**Object Oriented Development using Java**

OOD Week 1 – Module 2

Arrays & Lists

Tutorial

© FDM Group Ltd 2020. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce you to data types which hold multiple values.

# How long will the tutorial take to complete?

30 minutes

# What should you have already completed?

Module 1 (Data types)

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Apache Maven
* Eclipse IDE Kepler or above

# What does this tutorial cover?

* Arrays
* ArrayLists
* Collections methods
* Arrays methods

# What are Arrays & Lists

In the previous module you’ve seen data types which hold a single value. For instance an int holds one number, a String holds one piece of text. Arrays & Lists hold multiple values of the same data type.

# Arrays

An array holds a fixed number of variables of the same data type. Each array is based on a single data type. That type could be a primitive or an object type. It’s a very basic structure that has no methods.

Here’s a simple example of how to create an array:

**double**[] prices = {2.5, 1.99, 9.99, 7.75};

Notice that the data type is followed by []. This indicates that we’re creating an array or double values rather than just a single double. Any data type can be used this way to create an array.

This array contains 4 values (specified in the braces). The size of the array can never be changed. In this case it will always contain 4 elements. However we can change the values within the array.

If we try to print out the array, we won’t see anyting useful:

System.***out***.println(prices);

In fact all we’ll see is the memory address of the array and not its contents:

[D@7852e922

The elements of the array are indexed, counting up from zero. To access an element, you need to specify its index:

System.***out***.println(prices[0]);

System.***out***.println(prices[1]);

System.***out***.println(prices[2]);

System.***out***.println(prices[3]);

Try running the 4 lines of code above, and you’ll see it prints each of the 4 values within the array.

If we want to change a value within the array we can do this:

prices[3] = 7.49;

However, we can’t create a new element within the array:

prices[4] = 11.25;

If we try to run the code, we’ll see the following error:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4

at com.fdmgroup.datatypes.Runner.main(Runner.java:13)

If you want to see how many elements are contained in the array you can do this:

System.***out***.println(prices.length);

#### Other ways to create an array

There are some other ways to create an array. Here’s an example:

**double**[] prices = **new** **double**[5];

This creates an array of 5 doubles. Initially all of the values are set to zero.

Here’s another example:

String sentence = "this is a sentence";

String[] words = sentence.split(" ");

In this case the String’s split method splits the String using the space character into 4 separate Strings. Each of the 4 Strings contains one of the words from the sentence.

## The Arrays class

The Arrays class gives us a few utility methods to help manipulate an Array. The most commonly used of these methods is sort(). Here’s an example:

**double**[] prices = {2.5, 1.99, 9.99, 7.75};

Arrays.*sort*(prices);

System.***out***.println(prices[0]);

System.***out***.println(prices[1]);

System.***out***.println(prices[2]);

System.***out***.println(prices[3]);

In this case when you print out the 4 elements, you’ll see that they’re now sorted in ascending order.

# ArrayLists

You’ve seen that arrays can store multiple values of the same data type. But they’re limited in what they can do. Their size can’t be changed and they have no methods. ArrayLists are part of a wider category called ‘Lists’ which in turn is part of a wider category called ‘Collections’. You’ll cover both of these in OOD week 2.

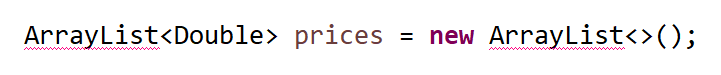
Like an array, an ArrayList holds multiple values of the same data type, but it’s size is completely flexible and it has lots of methods to help us manipulate its values.

Here’s an example of how to create an ArrayList:

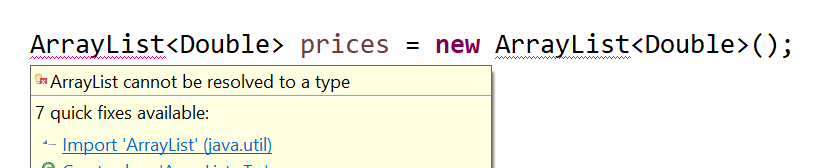
ArrayList<Double> prices = **new** ArrayList<Double>(); Java 7 syntax

ArrayList<Double> prices = **new** ArrayList<>(); Java 8+ syntax

In either case, when you first try to create an ArrayList in your method you’ll see a compile error:



This is easy enough to fix. It just needs to be imported. Your IDE will help you do this:



Choose the ‘Import ArrayList’ option and you’ll see that the following line of code is added at the top of your class:

**import** java.util.ArrayList;

Notice that the ArrayList is based on an object type i.e. Double. It’s not possible to base it on a primitive type:



This is not a problems as we have a wrapper equivalent for each of the 8 primitives.

At this point, you’ve created an empty ArrayList. You can add as many values as you like to it by using its add() method:

prices.add(2.5);

prices.add(7.99);

It’s also possible to initialise the ArrayList with a set of values when you create it:

ArrayList<Double> prices = **new** ArrayList<>(Arrays.*asList*(2.5, 7.99, 3.25));

Either way, the ArrayList is completely flexible and you can add or remove values from it at any time.

Let’s look at some other useful methods which an ArrayList has:

System.***out***.println(prices.size());

The size() method will tell us the number of elements in the ArrayList. It’s the equivalent of an array’s ‘length’ attribute.

You can remove an element from an ArrayList using the remove() method. There are two different ways of doing this:

prices.remove(2.5);

prices.remove(1);

The first example removes the first element from the ArrayList which has a value of 2.5. It won’t remove any subsequent elements with the same value.

The second example removes the element at position 1.

If you call the size() method after calling remove(), you’ll see that the size of the ArrayList has shrunk.

You might have realised there’s a potential issue with the remove() method if the ArrayList is based on an Integer value. Here’s an example:

ArrayList<Integer> idNumbers = **new** ArrayList<>(Arrays.*asList*(1,2,3,4,5));

idNumbers.remove(1);

The obvious question here is will it remove the value 1 or the index 1? Try running this to find out.

You should see that it actually removes the index 1 i.e. the value 2.

If you want to access a value within an ArrayList you can use the get() method:

System.***out***.println(prices.get(0));

System.***out***.println(prices.get(1));

System.***out***.println(prices.get(2));

The contains() method will tell you (true or false) whether an element exists in the ArrayList or not:

System.***out***.println(prices.contains(2.5));

The removeAll() method can also be very handy. It can remove the contents of one ArrayList from another ArrayList:

ArrayList<String> strings1 = **new** ArrayList<>(Arrays.*asList*("string1","string2","string3","string4"));

ArrayList<String> strings2 = **new** ArrayList<>(Arrays.*asList*("string2","string4"));

strings1.removeAll(strings2);

System.***out***.println(strings1.size());

System.***out***.println(strings1.get(0));

System.***out***.println(strings1.get(1));

Run this and you’ll see that the ArrayList strings1 now contains only “string1” and “string3”

# The Collections class

This is the equivalent of the Arrays class but for ArrayLists and other types of Collections that you’ll see in later weeks. You’ll see that it can do much more than the Arrays class does.

Let’s start off with a sort:

ArrayList<Double> prices = **new** ArrayList<>(Arrays.*asList*(2.5, 7.99, 3.25, 10.25, 7.99));

Collections.*sort*(prices);

Collections.sort() works in exactly the same way as Arrays.sort(), but it can do much more:

Collections.*reverse*(prices);

**double** highestPrice = Collections.*max*(prices);

**double** lowestPrice = Collections.*min*(prices);

**double** occurrencesOf7point99 = Collections.*frequency*(prices,7.99);

Collections.reverse() reverses the order of elements

Collections.max() tells us the highest number in the ArrayList

Collections.min() tells us the lowest number in the ArrayList

Collections.frequency() tells us how many times a particular value occurs in the ArrayList. In our example, 7.99 occurs twice.