Basic Concepts

Welcome to Java!

Java is a high level, modern programming language designed in the early 1990s by Sun Microsystems, and currently owned by Oracle.

Java is **Platform Independent**, which means that you only need to write the program once to be able to run it on a number of different platforms!
Java is **portable**, **robust**, and **dynamic**, with the ability to fit the needs of virtually any type of application.

> Java guarantees that you'll be able to **Write Once, Run Anywhere**.

Java

More than **3 billion** devices run Java.
Java is used to develop apps for Google's **Android** OS, various Desktop Applications, such as media players, antivirus programs, Web Applications, Enterprise Applications (i.e. banking), and many more!

> Learn, practice, and then join the huge community of Java developers around the world!

Your First Java Program

Let's start by creating a simple program that prints "Hello World" to the screen.

```
class MyClass {
  public static void main(String[ ] args) {
    System.out.println("Hello World");
  }
}
```

Try It Yourself

In Java, every line of code that can actually run needs to be inside a **class**.
In our example, we named the class **MyClass**. You will learn more about classes in the upcoming modules.

In Java, each application has an entry point, or a starting point, which is a method called **main**.
Along with main, the keywords **public** and static will also be explained later.

> As a summary:
> - Every program in Java must have a **class**.
> - Every Java program starts from the **main** method.

The main Method

To run our program, the **main** method must be identical to this signature:

```
public static void main(String[ ] args)
```

- **public**: anyone can access it
- **static**: method can be run without creating an instance of the class containing the main method
- **void**: method doesn't return any value
- **main**: the name of the method

For example, the following code declares a method called **test**, which does not return anything and has no parameters:

```
void test()
```

> The method's parameters are declared inside the parentheses that follow the name of the method.
> For **main**, it's an array of strings called **args**. We will use it in our next lesson, so don't worry if you don't understand it all now.

System.out.println()

Next is the body of the **main** method, enclosed in curly braces:

```
{
  System.out.println("Hello World!");
}
```

The **println** method prints a line of text to the screen.
The **System** class and its **out** stream are used to access the **println** method.

> In classes, methods, and other flow-control structures code is always enclosed in curly braces { }.

Semicolons in Java

You can pass a different text as the parameter to the **println** method to print it.

```
class MyClass {
  public static void main(String[ ] args) {
    System.out.println("I am learning Java");
  }
}
```

Try It Yourself

In Java, each code statement must end with a **semicolon**.

> Remember: do not use **semicolons** after method and class declarations that follow with the body defined using the curly braces.

Comments

The purpose of including comments in your code is to explain what the code is doing.
Java supports both single and multi-line comments. All characters that appear within a comment are ignored by the Java compiler.

A **single-line** comment starts with **two forward slashes** and continues until it reaches the end of the line.
For example:

```
// this is a single-line comment
x = 5; // a single-line comment after code
```

Adding comments as you write code is a good practice, because they provide clarification and understanding when you need to refer back to it, as well as for others who might need to read it.

Multi-Line Comments

Java also supports comments that span multiple lines.
You start this type of comment with a forward slash followed by an asterisk, and end it with an asterisk followed by a forward slash.
For example:

```
/*  This is also a
   comment spanning
   multiple lines */
```

Note that Java does not support nested multi-line comments.
However, you can nest single-line comments within multi-line comments.

```
/* This is a single-line comment:

   // a single-line comment

 */
```

Another name for a Multi-Line comment is a Block comment.

Documentation Comments

Documentation comments are special comments that have the appearance of multi-line comments, with the difference being that they generate external documentation of your source code. These begin with a forward slash followed by two asterisks, and end with an asterisk followed by a forward slash.
For example:

```
/** This is a documentation comment */

/** This is also a
   documentation comment */
```

Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code which has required documentation in a predefined format.

When a documentation comment begins with more than two asterisks, Javadoc assumes that you want to create a "box" around the comment in the source code. It simply ignores the extra asterisks.
For example:

```
/**********************

This is the start of a method

**********************/
```

This will retain just the text "This is the start of a method" for the documentation.

Variables

Variables store data for processing.
A variable is given a name (or **identifier**), such as area, age, height, and the like. The name uniquely identifies each variable, assigning a value to the variable and retrieving the value stored.

Variables have **types**. Some examples:
- int: for integers (whole numbers) such as 123 and -456
- **double**: for floating-point or real numbers with optional decimal points and fractional parts in fixed or scientific notations, such as 3.1416, -55.66.
- String: for texts such as "Hello" or "Good Morning!". Text strings are enclosed within double quotes.

You can declare a variable of a type and assign it a value. Example:

```
String name = "David";
```

This creates a variable called **name** of type String, and assigns it the value "David".

It is important to note that a variable is associated with a type, and is only capable of storing values of that particular type. For example, an **int** variable can store integer values, such as 123; but it cannot store real numbers, such as 12.34, or texts, such as "Hello".

Variables

Examples of variable declarations:

```
class MyClass {
  public static void main(String[ ] args) {
    String name ="David";
    int age = 42;
    double score =15.9;
    char group = 'Z';
  }
}
```

Try It Yourself

char stands for character and holds a single character.

Another type is the **Boolean** type, which has only two possible values: **true** and **false**.
This data type is used for simple flags that track true/false conditions.
For example:

```
boolean online = true;
```

> You can use a comma-separated list to declare more than one variable of the specified type. Example: int a = 42, b = 11;

The Math Operators

Java provides a rich set of operators to use in manipulating variables. A value used on either side of an operator is called an **operand**.
For example, in the expression below, the numbers 6 and 3 are operands of the plus operator:

```
int x = 6 + 3;
```

Java arithmetic operators:
+ addition
- subtraction
*** multiplication**
/ division
% modulo

> Arithmetic operators are used in mathematical expressions in the same way that they are used in algebraic equations.

Addition

The + operator adds together two values, such as two constants, a constant and a variable, or a variable and a variable. Here are a few examples of addition:

```
int sum1 = 50 + 10;
int sum2 = sum1 + 66;
int sum3 = sum2 + sum2;
```

Subtraction

The - operator subtracts one value from another.

```
int sum1 = 1000 - 10;
int sum2 = sum1 - 5;
int sum3 = sum1 - sum2;
```

> Just like in algebra, you can use both of the operations in a single line. For example: int val = 10 + 5 - 2;

Multiplication

The * operator multiplies two values.

```
int sum1 = 1000 * 2;
int sum2 = sum1 * 10;
int sum3 = sum1 * sum2;
```

Division

The / operator divides one value by another.

```
int sum1 = 1000 / 5;
int sum2 = sum1 / 2;
int sum3 = sum1 / sum2;
```

In the example above, the result of the division equation will be a whole number, as **int** is used as the data type. You can use **double** to retrieve a value with a decimal point.

Modulo

The **modulo** (or remainder) math operation performs an integer division of one value by another, and returns the remainder of that division.
The operator for the modulo operation is the percentage (%) character.
Example:

```
int value = 23;
int res = value % 6; // res is 5
```

Try It Yourself

Dividing 23 by 6 returns a quotient of 3, with a remainder of 5. Thus, the value of 5 is assigned to the **res** variable.

Increment Operators

An **increment** or **decrement** operator provides a more convenient and compact way to increase or decrease the value of a variable by **one**.
For example, the statement **x=x+1;** can be simplified to **++x;**
Example:

```
int test = 5;
++test; // test is now 6
```

Try It Yourself

The **decrement** operator (--) is used to decrease the value of a variable by one.

```
int test = 5;
--test; // test is now 4
```

Try It Yourself

Use this operator with caution to avoid calculation mistakes.

Prefix & Postfix

Two forms, **prefix** and **postfix**, may be used with both the increment and decrement operators. With prefix form, the operator appears before the operand, while in postfix form, the operator appears after the operand. Below is an explanation of how the two forms work:
Prefix: Increments the variable's value and uses the new value in the expression.
Example:

```
int x = 34;
int y = ++x; // y is 35
```

Try It Yourself

The value of x is first incremented to 35, and is then assigned to y, so the values of both x and y are now 35.
Postfix: The variable's value is first used in the expression and is then increased.
Example:

```
int x = 34;
int y = x++; // y is 34
```

Try It Yourself

x is first assigned to y, and is then incremented by one. Therefore, x becomes 35, while y is assigned the value of 34.

The same applies to the **decrement** operator.

Assignment Operators

You are already familiar with the **assignment** operator (=), which assigns a value to a variable.

```
int value = 5;
```

This assigned the value 5 to a variable called **value** of type **int**.

Java provides a number of assignment operators to make it easier to write code.
Addition and assignment (+=):

```
int num1 = 4;
int num2 = 8;
num2 += num1; // num2 = num2 + num1;

// num2 is 12 and num1 is 4
```

Try It Yourself

Subtraction and assignment (-=):

```
int num1 = 4;
int num2 = 8;
num2 -= num1; // num2 = num2 - num1;

// num2 is 4 and num1 is 4
```

Try It Yourself

Similarly, Java supports multiplication and assignment (*=), division and assignment (/=), and remainder and assignment (%=).

Strings

A String is an object that represents a sequence of characters.
For example, "Hello" is a string of 5 characters.

For example:

```
String s = "SoloLearn";
```

You are allowed to define an empty string. For example, String str = "";

String Concatenation

The + (plus) operator between strings adds them together to make a new string. This process is called **concatenation**.
The resulted string is the first string put together with the second string.
For example:

```
String firstName, lastName;
firstName = "David";
lastName = "Williams";

System.out.println("My name is " + firstName +" "+lastName);

// Prints: My name is David Williams
```

Try It Yourself

The **char** data type represents a single character.

Getting User Input

While Java provides many different methods for getting user input, the **Scanner** object is the most common, and perhaps the easiest to implement. Import the **Scanner** class to use the **Scanner** object, as seen here:

```
import java.util.Scanner;
```

In order to use the **Scanner** class, create an instance of the class by using the following syntax:

```
Scanner myVar = new Scanner(System.in);
```

You can now read in different kinds of input data that the user enters.
Here are some methods that are available through the Scanner class:
Read a byte - nextByte()
Read a short - nextShort()
Read an int - nextInt()
Read a long - nextLong()
Read a float - nextFloat()
Read a double - nextDouble()
Read a boolean - nextBoolean()
Read a complete line - nextLine()
Read a word - next()

Example of a program used to get user input:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[ ] args) {
    Scanner myVar = new Scanner(System.in);
    System.out.println(myVar.nextLine());
  }
}
```

Try It Yourself

This will wait for the user to input something and print that input.
The code might seem complex, but you will understand it all in the upcoming lessons.

End.