**Section 3: Constructing and Using Arrays**

# Subsection 1: Understand one-dimensional arrays (10 min)

## What is an array?

An array is a type of data storage that allows a collection of separated but logically related data to be stored together. Arrays are used to store things like: a list of pupil names, a list of favourite things and a list of scores, in fact anything that can logically be stored as a list can be stored in an array.

Because an array contains data and it has to declare a data type, just as a variable has to declare its data type. This data type can be a primitive or an object type.

We have discussed different primitive data types such as int, byte, short etc. and we have seen object types such as String and BankAccount. Arrays can be constructed to contain these data types.

### Declare and Create an Array

### Alternative creation and initialisation

There are other ways to create and initialise an array. For example, an array can be declared, created and initialised all on one line of code:

int[] testScores = new int[] {4, 7, 2};

The length of this array is determined by the quantity of initial values specified between curly braces. In this code snippet, its length is three. Take note that it is illegal to specify a length as well as a list of initial values. There could be a sneaky question on the exam that tries to trip you up.

### Alternative declaration and initialisation

Arrays can be declared and initialised in ways which vary only slightly. The table below gives a demonstration of some of these ways.

|  |  |
| --- | --- |
| Array declaration and initialisation | Notes |
| int[]scores = {180, 11, 12}; | Referred to as an anonymous array as we don’t specify the length or type. |
| int scores[] = new int[10]; | The square brackets can also follow the variable name. |
| int[] scores = new int[0] | You can create an array with zero elements. |
| int [] scores; | The additional space does not make this declaration illegal. |
| int scores []; |
| aMethodCall(new int[]{200, 100, 50}); | See discussion regarding anonymous arrays later on in this lesson. |
| int[] scores, results; | This declares two arrays of type int. |
| int scores[], results; | This declares one array of type int (scores) and one variable of type int (results). |

## Anonymous arrays

One final way to create arrays is to do so on-the-fly by creating an anonymous array as an argument to a method call. Take a look at the following code snippet.

public void aMethod(int[] scores){  
 // Do something  
 }

The method *aMethod* expects to be passed an integer array. This can be done by creating the array in the method call itself.

aMethod(new int[]{200, 100, 50});

The array is created and its reference passed to the variable scores. It becomes a local array to the method *aMethod*. There is no external reference to the array and can only be accessed within the method. Once method execution finishes it will be garbage collected.

## Object arrays

An array stores any type, including object types. This means we can have an array of String objects, an array of BankAccount objects, in fact any object whether it is part of the core Java classes or a class we designed ourselves, can be stored in an array.

BankAccount[] accounts = new BankAccount[5];  
accounts[0] = new BankAccount();  
  
String[] names = new String[5];  
names[0] = "Alex";

### Arrays and Object references

As mentioned above a primitive array is not a primitive itself, it is an object, and because it is an object its variable is a reference variable that refers to the memory address of the array.

The behaviour of object references that we saw in the previous chapter applies equally to the behaviour of array references. Let’s look at an example:

1: int[] scores = new int[5];  
2: int[] results = new int[5];  
3: scores = results;  
4: System.*out*.println(scores.equals(results));

In this code snippet we declare and create two integer arrays on line 1 and 2. On line 3 we assign the reference to the *results* array to the *scores* variable. Now both the *scores* and *results* array variables refer to the same integer array and no variable refers to the *scores* array (this array is now eligible for garbage collection). On line 4 we test if the *scores* array variable is referring to the same memory address as the *results* array variable. The *equals()* method compares the memory addresses of the two variables and if they are equal it returns true.

The output of line 4 is *true*, so they do refer to the same array.

Later on in this course you will meet the equals() method again. We use the *equals()* method quite a bit during this course.

Let’s look at another example. Here we have two object arrays: a *BankAccount* array and an *Object* array. We have not discussed the *Object* class yet, but for now it is enough to know that all Java classes are implicitly of type *Object*.

1: Object[] objects = new Object[5];  
2: BankAccount[] accounts = new BankAccount[5];  
3: objects = accounts;  
4: System.*out*.println(objects.equals(accounts));

As before, we create two arrays (line 1 and 2) and assign the reference to the *accounts* array to the *objects* array (line 3) and call the *equals()* method on the *objects* array. As before the output of line 4 is *true*.

The reason that an *Object* array can refer to a *BankAccount* array is that the *BankAccount* class inherits from the *Object* class. The concept of inheritance is discussed in **Lesson 3 Section 2: Understand Inheritance**, but you can think of the *Object* class as being ‘bigger’ that the *BankAccount* class and therefore, logically it can be used to refer to the *BankAccount* class.

The reverse is not true though. The *BankAccount* array variable cannot be used to refer to the *Object* array. An attempt to do so would result in the *Incompatible Types* error.

### Casting Object Arrays

If you remember from a previous section it is possible to cast one data type to another data type. It is also possible to do the same with object arrays. Take a look at this example:

1: String[] names = new String[5];  
2: Object[] objects = names;  
3: String[] newNames = (String[]) objects;

On line 1 we create a *String* array, online 2 the *String* array is assigned to an *Object* reference and then on line 3 it is cast back to a *String* array. The cast is required because we are changing to a more specific type.

Notice the square brackets after *String* on the third line, this ensures that we are casting to a String array and not a String object.

It is also possible to refer to the *String* array using an *Object* reference at the time of declaration.

1: Object[] names = new String[5];  
2: String[] newNames = (String[]) names;

This is called polymorphic referencing and it’s a topic we’ll discuss in **Lesson 3 Section 2: Understand Inheritance**.

## Casting Primitive Arrays

You might be thinking that if an object array can refer to another, then a primitive array can do the same. The answer is no. Why? It’s because primitive data types are not Object types and don’t extend from the *Object* class. It might seem logical for an integer array to be able to refer to a short array, after all, integers are larger than the shorts, but this thinking is flawed.

Examine the following code, it fails to compile with and reports an *Incompatible Types* error.

short[] totals = new short[5];  
int[] finals = new int[5];  
finals = (int)totals;

It is not possible to cast one primitive array to a different primitive type. This code fails to compile with an *Inconvertible Types* error on line 3.

1: short[] totals = new short[5];  
2: int[] finals = new int[5];  
3: finals = (int)totals;

## Using arrays We’ve seen how to put data into an array, now lets look at more things we can do with arrays.

### Array Copy

The *Arrays* class has a few methods that make it easy to copy arrays or a subset of elements in an array. Here are a few examples:

int[] scores = {99, 180, 43, -20};  
int result[] = Arrays.*copyOf*(scores, 4);

In this code snippet we copy the entire *scores* array into another array which we specify as having 4 elements.

int numbers[] = Arrays.*copyOfRange*(scores, 1, 3);

In this code snippet we specify that we only want to copy the elements at index 1 and index 2. Note that the end index is exclusive while the start index is inclusive.

## Varargs

You will know by now that a method can accept zero to many parameters, and the number of parameter is fixed at the time you write the method. However there is a way to write a method that will allow zero to many parameters, of the same type, and it looks like this:

public void doSomthing(String… args){}

Look familiar? Yes, it’s the parameter signature that you saw in the *public static void main* method. Its use is not restricted to the *main* method, it may be used in any method and with any type not just *String*. This type of parameter is called a *vararg*, which is short for variable argument. Let’s look at an example:

1: public static void main(String... args) {  
2: new VarArg().add(10, 20, 30);  
3: }  
4:  
5: private void add(int... numbers) {  
6: int total = 0;  
7: for (int number : numbers) {  
8: total = total + number;  
9: }  
10: System.*out*.println(total);  
11: }

The method *add()* defines a *vararg* method signature with an integer array called *numbers* on line 5. This method is called from the *main* method and passed three numbers on line 2.

In lines 7 to 9 there is a *for-each* loop that adds all the elements of the array together and prints the result to the console on line 10. You will learn about *for-each* loops in **Lesson 2 Section 2: Understand Loop Constructs**.

You can pass as many arguments to the *add* method as you like on line 2, and they will be collated together stored in an array of type *int* and passed to the *add* method.

Remember, from a previous section that the parameter to the *main* method could be expressed as *String[] args* or *String args[].* This is because a *vararg* is actually an array under the hood.

There is a lot to be said about *varargs*, so I will leave that until **Lesson 3 Section 1: Understand Methods and Encapsulation**, where it is better discussed in the context of methods.

## Java Collections API

We saw earlier in this section that an array is referred to with a reference just like objects are, but they don’t have any useful methods. This is a shortcoming in the Java platform and was resolved with the introduction of the Java Collections API. This collection of classes is part of the core Java classes and is effectively a sophisticated wrapper around a plain Java array we discussed in the section above.

The range of classes in the Java Collections API is extensive and we only have time to meet a fraction of those classes during this course. You are advised to get to know them very well. They form the basis of your Java toolkit and I guarantee that you will be using them every day you program something in Java.

In this section, I will look at one of those classes: the *ArrayList* class.

All collection classes are members of the *java.util* package and must be explicitly imported with an import statement like this:

import java.util.ArrayList;

or

import java.util.\*;

### Creating an ArrayList

An *ArrayList* is created like any other object:

ArrayList names = new ArrayList();

Unlike with the built in arrays, you are not required to specify a capacity because the *ArrayList* will automatically expand to make room for new elements, but if you want to you can specify a capacity by passing it to the constructor.

ArrayList names = new ArrayList(5);

If you do not specify a capacity it will be set to zero.

It is important to note that capacity and size are not the same. Size refers to the logical size of the array (i.e. the number of elements of data in the array) and the capacity is the amount of space reserved for elements. Think of it like a blank piece of paper that’s big enough for 10 lines but you haven’t written anything on it.

Another misconception is that the *ArrayList* is filled with 10 empty (or null) elements ready to be filled with data. This is not the case. The array has zero empty elements but the capacity to have up to 10 when you start adding data to the array.

The array will auto expand when you get pass the 10th data element, so there is no need to specify a new capacity, the *ArrayList* class is clever enough to handle this operation itself.

You can also create a new array based on an existing array, by passing it to the constructor:

ArrayList someNames = new ArrayList();  
 ArrayList moreNames = new ArrayList(someNames);

The size and contents of the new array, *moreNames*, will be the same as the, *someNames*, array.

You might notice there is something missing from these array declarations. It’s the type of data that the array can store. By default, an *ArrayList* can store any type of data and mix the data types. The following code demonstrates this:

ArrayList names = new ArrayList();  
 names.add("Alex");  
 names.add(new Integer(100));  
 names.add(new BankAccount());

This might seem like a very useful way to store data: all your data in one array, but actually it causes a lot of problems, especially when it comes to retrieving data from the *ArrayList*. This is because you have no idea what the data type is of the element you are retrieving.

An enhancement was included in Java 5 called *generics*. The type of data stored in the *ArrayList* is explicitly stated in the declaration.

ArrayList<String> names = new ArrayList<>();

In this code snippet, the data type stored in the *ArrayList* is *String*. You can see it specified between <> after the ArrayList type declaration.

Even though generics make using *ArrayList* type safe, the pre-Java 5 way of constructing *ArrayList* still exists in the language, and you still need to know about it.

## ArrayList Methods

The *ArrayList* class has many useful methods that make it easy to manage data stored in an array.

## Sorting ArrayList

The Java Collections API has an equivalent class to the Arrays class you learnt about earlier, called Collections and it has a range of useful methods including methods for sorting and searching.

## Converting between ArrayList and arrays

The *ArrayList* has a method that copies data from itself to an array.

1: ArrayList<String> names = new ArrayList<>();  
2: names.add("Alex");  
3: names.add("Zoe");  
4: names.add("Jane");  
5: String[] newNames = new String[3];  
6: names.toArray(newNames);

In this code snippet an *ArrayList* of *Strings* is created and populated on lines 1 to 4. On line 5 a new *String* array is created and populated with the data from the *String* *ArrayList*.

The reverse function is possible. The *Arrays* class has a static method that creates a *List* of data contained in an *Array*.

List<String> nextNames = Arrays.*asList*(newNames);

The *List* class is the parent class of the *ArrayList* class. You will learn about parent classes in **Lesson 3 Section 2: Understand Inheritance**.

## Exercises

Which of the following array declarations is legal?

1. short[] grades = new short[5] {45,23,76};
2. int numbers[] = new int[10];
3. short[] small = new byte[2];
4. char letters[] = new char[]{‘A’,’2’,’@’};
5. byte[] yum tum = new byte[70];
6. int[] empty = new int[0];

Answers: 2, 4, 6

Select the code that compiles when line 5 is replaced

4: byte[] tiny = new byte[10];  
5: // SELECT CODE TO INSERT HERE

1. int length = tiny.length;
2. int length = tiny.size;
3. int length = tiny.capacity;
4. int length = tiny.length();
5. Integer length = tiny.length;
6. Int length = size();
7. None of the above

Answer: 1, 5

Which of the following statements are true about arrays?

1. An array can be any type
2. An array is ordered automatically
3. An array is a primitive
4. An array cannot be copied
5. An array can be cast to another type
6. An array is an object reference
7. None of the above

Answers: 1, 6

What is the output of the following code?

int[] scores = {20, -100, 2, 0};  
 Arrays.*sort*(scores);  
 int index = Arrays.*binarySearch*(scores, 15);  
 System.*out*.println(index);

Answers:

1. 3
2. 2
3. -3
4. -4
5. Output unpredictable

Answer: 4

What is the outcome of the following code?

int[] scores = {20, -100, 2, 0};  
 Arrays.*sort*(scores);  
 scores[2] = -10;  
 int index = Arrays.*binarySearch*(scores, -10);  
 System.*out*.println(index);

Answers:

1. 1
2. -2
3. -3
4. 2
5. Output unpredictable

Answer: 5

# Subsection 2: Understand multidimensional arrays (10 min)

So far we have dealt with one-dimensional arrays that store data is a simple linear way. Now we are going to look at multiple-dimensional arrays that store data in two or more dimensions.

### Alternative declaration and initialisations

There is a short cut way to declare and initialise multidimensional arrays and it is very similar to the way we can declare and initialise one-dimensional arrays.

So instead of declaring a two-dimensional array like this:

int[][] i = new int[2][];  
 i[0][0] = 1;  
 i[0][1] = 2;  
 i[1][0] = 3;  
 i[1][1] = 4;

You can do it on one line like so:

int[][] i = {{1,2},{3,4}};

The dimensions must be declared from left to right:

int[] a[] = new int[4][];  
 int[][] a = new int[5][4];

However this is wrong:

int a[][] = new int[][4];

## Exercises

Which of the following declaration of a multi-dimensional array is legal?

1. int[] a[] = new int[10][];
2. int[][] a = new int[8][8];
3. int a[][] = new int[][89];
4. int[][] i = {{10,30},{99,0}};
5. int[] a[] = new int[5][];

Answer: 1, 2, 4, 5