**Object Oriented Development using Java**

OOD Week 1 – Module 1

Datatypes

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 common datatypes in Java.

# How long will the tutorial take to complete?

1 hour

# What should you have already completed?

Be able to create a project in your IDE (Eclipse, intelliJ etc). Be able to create a package within your project. Be able to create a class containing a main method in your package. All code in this tutorial will be written in the main method.

# 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?

* Primitives
* Primitive casting
* Objects
* Strings
* Wrappers
* Enums

# What is a data type?

In computing, a variable is something which holds some data in memory for use at a later point in the program. In some computer languages, a variable can hold any type of data. For instance in a simple language such as Bash scripting in UNIX we could make a variable called ‘var’ and store a number in it. Later on in our code we could change var’s value to a word.

In Java things are different. Each variable can only hold a specific type of data. So we can have a variable that can store numbers and nothing but numbers or another variable which can store a single character and nothing but a single character.

Each time we create a variable in Java we must specify what type of data it will hold. We do this by using data types. For example if we create a variable of type int, the only thing it will be able to hold is a whole number. We won’t be able to change its value to a word later in the program.

We’ll see that there are two main categories of data types in Java: primitives and objects.

# Primitives

This is the most simple of the two categories of data type. It holds a single value. There are only 8 primitive data types in Java:

* byte – holds a whole number between -128 and 127
* short – holds a whole number between -32,768 and 32,767
* int – holds a whole number between -2,147,483,648 and 2,147,483,647
* long – holds a whole number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807
* float – holds a 32 bit decimal number
* double – holds a 64 bit decimal number
* char – holds a single character
* boolean – holds the values ‘true’ or ‘false’

Let’s try a few basic examples. First of all we’ll create two byte variables:

**byte** availableMarks = 100;

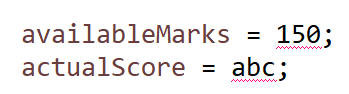
**byte** actualScore;

We’ve given the first variable availableMarks an initial value of 100. The second variable actualScore hasn’t got an initial value, so we’ll have to give it a value later on in the code. Both variables can be given different values at any point in the code after they’ve been created:

availableMarks = 90;

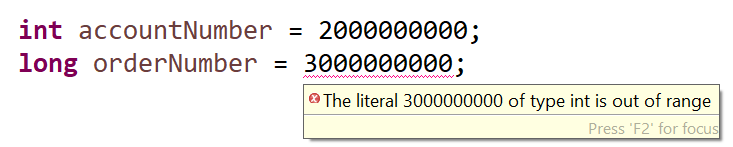
actualScore = 85;

However we can only give them values which are valid for a byte variable. The following wouldn’t work:



## int and long

The datatypes byte and short are rarely used. If we want to store a whole number generally we’ll use int or long. Let’s try an example:



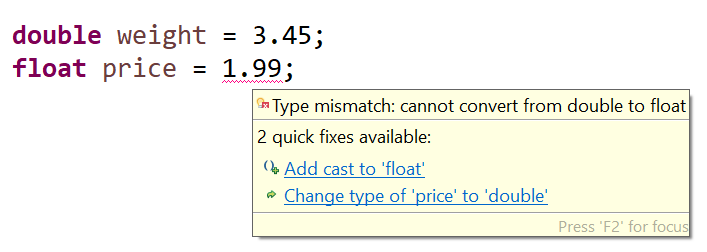
The account number variable is an int and has no problem fitting the number 2000000000. However there seems to be a problem with putting the number 3000000000 into the long variable orderNumber. This is strange as a long can hold numbers of up to 19 digits. The number 3000000000 really should fit.

The error message gives us a big clue about what’s going on ‘The literal 3000000000 of type **int** is out of range’. This tells us that by default Java sees all whole numbers as being ints. If we want 3000000000 to be treated as a long we need to tell Java by adding the letter ‘l’ onto the end of it:



## float and double

You’ve seen that the first four primitive types all hold whole numbers. The next two types hold decimal numbers. Let’s look at an example:



The value 3.45 fits comfortably into our weight variable which has datatype double. This should be no surprise as double holds a 64 bit number.

We have a compile error with float though. The value 1.99 is small, it should fit comfortably enough into a 32 bit data type like a float. The problem here is much like we saw with int and long. By default Java assumes that all decimal numbers are doubles. As with long, we need to tell Java that our number is a float. This time we’ll do it by adding the letter ‘f’ to the end:



## Arithmetic operators

Here are some examples of doing arithmetic on an int variable. These will work for any of the numeric primitives.

**int** score = 0;

score ++;

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

score += 2;

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

Try running this code. You’ll see that it prints out the numbers 1 and 3.

The ++ operator increases a variable by 1. If we want to decrease it by 1 we can do - -

The += operator increases a variable by a specified amount (in our example 2). We can do the same with subtract -=, multiply \*= or divide /=

## Precision problems with floats and doubles

Try doing the following arithmetic on a calculator: 1586.6 - 708.75

You should get 877.85.

Now try the same thing again in Java:

System.***out***.println(1586.6 - 708.75);

This time we get a different result of 877.8499999999999. It’s almost correct, but not quite. This is a problem that you’ll encounter sometimes in Java when doing arithmetic with floats and doubles. Provided you’re aware that this can happen, you can take steps to deal with it. It’s something we’ll have to deal with in some of our upcoming modules.

## char

Variables of the char data type hold a single character. Here’s a simple example:

**char** letter1 = 'a';

**char** letter2 = 'b';

When assigning the value of a char variable, the character must always be contained within single quotes.

What do you think the following line of code will print?

System.***out***.println(letter1+letter2);

If you guessed “ab”, you’d be wrong. Try running the code and you’ll see that actually it prints: 195

This tells us something very important, not just about char variables but about primitive variables in general:

**If you perform arithmetic on two or more primitive variables of the same type, the result will also be of that type.**

So a char value plus a char value must result in a char value. Therefore ‘a’ + ‘b’ cannot be ‘ab’ as a char can only fit one character.

At this point you’re probably thinking that 195 doesn’t look like a char value either. So let’s look a bit deeper.

Each char value has an accompanying integer code. The code for ‘a’ is 97 and the code for ‘b’ is 98. Actually we can set char values like this:

**char** letter1 = 97;

**char** letter2 = 98;

If we print out the values in turn:

System.***out***.println(letter1);

System.***out***.println(letter2);

We’ll see that letter1 has the value ‘a’ and letter2 has the value ‘b’.

Now it should be clear where the value 195 came from when we did letter1 + letter2. It was the sum of the integer values of the two char variables. Theoretically this could have been displayed as a char. The code 195 has a char value of Ã but displaying this seems very pointless!

## boolean

This is the most simple of the primitives. It can only store two different values ‘true’ or ‘false’. We’ll see later on that it’s commonly used to store the results of conditions:

**boolean** isTrue = **true**;

**boolean** isFalse = **false**;

**boolean** aEqualsb = ('a' == 'b');

**boolean** aEqualsa = ('a' == 'a');

Notice that the third and fourth examples don’t literally use the words ‘true’ and ‘false’. Instead they have a statement that is either false (a is not equal to b) or true (a is equal to a).

## Primitive casting

Casting is a process where we can change the datatype of a value. Let’s start with a simple but pointless example using a byte. As we’ve seen, numbers above 127 won’t fit into a byte:



But we could force the number 200 to become a byte like this:

**byte** score = (**byte**) 200;

A datatype in parentheses before a value is what we call a ‘cast’. It changes the value to the data type in the parentheses. So in this case (byte) turns the value 200 into a byte.

The problem here is that a byte data type can only hold an 8 bit binary number. The number 200 has 9 bits in binary. So all the cast does is to remove one of those 9 bits. The result, if we print the value score is -56. Very pointless!!

Let’s look at a much more useful way of using a cast. First of all let’s try an example without a cast:

**int** availableMarks = 80;

**int** actualScore = 60;

**int** percentageScore = actualScore / availableMarks \* 100;

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

Before you run the code try to guess what you think will be printed out.

There’s a good chance you’re thinking that the program will print out the number 75. After all if you did the maths 60 / 80 \* 100 on a calculator, 75 would be the result.

Actually the result in Java is 0.

This is because of the rule we saw earlier when we looked at adding two char values:

**If you perform arithmetic on two or more primitive variables of the same type, the result will also be of that type.**

So let’s work through what happens:

60 / 80 is 0.75, but since both values are ints, the result must also be an int. Therefore it’s rounded down to the nearest whole number: zero.

In the next part of the arithmetic we multiply the result by 100, so 0 \* 100. The answer is still zero.

To get around this we need to make sure that one of the first 2 numbers is of type double. You could change the datatype of one of the two variables but this would be a bad idea. A much better idea is to use a cast to temporarily make one of the variables a double during the arithmetic:

**double** percentageScore = (**double**) actualScore /availableMarks \* 100;

Notice that since the result is now a double, we need to change the data type of the percentageScore into a double as well.

Converting ints to doubles and vice versa is probably the most common usage of casting with primitives.

# Object data types

You’ve seen that there are only 8 primitive data types. Most data types in Java fall into the category of object types. There are literally thousands of built in object data types in Java. You can even make your own object data types.

So what is an object data type? An object is a data type which can contain one or more values and has functionality attached to it. It’s commonly (but not always) used to model a real life entity. In this tutorial, we’re going to look at the most common types of built in Java object.

## String

This is probably the most common of the object data types. In the previous section on primitives you saw that a char variable can hold a single letter. Imagine that we wanted to write a sentence. Using lots of char values to make the sentence would be very impractical. A String object contains multiple char values as well as functionality to help us manipulate those values.

Here's a simple example:

String message = "Hello";

Notice that this time double quotes are used. Unlike some other languages, double and single quotes are not interchangeable in Java. Single quotes are used for a char, double quotes for a String.

Notice also a couple of differences between the data type String and the data type **char**. Firstly, String starts with a capital letter while char starts with lower case. Secondly, the colours are different if you’re using an IDE, char is in purple (in Eclipse) indicating that it’s a Java keyword. String is just in plain black.

All object data types will start with an upper case letter. Primitives start with lower case.

Let’s look at some of String’s functionality:

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

System.***out***.println(message.charAt(0));

System.***out***.println(message.substring(1));

When you run this you should see:

5 (the length of the String)

H (the first character of the String – character indexes count up from zero)

ello (all of the characters of the String from the second position onwards)

length(), charAt() and substring() are known as ‘methods’ in Java. A method is just some named functionality attached to an object.

String has lots more methods. Your IDE will show you a list of available methods when you type the String variable’s name followed by a dot.

String variables can hold any character, in our previous example we only used letters, but a String could also contain numbers:

String number1 = "3";

String number2 = "2";

What do you think the following code would print:

System.***out***.println(number1+number2);

The result is definitely not 5.

In fact it’s 32. It doesn’t matter that the Strings contain only numbers. Within a String they are just text. The + symbol simply adds one String to another String to make a longer String.

In fact if we add anything to a String, the result will always be a String. What do you think the code below will print out:

System.***out***.println(2+number1+5);

The answer is 235.

## Wrappers

A wrapper is an object datatype which contains a single primitive and adds some functionality to it. There are 8 wrapper types:

Byte

Short

Integer

Long

Float

Double

Boolean

Character

Six of the 8 wrappers have the same name as the equivalent primitive, but start with a capital letter. The two exceptions are Integer (int) and Character (char).

Most of the time, wrappers can be used in the same way as primitives to store data:

**double** price1 = 2.5;

Double price2 = 3.99;

Their main benefit comes with the functionality that they add. Here are some examples:

String string1 = "3";

String string2 = "2";

**int** number1 = Integer.*parseInt*(string1);

**int** number2 = Integer.*parseInt*(string2);

System.***out***.println(number1+number2);

The parseInt() method converts a String containing only digits into an int. In our example this allows us to add up the numbers in the two Strings.

All numeric wrapper classes have equivalent parse methods.

The Character data type has methods to check if a character is a letter or a digit:

**char** character1 = 'a';

**char** character2 = '1';

System.***out***.println(Character.*isDigit*(character1));

System.***out***.println(Character.*isDigit*(character2));

System.***out***.println(Character.*isLetter*(character1));

System.***out***.println(Character.*isLetter*(character2));

Generally, if you just want to store data and aren’t interested in functionality, you should use a primitive datatype. This is because wrappers take up more memory.

# Clean code

It’s very important that you write code that’s easy to read. This is especially important on site where different developers may need to read each other’s code. Debugging is a good example of this. A large part of writing readable or ‘clean’ code is choosing good variable names.

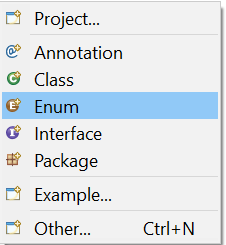
You’ll have noticed that thoughout this tutorial, we’ve used meaningful variable names such as price, score & message. You should always do this when writing your own code. On the internet there are lots of code examples using poor variable names like ‘x’ and ‘y’. You should avoid doing this.

# Enums

We’ve seen that Strings can hold any text. In an application this could be useful where we wanted a user to type in some information. But what about situations where we wanted the user to choose from a limited number of options such as in a dropdown menu. This is where something called an ‘enum’ can help.

Enum is a third category of data type which can allow us to create sets of pre-defined values.

Up until now we’ve written all of our code in the main method of a class file. An enum will have its own separate file. In Eclipse you can create an enum in a similar way to how you create a class. You should right click on the package where you want to create the enum, choose ‘New’ and then choose ‘Enum’:



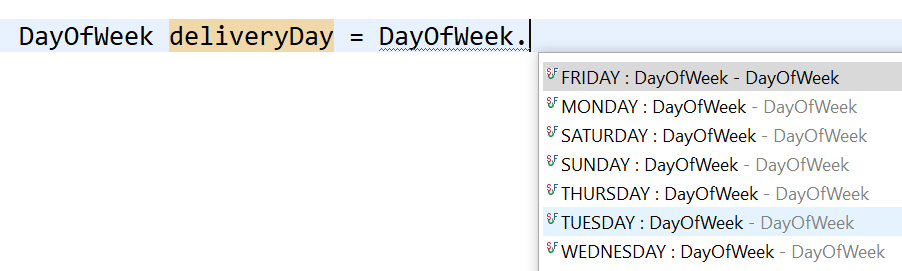
We’re going to create an enum called dayOfWeek. It could potentially be used in a dropdown where a customer has to choose which day of the week they want their order delivered. The code is very simple:

**public** **enum** DayOfWeek {

***MONDAY***, ***TUESDAY***, ***WEDNESDAY***, ***THURSDAY***, ***FRIDAY***, ***SATURDAY***, ***SUNDAY***;

}

You can now click ‘save’ and close the enum file. That’s it. The enum is now ready to use in your code. You can use it in any class throughout your package. Here’s an example:



Notice that the name of the enum (DayOfWeek) is our data type. In Eclipse we get a dropdown offering us just 7 values to choose from.

DayOfWeek deliveryDay = DayOfWeek.***TUESDAY***;

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

So in this case our deliveryDay variable is given a value of TUESDAY. This is one of only 7 values it could have been given.