Java Hello World Program

A "Hello, World!" is a simple program that outputs `Hello, World!` on the screen. Since it's a very simple program, it's often used to introduce a new programming language to a newbie.

Let's explore how Java "Hello, World!" program works.

Java "Hello, World!" Program

```
// Your First Program

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

Output

```
Hello, World!
```

How Java "Hello, World!" Program Works?

1. // Your First Program

   In Java, any line starting with // is a comment. Comments are intended for users reading the code to understand the intent and functionality of the program. It is completely ignored by the Java compiler (an application that translates Java program to Java bytecode that computer can execute).

2. `class HelloWorld { ... }`

   In Java, every application begins with a class definition. In the program, *HelloWorld* is the name of the class, and the class definition is:

   ```
   class HelloWorld {

   ... .. ...

   }
   ```

For now, just remember that every Java application has a class definition, and the name of the class should match the filename in Java.

3. `public static void main(String[] args) { ... }`

   This is the main method. Every application in Java must contain the main method. The Java compiler starts executing the code from the main method.

   How does it work? Good question. However, we will not discuss it in this article. After all, it's a basic program to introduce Java programming language to a newbie. We will learn the meaning of public, static, void, and how methods work? in later chapters.

   For now, just remember that the main function is the entry point of your Java application, and it's mandatory in a Java program. The signature of the main method in Java is:

   ```
   public static void main(String[] args) {
   ... .. ...
   }
   ```

4. `System.out.println("Hello, World!");`

   The code above is a print statement. It prints the text Hello, World! to standard output (your screen). The text inside the quotation marks is called String in Java.

   Notice the print statement is inside the main function, which is inside the class definition.

Things to take away

- Every valid Java Application must have a class definition (that matches the filename).
- The main method must be inside the class definition.
- The compiler executes the codes starting from the main function.

This is a valid Java program that does nothing.

```
public class HelloWorld {
    public static void main(String[] args) {
        // Write your code here
    }
}
```

Don't worry if you don't understand the meaning of class, static, methods, and so on for now.

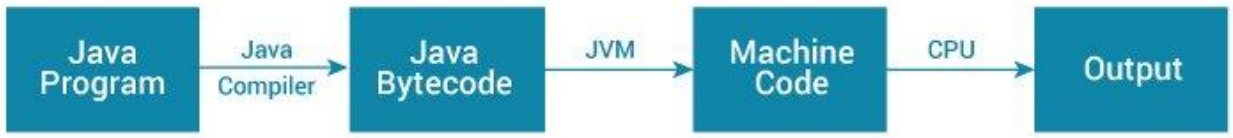
Java JDK, JRE and JVM

What is JVM?

JVM (Java Virtual Machine) is an abstract machine that enables your computer to run a Java program.

When you run the Java program, Java compiler first compiles your Java code to bytecode. Then, the JVM translates bytecode into native machine code (set of instructions that a computer's CPU executes directly).

Java is a platform-independent language. It's because when you write Java code, it's ultimately written for JVM but not your physical machine (computer). Since JVM executes the Java bytecode which is platform-independent, Java is platform-independent.



Working of Java Program

What is JRE?

JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications.

JRE is the superset of JVM.

Java Runtime Environment

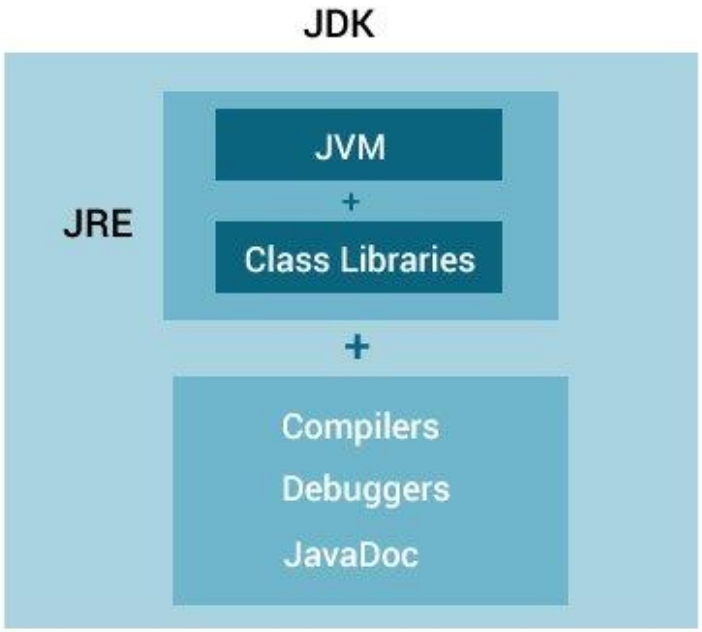
What is JDK?

JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.

In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).

Java Development Kit

Relationship between JVM, JRE, and JDK.



Relationship between JVM, JRE, and JDK

Java Variables and Literals

Java Variables

A variable is a location in memory (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier).

Create Variables in Java

Here's how we create a variable in Java,

```
int speedLimit = 80;
```

Here, *speedLimit* is a variable of *int* data type and we have assigned value **80** to it.

The int data type suggests that the variable can only hold integers.

In the example, we have assigned value to the variable during declaration. However, it's not mandatory.

You can declare variables and assign variables separately. For example,

```
int speedLimit;
speedLimit = 80;
```

Note: Java is a statically-typed language. It means that all variables must be declared before they can be used.

Change values of variables

The value of a variable can be changed in the program, hence the name **variable**. For example,

```
int speedLimit = 80;
... .. ...
speedLimit = 90;
```

Here, initially, the value of *speedLimit* is **80**. Later, we changed it to **90**.

However, we cannot change the data type of a variable in Java within the same scope.

What is the variable scope?

Don't worry about it for now. Just remember that we can't do something like this:

```
int speedLimit = 80;
... .. ...
float speedLimit;
```

Rules for Naming Variables in Java

Java programming language has its own set of rules and conventions for naming variables. Here's what you need to know

 Java is case sensitive. Hence, age and AGE are two different variables. For example,

```
int age = 24;
int AGE = 25;

System.out.println(age);  // prints 24
System.out.println(AGE);  // prints 25
```

 Variables must start with either a letter or an underscore, _ or a dollar, $ sign. For example,

```
int age;  // valid name and good practice
int _age;  // valid but bad practice
int $age;  // valid but bad practice
```

 Variable names cannot start with numbers. For example,

```
int 1age;  // invalid variables
```

 Variable names can't use whitespace. For example,

```
int my age;  // invalid variables
```

- Here, is we need to use variable names having more than one word, use all lowercase letters for the first word and capitalize the first letter of each subsequent word. For example, *myAge*.
- When creating variables, choose a name that makes sense. For example, *score*, *number*, *level* makes more sense than variable names such as *s*, *n*, and *l*.

- If you choose one-word variable names, use all lowercase letters. For example, it's better to use *speed* rather than *SPEED*, or *sPEED*.

There are 4 types of variables in Java programming language:

- Instance Variables (Non-Static Fields)
- Class Variables (Static Fields)
- Local Variables
- Parameters

Java literals

Literals are data used for representing fixed values. They can be used directly in the code. For example,

int a = 1;
float b = 2.5;
char c = 'F';

Here, 1, 2.5, and 'F' are literals.

Here are different types of literals in Java.

1. Boolean Literals

In Java, boolean literals are used to initialize boolean data types. They can store two values: true and false. For example,

```
boolean flag1 = false;
boolean flag2 = true;
```

Here, false and true are two boolean literals.

2. Integer Literals

An integer literal is a numeric value(associated with numbers) without any fractional or exponential part. There are 4 types of integer literals in Java:

1. binary (base 2)
2. decimal (base 10)
3. octal (base 8)
4. hexadecimal (base 16)

For example:

```
// binary
int binaryNumber = 0b10010;
// octal
int octalNumber = 027;

// decimal
int decNumber = 34;

// hexadecimal
int hexNumber = 0x2F; // 0x represents hexadecimal
// binary
int binNumber = 0b10010; // 0b represents binary
```

In Java, binary starts with **0b**, octal starts with **0**, and hexadecimal starts with **0x**.

Note: Integer literals are used to initialize variables of integer types like byte, short, int, and long.

3. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponential form. For example,

```
class Main {
  public static void main(String[] args) {

    double myDouble = 3.4;
    float myFloat = 3.4F;

    // 3.445*10^2
    double myDoubleScientific = 3.445e2;

    System.out.println(myDouble);  // prints 3.4
    System.out.println(myFloat);    // prints 3.4
    System.out.println(myDoubleScientific);   // prints 344.5
  }
}
```

Note: The floating-point literals are used to initialize float and double type variables.

4. Character Literals

Character literals are unicode character enclosed inside single quotes. For example,

```
char letter = 'a';
```

Here, a is the character literal.

We can also use escape sequences as character literals. For example, **\b** (backspace), **\t** (tab), **\n** (new line), etc.

5. String literals

A string literal is a sequence of characters enclosed inside double-quotes. For example,

```
String str1 = "Java Programming";
String str2 = "Programiz";
```

Here, Java Programming and Programiz are two string literals.

Java Data Types (Primitive)

As the name suggests, data types specify the type of data that can be stored inside variables in Java.

Java is a statically-typed language. This means that all variables must be declared before they can be used.

int speed;

Here, *speed* is a variable, and the data type of the variable is int.

The *int* data type determines that the *speed* variable can only contain integers.

There are 8 data types predefined in Java programming language, known as primitive data types.

Note: In addition to primitive data types, there are also referenced types (object type).

8 Primitive Data Types

1. boolean type

- The boolean data type has two possible values, either true or false.
- Default value: false.
- They are usually used for **true/false** conditions.

Example 1: Java boolean data type

```
class Main {
  public static void main(String[] args) {

    boolean flag = true;
    System.out.println(flag);    // prints true
  }
}
```

2. byte type

- The byte data type can have values from **-128** to **127** (8-bit signed two's complement integer).
- If it's certain that the value of a variable will be within -128 to 127, then it is used instead of int to save memory.
- Default value: 0

Example 2: Java byte data type

```
class Main {
  public static void main(String[] args) {

    byte range;
    range = 124;
    System.out.println(range);    // prints 124
  }
}
```

3. short type

- The short data type in Java can have values from **-32768** to **32767** (16-bit signed two's complement integer).

- If it's certain that the value of a variable will be within -32768 and 32767, then it is used instead of other integer data types (int, long).
- Default value: 0

Example 3: Java short data type

```
class Main {
  public static void main(String[] args) {

    short temperature;
    temperature = -200;
    System.out.println(temperature);  // prints -200
  }
```

4. int type

- The int data type can have values from **-231** to **231-1** (32-bit signed two's complement integer).
- If you are using Java 8 or later, you can use an unsigned 32-bit integer. This will have a minimum value of 0 and a maximum value of 232-1.
- Default value: 0

Example 4: Java int data type

```
class Main {
  public static void main(String[] args) {

    int range = -4250000;
    System.out.println(range);  // print -4250000
  }
}
```

5. long type

- The long data type can have values from **-263** to **263-1** (64-bit signed two's complement integer).
- If you are using Java 8 or later, you can use an unsigned 64-bit integer with a minimum value of **0** and a maximum value of **264-1**.
- Default value: 0

Example 5: Java long data type

```
class LongExample {
  public static void main(String[] args) {

    long range = -42332200000L;
    System.out.println(range);    // prints -42332200000
```

```
  }
}
```

Notice, the use of L at the end of -42332200000. This represents that it's an integral literal of the long type.

6. double type

- The double data type is a double-precision 64-bit floating-point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0d)

Example 6: Java double data type

```
class Main {
  public static void main(String[] args) {

    double number = -42.3;
    System.out.println(number);  // prints -42.3
  }
}
```

7. float type

- The float data type is a single-precision 32-bit floating-point.
- It should never be used for precise values such as currency.
- Default value: 0.0 (0.0f)

Example 7: Java float data type

```
class Main {
  public static void main(String[] args) {

    float number = -42.3f;
    System.out.println(number);  // prints -42.3
  }
}
```

Notice that, we have used -42.3f instead of -42.3in the above program. It's because -42.3 is a double literal.

To tell the compiler to treat -42.3 as float rather than double, you need to use *f* or *F*.

8. char type

- It's a 16-bit Unicode character.
- The minimum value of the char data type is '\u0000' (0) and the maximum value of the is '\uffff'.
- Default value: '\u0000'

Example 8: Java char data type

```
class Main {
  public static void main(String[] args) {

    char letter = '\u0051';
    System.out.println(letter);  // prints Q
  }
}
```

Here, the Unicode value of Q is **\u0051**. Hence, we get Q as the output.

Here is another example:

```
class Main {
  public static void main(String[] args) {

    char letter1 = '9';
    System.out.println(letter1);  // prints 9

    char letter2 = 65;
    System.out.println(letter2);  // prints A

  }
}
```

Here, we have assigned 9 as a character (specified by single quotes) to the *letter1* variable. However, the *letter2* variable is assigned 65 as an integer number (no single quotes).

Hence, A is printed to the output. It is because Java treats characters as integral types and the ASCII value of A is 65.

9. String type

Java also provides support for character strings via java.lang.String class. Strings in Java are not primitive types. Instead, they are objects. For example,

```
String myString = "Java Programming";
```

Here, *myString* is an object of the String class.

Java Operators

Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while * is also an operator used for multiplication.

Operators in Java can be classified into 5 types:

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Unary Operators
6. Bitwise Operators

1. Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

```
a + b;
```

Here, the + operator is used to add two variables *a* and *b*. Similarly, there are various other arithmetic operators in Java.

Operator	Operation
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Example 1: Arithmetic Operators

```
class Main {
  public static void main(String[] args) {

    // declare variables
    int a = 12, b = 5;

    // addition operator
    System.out.println("a + b = " + (a + b));

    // subtraction operator
    System.out.println("a - b = " + (a - b));

    // multiplication operator
    System.out.println("a * b = " + (a * b));

    // division operator
    System.out.println("a / b = " + (a / b));

    // modulo operator
    System.out.println("a % b = " + (a % b));
  }
}
```

Output

```
a + b = 17
a - b = 7
a * b = 60
a / b = 2
a % b = 2
```

In the above example, we have used +, -, and * operators to compute addition, subtraction, and multiplication operations.

/ Division Operator

Note the operation, a / b in our program. The / operator is the division operator.

If we use the division operator with two integers, then the resulting quotient will also be an integer. And, if one of the operands is a floating-point number, we will get the result will also be in floating-point.

```
In Java,

(9 / 2) is 4
```

```
(9.0 / 2) is 4.5
(9 / 2.0) is 4.5
(9.0 / 2.0) is 4.5
```

% Modulo Operator

The modulo operator % computes the remainder. When a = 7 is divided by b = 4, the remainder is **3**.

Note: The % operator is mainly used with integers.

2. Java Assignment Operators

Assignment operators are used in Java to assign values to variables. For example,

```
int age;
age = 5;
```

Here, = is the assignment operator. It assigns the value on its right to the variable on its left. That is, **5** is assigned to the variable *age*.

Let's see some more assignment operators available in Java.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Example 2: Assignment Operators

```
class Main {
  public static void main(String[] args) {

    // create variables
    int a = 4;
    int var;
```

```
    // assign value using =
    var = a;
    System.out.println("var using =: " + var);

    // assign value using =+
    var += a;
    System.out.println("var using +=: " + var);

    // assign value using =*
    var *= a;
    System.out.println("var using *=: " + var);
  }
}
```

Output

```
var using =: 4
var using +=: 8
var using *=: 32
```

3. Java Relational Operators

Relational operators are used to check the relationship between two operands. For example,

```
// check is a is less than b
a < b;
```

Here, > operator is the relational operator. It checks if *a* is less than *b* or not.

It returns either true or false.

Operator	Description	Example
==	Is Equal To	3 == 5 returns **false**
!=	Not Equal To	3 != 5 returns **true**
>	Greater Than	3 > 5 returns **false**
<	Less Than	3 < 5 returns **true**
>=	Greater Than or Equal To	3 >= 5 returns **false**

<= Less Than or Equal To `3 <= 5` returns **true**

Example 3: Relational Operators

```
class Main {
  public static void main(String[] args) {

    // create variables
    int a = 7, b = 11;

    // value of a and b
    System.out.println("a is " + a + " and b is " + b);

    // == operator
    System.out.println(a == b);  // false

    // != operator
    System.out.println(a != b);  // true

    // > operator
    System.out.println(a > b);  // false

    // < operator
    System.out.println(a < b);  // true

    // >= operator
    System.out.println(a >= b);  // false

    // <= operator
    System.out.println(a <= b);  // true
  }
}
```

Note: Relational operators are used in decision making and loops.

4. Java Logical Operators

Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
`&&` (Logical AND)	expression1 **&&** expression2	`true` only if both *expression1* and *expression2* are `true`
`||` (Logical OR)	expression1 **||** expression2	`true` if either *expression1* or *expression2* is `true`
`!` (Logical NOT)	**!**expression	`true` if *expression* is `false` and vice versa

Example 4: Logical Operators

```
class Main {
  public static void main(String[] args) {

    // && operator
    System.out.println((5 > 3) && (8 > 5));  // true
    System.out.println((5 > 3) && (8 < 5));  // false

    // || operator
    System.out.println((5 < 3) || (8 > 5));  // true
    System.out.println((5 > 3) || (8 < 5));  // true
    System.out.println((5 < 3) || (8 < 5));  // false

    // ! operator
    System.out.println(!(5 == 3));  // true
    System.out.println(!(5 > 3));  // false
  }
}
```

Working of Program

- `(5 > 3) && (8 > 5)` returns `true` because both `(5 > 3)` and `(8 > 5)` are `true`.
- `(5 > 3) && (8 < 5)` returns `false` because the expression `(8 < 5)` is `false`.
- `(5 < 3) || (8 > 5)` returns `true` because the expression `(8 > 5)` is `true`.
- `(5 > 3) && (8 > 5)` returns `true` because the expression `(5 > 3)` is `true`.
- `(5 > 3) && (8 > 5)` returns `false` because both `(5 < 3)` and `(8 < 5)` are `false`.
- `!(5 == 3)` returns true because `5 == 3` is `false`.
- `!(5 > 3)` returns false because `5 > 3` is `true`.

5. Java Unary Operators

Unary operators are used with only one operand. For example, ++ is a unary operator that increases the value of a variable by **1**. That is, ++5 will return **6**.

Different types of unary operators are:

Operator	Meaning
+	**Unary plus**: not necessary to use since numbers are positive without using it
-	**Unary minus**: inverts the sign of an expression
++	**Increment operator**: increments value by 1
--	**Decrement operator**: decrements value by 1

!	**Logical complement operator**: inverts the value of a boolean

Increment and Decrement Operators

Java also provides increment and decrement operators: `++` and `--` respectively. `++` increases the value of the operand by **1**, while `--` decrease it by **1**. For example,

```
int num = 5;

// increase num by 1
++num;
```

Here, the value of *num* gets increased to **6** from its initial value of **5**.

Example 5: Increment and Decrement Operators

```
class Main {
  public static void main(String[] args) {

    // declare variables
    int a = 12, b = 12;
    int result1, result2;

    // original value
    System.out.println("Value of a: " + a);

    // increment operator
    result1 = ++a;
    System.out.println("After increment: " + result1);

    System.out.println("Value of b: " + b);

    // decrement operator
    result2 = --b;
    System.out.println("After decrement: " + result2);
  }
}
```

Output

```
Value of a: 12
After increment: 13
Value of b: 12
After decrement: 11
```

In the above program, we have used the ++ and -- operator as **prefixes (++a, --b)**. We can also use these operators as **postfix (a++, b++)**.

There is a slight difference when these operators are used as prefix versus when they are used as a postfix.

6. Java Bitwise Operators

Bitwise operators in Java are used to perform operations on individual bits. For example,

```
Bitwise complement Operation of 35

35 = 00100011 (In Binary)

~ 00100011
  ________
   11011100  = 220 (In decimal)
```

Here, ~ is a bitwise operator. It inverts the value of each bit (**0** to **1** and **1** to **0**).

The various bitwise operators present in Java are:

Operator	Description
~	Bitwise Complement
<<	Left Shift
>>	Right Shift
>>>	Unsigned Right Shift
&	Bitwise AND
^	Bitwise exclusive OR

These operators are not generally used in Java.

Other operators

Besides these operators, there are other additional operators in Java.

Java instanceof Operator

The instanceof operator checks whether an object is an instanceof a particular class. For example,

```
class Main {
  public static void main(String[] args) {

    String str = "Programiz";
    boolean result;

    // checks if str is an instance of
    // the String class
    result = str instanceof String;
    System.out.println("Is str an object of String? " + result);
  }
}
```

Output

```
Is str an object of String? true
```

Here, *str* is an instance of the String class. Hence, the instanceof operator returns true.

Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the if-then-else statement. For example,

```
variable = Expression ? expression1 : expression2
```

Here's how it works.

- If the Expression is true, expression1 is assigned to the *variable*.
- If the Expression is false, expression2 is assigned to the *variable*.

Let's see an example of a ternary operator.

```
class Java {
  public static void main(String[] args) {

    int februaryDays = 29;
    String result;

    // ternary operator
    result = (februaryDays == 28) ? "Not a leap year" : "Leap year";
    System.out.println(result);
  }
}
```

Output

```
Leap year
```

In the above example, we have used the ternary operator to check if the year is a leap year or not.

Java Basic Input and Output

Java Output

In Java, you can simply use

```
System.out.println(); or

System.out.print(); or

System.out.printf();
```

to send output to standard output (screen).

Here,

- `System` is a class
- `out` is a `public` `static` field: it accepts output data.

Don't worry if you don't understand it. We will discuss class, public, and static in later chapters.

Let's take an example to output a line.

```
class AssignmentOperator {
    public static void main(String[] args) {

        System.out.println("Java programming is interesting.");
    }
}
```

Output:

```
Java programming is interesting.
```

Here, we have used the println() method to display the string.

Difference between println(), print() and printf()

- print() - It prints string inside the quotes.
- println() - It prints string inside the quotes similar like print() method. Then the cursor moves to the beginning of the next line.
- printf() - It provides string formatting (similar to printf in C/C++ programming).

Example: print() and println()

```
class Output {
    public static void main(String[] args) {

        System.out.println("1. println ");
        System.out.println("2. println ");

        System.out.print("1. print ");
        System.out.print("2. print");
    }
}
```

Output:

```
1. println
2. println
1. print 2. print
```

In the above example, we have shown the working of the print() and println() methods.

Example: Printing Variables and Literals

```
class Variables {
    public static void main(String[] args) {

        Double number = -10.6;

        System.out.println(5);
        System.out.println(number);
    }
}
```

When you run the program, the output will be:

```
5
-10.6
```

Here, you can see that we have not used the quotation marks. It is because to display integers, variables and so on, we don't use quotation marks.

Example: Print Concatenated Strings

```
class PrintVariables {
    public static void main(String[] args) {

        Double number = -10.6;

        System.out.println("I am " + "awesome.");
        System.out.println("Number = " + number);
    }
}
```

Output:

```
I am awesome.
Number = -10.6
```

In the above example, notice the line,

```
System.out.println("I am " + "awesome.");
```

Here, we have used the + operator to concatenate (join) the two strings: *"I am "* and *"awesome."*.

And also, the line,

```
System.out.println("Number = " + number);
```

Here, first the value of variable *number* is evaluated. Then, the value is concatenated to the string: *"Number = "*.

Java Input

Java provides different ways to get input from the user. However, in this tutorial, you will learn to get input from user using the object of Scanner class.

In order to use the object of Scanner, we need to import java.util.Scanner package.

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

To learn more about importing packages in Java, visit Java Import Packages.

Then, we need to create an object of the Scanner class. We can use the object to take input from the user.

```
// create an object of Scanner
Scanner input = new Scanner(System.in);

// take input from the user
int number = input.nextInt();
```

Example: Get Integer Input From the User

```
import java.util.Scanner;

class Input {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("You entered " + number);

        // closing the scanner object
        input.close();
    }
}
```

Output:

```
Enter an integer: 23
You entered 23
```

In the above example, we have created an object named *input* of the Scanner class. We then call the nextInt() method of the Scanner class to get an integer input from the user.

Similarly, we can use nextLong(), nextFloat(), nextDouble(), and next() methods to get long, float, double, and string input respectively from the user.

Note: We have used the close() method to close the object. It is recommended to close the scanner object once the input is taken.

Example: Get float, double and String Input

```
import java.util.Scanner;

class Input {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        // Getting float input
        System.out.print("Enter float: ");
        float myFloat = input.nextFloat();
        System.out.println("Float entered = " + myFloat);

        // Getting double input
        System.out.print("Enter double: ");
        double myDouble = input.nextDouble();
        System.out.println("Double entered = " + myDouble);

        // Getting String input
        System.out.print("Enter text: ");
        String myString = input.next();
        System.out.println("Text entered = " + myString);
    }
}
```

Output:

```
Enter float: 2.343
Float entered = 2.343
Enter double: -23.4
Double entered = -23.4
Enter text: Hey!
Text entered = Hey!
```

As mentioned, there are other several ways to get input from the user.

Java Expressions, Statements and Blocks

In previous chapters, we have used expressions, statements, and blocks without much explaining about them. Now that you know about variables, operators, and literals, it will be easier to understand these concepts.

Java Expressions

A Java expression consists of variables, operators, literals, and method calls. To know more about method calls, visit Java methods. For example,

```
int score;
score = 90;
```

Here, score = 90 is an expression that returns an int. Consider another example,

```
Double a = 2.2, b = 3.4, result;
result = a + b - 3.4;
```

Here, a + b - 3.4 is an expression.

```
if (number1 == number2)
    System.out.println("Number 1 is larger than number 2");
```

Here, number1 == number2 is an expression that returns a boolean value. Similarly, "Number 1 is larger than number 2" is a string expression.

Java Statements

In Java, each statement is a complete unit of execution. For example,

```
int score = 9*5;
```

Here, we have a statement. The complete execution of this statement involves multiplying integers 9 and 5 and then assigning the result to the variable score.

In the above statement, we have an expression 9 * 5. In Java, expressions are part of statements.

Expression statements

We can convert an expression into a statement by terminating the expression with a ;. These are known as expression statements. For example,

```
// expression
number = 10
// statement
number = 10;
```

In the above example, we have an expression number = 10. Here, by adding a semicolon (;), we have converted the expression into a statement (number = 10;).

Consider another example,

```
// expression
++number
// statement
++number;
```

Similarly, ++number is an expression whereas ++number; is a statement.

Declaration Statements

In Java, declaration statements are used for declaring variables. For example,

```
Double tax = 9.5;
```

The statement above declares a variable *tax* which is initialized to 9.5.

Note: There are control flow statements that are used in decision making and looping in Java. You will learn about control flow statements in later chapters.

Java Blocks

A block is a group of statements (zero or more) that is enclosed in curly braces { }. For example,

```
class Main {
    public static void main(String[] args) {

        String band = "Beatles";

        if (band == "Beatles") { // start of block
```

```
            System.out.print("Hey ");
            System.out.print("Jude!");
        } // end of block
    }
}
```

Output:

```
Hey Jude!
```

In the above example, we have a block if {....}.

Here, inside the block we have two statements:

- `System.out.print("Hey ");`
- `System.out.print("Jude!");`

However, a block may not have any statements. Consider the following examples,

```
class Main {
    public static void main(String[] args) {

        if (10 > 5) { // start of block

        } // end of block
    }
}
```

This is a valid Java program. Here, we have a block if {...}. However, there is no any statement inside this block.

```
class AssignmentOperator {
    public static void main(String[] args) {  // start of block

    } // end of block
}
```

Here, we have block public static void main() {...}. However, similar to the above example, this block does not have any statement.

Java Comments

In computer programming, comments are a portion of the program that are completely ignored by Java compilers. They are mainly used to help programmers to understand the code. For example,

```
// declare and initialize two variables
int a =1;
int b = 3;

// print the output
System.out.println("This is output");
```

Here, we have used the following comments,

- declare and initialize two variables
- print the output

Types of Comments in Java

In Java, there are two types of comments:

- single-line comment
- multi-line comment

Single-line Comment

A single-line comment starts and ends in the same line. To write a single-line comment, we can use the // symbol. For example,

```
// "Hello, World!" program example

class Main {
    public static void main(String[] args) {
    {
        // prints "Hello, World!"
        System.out.println("Hello, World!");
    }
}
```

Output:

```
Hello, World!
```

Here, we have used two single-line comments:

- *"Hello, World!" program example*
- *prints "Hello World!"*

The Java compiler ignores everything from // to the end of line. Hence, it is also known as **End of Line** comment.

Multi-line Comment

When we want to write comments in multiple lines, we can use the multi-line comment. To write multi-line comments, we can use the /*....*/ symbol. For example,

```
/* This is an example of  multi-line comment.
 * The program prints "Hello, World!" to the standard output.
 */

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

Output:

```
Hello, World!
```

Here, we have used the multi-line comment:

```
/* This is an example of multi-line comment.
* The program prints "Hello, World!" to the standard output.
*/
```

This type of comment is also known as **Traditional Comment**. In this type of comment, the Java compiler ignores everything from /* to */.

Use Comments the Right Way

One thing you should always consider that comments shouldn't be the substitute for a way to explain poorly written code in English. You should always write well structured and self explaining code. And, then use comments.

Some believe that code should be self-describing and comments should be rarely used. However, in my personal opinion, there is nothing wrong with using comments. We can use comments to explain complex algorithms, regex or scenarios where we have to choose one technique among different technique to solve problems.

Note: In most cases, always use comments to explain '**why**' rather than '**how**' and you are good to go.

Java Package

A package is simply a container that groups related types (Java classes, interfaces, enumerations, and annotations). For example, in core Java, the ResultSet interface belongs to the java.sql package. The package contains all the related types that are needed for the SQL query and database connection.

If you want to use the ResultSet interface in your code, just import the java.sql package (Importing packages will be discussed later in the article).

As mentioned earlier, packages are just containers for Java classes, interfaces and so on. These packages help you to reserve the class namespace and create a maintainable code.

For example, you can find two Date classes in Java. However, the rule of thumb in Java programming is that only one unique class name is allowed in a Java project.

How did they manage to include two classes with the same name Date in JDK?

This was possible because these two Date classes belong to two different packages:

- java.util.Date - this is a normal Date class that can be used anywhere.
- java.sql.Date - this is a SQL Date used for the SQL query and such.

Based on whether the package is defined by the user or not, packages are divided into two categories:

Built-in Package

Built-in packages are existing java packages that come along with the JDK. For example, java.lang, java.util, java.io, etc. For example:

```
import java.util.ArrayList;

class ArrayListUtilization {
    public static void main(String[] args) {

        ArrayList<Integer> myList = new ArrayList<>(3);
        myList.add(3);
        myList.add(2);
        myList.add(1);

        System.out.println(myList);
    }
}
```

Output:

```
myList = [3, 2, 1]
```

The ArrayList class belongs to java.util package. To use it, we have to import the package first using the import statement.

```
import java.util.ArrayList;
```

User-defined Package

Java also allows you to create packages as per your need. These packages are called user-defined packages.

How to define a Java package?

To define a package in Java, you use the keyword package.

```
package packageName;
```

Java uses file system directories to store packages. Let's create a Java file inside another directory.

For example:

```
└── com
    └── test
        └── Test.java
```

Now, edit **Test.java** file, and at the beginning of the file, write the package statement as:

```
package com.test;
```

Here, any class that is declared within the test directory belongs to the **com.test** package.

Here's the code:

```
package com.test;

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

Output:

```
Hello World!
```

Here, the *Test* class now belongs to the **com.test** package.

Package Naming convention

The package name must be **unique** (like a domain name). Hence, there's a convention to create a package as a domain name, but in reverse order. For example, **com.company.name**

Here, each level of the package is a directory in your file system. Like this:

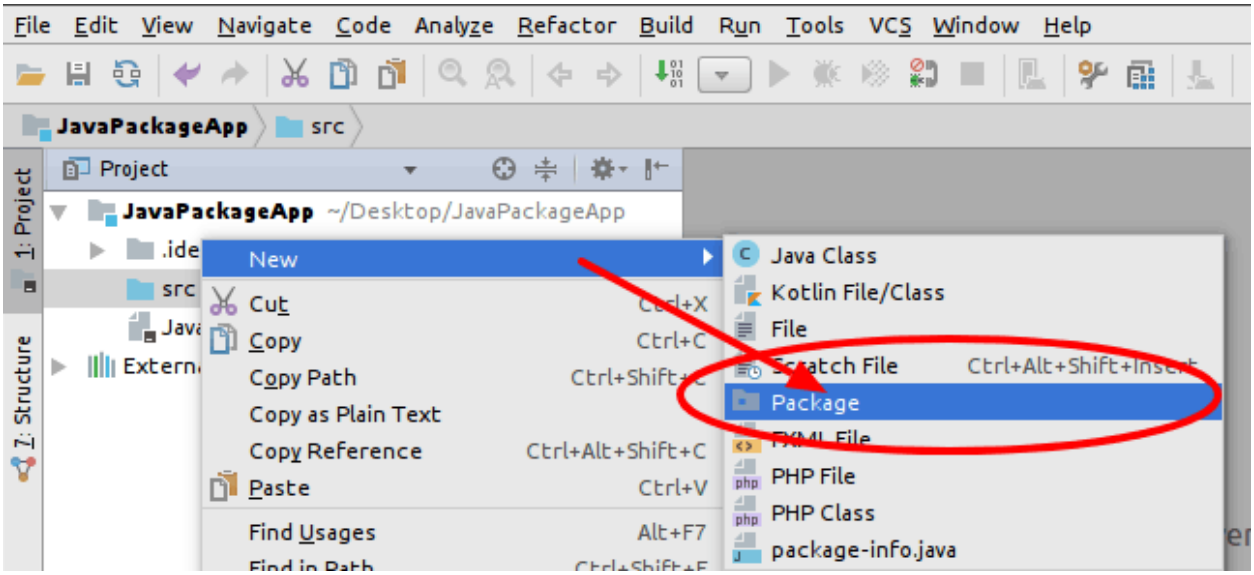
```
└── com
    └── company
        └── name
```

And, there is no limitation on how many subdirectories (package hierarchy) you can create.

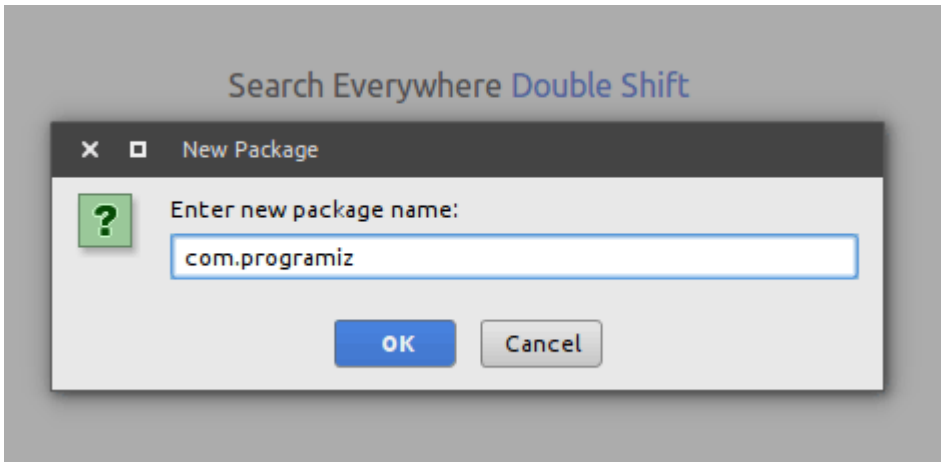
How to create a package in Intellij IDEA?

In IntelliJ IDEA, here's how you can create a package:

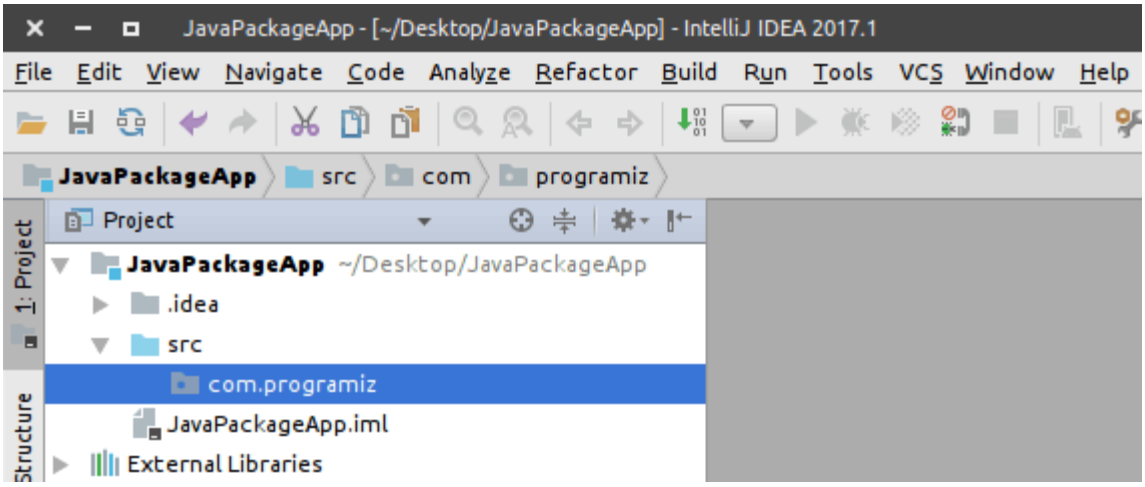
1. Right-click on the source folder.
2. Go to **new** and then **package**.



3. A pop-up box will appear where you can enter the package name.



Once the package is created, a similar folder structure will be created on your file system as well. Now, you can create classes, interfaces, and so on inside the package.



How to import packages in Java?

Java has an import statement that allows you to import an entire package (as in earlier examples), or use only certain classes and interfaces defined in the package.

The general form of import statement is:

```
import package.name.ClassName;   // To import a certain class only
import package.name.*   // To import the whole package
```

For example,

```
import java.util.Date; // imports only Date class
import java.io.*;      // imports everything inside java.io package
```

The import statement is optional in Java.

If you want to use class/interface from a certain package, you can also use its **fully qualified name**, which includes its full package hierarchy.

Here is an example to import a package using the import statement.

```
import java.util.Date;

class MyClass implements Date {
    // body
}
```

The same task can be done using the fully qualified name as follows:

```
class MyClass implements java.util.Date {
    //body
}
```

Example: Package and importing package

Suppose, you have defined a package **com.programiz** that contains a class *Helper*.

```
package com.programiz;

public class Helper {
    public static String getFormattedDollar (double value){
        return String.format("$%.2f", value);
    }
}
```

Now, you can import *Helper* class from **com.programiz** package to your implementation class. Once you import it, the class can be referred directly by its name. Here's how:

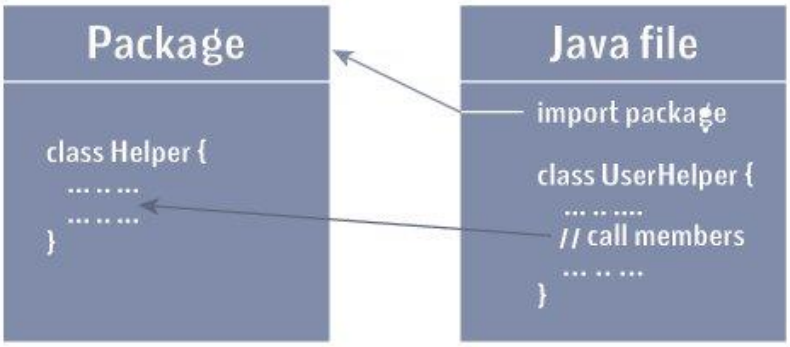
```
import com.programiz.Helper;

class UseHelper {
    public static void main(String[] args) {

        double value = 99.5;
        String formattedValue = Helper.getFormattedDollar(value);
        System.out.println("formattedValue = " + formattedValue);
    }
}
```

Output:

```
formattedValue = $99.50
```

Here,

1. the *Helper* class is defined in **com.programiz** package.
2. the *Helper* class is imported to a different file. The file contains *UseHelper* class.
3. The getFormattedDollar() method of the *Helper* class is called from inside the *UseHelper* class.



Java import package

In Java, the import statement is written directly after the package statement (if it exists) and before the class definition.

For example,

```
package package.name;
import package.ClassName; // only import a Class

class MyClass {
    // body
}
```