**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 12 – Arrays**

In this lab sheet you will see how arrays are passed into methods and how the contents of an array can be sorted and searched.

**Call-by-value**

In Java, when you pass information into a method, the technique used is referred to as **call-by-value**. This technique means that a **copy** of the original variable’s value is made into the method argument, when the method is called e.g. consider the following fragment of code from main()

System.out.print(“Please enter a number: ”);

number = input.nextDouble();

System.out.println(“The square root of the number you entered is ” + Math.sqrt(number));

Now consider the sqrt() method definition:

public static double sqrt(double x)

{

// sqrt() method body processes the argument x

}

Imagine that at runtime the user enters 17.56 for the number. This gets stored in the double variable number and then the Math.sqrt() method gets called. Note that when the sqrt() method is called, the value stored in the variable number is then copied into the sqrt() method argument x. After the copy is made there is no longer a link between the main() method and the sqrt() method. A link will be reestablished later when the sqrt() method eventually returns the square root of the number. The important thing to note with call-by-value is that the method receives only a copy of the value stored in some variable and so **cannot directly alter the original variable value**. This is generally considered a good feature because it could happen otherwise that the value might be accidentally (or deliberately) altered.

**Passing Ordinary Variables versus Object References**

In Java, **all** **values are passed to methods using the call-by-value techniqu**e. This applies not only for values associated with primitive data types such as int, char, float etc. but also for values associated with more complex data types i.e. objects.

However, it should be noted that **objects are never passed to a method**. Instead, a **reference to the object is passed to the method** (using call-by-value). If you recall the analogy of object references and **remote controls**, it means that, because the method is passed a copy of the remote control, it can **directly modify the object** which that remote control is pointing to currently. This is how things work for arrays since **an array is an object**.

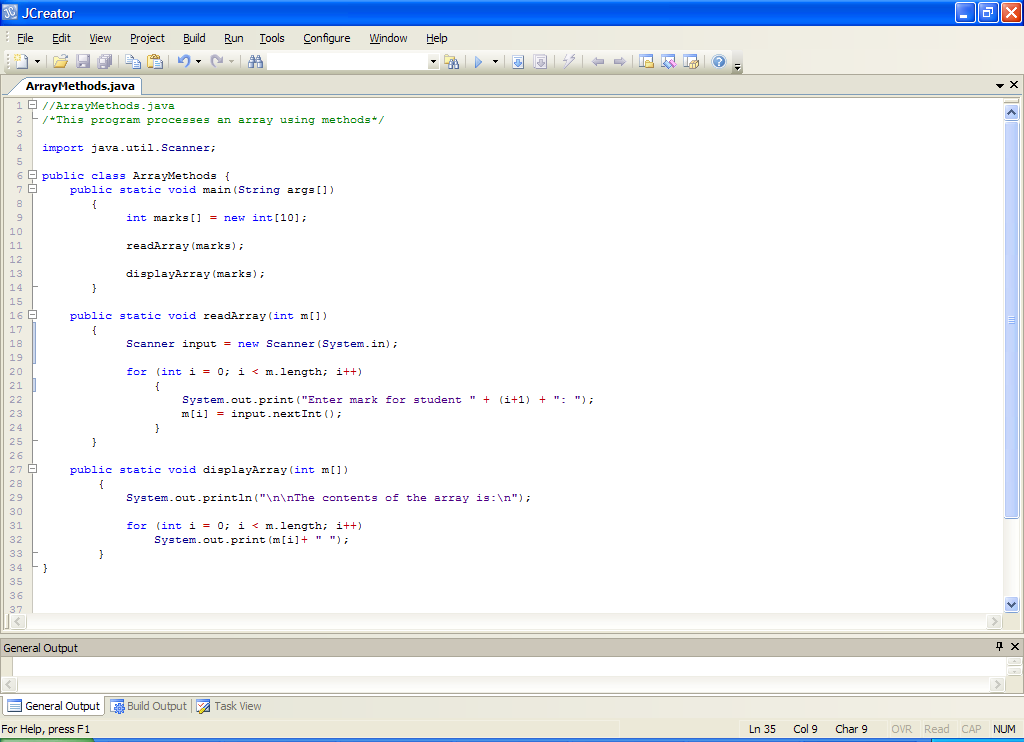
So there is an important, but subtle, difference between passing a value of an ordinary variable and passing an object reference value to a method. **Both use the call-by-value technique** but it is only in the case of passing object references that the method can directly modify the original information.

Although the ability of a method to directly modify an object is usually considered a **security threat**, it is important to be able to pass object references rather than whole objects because an object can potentially grow to an arbitrarily large size and making a copy of each and every individual data value associated with the object could take a lot of time and consume considerable storage. Therefore, for **performance considerations**, object references are passed rather than objects.

The next program we examine looks at passing an array into a method

**Aim:** We wish to write a program that will use an array of 10 integer values to store 10 marks and then display these marks. Both the storage and displaying will take place via user-defined methods

**Java Code**:



**Program Analysis:**

● The program declares an integer array of size 10. marks is the object reference that “points to” this array. Then the readArray() method is called and the object reference marks is passed into the method. Note that this is the one place where **square brackets are omitted**. Now the readArray() method can directly modify the contents of the array and this is exactly what happens within the code of readArray(), where the user is prompted for a list of 10 numbers and these are added, one by one, to the array marks.

If you look at the code for readArray(), you see that its method argument is called **m**. There are **square brackets** after it to indicate that the method expects to be passed an object reference of type integer array as an argument. This is exactly what happens in main() when we call readArray() – we pass in an object reference of type integer array called marks. So, temporarily, while the method is executing, the array object we created on line 9 is being pointed to (referenced by) two different object references – marks and m, as indicated in the diagram below

The **array** **object**

Object reference **marks**

Object reference **m**

This means that, whatever changes are made to the object through the object reference m in the method readArray() are also “visible” to the object reference marks back in main(). In this way, the **method is** **directly modifying the original array object** created in main().

● When the readArray() method has completed, the array we created will be completed populated with values. Next we call the displayArray() method and again pass as an argument, the object reference marks. Note that we couldn’t pass the object reference **m** here because it is **not visible** outside of the method it is defined in i.e. readArray(). It is for this same reason that we get away with calling the integer array argument m in the displayArray() method also.

The displayArray() method uses the object reference m to refer to the array object and in this case it uses it to display the entire contents of the array via a for loop.

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab12** within OOP1Stuff to save your work from this lab session.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **ArrayMethods.java** in your Lab12 folder. Now, for practice, type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

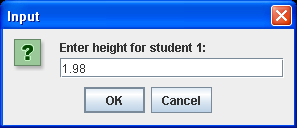
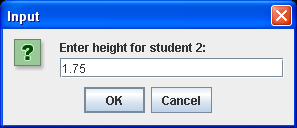
Once you are free from errors, run the program and test it fully. With proper testing you should see that this program is **not validated**

**Exercise 1**

Write a program called **Exercise1.java** which makes use of an array of floats called heights of size 10. Apart from main(), this program should have 5 user-defined methods. The purpose of these 5 methods, respectively, is as follows:

1. prompt for and read in the heights of the 10 students
2. calculate and return the average height of the 10 students
3. calculate and return the height of the tallest student
4. calculate and return the height of the smallest student
5. display a list of the heights that were above 1.7 metres

The program should run as indicated in the following sample screenshots:

……. **8 more input values supplied** ……

**Searching an Array**

On occasions it may be important for us to determine whether or not an array contains a value that matches some **key value** that has been specified. For example, suppose we have an array of names and a corresponding (parallel) array of telephone numbers. You can imagine how we might wish to search the array of names based on a key value (the person we seek) in order to get their telephone number.

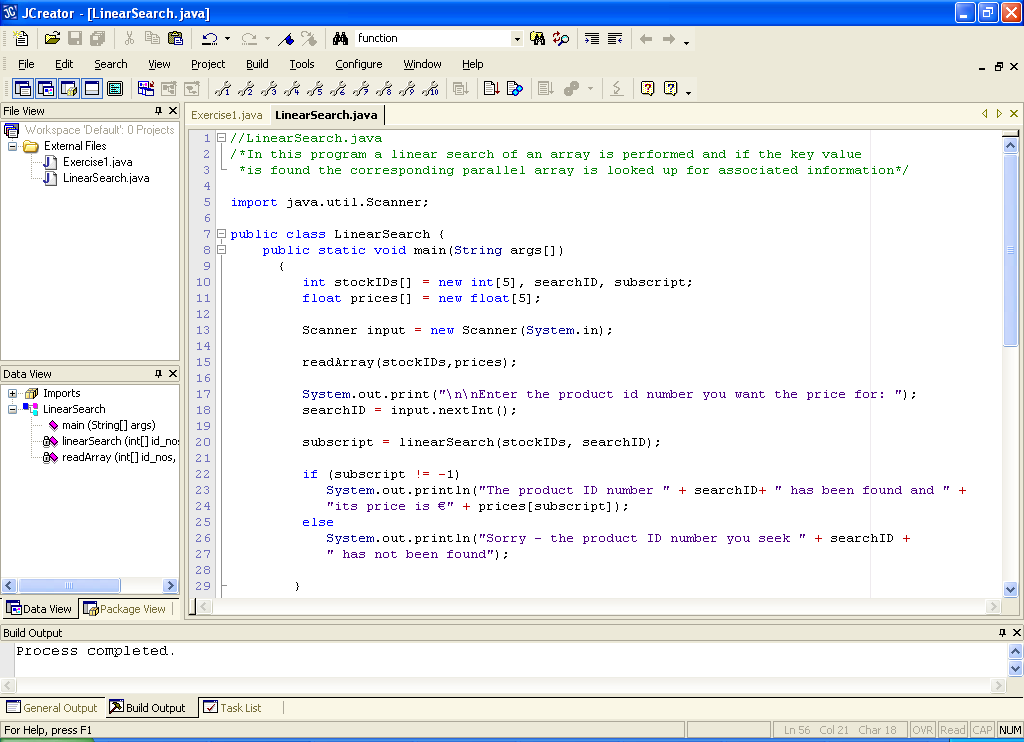
**Linear Search**

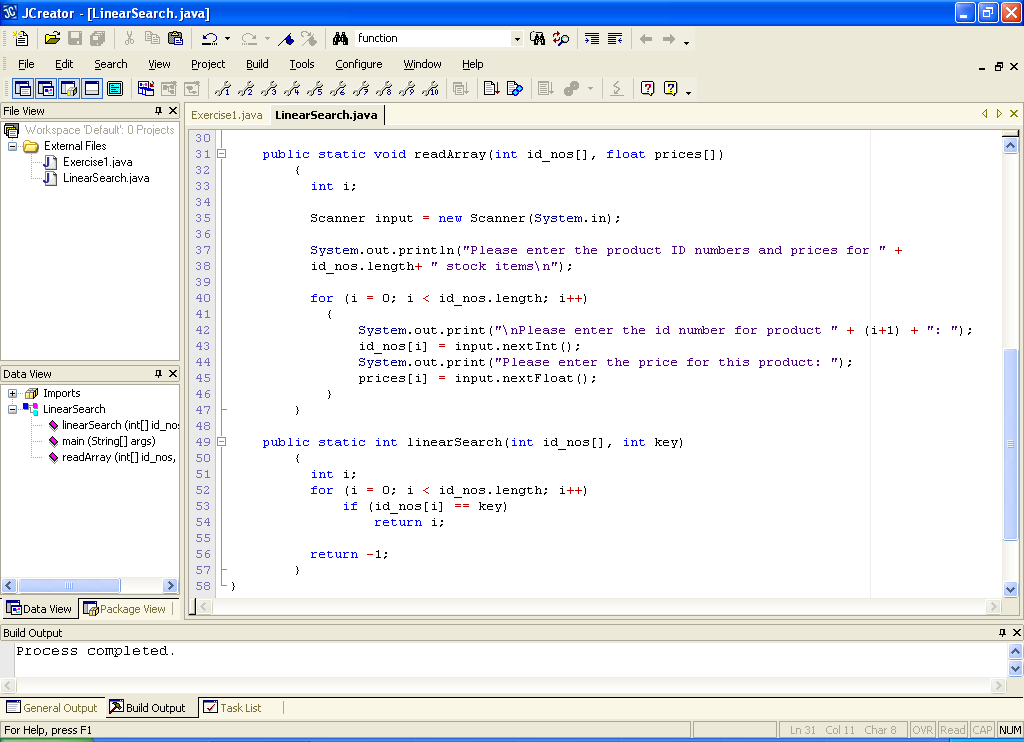
There are many different searching algorithms and some are certainly more efficient than others but for **simplicity** sake, we shall consider a technique called a **linear search**. As the name implies we are **searching through an array in a straight line** as it were, from the start of the array until we find the value we seek (which we may not find).

The following sample program makes use of a user-defined method called **linearSearch**() for this purpose.

**Aim:** We wish to write a program that will use 2 related (parallel) arrays. One of these will store the stock ID numbers for products in a shop and the other the corresponding price of that product. The arrays will be populated and then the user asked to enter a particular stock ID number. A linearSearch() method then searches the stock ID array and if the product exists, the corresponding product price is displayed.

**Java Code**:





**Program Analysis:**

● The program declares 2 parallel (related and of the same size) arrays called stockIDs and prices which will respectively store the product ID numbers and corresponding prices of items in the corresponding slots of their arrays.

The variable searchID is used in the program to store the stock ID number of the user’s search key and subscript is used to store the value returned by the call to the user-defined method linearSearch().

● The readArray() method populates the arrays with user-supplied values with the ID numbers and prices for a given item stored in corresponding slots of each array. Note again that when the method is called, the arrays are passed by only using the array names – there are **no square brackets**.

● After this the user is prompted to enter the stock ID number for the product they wish to find the price of and this is stored in the variable searchID.

● Next the linearSearch() method is called. This takes 2 arguments – the array that will be searched and the search key. We need to search the stockIDs array using the search key stored in searchID, so these are the variables passed to the method. The order is dictated by the order they were written in the method definition itself, so the array is first, the key second.

● When the linearSearch() method is called, it will loop through the stockIDs array (via the **object reference** variable id\_nos) and, on each loop iteration it will check to see whether the key value stored in key matches the value contained in that slot of the array.

id\_nos[i] == key

If there is a match then the product with that ID number must exist in the array and then the subscript number associated with that slot is returned and stored in the variable subscript. The code

return i;

returns the subscript number. It should be noted that as soon as a match is found, the subscript is returned and the method exits.

Returning this subscript number here is crucial since we use it eventually to pick out the corresponding item price from the prices array with the code:

if (subscript != -1)

System.out.println("The product ID number " + searchID+ " has been found and " + "its price is €" + prices[subscript]);

Note here that the code checks to see if the value of subscript is -1. If it isn’t then the value contained in subscript must be the subscript number of the element in the array containing the search key.

If it did happen to be -1, then this signals that the search key was not found in the array.

The code

return -1;

returns the value -1 to signal that the product ID entered by the user was not found in the array.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **LinearSearch.java** in your Lab12 folder. Now, for practice, type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

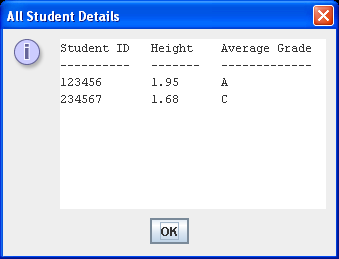
Once you are free from errors, run the program and test it fully. With proper testing you should see that this program is **not validated**

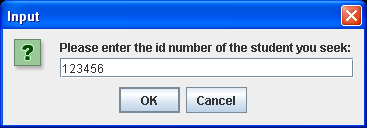
**Exercise 2**

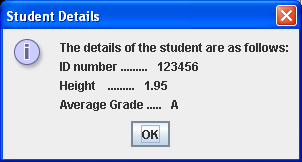
Write a program called **Exercise2.java** which creates 3 parallel arrays. The first should be an integer array of 5 student id numbers. The second should be a float array of associated heights and the third a character array of associated alphabetical grades (i.e. A,B,C etc). The program should use a method called **populateArrays**() to dynamically enter the information for the 3 related arrays and then use a method called **displayTable**() to output the information in a nice, tabular format as follows using a text area.

Finally, the program should ask the user to enter the id number of a particular student and determine, using a **linearSearch**() method, if the student exists in the array of student ids. If it does, the student id, along with the rest of that student’s details should be output. If it does not then a message should be given to indicate that the particular student sought did not exist in the array at all.

Your program should run as indicated in the following sample screenshots (note that I only had arrays of size 2 for my shots below and I haven’t shown the populateArrays() inputs for brevity)







**Sorting an Array**

The ability to sort data i.e. place data into some particular order such as ascending or descending is, like searching, one of the most important computing applications. A bank sorts all cheques by account number so that individual bank statements can be prepared at the end of each month. The “Golden Pages” sort the list of entries by last name, and within that, by first name to make it easy to find phone numbers. Virtually every organisation needs to sort some data. Sorting data is so important that it is one of the most intensely researched areas of computer science. There are a plethora of various sorting algorithms available. Some are more efficient in terms of execution time than others and these algorithms are normally more complex than the simpler sorting algorithms.

Some of you will spend a share of time next year in your **Algorithms** **module** dealing with arrays and these sorting algorithms and trying to understand how they operate at a relatively low level. Therefore, for this module, **we will deal with sorting an array at a higher level** (remember the term “**abstraction”** from the problem-solving section of last semester’s module?). In other words, you will not be inspecting the code that does the sorting at all – you will simply use an **existing Java API method** that has already been defined to do that job. This makes our task much, much easier and it is another example of **software re-use**.

**The Arrays Class**

You may be surprised to learn that there is a class called **Arrays** already in the Java API. Remember that the Java API has literally hundreds of predefined classes and each of these (normally) contains many methods to allow you to carry out various tasks.

This class allows you to easily manipulate arrays within your programs and is especially geared towards searching and sorting. If you examine the **Java API documentation** for this class (open it up now) you will see that there are many methods associated with the class. The ones we are most interested in using are the **binarySearch**() and the **sort**().

**Exercise 3**

Using the Java API documentation, find out the name of the particular sorting algorithm used by all of the various sort() methods.

***The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.***

**Exercise 4**

Using the Java API documentation, determine the method definition header for the method that would be used to make a copy of an existing double array. You can take it that the copy will be exactly the same size as the original.

|  |  |
| --- | --- |
| static double[] | [**copyOf**](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#copyOf%28double%5B%5D,%20int%29)(double[] original, int newLength)  Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length. |

**Exercise 5**

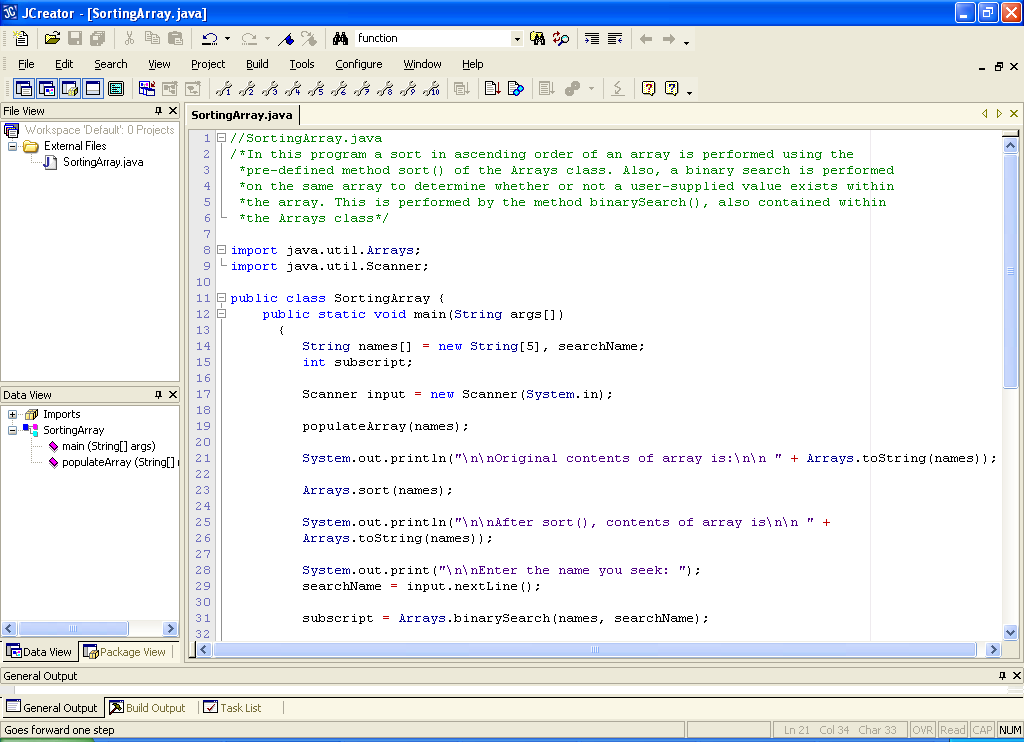
Using the Java API documentation, determine the method definition header for the method that would be used to return a string representation of the contents of an array of character values.

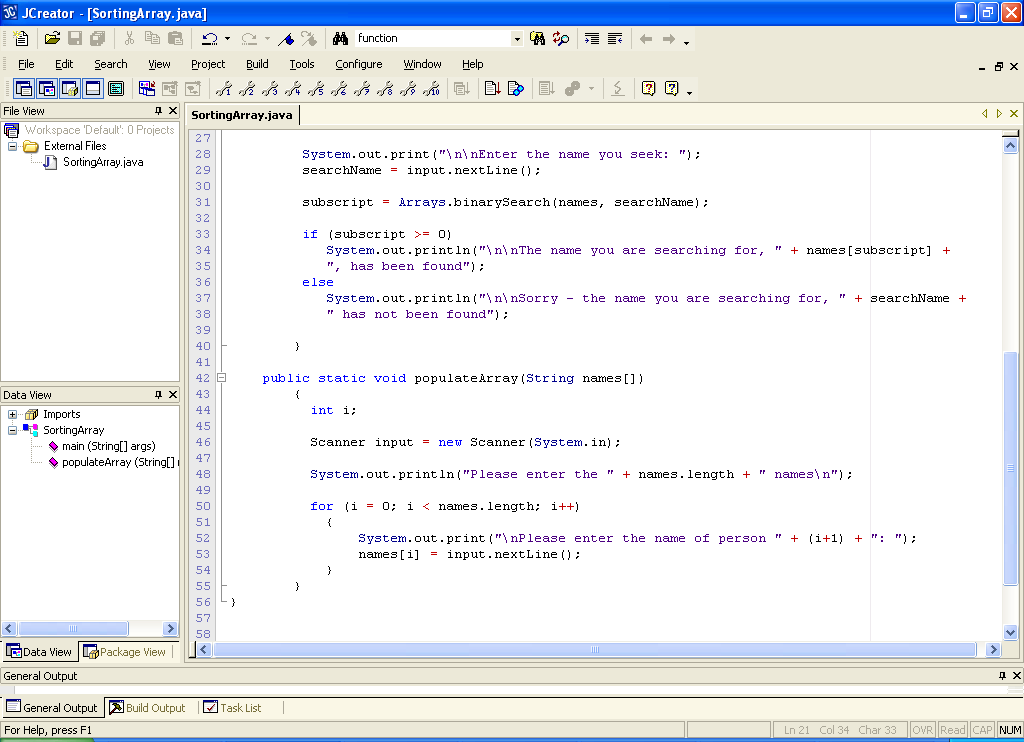
|  |  |
| --- | --- |
| static [String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**toString**](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#toString%28char%5B%5D%29)(char[] a)  Returns a string representation of the contents of the specified array. |

Next we will examine a program that uses some of the predefined methods contained in the **Arrays** class.

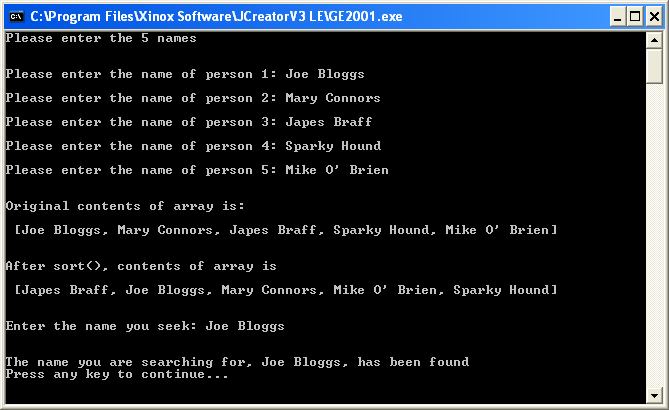
**Aim:** We wish to write a program that will use a method called populateArray() to read in exactly 5 user-supplied names and store them in a String array. This will then be sorted into ascending order by first name and then the user will be asked for a name and a binary search will be performed on the array to determine whether or not the name exists in the array.

**Java Code**:





**Sample Output**



**Analysis of Program**

• The program begins by declaring an array of String of size 5. This is then populated using the populateArray() method.

• The contents of the array is then displayed to the screen using the Arrays class toString() method. This method returns a comma-separated list of the contents of the array it takes as an argument, surrounded by square brackets. The version of the toString() method used in our program is defined as follows:

|  |  |
| --- | --- |
| static [String](http://java.sun.com/javase/6/docs/api/java/lang/String.html) | [**toString**](http://java.sun.com/javase/6/docs/api/java/util/Arrays.html#toString(java.lang.Object[]))([Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html)[] a) |

You can see that the method is **static** – remember that this always means we should **put the name of the class it belongs to in front of it** when we call it (separated by a dot) i.e. our code here is:

System.out.println("\n\nOriginal contents of array is:\n\n " + **Arrays.toString**(names));

We can also see the method returns a String. This means that we can put it into our println().

One thing you may be wondering about is why the argument type here is **Object[]** and not **String[]**. The funny thing is, there is no toString() method available that specifically takes a String array as an argument. It isn’t necessary because it is covered by this method that takes an Object array as an argument. Object is a special class in Java and, for now, you can just take it that if Object appears in an argument list as a type, then literally **any type** of object reference (including String object references) can be passed in for that method argument. You will learn more about this in your OOP2 module next September.

• The next line of code

Arrays.sort(names);

just calls the sort() method to perform a quicksort algorithm on the names array, to sort it into ascending order by firstname. The version of the sort() method used in our program is:

|  |  |
| --- | --- |
| static void | [**sort**](http://java.sun.com/javase/6/docs/api/java/util/Arrays.html#sort(java.lang.Object[], int, int))([Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html)[] a) |

Again, this method is static. However its return type is void, so it returns nothing to the method calling it (main() in this case). It receives an Object array as an argument but as we now know, this also covers String arrays. When the array is received, the quicksort is perfomed and the array will have been sorted into ascending order by the time the method exits.

This is proven when the next line of code executes, using toString() once more to display the array’s contents.

• The next section of the program asks the user to enter a search key. The user enters this and then the binarySearch() method is called to determine whether or not the search key exists within the array. Recall that we wrote our own linearSearch() method to do the same thing recently. However, a **binary search is much more efficient than a linear search** and effectively “chops” the search range in half every time it iterates. This is another algorithm that some of you will probably spend time studying next year in your **Algorithms module**. It’s simple for us though – we just need to be able to call the method correctly and we do this as follows:

subscript = Arrays.binarySearch(names, searchName);

Looking at this code, hopefully you can see that binarySearch() must be a static method. Hopefully you can also see that it takes 2 arguments, the first of which is the array we wish to perform the search on and the second is the search key value. The version of the binarySearch() method used in this program is:

|  |  |
| --- | --- |
| static int | [**binarySearch**](http://java.sun.com/javase/6/docs/api/java/util/Arrays.html#binarySearch(java.lang.Object[], java.lang.Object))([Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html)[] a, [Object](http://java.sun.com/javase/6/docs/api/java/lang/Object.html) key) |

The argument returned by binarySearch() is an int which is the subscript number of the element in the array containing the key if it actually exists. If it does not exist then the method will return a value that is negative and this is how we can test to see if the key was found.

if (subscript >= 0)

Note that both arguments involve Object, but this covers String again.

**It should be mentioned that a binary search can only be carried out provided that the array has already been sorted. Without having been sorted in advance, the result is undefined and constitutes a logical error.**

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the file as **SortingArray.java** in your Lab12 folder. Now, for practice, type in the code for the program above.

If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and test it fully.

**Exercise 6**

Recall that some time back you wrote a program which generated 7 random numbers between 1 and 42 to simulate the Irish lotto draw. The original version was flawed because it was possible for numbers to appear multiple times in the same “draw”. You were given an exercise that used a “band-aid” solution to sort out this problem. Although this solution works perfectly, it is not really the most elegant solution. Ideally, you would use some **arrays** to solve the problem nicely.

You should now use an array as follows to store the chosen numbers:

int lottoNumbers[] = new int[7];

To check if a number has already been picked, use another array as follows:

boolean alreadyPicked[] = new boolean[43];

When a number n is chosen, you will set alreadyPicked[n] to true.

Write this program and save it as **Exercise6.java**

You should use the **Arrays** class **toString**() method to display the contents of the lottoNumbers array to the message dialog window. The lotto numbers should also be **sorted** in ascending order (don’t worry about the fact that the “bonus” number gets mixed up by doing this). Run your program several times to ensure it is giving 7 different numbers each time. Your program should run as indicated in the following sample screenshot.

