1. Provide short answers (at most two sentences each) to the following questions.

1.a What is the purpose of casting?
A: Casting allows to “convert” a value of one type to a value of a different type, in the sense that it creates a new value which is the representation of the original value in the new type. Casting does not convert a variable, or in any way change a variable. It could convert the value of the variable, but the variable remains unchanged.

1.b How do you generate an integer random number between 1 and 12 (inclusive) and assign it to a int variable num?
A: int num = 1 + (int) (Math.random() * 12);

The function Math.random() returns a random number between 0 (inclusive) and 1 (exclusive), so when we multiply this number by 12 and cast the result to an int, we obtain a random integer between 0 (inclusive) and 11 (inclusive), so we need to add 1 to obtain what was requested.

1.c What is wrong with the following code? Please give a way to correct it.
public static void giveBack(int r) {
    ... // Some instructions doing something with r
    return false;
}
A: The method is returning a boolean, but is declared as being a void method. The correction is to either change void to boolean or to remove the line return false;

1.d The variable a, of type int[] refers to an array of length 15. You now would like it to refer to an array of length 12. How can you achieve that (write the code)?
A: a = new int[12]; As the question makes clear, the variable already exists, so we do not need to declare it again.
2. What is the output when the following program ArrayCall is run?

```java
public class ArrayCall {
    public static void main(String[] args) {
        int[] q = new int[16];
        for (int i = 0; i < q.length; i++) {
            q[i] = (i * 2) / 3;
        }
        int z = 13;
        int y = duck(z, q);
        int[] a = q;
        z = duck(z, a);
        System.out.println(y);
        System.out.println(a[z - 1]);
        System.out.println(q[z]);
        System.out.println(q[z - 1]);
        System.out.println(q[z + 1]);
        System.out.println(a[z + 1]);
        System.out.println(q[q.length - 1]);
    }
    public static int duck(int x, int[] a) {
        x = x - 1;
        int t = a[x];
        a[x] = a[x] + 3;
        return t;
    }
}
```

A: The most important part is to notice that there is only one array, and the `int[]` variables `a` and `q` in the method `main`, are “pointing” to the same array, which is the only existing array. So `a[j]` and `q[j]` refer to the same element of the only existing array, for every value of `j`.

The array is filled as follows in the for-loop at the beginning of `main` (notice the integer divisions):

```
0 0 1 2 2 3 4 4 5 6 6 7 8 8 9 10
```

We then call `duck(z, q)` with `z=13`. In the ensuing call to the `duck` method, we therefore have `x=13` and `a` referring to the only array that exists (not to a copy of the array). In the method, the variable `t` gets the value in the cell of index 12 of the array. The value in this cell is 8. Then, the value in this cell is incremented by 3. At this point, the array is

```
0 0 1 2 2 3 4 4 5 6 6 7 11 8 9 10
```

The value of the variable `t` (which is 8) is then returned to the `main` method, and stored in the variable `y`. In `main`, after having created the variable `a` and having it pointing to the same array that the variable `q` points to (which is the only array that exists), we call `duck` again, with
parameter values 13 (the value of the variable \(z\)) and the address of the only existing array, which is stored in \(a\) (and also in \(q\)). So the values for the parameters of this call to \(duck\) are exactly the same as in the first call to \(duck\). So \(duck\) once again saves in \(t\) the value of the cell of index 12 of the array. The value in this cell is 11. Then, the value in this cell is incremented by 3. At this point, the array is

\[
\begin{array}{cccccccccccc}
0 & 0 & 1 & 2 & 2 & 3 & 4 & 4 & 5 & 6 & 6 & 7 & 14 & 8 & 9 & 10
\end{array}
\]

The value of the variable \(t\) (which is 11) is then returned to the \texttt{main} method, and stored in the variable \(z\). So \(z\) has value 11, \(z+1\) is 12, and \(z-1\) is 10.

Since \(a[z-1]\) and \(q[z-1]\) refer to the same element of the only existing array, the same value will be printed (which is 6), and similarly for \(a[z+1]\) and \(q[z+1]\), which contain the value 14. The other values can be obtained from the described values for \(y\), \(z\), and the above list of the elements in the array. So the final output is:

8
6
7
6
14
14
10
3. Write a method findAll with the following specifications. Its parameters are two arrays of integers called d and a. The array d contains distinct integers (i.e., all values in this array are different), while a may contain duplicate values. The method should return true if all values stored in d are present in a, and false otherwise. The array a may contain values that are not in d. Do not assume that d and a are sorted.

A: First of all, notice that it is not said anywhere that the arrays have the same length, so you should not assume that. Also, it should be clear that the arrays already contain values, so you do not need to set the values to arbitrary (or not so arbitrary) values.

The main idea is to iterate over the elements of d, and for each of them, iterate over the elements of a: if we do not find any element of a equal to the element of d that we look for, then we can safely return false. If we manage to iterate over all elements of d without returning false, then we can safely return true, because it means that for each of the elements in d we found at least one element of a equal to it.

Reading the instructions one may notice that if a is shorter than d, then we can safely return false immediately. This is a nice observation that we can use in our code, but it is not necessary.

Here is the code implementing the above idea. Many other implementations of the same idea are possible, and some of you came up with very nice ones.

```java
public static boolean findAll(int[] d, int[] a) {
    if (a.length < d.length) {
        return false;
    }
    for (int i=0; i < d.length; i++) {
        int j = 0;
        boolean found = false;
        while (j < a.length && ! found) {
            if (a[j] == d[i]) {
                found = true;
            }
            j++;
        }
        if (! found) {
            return false;
        }
    }
    return true;
}
```
4. Write a method `findAllSorted` that behaves like the method `findAll` from Exercise 3, but assumes the arrays `d` and `a` are sorted in non-decreasing order.

A: While the code from exercise 3 would have worked for this exercise, it is not using in any way the fact that the two arrays are sorted.

The idea for how to modify the code from exercise 3 is best given with an example. If we found that `a[3]` is equal to `d[2]`, then when we want to look for `d[3]` we do not have to start our search from `a[0]` because the arrays are sorted and `d[3]` is greater than `d[2]` so it can only appear in `a` at an index greater than the one where we found `d[2]`. So we can start our search for `d[3]` from `a[4]`.

The modifications to the code from exercise 3 are minimal: we just need to avoid declaring a new variable `j` and initialize it to zero at every iteration of the for loop. That corresponds to moving `int j=0;` to before the for loop.

```java
public static boolean findAllSorted(int[] d, int[] a) {
    if (a.length < d.length) {
        return false;
    }
    int j = 0;
    for (int i=0; i < d.length; i++) {
        boolean found = false;
        while (j < a.length && ! found) {
            if (a[j] == d[i]) {
                found = true;
            }
            j++;
        }
        if (! found) {
            return false;
        }
    }
    return true;
}
```
5. Write a variant `zdInsertionSort` of `insertionSort` that takes an array `a` of positive integers, and sort the array, while replacing all but one copy of every duplicate value with a 0 (zero). All the zeroes must appear, when the array is sorted, in positions with lower index than any positive value. For example, the input array |4|6|3|4|2|4|3| should become |0|0|0|2|3|4|6|.

**A:** There are many good answers to this question. One of them is to use `insertionSort` to sort the array, then set the duplicates (which are now adjacent to each other) to zero, and then sort the array again. Another more efficient solution is to do a nested for-loop over the array to set the duplicates to zero, and then sort the array with `insertionSort`: as the duplicates are now zero, they will magically go to the beginning of the array.

The following solution is (likely) the most efficient: it literally modifies `insertionSort`, noticing that we can slightly change the while loop condition so that, in the body of the while loop, we can check whether the value stored in `t` is a duplicate, in which case we can set `t` to zero and continue with our sorting. The code can be optimized to perform an earlier exit from the while loop in case `t` is already zero.

```java
class MyArray {
  public static void zdInsertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
      int t = a[i];
      int j = i - 1;
      while (j >= 0 && t <= a[j]) {
        if (t == a[j]) {
          t = 0;
        }
        a[j + 1] = a[j];
        j--;
      }
      a[j + 1] = t;
    }
  }
}
```