a)

```
1  // Reads a series of high temperatures and reports the
2  // average and the number of days above average.
3
4  import java.util.*;
5
6  public class Temperature2 {
7      public static void main(String[] args) {
8          Scanner console = new Scanner(System.in);
9          System.out.print("How many days' temperatures? ");
10         int numDays = console.nextInt();
11         int[] temps = new int[numDays];
12
13         // record temperatures and find average
14         int sum = 0;
15         for (int i = 0; i < numDays; i++) {
16             System.out.print("Day " + (i + 1) + "'s high temp: ");
17             temps[i] = console.nextInt();
18             sum += temps[i];
19         }
20         double average = (double) sum / numDays;
21
22         // count days above average
23         int above = 0;
24         for (int i = 0; i < temps.length; i++) {
25             if (temps[i] > average) {
26                 above++;
27             }
28         }
29
30         // report results
31         System.out.println();
32         System.out.println("Average = " + average);
33         System.out.println(above + " days above average");
34     }
35 }
```

Here is a sample execution of the program:

```
How many days' temperatures? 9
Day 1's high temp: 75
Day 2's high temp: 78
Day 3's high temp: 85
Day 4's high temp: 71
Day 5's high temp: 69
Day 6's high temp: 82
Day 7's high temp: 74
Day 8's high temp: 80
Day 9's high temp: 87

Average = 77.88888888888889
5 days above average
```

b)

```
1  // Sample program with arrays passed as parameters
2
3  public class IncrementOdds {
4      public static void main(String[] args) {
5          int[] list = buildOddArray(5);
6          incrementAll(list);
7          for (int i = 0; i < list.length; i++) {
8              System.out.print(list[i] + " ");
9          }
10         System.out.println();
11     }
12
13     // returns array of given size composed of consecutive odds
14     public static int[] buildOddArray(int size) {
15         int[] data = new int[size];
16         for (int i = 0; i < data.length; i++) {
17             data[i] = 2 * i + 1;
18         }
19         return data;
20     }
21
22     // adds one to each array element
23     public static void incrementAll(int[] data) {
24         for (int i = 0; i < data.length; i++) {
25             data[i]++;
26         }
27     }
28 }
```

The program produces the following output:

```
2 4 6 8 10
```

c)

Table 7.2 Useful Methods of the Arrays Class

Method	Description
`copyOf(array, newSize)`	returns a copy of the array with the given size
`copyOfRange(array, startIndex, endIndex)`	returns a copy of the given subportion of the given array from `startIndex` (inclusive) to `endIndex` (exclusive)
`equals(array1, array2)`	returns `true` if the arrays contain the same elements
`fill(array, value)`	sets every element of the array to be the given value
`sort(array)`	rearranges the elements so that they appear in sorted (nondecreasing) order
`toString(array)`	returns a `String` representation of the array, as in `[3, 5, 7]`

The third limitation is that you can't compare arrays for equality using a simple == test. We saw that this was true of `String`s as well. If you want to know whether two arrays contain the same set of values, you should call the `Arrays.equals` method:

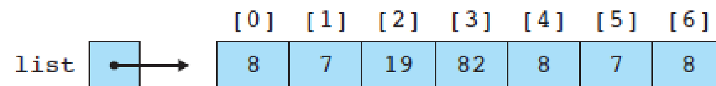
```
int[] data1 = {1, 1, 2, 3, 5, 8, 13, 21};
int[] data2 = {1, 1, 2, 3, 5, 8, 13, 21};
if (Arrays.equals(data1, data2)) {
    System.out.println("They store the same data");
}
```

This code prints the message that the arrays store the same data. It would not do so if we used a direct comparison with ==.

D)

Searching and Replacing

Often you'll want to search for a specific value in an array. For example, you might want to count how many times a particular value appears in an array. Suppose you have an array of `int` values like the following:



Counting occurrences is the simplest search task, because you always examine each value in the array and you don't need to change the contents of the array. You can accomplish this task with a for-each loop that keeps a count of the number of occurrences of the value for which you're searching:

```
public static int count(int[] list, int target) {
    int count = 0;
    for (int n : list) {
        if (n == target) {
            count++;
        }
    }
    return count;
}
```

You can use this method in the following call to figure out how many 8s are in the list:

```
int number = count(list, 8);
```

E)

Remember that a `return` statement terminates a method, so you'll break out of this loop as soon as the target value is found. But what if the value isn't found? What if you traverse the entire array and find no matches? In that case, the `for` loop will finish executing without ever returning a value.

There are many things you can do if the value is not found. The convention used throughout the Java class libraries is to return the value `-1` to indicate that the value is not anywhere in the list. So you can add an extra `return` statement after the loop that will be executed only when the target value is not found. Putting all this together, you get the following method:

```
public static int indexOf(int[] list, int target) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == target) {
            return i;
        }
    }
    return -1;
}
```

F)

As a final variation, consider the problem of replacing all the occurrences of a value with some new value. This is similar to the counting task. You'll want to traverse the array looking for a particular value and replace the value with something new when you find it. You can't accomplish that task with a for-each loop, because changing the loop variable has no effect on the array. Instead, use a standard traversing loop:

```
public static void replaceAll(int[] list, int target, int replacement) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == target) {
            list[i] = replacement;
        }
    }
}
```

Notice that even though the method is changing the contents of the array, you don't need to return it in order to have that change take place.

As we noted at the beginning of this section, these examples involve an array of integers, and you would have to change the type if you were to manipulate an array of a different type (for example, changing `int[]` to `double[]` if you had an array of `double` values). But the change isn't quite so simple if you have an array of objects, such as `String`s. In order to compare `String` values, you must make a call on the `equals` method rather than using a simple == comparison. Here is a modified version of the `replaceAll` method that would be appropriate for an array of `String`s:

```
public static void replaceAll(String[] list, String target,
                              String replacement) {
    for (int i = 0; i < list.length; i++) {
        if (list[i].equals(target)) {
            list[i] = replacement;
        }
    }
}
```

G)

Testing for Equality

Because arrays are objects, testing them for equality is more complex than testing primitive values like integers and doubles for equality. Two arrays are equivalent in value if they have the same length and store the same sequence of values. The method Arrays.equals performs this test:

```
if (Arrays.equals(list1, list2)) {
    System.out.println("The arrays are equal");
}
```

Like the Arrays.toString method, often the Arrays.equals method will be all you need. But sometimes you'll want slight variations, so it's worth exploring how to write the method yourself.

The method will take two arrays as parameters and will return a boolean result indicating whether or not the two arrays are equal. So, the method will look like this:

```
public static boolean equals(int[] list1, int[] list2) {
    ...
}
```

When you sit down to write a method like this, you probably think in terms of defining equality: "The two arrays are equal if their lengths are equal and they store the same sequence of values." But this isn't the easiest approach. For example, you could begin by testing that the lengths are equal, but what would you do next?

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length == list2.length) {
        // what do we do?
        ...
    }
    ...
}
```

Methods like this one are generally easier to write if you think in terms of the opposite condition: What would make the two arrays *unequal?* Instead of testing for the lengths being equal, start by testing whether the lengths are unequal. In that case, you know exactly what to do. If the lengths are not equal, the method should return a value of false, and you'll know that the arrays are not equal to each other:

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length != list2.length) {
        return false;
    }
    ...
}
```

If you get past the `if` statement, you know that the arrays are of equal length. Then you'll want to check whether they store the same sequence of values. Again, test for inequality rather than equality, returning `false` if there's a difference:

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length != list2.length) {
        return false;
    }
    for (int i = 0; i < list1.length; i++) {
        if (list1[i] != list2[i]) {
            return false;
        }
    }
    ...
}
```

If you get past the `for` loop, you'll know that the two arrays are of equal length and that they store exactly the same sequence of values. In that case, you'll want to return the value `true` to indicate that the arrays are equal. This addition completes the method:

If you get past the `for` loop, you'll know that the two arrays are of equal length and that they store exactly the same sequence of values. In that case, you'll want to return the value `true` to indicate that the arrays are equal. This addition completes the method:

```
public static boolean equals(int[] list1, int[] list2) {
    if (list1.length != list2.length) {
        return false;
    }
    for (int i = 0; i < list1.length; i++) {
        if (list1[i] != list2[i]) {
            return false;
        }
    }
    return true;
}
```

This is a common pattern for a method like `equals`: You test all of the ways that the two objects might not be equal, returning `false` if you find any differences, and returning `true` at the very end so that if all the tests are passed the two objects are declared to be equal.

H)

Before we leave the subject of reference semantics, we should describe in more detail the concept of the special value `null`. It is a special keyword in Java that is used to represent “no object”.

> **`null`**
> A Java keyword signifying no object.

The concept of null doesn’t have any meaning for value types like `int` and `double` that store actual values. But it can make sense to set a variable that stores a reference to `null`. This is a way of telling the computer that you want to have the variable, but you haven’t yet come up with an object to which it should refer. So you can use `null` for variables of any object type, such as a `String` or array:

```
String s = null;
int[] list = null;
```

There is a difference between setting a variable to an empty string and setting it to `null`. When you set a variable to an empty string, there is an actual object to which your variable refers (although not a very interesting object). When you set a variable to `null`, the variable doesn’t yet refer to an actual object. If you try to use the variable to access the object when it has been set to `null`, Java will throw a `NullPointerException`.

I)

Two-Dimensional Array as Parameter

Example 1

Here is a method that counts the number of negative values in a matrix.

```
//Precondition: mat is initialized with integers.
//Postcondition: Returns count of negative values in mat.
public static int countNegs (int[][] mat)
{
    int count = 0;
    for (int r = 0; r < mat.length; r++)
        for (int c = 0; c < mat[0].length; c++)
            if (mat[r][c] < 0)
                count++;
    return count;
}
```

A method in the same class can invoke this method with a statement such as

```
int negs = countNegs(mat);
```

Example 2

Reading elements into a matrix:

```
//Precondition: Number of rows and columns known.
//Returns matrix containing rows × cols integers
//        read from the keyboard.
public static int[][] getMatrix(int rows, int cols)
{
    int[][] mat = new int[rows][cols];   //initialize slots
    System.out.println("Enter matrix, one row per line:");
    System.out.println();

    //read user input and fill slots
    for (int r = 0; r < rows; r++)
        for (int c = 0; c < cols; c++)
            mat[r][c] = IO.readInt();   //read user input
    return mat;
}
```

To call this method:

```
//prompt for number of rows and columns
int rows = IO.readInt();      //read user input
int cols = IO.readInt();      //read user input
int[][] mat = getMatrix(rows, cols);
```