**Object Oriented Development using Java**

OOD Week 1 – Module 4

Loops

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 the idea of loops. You’ll learn the syntax of 4 different types of loops and find out the best times to use each type.

# How long will the tutorial take to complete?

1 hour

# What should you have already completed?

Modules 1 to 3 (up to and including Conditionals)

# 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?

* Why we use loops
* For each loops
* For loops
* While loops
* Do while loops
* Infinite loops
* Break & continue statements

# What is a loop and why do we use them?

Look at the code example below. Why do you think it’s inefficient?

String[] strings = {"string1","string2","string3"};

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

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

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

The obvious problem here is that there’s a lot of repetition. We’ve had to write three println statements which do virtually the same thing. Imagine if our array of Strings contained 1000 elements. It really wouldn’t be practical to write 1000 lines of code to print out all the elements.

This is where a loop comes in. A loop allows us to run the same line or lines of code as many times as we need to without any duplication. There are 4 different types of loop in Java: for each, for, while and do while. We’ll look at each in turn.

In this tutorial we’ll be using arrays and ArrayLists as examples. You’ll see in the Collections module of OOD week 2 that there are many other collections of elements that can be looped through.

## For each loops

This is the most simple type of loop. It exists for one purpose only. It’s purpose is to loop through every element of an array or a collection of elements such as an ArrayList. It processes the elements in order starting with the first one (index 0) and ending with the last one.

Let’s re-try our example from above using a for each loop:

String[] strings = {"string1","string2","string3"};

**for** (String string : strings) {

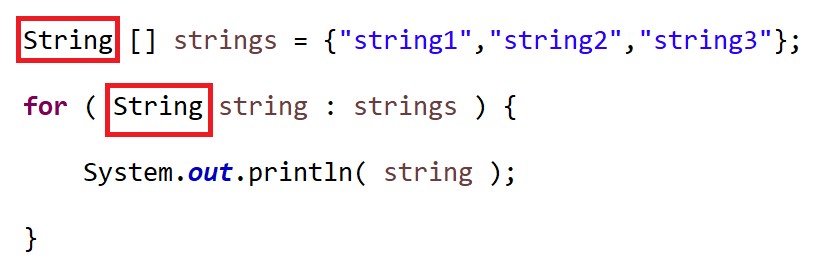
System.***out***.println(string);

}

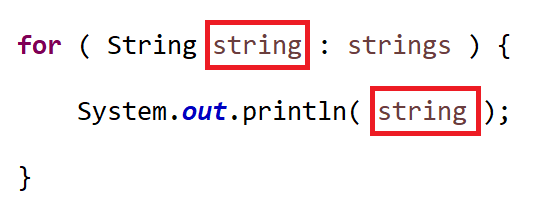
It doesn’t matter how many elements are in our array of Strings, this loop will print them all.

Let’s look at it in more detail. The for each loop is defined by 3 elements in parentheses. Let’s look at each in turn:

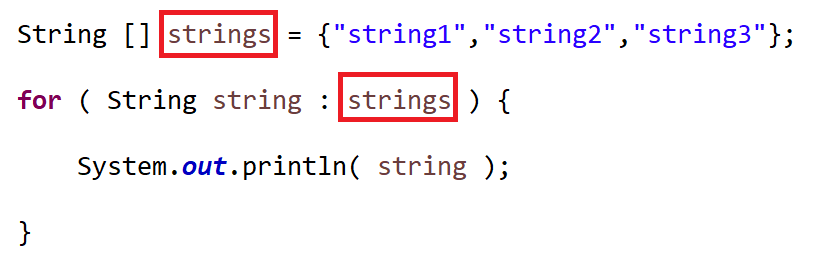
The first part is the datatype of the array or ArrayList we’re looping through:



The second part is the name of a temporary variable which will hold one element of the array or ArrayList at a time. This temporary variable will be used by the code within the loop.



The third and final part is the name of the array or ArrayList that we’re looping through:



Notice finally, that the second and third parts of the loop definition have a colon symbol in between them.

For each loops will always follow this syntax. This makes them easy to use. They are the most commonly used type of loop in Java. Use them any time you need to loop through all elements of an array or ArrayList.

Let’s repeat the example with an ArrayList just to prove that the syntax is the same:

ArrayList<String> strings = **new** ArrayList<String>();

strings.add("string1");

strings.add("string2");

strings.add("string3");

**for** ( String string : strings ) {

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

}

## For loops

For loops are much more flexible than for each loops and can be used for a variety of purposes. Common uses are for producing a series of numbers or for looping through a selection of indexes of an array or ArrayList.

Let’s start with a simple example of a counter:

**for** (**int** count = 1; count <= 10; count ++) {

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

}

If you run the loop you’ll see that it displays the numbers from 1 to 10.

All for loops are based on a variable. The most common datatypes for the variable are int or long. Although sometimes you might use other types such as char. In our example we’re using an int.

You can see that the loop definition consists of three parts separated by semi colons. Let’s look at each of the three parts in turn:

1. **int** count = 1
   1. This defines the variable the loop is based on and sets its initial value:
2. count <= 10
   1. This a condition under which the loop will keep on running. It effectively defines an end point for the loop.
3. count ++
   1. This determines how much the variable changes each time the loop repeats. In this case the count will be increased by one each time. However it could be increased by a different amount (e.g. count += 2) or even decreased (count --).

### Example – counting backwards

What do you think the code below will print out?

**for** (**int** count = 10; count >= 1; count --) {

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

}

If you run the code you should see that it prints the numbers from 10 to 1 in descending order. If we look at the definition we can see that the start point and end point from the previous example have been swapped around.

Another thing to notice is that this time we’re using the greater than or equals operator instead of the less than or equals operator.

Finally we’re counting down by one each time by using - - instead of ++

### Example – looping through an array

What do you think the code below will do?

String[] strings = {"string1","string2","string3"};

**for** (**int** index = 0; index < strings.length; index ++) {

System.***out***.println(strings[index]);

}

Hopefully you’ll have realised that this does the exact same thing as a for each loop. Notice that the end point of the loop is defined as the index being less than the length of the array. This is because indexes count up from 0. So the final index of an array with 10 elements would be 9.

A very common error when writing this type of loop is to use the <= operator. This will cause the code to crash, and display an ‘array index out of bounds’ exception.

In reality we’re unlikely to use a for loop to do exactly the same thing as a for each loop. The for each loop is much easier to write.

However, we would use a for loop where we don’t want to loop through every element of the array or ArrayList. A common example is where we want to write code which compares consecutive pairs of elements. In this case the index would count up from 1 instead of 0. Here’s an example:

String[] strings = {"string1","string2","string3"};

**for** (**int** index = 1; index < strings.length; index ++) {

String previousString = strings[index-1];

String currentString = strings[index];

// code to compare current string with previous string

}

### Example – looping through the alphabet

In this example the loop is based on a char instead of an int:

**for** (**char** letter = 'a'; letter <= 'z'; letter ++) {

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

}

It will display all 26 letters of the alphabet in lower case.

## A reminder about clean code

You’ll have noticed that in the previous examples our for loop definitions used a variety of variable names:

* count – when producting a sequence of consecutive numbers
* index – when looping through an array or ArrayList
* letter – when producing a sequence of letters

Please remember to always use meaningful variable names when defining a loop (actually when defining all variables).

You’ll see a lot of code examples on the internet where loops use variable names like ‘i’ or ‘j’. Please do not copy this. It’s poor practice and not what our clients want you to use.

## While loops

We’ve seen that with for each loops and for loops, the loop definition tells us exactly how many times the loop will repeat.

Sometimes we don’t know how many times we want a loop to repeat. A good example would be where we’re processing user input. Let’s say that a user is trying to set a password and that they enter some text which doesn’t have enough characters. In this case we have no idea how many times the user will need to be prompted to try again. It could be that they get it right first time or it could be that they need 10 attempts. We just don’t know.

In this case a for loop or a for each loop would be inappropriate. This is where while loops come in. A while loop is a loop which repeats an unknown number of times. It’s based on a condition and while that condition is true the loop will continue to repeat.

Here’s an example:

String password = "";

**while** (password.length() < 6) {

password = // code to get user input;

}

Since you haven’t yet learned about getting user input you can’t try this one out. So let’s try another example which you can run.

**int** passCode = 57;

**int** guess = 0;

**int** numberOfTries = 0;

**while** (guess != passCode) {

guess = (**int**) (Math.*random*() \* 100);

numberOfTries ++;

}

System.***out***.println("you took "+numberOfTries+" tries to guess the pass code");

Try running this a few times and you’ll see that the loop repeats a different number of times on each occasion.

## Do while loops

This is a variation on a while loop. Again we don’t know exactly how many times its going to run, but there’s one big difference with a while loop.

If the while loop’s condition is false before the loop starts, the loop won’t run at all. This is really not a problem. If the loop’s task is complete before the loop starts, then there’s no reason to run the loop.

The do while loop is guaranteed to run at least once.

Let’s re-try our password example. Let’s say that this time the user already has a password and it’s longer than 6 characters. But we want them to set a new password. If we were to use our previous while loop example, the loop would never run and the user wouldn’t get a chance to set a new password. The do while loop gets around this:

String password = "this is longer than 6 characters";

**do** {

password = // code to get user input

} **while** (password.length() < 6);

In our example, the user gets to enter their new password before the condition checks that it’s more than 6 characters long.

## The break keyword

The break keyword allows us to terminate a loop early. It can be used in any type of loop.

One type of loop where a break statement is essential is an infinite loop. This takes the form of a while loop with a condition which is always true. Let’s re-do our example where we try to guess a pass code:

**int** passCode = 57;

**int** guess = 0;

**int** numberOfTries = 0;

**while** (**true**) {

guess = (**int**) (Math.*random*() \* 100);

numberOfTries ++;

**if** (guess == passCode) {

**break**;

}

}

System.***out***.println("you took "+numberOfTries+" tries to guess the pass code");

Notice that the condition used here in the if statement (guess == passCode) is the opposite of the one used in the previous example when the condition was in the loop definition (guess != passCode).

In this example there’s no real gain from using break. The best times to use break are for complex conditions such as nested if statements or multiple conditions in different parts of the loop’s code.

What’s wrong with the code below?

**while** (**true**) {

guess = (**int**) (Math.*random*() \* 100);

numberOfTries ++;

**break**;

}

The problem in the code above is that the break statement is not inside a conditional (e.g. an if statement or a switch/case statement). This means the loop will always run exactly once. A loop which always runs exactly once is very pointless!!

## The continue keyword

The continue keyword allows us to skip code below it in a loop. As with break it needs to be contained in a conditional. It can be used with any kind of loop.

Let’s try a variation on our password example. This time the password should not only be more than 6 characters long, it should also not match the previous password:

String previousPassword = "abcdefg";

String newPassword = "";

**while** (**true**) {

newPassword = // code to get user input

**if** (newPassword == previousPassword) {

**continue**;

}

**if** (newPassword.length() > 6) {

**break**;

}

}

If the user enters a password which is identical to the previous password, the loop will skip the code below it and go straight back to the top for its next iteration. This means that the second if statement will only ever run if the new password is different to the previous password.

There are a couple of benefits of this. One is that the code runs more quickly. We don’t need to waste time running code which is not needed.

We could have done this without using continue. A nested if statement like the one below would have achieved the same thing:

**if** (newPassword != previousPassword) {

**if** (newPassword.length() > 6) {

**break**;

}

}

This is much less readable than our previous code. If there was more code within the two if statements it would be even less readable. The use of continue helps us make our code clean.