sort.

*****20.5** (*Combine colliding bouncing balls*) The example in Section 20.8 displays multiple bouncing balls. Extend the example to detect collisions. Once two balls collide, remove the later ball that was added to the pane and add its radius to the other ball, as shown in Figure 20.17b. Use the *Suspend* button to suspend the animation, and the *Resume* button to resume the animation. Add a mouse-pressed handler that removes a ball when the mouse is pressed on the ball.

20.6 (*Use iterators on ArrayList*) Write a test program that stores 10 million integers in ascending order (i.e., 1,2, . . ., 10m) in an `ArrayList`, displays the execution time taken to traverse the list, and searches for the 10 millionth element using the `get(index)` vs using the `iterator` method.

*****20.7** (*Game: hangman*) Programming Exercise 7.35 presents a console version of the popular hangman game. Write a GUI program that lets a user play the game. The user guesses a word by entering one letter at a time, as shown in Figure 20.18. If the user misses seven times, a hanging man swings. Once a word is finished, the user can press the *Enter* key to continue to guess another word.

****20.8** (*Game: lottery*) Revise Programming Exercise 3.15 to add an additional $2,000 award if two digits from the user input are in the lottery number. (*Hint*: Sort the three digits in the lottery number and three digits in the user input into two lists, and use the `Collection`'s `containsAll` method to check whether the two digits in the user input are in the lottery number.)

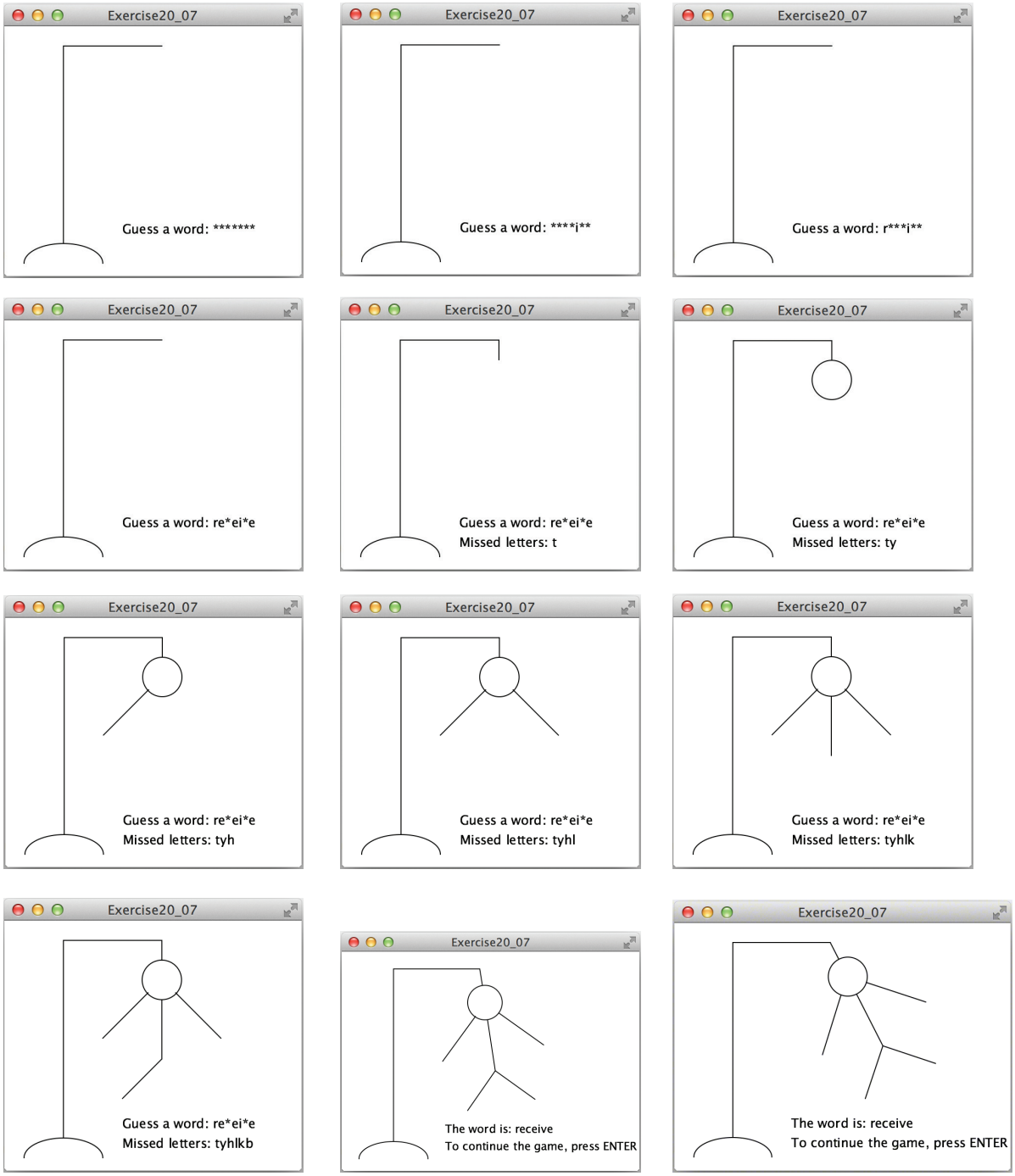


FIGURE 20.18 The program displays a hangman game. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Sections 20.8–20.10

***20.9** (*Remove the largest ball first*) Modify Listing 20.10, `MultipleBallApp.java` to assign a random radius between 2 and 20 when a ball is created. When the − button is clicked, one of largest balls is removed.

20.10 (*Perform set operations on priority queues*) Given two stacks of textbooks of the following subjects {`"Chemistry"`, `"Mathematics"`, `"Biology"`,

"English"} and {"Biology", "English", "Geography", "Physics"}, find the subjects that are (1) only present in the first stack; (2) only present in the second stack; (3) present in both stacks.

*20.11 (*Remove Consecutive Integers*) Write a program that reads 10 integers and displays them in the reverse of the order in which they were read. If two consecutive numbers are identical, then only display one of them. Implement your program using only stack and not arrays or queues.

20.12 (*Create Stack from list*) Define a class `MyStack` that extends `Stack` to be able to have its constructor use a list of objects instead of pushing each object individually.

20.13 (*Game: the 24-point card game*) The 24-point card game is to pick any four cards from 52 cards, as shown in Figure 20.19. Note the Jokers are excluded. Each card represents a number. An Ace, King, Queen, and Jack represent 1, 13, 12, and 11, respectively. You can click the *Shuffle* button to get four new cards. Enter an expression that uses the four numbers from the four selected cards. Each number must be used once and only once. You can use the operators (addition, subtraction, multiplication, and division) and parentheses in the expression. The expression must evaluate to 24. After entering the expression, click the *Verify* button to check whether the numbers in the expression are currently selected and whether the result of the expression is correct. Display the verification in a label before the *Shuffle* button. Assume that images are stored in files named **1.png, **2.png**, . . . , **52.png**, in the order of spades, hearts, diamonds, and clubs. Thus, the first 13 images are for spades 1, 2, 3, . . . , and 13.

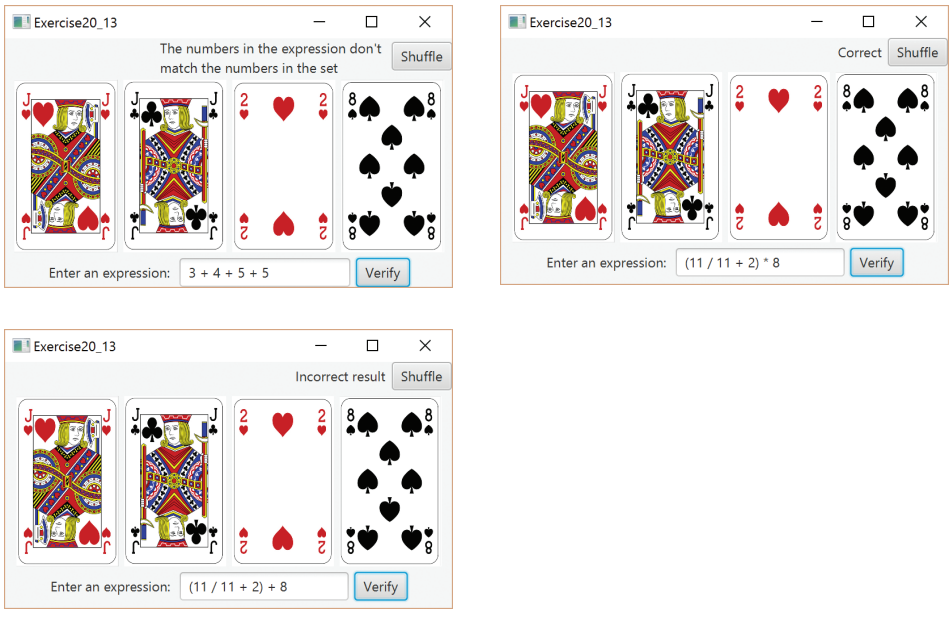
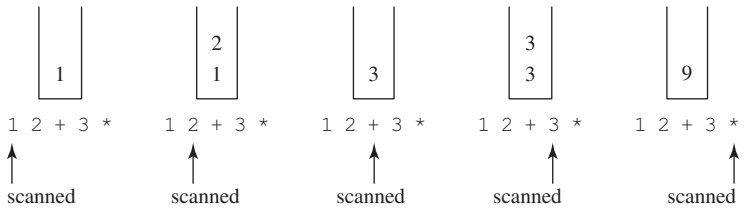


FIGURE 20.19 The user enters an expression consisting of the numbers in the cards and clicks the Verify button to check the answer. *Source*: Fotolia.

**20.14 (*Postfix notation*) Postfix notation is a way of writing expressions without using parentheses. For example, the expression `(1 + 2) * 3` would be written as `1 2 + 3 *`. A postfix expression is evaluated using a stack. Scan a postfix expression from left to right. A variable or constant is pushed into the stack. When an operator is encountered, apply the operator with the top two operands in the stack and replace the two operands with the result. The following diagram shows how to evaluate `1 2 + 3 *`:



Write a program to evaluate postfix expressions. Pass the expression as a command-line argument in one string.

***20.15 (*Game: the 24-point card game*) Improve Programming Exercise 20.13 to enable the computer to display the expression if one exists, as shown in Figure 20.20. Otherwise, report that the expression does not exist. Place the label for verification result at the bottom of the UI. The expression must use all four cards and evaluated to 24.

FIGURE 20.20 The program can automatically find a solution if one exists. *Source*: Fotolia.

**20.16 (*Convert infix to postfix*) Write a method that converts an infix expression into a postfix expression using the following header:

```
public static String infixToPostfix(String expression)
```

For example, the method should convert the infix expression `(1 + 2) * 3` to `1 2 + 3 *` and `2 * (1 + 3)` to `2 1 3 + *`. Write a program that accepts an expression in one argument from the command line and displays its corresponding postfix expression.

***20.17 (*Game: the 24-point card game*) This exercise is a variation of the 24-point card game described in Programming Exercise 20.13. Write a program to check whether there is a 24-point solution for the four specified numbers. The program lets the user enter four values, each between `1` and `13`, as shown in Figure 20.21. The user can then click the *Solve* button to display the solution or display "No solution" if none exists:

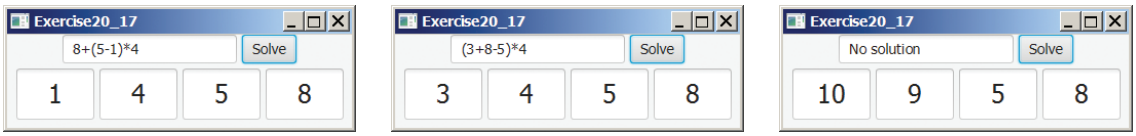


FIGURE 20.21 The user enters four numbers and the program finds a solution. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*20.18 (*Directory size*) Listing 18.7, DirectorySize.java, gives a recursive method for finding a directory size. Rewrite this method without using recursion. Your program should use a queue to store the subdirectories under a directory. The algorithm can be described as follows:

```
long getSize(File directory) {
  long size = 0;
  add directory to the queue;

  while (queue is not empty) {
    Remove an item from the queue into t;
    if (t is a file)
      size += t.length();
    else
      add all the files and subdirectories under t into the
        queue;
  }

  return size;
}
```

***20.19 (*Use `Comparator`*) Write the following generic method using selection sort and a comparator:

```
public static <E> void selectionSort(E[] list,
  Comparator<? super E> Comparator)
```

Write a test program that creates an array of 10 `GeometricObject`s and invokes this method using the `GeometricObjectComparator` introduced in Listing 20.4 to sort the elements. Display the sorted elements. Use the following statement to create the array:

```
GeometricObject[] list1 = {new Circle(5), new Rectangle(4, 5),
   new Circle(5.5), new Rectangle(2.4, 5), new Circle(0.5),
   new Rectangle(4, 65), new Circle(4.5), new Rectangle(4.4, 1),
   new Circle(6.5), new Rectangle(4, 5)};
```

*20.20 (*Nonrecursive Tower of Hanoi*) Implement the `moveDisks` method in Listing 18.8 using a stack instead of using recursion.

*20.21 (*Use `Comparator`*) Write the following generic method using selection sort and a comparator:

```
public static <E> void selectionSort(E[] list,
  Comparator<? super E> comparator)
```

Write a test program that prompts the user to enter six strings, invokes the sort method to sort the six strings by their last character, and displays the sorted strings. Use `Scanner`'s `next()` method to read a string.

*20.22 (*Nonrecursive Tower of Hanoi*) Implement the `moveDisks` method in Listing 18.8 using a stack instead of using recursion.

**20.23 (*Evaluate expression*) Modify Listing 20.12, EvaluateExpression class to make all operators (`+-*/`) have equal precedence. For example, `4 + 3 - 2 * 10` is `50`. Write a test program that prompts the user to enter an expression and displays the result.