12 return mid;
13 else
14     low = mid + 1;
15 }
16 17 return low - 1; // Now high < low, key not found
18 19 }

The binary search returns the index of the search key if it is contained in the list (line 12). Otherwise, it returns low - 1 (line 17).

What would happen if we replaced high := low in line 7 with high > low? The search would miss a possible matching element. Consider a list with just one element. The search would miss the element.

Does the method still work if there are duplicate elements in the list? Yes, as long as the elements are sorted in increasing order. The method returns the index of one of the matching elements if the element is in the list.

To better understand this method, trace it with the following statements and identify low and high when the method returns:

```java
int[] list = {1, 2, 4, 7, 10, 11, 13, 15, 20, 50, 60, 66, 69, 70, 79};
int i = binarySearch(binarySearch(list, 2), 0); // Returns 0
int j = binarySearch(binarySearch(list, 10), 0); // Returns 0
int k = binarySearch(binarySearch(list, 1), 0); // Returns -6
int l = binarySearch(binarySearch(list, 3), 0); // Returns -1
int m = binarySearch(binarySearch(list, 3), 0); // Returns -2
```

Here is the table that lists the low and high values when the method exits and the value returned from invoking the method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Low</th>
<th>High</th>
<th>Value Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(list, 2)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>binarySearch(list, 11)</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>binarySearch(list, 12)</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>binarySearch(list, 1)</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>binarySearch(list, 3)</td>
<td>1</td>
<td>0</td>
<td>-2</td>
</tr>
</tbody>
</table>

**Note**

Linear search is useful for finding an element in a small array or an unsorted array, but it is inefficient for large arrays. Binary search is more efficient, but it requires that the array be sorted.

### 6.11 Sorting Arrays

**Key Point**

There are many strategies for sorting elements in an array. Selection sort and insertion sort are two common approaches.

Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces two simple, intuitive sorting algorithms: selection sort and insertion sort.

#### 6.11.1 Selection Sort

Suppose that you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains. Figure 6.12 shows how to sort the list [2, 4, 3, 6, 5, 9, 1, 6] using selection sort.

- **Select 1 (the smallest) and swap it with 2 (the first) in the list.**
  - The number 1 is now in the correct position and no longer needs to be considered.
  - Select 2 (the smallest) and swap it with 9 (the first) in the remaining list.

- **Select 4 (the smallest) and swap it with 5 (the first) in the remaining list.**
  - The number 4 is now in the correct position and no longer needs to be considered.

- **Select 6 (the smallest) and swap it with 9 (the first) in the remaining list.**
  - The number 6 is now in the correct position and no longer needs to be considered.

- **Select 8 (the smallest) and swap it with 9 (the first) in the remaining list.**

You know how the selection-sort approach works. The task now is to implement it in Java. Beginners find it difficult to develop a complete solution on the first attempt. Start by writing the code for the first iteration to find the smallest element in the list and swap it with the first element, and then observe what would be different for the second iteration, the third, and so on. The insight this gives will enable you to write a loop that generalizes all the iterations.

The solution can be described as follows:

```java
for (int i = 0; i < list.length - 1; i++) {
    select the smallest element in list[i..list.length-1];
    swap the smallest with list[i], if necessary;
    // list[i] is in its correct position.
    // The next iteration apply on list[i+1..list.length-1]
}
```

Since there is only one element remaining in the list, the sort is completed.
Listing 6.8 SelectionSort.java

```java
1 public class SelectionSort {
2     /** The method for sorting the numbers */
3     public static void selectionSort(double[] list) {
4         for (int i = 0; i < list.length - 1; i++) {
5             // Find the minimum in the list[i..list.length-1]
6             double currentMin = list[i];
7             int currentMinIndex = i;
8             for (int j = i + 1; j < list.length; j++) {
9                 if (currentMin > list[j]) {
10                     currentMin = list[j];
11                     currentMinIndex = j;
12                 }
13             }
14             // Swap list[i] with list[currentMinIndex] if necessary
15             if (currentMinIndex != i) {
16                 list[currentMinIndex] = list[i];
17                 list[i] = currentMin;
18             }
19         }
20     }
21 }
22 }
23 }

The selectionSort(double[] list) method sorts any array of double elements. The method is implemented with a nested for loop. The outer loop (with the loop control variable i) (line 4) is iterated in order to find the smallest element in the list, which ranges from list[0] to list[list.length-1], and exchange it with list[i].

The variable i is initially 0. After each iteration of the outer loop, list[i] is in the right place. Eventually, all the elements are put in the right place; therefore, the whole list is sorted.

To understand this method better, trace it with the following statements:

double[] list = {1, 9, 4.5, 6.6, 5.7, -4.5};
SelectionSort.selectionSort(list);
```

### 6.11.2 Insertion Sort

Suppose that you want to sort a list in ascending order. The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted. Figure 6.13 shows how to sort the list [1, 2, 4, 5, 6] using insertion sort.

The algorithm can be described as follows:

```java
for (int i = 1; i < list.length; i++) {
    insert list[i] into a sorted sublist list[0..i-1] so that list[0..i] is sorted.
}
```

To insert list[i] into list[0..i-1], save list[i] into a temporary variable, say currentElement, move list[1..i-1] to list[i] if list[i-1] > currentElement, move list[1] to list[1] if list[1] > currentElement, and so on, until list[i-1] <= currentElement or k > i (we pass the first element of the sorted list).

Assign currentElement to list[i-1]. For example, to insert 4 into [1, 2, 5, 6] in Step 4 in Figure 6.14, move list[2] (9) to list[3] since 9 > 4, and move list[1] (5) to list[2] since 5 > 4. Finally, move currentElement (4) to list[1].

```
Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.
Step 2: The sorted sublist is [2, 9]. Insert 5 into the sublist.
Step 3: The sorted sublist is [2, 5, 9]. Insert 4 into the sublist.
Step 4: The sorted sublist is [2, 4, 5, 9]. Insert 8 into the sublist.
Step 5: The sorted sublist is [2, 4, 5, 8, 9]. Insert 1 into the sublist.
Step 6: The sorted sublist is [1, 2, 4, 5, 8, 9]. Insert 6 into the sublist.
Step 7: The entire list is now sorted.
```

**Figure 6.13** Insertion sort repeatedly inserts a new element into a sorted sublist.

```
[0][1][2][3][4][5][6]
```

**Figure 6.14** A new element is inserted into a sorted sublist.

The algorithm can be expanded and implemented as in Listing 6.9.

Listing 6.9 InsertionSort.java

```java
1 public class InsertionSort {
2     /** The method for sorting the numbers */
3     public static void insertionSort(double[] list) {
4         for (int i = 1; i < list.length; i++) {
5             /* Insert list[i] into a sorted sublist list[0..i-1] so that list[0..i] is sorted. */
6             double currentElement = list[i];
7             int k = i;
8             for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
9                 shift
10                 list[k + 1] = list[k];
11             }
12             // Insert the current element into list[k + 1]
13         }
14 }
```
The `insert(double[] list)` method sorts any array of `double` elements. The method is implemented with a nested `for` loop. The outer loop (with the loop control variable `i`) (line 4) is initiated in order to obtain a sorted sublist, which ranges from `list[0]` to `list[5]`. The inner loop (with the loop control variable `k`) inserts `list[i]` into the sublist from `list[0]` to `list[i-1].`

To better understand this method, trace it with the following statements:

```java
double[] list = {1.9, 4.5, 6.6, 5.7, -4.5};
insertSort.insertSort(list);
```

6.19 Use Figure 6.13 as an example to show how to apply the binary search approach to a search for key 10 and key 12 in list (2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79).

6.20 If the binary search method returns -4, is the key in the list? Where should the key be inserted if you want to insert the key into the list?

6.21 Use Figure 6.13 as an example to show how to apply the selection-sort approach to sort (1, 4, 5, 3, 3, 5, 2, 2, 1, 9, 2).

6.22 Use Figure 6.13 as an example to show how to apply the insertion-sort approach to sort (1, 4, 5, 3, 3, 5, 2, 2, 1, 9, 2).

6.23 How do you modify the `selectionSort` method in Listing 6.8 to sort numbers in decreasing order?

6.24 How do you modify the `insertSort` method in Listing 6.9 to sort numbers in decreasing order?

6.12 The Arrays Class

The `java.util.Arrays` class contains useful methods for common array operations such as sorting and searching.

The `java.util.Arrays` class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and returning a string representation of the array. These methods are overloaded for all primitive types. You can use the `sort()` method to sort a whole array or a partial array. For example, the following code sorts an array of numbers and an array of array of characters.

```java
double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers); // Sort the whole array
char[] chars = {'a', 'A', 'a', 'A', 'E', 'D', 'D', 'P'};
java.util.Arrays.sort(chars, 1, 3); // Sort part of the array
```

Invoking `sort(numbers)` sorts the whole array. Invoking `sort(chars, 1, 3)` sorts a partial array from `chars[3]` to `chars[3-1]`. You can use the `binarySearch` method to search for a key in an array. The array must be processed in increasing order. If the key is not in the array, the method returns `(-insertionPoint+1)`. For example, the following code searches the keys in an array of integers and an array of characters.

```java
int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("Index is "+
java.util.Arrays.binarySearch(list, 12));
```

The output of the preceding code is

1. Index is 4
2. Index is -6
3. Index is 0
4. Index is -4

You can use the `equals` method to check whether two arrays are equal. Two arrays are equal if they have the same contents. In the following code, `list1` and `list2` are equal, but `list2` and `list3` are not.

```java
int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {2, 4, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false
```

You can use the `fill` method to fill an array part of the array. For example, the following code fills list with 5 and fills 6 into elements list[11] and list[12].

```java
int[] list = {2, 4, 7, 10};
list[11] = 7;
list[12] = 10;
java.util.Arrays.fill(list, 1, 3, 5); // Fill 5 to the whole array
java.util.Arrays.fill(list, 1, 3, 3); // Fill 3 to a partial array
```

You can also use the `toString` method to return a string that represents all elements in the array. This is a quick and simple way to display all elements in the array. For example, the following code displays

```java
int[] list = {2, 4, 7, 10};
System.out.println(new java.util.Arrays.toString(list));
```

6.25 What types of array can be sorted using the `java.util.Arrays.sort()` method? Does this sort method create a new array?

6.26 To apply `java.util.Arrays.binarySearch(array, key)`, should the array be sorted in increasing order, in decreasing order, or neither?

6.27 Show the output of the following code:

```java
int[] list1 = {2, 4, 7, 10};
java.util.Arrays.fill(list1, 7);
System.out.println(java.util.Arrays.toString(list1));
```

```java
int[] list2 = {2, 4, 7, 10};
System.out.println(new java.util.Arrays.toString(list2));
```