**Java Syntax: -**

**Java Class**

**public** **class** Syntax {

}

1. Every line of code that runs in Java must be inside a class.
2. A class should always start with an uppercase first letter.
3. Java is case-sensitive: "MyClass" and "myclass" has different meaning.
4. The name of the java class file must match the class name.

**Java Main**

**public** **class** Syntax {

**public** **static** **void** main(String[] args) {

System.***out***.println("Hello Rathan!!");

}

}

1. The main() method is required and you will see it in every Java program

**System.out.println()**

**public** **class** Syntax {

**public** **static** **void** main(String[] args) {

System.***out***.println("Hello Rathan!!");

}

}

1. Inside the main() method, we can use the println() method to print a line of text to the screen.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Output/ Print: -**

**Print Text**

1. When you are working with text, it must be wrapped inside double quotations marks "".
2. If you forget the double quotes, an error occurs.

System.***out***.println("This section will work");

System.***out***.println(This section will not work);

1. There is also a print() method, which is similar to println().
2. here only difference is that it does not insert a new line at the end of the output.

System.***out***.print("Hello Rathan");

System.***out***.print("This section will be printed in the same line");

In the Above Example Section will be printed in the same line

**Print Number**

1. You can also use the println() method to print numbers.
2. However, unlike text, we don't put numbers inside double quotes “”

System.***out***.println(3);

System.***out***.println(358);

System.***out***.println(50000);

1. You can also perform mathematical calculations inside the println() method.

System.out.println(3 + 3);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Comment: -**

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

**Single-line Comment**

1. Single-line comments start with two forward slashes (//).
2. Any text after // will be ignored by Java (will not be executed).

//This is comment line will not be Executed

System.***out***.println("Hello World");

**Multi-line Comment**

1. Multi-line comments start with /\* and ends with \*/.
2. Any text between /\* and \*/ will be ignored by Java

/\*This is comment line will not be Executed

because these Texts are written in Multi line\*/

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Variables:-**

**Variable Types**

Variables are containers for storing data values.

In Java, there are different **types** of variables, for example:

* String - stores text, such as "Hello". String values are surrounded by double quotes.
* int - stores integers (whole numbers), without decimals, such as 123 or -123
* float - stores floating point numbers, with decimals, such as 19.99 or -19.99
* char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes.
* boolean - stores values with two states: true or false.

**Declaring Variable**

*variabletype variableName = value;* // Syntax of Declaring Variable

**int** myNum = 5;

**float** myFloatNum = 5.99f;

**char** myLetter = 'D';

**boolean** myBool = **true**;

**String** myText = "Hello";

**double** myDouble = 5.99;

**Declare Many Variables**

Instead of writing like this

**int** x = 5;

**int** y = 6;

**int** z = 50;

System.***out***.println(x + y + z);

We can write like this

**int** x = 5, y = 6, z = 50;

System.***out***.println(x + y + z);

One value to Multiple Variable

**int** x, y, z;

x = y = z = 50;

System.***out***.println(x + y + z);

**Declaring Variable Names**

* Names can contain letters, digits, underscores, and dollar signs
* Names must begin with a letter
* Names can start with Uppercase or Lowercase letter.
* Names can also begin with $ and \_ but it is not a good practise
* Names are case sensitive ("myVar" and "myvar" are different variables)
* Reserved words (like Java keywords, such as int or boolean) cannot be used as names

**Java DataType:-**

Data types are divided into two groups:

* Primitive data types - includes byte, short, int, long, float, double, boolean and char
* Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp) (you will learn more about these in a later chapter)

Primitive Data Type: -

**byte** ---> 1 **byte** ---> Stores whole numbers from -128 to 127

**short** ---> 2 bytes ---> Stores whole numbers from -32,768 to 32,767

**int** ---> 4 bytes ---> Stores whole numbers from -2,147,483,648 to 2,147,483,647

**long** ---> 8 bytes ---> Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

**float** ---> 4 bytes ---> Stores fractional numbers. Sufficient **for** storing 6 to 7 decimal digits

**double** ---> 8 bytes ---> Stores fractional numbers. Sufficient **for** storing 15 decimal digits

**Boolean** ---> 1 bit ---> Stores **true** or **false** values

**Char** ---> 2 bytes ---> Stores a single character/letter or ASCII values \

Non-Primitive Data Type: -

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

* Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
* Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type has always a value, while non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
* Non-primitive data types - such as [String](https://www.w3schools.com/java/java_strings.asp), [Arrays](https://www.w3schools.com/java/java_arrays.asp) and [Classes](https://www.w3schools.com/java/java_classes.asp) (you will learn more about these in a later chapter)

Boolean Type

Very often in programming, you will need a data type that can only have one of two values, like:

* YES / NO
* ON / OFF
* TRUE / FALSE

For this, Java has a boolean data type, which can only take the values true or false.

System.***out***.println(10 > 9); // returns true, because 10 is higher than 9

**int** x = 10;

System.***out***.println(x == 10); // returns true, because the value of x is equal to 10

System.***out***.println(10 == 15); // returns false, because 10 is not equal to 15

**int** myAge = 25;

**int** votingAge = 18;

System.***out***.println(myAge >= votingAge);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Type Casting: -**

**Widening Casting**

**Widening Casting** (automatically) - converting a smaller type to a larger type size  
byte -> short -> char -> int -> long -> float -> double

**int** myInt = 9;

**double** myDouble = myInt; // Automatic casting: int to double

System.***out***.println(myInt); // Outputs 9

System.***out***.println(myDouble); // Outputs 9.0

**Narrow Casting**

**Narrowing Casting** (manually) - converting a larger type to a smaller size type  
double -> float -> long -> int -> char -> short -> byte

**double** myDouble = 9.78d;

**int** myInt = (**int**) myDouble; // Manual casting: double to int

System.***out***.println(myDouble); // Outputs 9.78

System.***out***.println(myInt); // Outputs 9

**Java Operator:-**

Java divides the operators into the following groups:

* Arithmetic operators
* Assignment operators
* Comparison operators
* Logical operators
* Bitwise operators

**Arithmetic Operator**

Arithmetic operators are used to perform common mathematical operations.

(+) ---> Addition ---> Adds together two values ---> x + y

(-) ---> Subtraction ---> Subtracts one value from another ---> x – y

(\*) ---> Multiplication ---> Multiplies two values ---> x \* y

(/) ---> Division ---> Divides one value by another ---> x / y

(%) ---> Modulus ---> Returns the division remainder ---> x % y

(++) ---> Increment ---> Increases the value of a variable by 1 ---> ++x

(--) ---> Decrement ---> Decreases the value of a variable by 1 ---> --x

**Assignment Operator**

Assignment operators are used to assign values to variables

(=) ---> x = 5 ---> x = 5

(+=) ---> x += 3 ---> x = x + 3

(-=) ---> x -= 3 ---> x = x – 3

(\*=) ---> x \*= 3 ---> x = x \* 3

(/=) ---> x /= 3 ---> x = x / 3

(%=) ---> x %= 3 ---> x = x % 3

(&=) ---> x &= 3 ---> x = x & 3

(|=) ---> x |= 3 ---> x = x | 3

(^=) ---> x ^= 3 ---> x = x ^ 3

(>>=) ---> x >>= 3 ---> x = x >> 3

<= ---> x <<= 3 ---> x = x << 3

Syntax:-

**int** y = 10; // Assign value to a Variable

**Comparison Operator**

(==) ---> Equal to ---> x == y

(!=) ---> Not equal ---> x != y

(>) ---> Greater than ---> x > y

(<) ---> Less than ---> x < y

(>=) ---> Greater than or equal to ---> x >= y

(<=) ---> Less than or equal to ---> x <= y

**Logical Operator**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java String: -**

**String Length**

In Java, length() is a method used to retrieve the length of a string

String txt = "Rathan Rajiv Singh";

System.***out***.println("The length of the txt string is: " + txt.length());

**toUpperCase() & toLowerCase()**

toUpperCase() is used to conver the text to Uppercase letter

toLowerCase() is used to convert the text to LowerCase

String txt = "Hello World";

System.***out***.println(txt.toUpperCase()); // Outputs "HELLO WORLD"

System.***out***.println(txt.toLowerCase()); // Outputs "hello world"

**indexOf()**

The indexOf() method returns the **index** (the position) of the first occurrence of a specified text in a string (including whitespace):

String txt = "Rathan Rajiv Singh";

System.***out***.println(txt.indexOf("Rajiv")); // Outputs 7

**Java String Concatenation**

In Java, String concatenation forms a new String that is the combination of multiple strings.

1. The + operator can be used between strings to combine them

String firstName = "Rathan";

String lastName = "Rajiv";

System.***out***.println(firstName + " " + lastName);

1. You can also use the concat() method to concatenate two strings.

String firstName = "Rathan ";

String lastName = "Rajiv";

System.***out***.println(firstName.concat(lastName));

**Adding Numbers & String**

Java uses the + operator for both addition and concatenation.Numbers are added. Strings are concatenated.

1. If you add two numbers, the result will be a number.

**int** x = 10;

**int** y = 20;

**int** z = x + y; // z will be 30 (an integer/number)

1. If you add two strings, the result will be a string concatenation.

String x = "10";

String y = "20";

String z = x + y; // z will be 1020 (a String)

1. If you add a number and a string, the result will be a string concatenation.

String x = "10";

**int** y = 20;

String z = x + y; // z will be 1020 (a String)

**String Special Characters**

Because strings must be written within quotes, Java will misunderstand this string, and generate an error.

String txt = "We are the so-called "Vikings" from the north."; // This will throw Error

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (\) escape character turns special characters into string characters.

\' ---> ' ---> This sequence inserts a Single quote in string

\" ---> " ---> This sequence inserts a Double quote in string

\\ ---> \ ---> This sequence inserts a Backslash in string

Other common escape sequences that are valid in Java are.

\n ---> New Line

\r ---> Carriage Return

\t ---> Tab

\b ---> Backspace

\f ---> Form Feed

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Math**

The Java Math class has many methods that allows you to perform mathematical tasks on numbers.

**Math.max(***x,y***)**

The Math.max(x,y) method can be used to find the highest value of x and y.

System.***out***.println(Math.*max*(5, 10)); // Output will be 10 because 10 is the Max value

**Math.min(***x,y***)**

The Math.min(x,y) method can be used to find the lowest value of x and y.

System.***out***.println(Math.*min*(5, 10)); //Output will be 5 because 5 is the Min value

**Math.sqrt(*x*)**

The Math.sqrt(x) method returns the square root of x.

System.***out***.println(Math.*sqrt*(3)); //Output will be 3 (3\*3)

**Math.abs(***x***)**

The Math.abs(x) method returns the absolute (positive) value of x.

System.***out***.println(Math.*abs*(-45.25) );

**Math.random()**

Math.random() returns a random number between 0.0 (inclusive), and 1.0 (exclusive):

System.***out***.println(Math.*random*());

To get more control over the random number, for example, if you only want a random number between 0 and 100, you can use the following formula.

**int** randomNum = (**int**)(Math.*random*() \* 101); // 0 to 100

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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Statements**

Java has the following conditional statements:

* Use if to specify a block of code to be executed, if a specified condition is true
* Use else to specify a block of code to be executed, if the same condition is false
* Use else if to specify a new condition to test, if the first condition is false
* Use switch to specify many alternative blocks of code to be executed

You already know that Java supports the usual logical conditions from mathematics:

* Less than: a < b
* Less than or equal to: a <= b
* Greater than: a > b
* Greater than or equal to: a >= b
* Equal to a == b
* Not Equal to: a != b

**If Statement**

Use the if statement to specify a block of Java code to be executed if a condition is true.

**if** (20 > 18) {

System.***out***.println("20 is greater than 18");

}

**Else Statement**

Use the else statement to specify a block of code to be executed if the condition is false.

**int** Age = 29;

**if** (Age < 18) {

System.***out***.println("Not Old Enough to Vote.");

} **else** {

System.***out***.println("Old Enough Vote.");

}

// Outputs "Old Enough Vote."

**Else-If Statement**

Use the else if statement to specify a new condition if the first condition is false.

**int** Age = 18;

**if** (Age < 18) {

System.***out***.println("Not Old Enough to Vote.");

}**else** **if** (Age==18){

System.***out***.println("Far enough to Vote but Need to check the DOB with current Date");

}

**else** {

System.***out***.println("Old Enough Vote.");

}

// Outputs "Old Enough Vote."

Short Hand If...Else (Ternary Operator)

It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements.

Syntax

variable = (condition) ? expressionTrue : expressionFalse;

**int** Age = 20;

String result = (Age < 18) ? "Not Eligible to Vote." : "Eligible to Vote.";

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

Switch Statement

Instead of writing **many** else if statements, you can use the switch statement.

The default keyword specifies some code to run if there is no case match.

When Java reaches a break keyword, it breaks out of the switch block.

**int** day = 7;

**switch** (day) {

**case** 1:

System.***out***.println("Monday");

**break**;

**case** 2:

System.***out***.println("Tuesday");

**break**;

**case** 3:

System.***out***.println("Wednesday");

**break**;

**case** 4:

System.***out***.println("Thursday");

**break**;

**case** 5:

System.***out***.println("Friday");

**break**;

**case** 6:

System.***out***.println("Saturday");

**break**;

**case** 7:

System.***out***.println("Sunday");

**break**;

**default**:

System.***out***.println("Not the Specified day");

}

// Outputs "Not the Specified day" (default)

Instead of the below If-Else statement we can write as above.

**int** day = 4;

**if** (day == 1) {

System.***out***.println("Your selected day is Monday.");

}**else** **if** (day==2){

System.***out***.println("Your selected day is Tuesday.");

}

**else** **if** (day==3){

System.***out***.println("Your selected day is Wednesday.");

}

**else** **if** (day==4){

System.***out***.println("Your selected day is Thursday.");

}

**else** **if** (day==5){

System.***out***.println("Your selected day is Friday.");

}

**else** **if** (day==6){

System.***out***.println("Your selected day is Saturday.");

}

**else** **if** (day==7){

System.***out***.println("Your selected day is Sunday.");

}

**else** {

System.***out***.println("Out of the day specified.");

}

// Outputs " Your selected day is Thursday."

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java loops**

Loops can execute a block of code as long as a specified condition is reached.

While Loop

The while loop loops through a block of code as long as a specified condition is true.

**int** countdown = 10;

System.***out***.println("Countdown starts for the rocket launch:");

**while** (countdown > 0) {

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

countdown--;

}

System.***out***.println("Blastoff!");

Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

**int** i = 0;

**do** {

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

i++;

}

**while** (i < 5);

For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop.

Syntax

for (statement 1; statement 2; statement 3) {

// code block to be executed

}

**Statement 1** is executed (one time) before the execution of the code block.

**Statement 2** defines the condition for executing the code block.

**Statement 3** is executed (every time) after the code block has been executed.

**for** (**int** i = 0; i < 5; i++) {

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

}

**Statement 1** sets a variable before the loop starts (int i = 0).

**Statement 2** defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

**Statement 3** increases a value (i++) each time the code block in the loop has been executed.

Nested Loop

It is also possible to place a loop inside another loop. This is called a **nested loop**.

The "inner loop" will be executed one time for each iteration of the "outer loop".

// Outer loop

**for** (**int** i = 1; i <= 2; i++) {

System.***out***.println("Outer: " + i); // Executes 2 times

// Inner loop

**for** (**int** j = 1; j <= 3; j++) {

System.***out***.println(" Inner: " + j); // Executes 6 times (2 \* 3)

}

}

Output: -

Outer: 1

Inner: 1

Inner: 2

Inner: 3

Outer: 2

Inner: 1

Inner: 2

Inner: 3

Real Time example for Nested loop using Multiple Table

//Outer Loop

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

System.***out***.println("< " + j + "x >");

//Inner loop

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

System.***out***.println(j + "x" + i + "=" + j\*i);

}

}

< 1x >

1x1=1

1x2=2

1x3=3

1x4=4

1x5=5

1x6=6

1x7=7

1x8=8

1x9=9

1x10=10

< 2x >

2x1=2

2x2=4

2x3=6

2x4=8

2x5=10

2x6=12

2x7=14

2x8=16

2x9=18

2x10=20

< 3x >

3x1=3

3x2=6

3x3=9

3x4=12

3x5=15

3x6=18

3x7=21

3x8=24

3x9=27

3x10=30

< 4x >

4x1=4

4x2=8

4x3=12

4x4=16

4x5=20

4x6=24

4x7=28

4x8=32

4x9=36

4x10=40

< 5x >

5x1=5

5x2=10

5x3=15

5x4=20

5x5=25

5x6=30

5x7=35

5x8=40

5x9=45

5x10=50

< 6x >

6x1=6

6x2=12

6x3=18

6x4=24

6x5=30

6x6=36

6x7=42

6x8=48

6x9=54

6x10=60

< 7x >

7x1=7

7x2=14

7x3=21

7x4=28

7x5=35

7x6=42

7x7=49

7x8=56

7x9=63

7x10=70

< 8x >

8x1=8

8x2=16

8x3=24

8x4=32

8x5=40

8x6=48

8x7=56

8x8=64

8x9=72

8x10=80

< 9x >

9x1=9

9x2=18

9x3=27

9x4=36

9x5=45

9x6=54

9x7=63

9x8=72

9x9=81

9x10=90

< 10x >

10x1=10

10x2=20

10x3=30

10x4=40

10x5=50

10x6=60

10x7=70

10x8=80

10x9=90

10x10=100

For-Each Loop (Enhanced for-each Loop:)

There is also a "**for-each**" loop, which is used exclusively to loop through elements in an Array.

String[] cars = {"Rolls Royce", "BMW", "Benz", "Vintage"};

**for**(String car : cars) {

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

}

Output: -

Rolls Royce

BMW

Benz

Vintage

Difference Between For & For-Each loop in Array

In Java, there are two different types of loops: the traditional for loop and the enhanced for-each loop (also known as the "enhanced for loop" or "for-each loop"). These two types of loops have distinct purposes and usage.

## Traditional for Loop:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

**for** (**int** i = 0; i < cars.length; i++) {

System.***out***.println(cars[i]);

}

In a traditional for loop, you use an index variable (in this case, i) to iterate through an array (cars in this example). You specify the loop condition (i < cars.length) and the increment step (i++). The loop iterates through the array from the beginning to the end. Inside the loop, you can access individual elements of the array using the index (cars[i]) and perform operations on each element.

## Enhanced for-each Loop:

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

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

// Outputs 4

The enhanced for-each loop is designed to simplify iteration through arrays or collections. It iterates over the elements of an array (or collection) directly, without the need for an index. In this loop, car represents each element of the cars array in each iteration. You don't need to manually manage an index or check the array length; the loop automatically iterates through all the elements in the array.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Array**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

Syntax: -

String[] cars;

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

**int**[] myNum = {10, 20, 30, 40};

You can access an array element by referring to the index number.

## Access the Elements of an Array

Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

String[] cars = {"Rolls Royce", "BMW", "Benz", "Vintage"};

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

// Outputs Rolls Royce

## Change an Array Element

To change the value of a specific element, refer to the index number.

cars[1] **=** Thar

## Array Length

To find out how many elements an array has, use the length property.

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

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

// Outputs 4

Loop through an Array (traditional for loop)

String[] cars = {"Rolls Royce", "BMW", "Benz", "Vintage"};

**for** (**int** i = 0; i < cars.length; i++) {

System.***out***.println(cars[i]);

}

Loop through an Array (enhanced for-each loop)

String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

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

// Outputs 4

Multi Dimensional Array

A multidimensional array is an array of arrays.

Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.

To create a two-dimensional array, add each array within its own set of **curly braces**:

**int**[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };

**Java Methods**

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Why use methods? To reuse code: define the code once and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **().**

**public** **class** Syntax {

**static** **void** myMethod() {

}

}

myMethod() is the name of the method

static means that the method belongs to the Main class and not an object of the Main class.

void means that this method does not have a return value.

**public** **class** Syntax {

**static** **void** myMethod() {

System.***out***.println("I just got executed!");

}

**public** **static** **void** main(String[] args) {

*myMethod*();

}

}

// Outputs "I just got executed!"

A method can also be called multiple times.

**public** **class** Syntax {

**static** **void** myMethod() {

System.***out***.println("I just got executed!");

}

**public** **static** **void** main(String[] args) {

*myMethod*();

*myMethod*();

*myMethod*();

*myMethod*();

*myMethod*();

}

}

//Outputs "I just got executed!"

//Outputs "I just got executed!"

//Outputs "I just got executed!"

//Outputs "I just got executed!"

//Outputs "I just got executed!"

Method Single Parameters

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

**public** **class** Syntax {

**static** **void** myMethod(String fname) {

System.***out***.println(fname + " Singh");

}

**public** **static** **void** main(String[] args) {

*myMethod*("Rathan");

*myMethod*("Jack");

*myMethod*("Aquila");

}

}

// Rathan Singh

// Jack Singh

// Aquila Singh

Method with Multiple Parameter

Below method with Integer & String as parameter

**public** **class** Syntax {

**static** **void** myMethod(String fname, **int** age) {

System.***out***.println(fname + " Singh" + " is " + age + " Years old");

}

**public** **static** **void** main(String[] args) {

*myMethod*("Rathan", 24);

*myMethod*("Jack", 22);

*myMethod*("Aquila", 25);

}

}

// Rathan Singh is 24 years old

// Jack Singh is 22 years old

// Aqila Singh is 25 years old

Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method.

**public** **class** Syntax {

**static** **int** myMethod(**int** x) {

**return** 5 + x;

}

**public** **static** **void** main(String[] args) {

System.***out***.println(*myMethod*(3));

}

}

// Outputs 8 (5 + 3)

This example returns the sum of a method’s two parameters.

**public** **class** Syntax {

**static** **int** myMethod(**int** x, **int** y) {

**return** x + y;

}

**public** **static** **void** main(String[] args) {

System.***out***.println(*myMethod*(5, 3));

}

}

// Outputs 8 (5 + 3)

You can also store the result in a variable (recommended, as it is easier to read and maintain).

**public** **class** Syntax {

**static** **int** myMethod(**int** x, **int** y) {

**return** x + y;

}

**public** **static** **void** main(String[] args) {

**int** z = *myMethod*(5, 3);

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

}

}

// Outputs 8 (5 + 3)

Method with If...Else

It is common to use if...else statements inside methods.

**public** **class** Syntax {

// Create a checkAge() method with an integer variable called age

**static** **void** checkAge(**int** age) {

// If age is less than 18, print "access denied"

**if** (age < 18) {

System.***out***.println("Access denied - You are not old enough!");

// If age is greater than, or equal to, 18, print "access granted"

} **else** {

System.***out***.println("Access granted - You are old enough!");

}

}

**public** **static** **void** main(String[] args) {

*checkAge*(20); // Call the checkAge method and pass along an age of 20

}

}

// Outputs "Access granted - You are old enough!"

Method Overloading

Method overloading in Java is a feature that allows you to define multiple methods in a class with the same name but different parameters. The parameters could differ in terms of the number of parameters, data types of parameters, or both. Java uses the number and types of parameters to determine which method to call when the overloaded method is invoked.

int myMethod(int x)

float myMethod(float x)

double myMethod(double x, double y)

Consider the following example, which has two methods that add numbers of different type.

**public** **class** Syntax {

**static** **int** plusMethodInt(**int** x, **int** y) {

**return** x + y;

}

**static** **double** plusMethodDouble(**double** x, **double** y) {

**return** x + y;

}

**public** **static** **void** main(String[] args) {

**int** myNum1 = *plusMethodInt*(8, 5);

**double** myNum2 = *plusMethodDouble*(4.3, 6.26);

System.***out***.println("int: " + myNum1);

System.***out***.println("double: " + myNum2);

}

}

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the plusMethod method to work for both int and double.

**public** **class** Syntax {

**static** **int** plusMethod(**int** x, **int** y) {

**return** x + y;

}

**static** **double** plusMethod(**double** x, **double** y) {

**return** x + y;

}

**public** **static** **void** main(String[] args) {

**int** myNum1 = *plusMethod*(8, 5);

**double** myNum2 = *plusMethod*(4.3, 6.26);

System.***out***.println("int: " + myNum1);

System.***out***.println("double: " + myNum2);

}

}

Method Overloading with same parameter Type will not work throws error, Only the above method Overloading with different parameter type will work in Method overloading.

**public** **class** Syntax {

**static** **int** plusMethod(**int** x, **int** y) {

**return** x + y;

}

**static** **int** plusMethod(**int** x, **int** y) {

**return** x + y;

}

**public** **static** **void** main(String[] args) {

**int** myNum1 = *plusMethod*(8, 5);

**int** myNum2 = *plusMethod*(4, 6);

System.***out***.println("int: " + myNum1);

System.***out***.println("int: " + myNum2);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Scope**

In Java, scope refers to the region or context in which a variable, method, or other identifier is valid and can be used. Java has several types of scopes, including.

**Class Scope (Global Scope):**

Variables and methods declared within a class, but outside of any method, have class scope. These are accessible from any method within the class. Class scope variables are often called instance variables or fields. They are created when an object of the class is instantiated and destroyed when the object is no longer in use.

**Method Scope:**

Variables declared within a method are said to have method scope. They are only accessible within the method where they are defined and are destroyed when the method exits.

**public** **class** Syntax {

**int** value = 1000; // Class scope variable

**static** **int** myMethod(**int** x, **int** y) {

**return** x+y; //Method scope Variable

}

}

**Block Scope:**

Variables declared within a code block (within curly braces) have block scope. They are only accessible within that block. Block scope is commonly found in conditional statements (if, while, for) and in code blocks defined within methods.

**int** Age = 29;

**if** (Age < 18) {

System.***out***.println("Not Old Enough to Vote.");

} **else** {

System.***out***.println("Old Enough Vote.");

}

// Outputs "Old Enough Vote."

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Recursion**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java OOP**

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

* OOP is faster and easier to execute.
* OOP provides a clear structure for the programs.
* OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug.
* OOP makes it possible to create full reusable applications with less code and shorter development time.

**Tip:** The "Don't Repeat Yourself" (DRY) principle is about reducing the repetition of code. You should extract out the codes that are common for the application, and place them at a single place and reuse them instead of repeating it.

**Java - What are Classes and Objects?**

Classes and objects are the two main aspects of object-oriented programming.

A blue and white rectangular box with text

Description automatically generated

So, a class is a template for objects, and an object is an instance of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

**Java Classes/Objects**

Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a volvo is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

A Class is like an object constructor, or a "blueprint" for creating objects.

**Create a Class**

To create a class, use the keyword class.

Create a class named "Syntax" with a variable x:

**public** **class** Syntax {

String name = "Rathan Rajiv Singh B";

}

Remember that a class should always start with an uppercase first letter, and that the name of the java file should match the class name.

**Create an Object**

In Java, an object is created from a class. We have already created the class named Syntax, so now we can use this to create objects.

To create an object of Syntax, specify the class name, followed by the object name, and use the keyword new:

**public** **class** Syntax {

**int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj = **new** Syntax();

System.***out***.println(myObj.x);

}

}

**Multiple Objects**

You can create multiple objects of one class.

**public** **class** Syntax {

**int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj1 = **new** Syntax();

Syntax myObj2 = **new** Syntax();

System.***out***.println(myObj1.x);

System.***out***.println(myObj2.x);

}

}

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other.

**public** **class** Syntax {

**int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj1 = **new** Syntax(); // Object 1

Syntax myObj2 = **new** Syntax(); // Object 2

myObj2.x = 25; // Changing the X Value in Object 2

System.***out***.println(myObj1.x); // Outputs 5

System.***out***.println(myObj2.x); // Outputs 25

}

}

**Multiple Class**

Only 1 public class can be provided in a file.

**public** **class** Syntax {

**int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj1 = **new** Syntax();

Syntax myObj2 = **new** Syntax();

System.***out***.println(myObj1.x);

System.***out***.println(myObj2.x);

}

}

**class** Second {

//int y = 10;

**public** **static** **void** main(String[] args) {

Syntax myObj = **new** Syntax();

System.***out***.println(myObj.x);

}}

**Java Class Attributes**

In the below Example I have given the variable name “**x**” & “**y**” under the class, it is actually called as the Attribute of the class.

Create a class called "Main" with two attributes: x and y:

**public** **class** Syntax {

**int** x = 5;

**int** y = 10;

}

**Accessing Attributes**

You can access attributes by creating an object of the class, and by using the dot syntax (.):

The following example will create an object of the Syntax class, with the name myObj. We use the x attribute on the object to print its value:

**public** **class** Syntax {

**int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj = **new** Syntax();

System.***out***.println(myObj.x);

}

}

**Modify Attributes**

You can also modify attribute values:

**public** **class** Syntax {

**int** x;

**public** **static** **void** main(String[] args) {

Syntax myObj = **new** Syntax();

myObj.x = 10; // X is now 10

System.***out***.println(myObj.x);

}

}

You can also Override attribute values:

**public** **class** Syntax {

**int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj = **new** Syntax();

myObj.x = 10; // X is now 10

System.***out***.println(myObj.x);

}

}

If you don't want the ability to override existing values, declare the attribute as final:

**public** **class** Syntax {

**final** **int** x = 5;

**public** **static** **void** main(String[] args) {

Syntax myObj = **new** Syntax();

myObj.x = 10; // The final field Syntax.x cannot be assigned

System.***out***.println(myObj.x);

}

}

**Multiple Attributes**

You can specify as many attributes as you want.

**public** **class** Main {

String fname = "John";

String lname = "Doe";

**int** age = 24;

**public** **static** **void** main(String[] args) {

Main myObj = **new** Main();

System.***out***.println("Name: " + myObj.fname + " " + myObj.lname);

System.***out***.println("Age: " + myObj.age);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Class Methods**

myMethod() prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**.**

Inside main, call myMethod()

**public** **class** Main {

**static** **void** myMethod() {

System.***out***.println("Hello World!");

}

**public** **static** **void** main(String[] args) {

*myMethod*();

}

}

// Outputs "Hello World!"

**Static vs. Public**

You will often see Java programs that have either static or public attributes and methods.

In the example above, we created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:

**public** **class** Main {

// Static method

**static** **void** myStaticMethod() {

System.***out***.println("Static methods can be called without creating objects");

}

// Public method

**public** **void** myPublicMethod() {

System.***out***.println("Public methods must be called by creating objects");

}

// Main method

**public** **static** **void** main(String[] args) {

*myStaticMethod*(); // Call the static method

// myPublicMethod(); This would compile an error

Main myObj = **new** Main(); // Create an object of Main

myObj.myPublicMethod(); // Call the public method on the object

}

}

**Example: -**

// Create a Main class

**public** **class** Main {

// Create a fullThrottle() method

**public** **void** fullThrottle() {

System.***out***.println("The car is going as fast as it can!");

}

// Create a speed() method and add a parameter

**public** **void** speed(**int** maxSpeed) {

System.***out***.println("Max speed is: " + maxSpeed);

}

// Inside main, call the methods on the myCar object

**public** **static** **void** main(String[] args) {

Main myCar = **new** Main(); // Create a myCar object

myCar.fullThrottle(); // Call the fullThrottle() method

myCar.speed(200); // Call the speed() method

}

}

// The car is going as fast as it can!

// Max speed is: 200

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java Constructor**

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

// Create a Main class

**public** **class** Main {

**int** x; // Create a class attribute

**// Create a class constructor for the Main class**

**public** Main() {

x = 5; // Set the initial value for the class attribute x

}

**public** **static** **void** main(String[] args) {

Main myObj = **new** Main(); // Create an object of class Main (This will call the constructor)

System.***out***.println(myObj.x); // Print the value of x

}

}

// Outputs 5

Note that the constructor’s name must **match the class name**, and it cannot have a **return type** (like void).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

**public** **class** Main {

String CarBrand;

String CarModel;

**int** ManufacturingYear;

// Constructor with Parameter

**public** Main (String Brand, String Model, **int** Year) {

CarBrand = Brand;

CarModel = Model;

ManufacturingYear = Year;

}

**public** **static** **void** main(String[] args) {

Main myCar = **new** Main("RollsRoyce", "Ghost", 2022);

System.***out***.println("My Car name is " + myCar.CarBrand + " " + myCar.CarModel + " which is built on " + myCar.ManufacturingYear );

}

}

**Modifiers**

**public** **class** Main

The public keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

* **Access Modifiers** - controls the access level.
* **Non-Access Modifiers** - do not control access level but provides other functionality.

**Access Modifiers**

For **classes**, you can use either public or default.

Public - The class is accessible by any other class.

Default - The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

For **attributes, methods and constructors**, you can use the one of the following.

Public - The class is accessible by any other class.

Default - The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

Private - The code is only accessible within the declared class.

Protected - The code is accessible in the same package and **subclasses**.

**Non-Access Modifiers**

For **classes**, you can use either final or abstract.

Final - The class cannot be inherited by other classes.

Abstract - The class cannot be used to create objects (To access an abstract class, it must be inherited from another class).

For **attributes and methods**, you can use the one of the following.

Final - Attributes and methods cannot be overridden/modified.

Static - Attributes and methods belongs to the class, rather than an object.

Abstract - Can only be used in an abstract class and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from)

Transient - Attributes and methods are skipped when serializing the object containing them.

Synchronized - Methods can only be accessed by one thread at a time.

Volatile - The value of an attribute is not cached thread-locally and is always read from the "main memory".

**Final**

If you don't want the ability to override existing attribute values, declare attributes as final:

**public** **class** Main {

**final** **int** x = 10;

**final** **double** PI = 3.14;

**public** **static** **void** main(String[] args) {

Main myObj = **new** Main();

myObj.x = 50; // will generate an error: cannot assign a value to a final variable

myObj.PI = 25; // will generate an error: cannot assign a value to a final variable

System.***out***.println(myObj.x);

}

}

**Static**

A static method means that it can be accessed without creating an object of the class, unlike public:

**public** **class** Main {

// Static method

**static** **void** myStaticMethod() {

System.***out***.println("Static methods can be called without creating objects");

}

// Public method

**public** **void** myPublicMethod() {

System.***out***.println("Public methods must be called by creating objects");

}

// Main method

**public** **static** **void** main(String[ ] args) {

*myStaticMethod*(); // Call the static method

// myPublicMethod(); This would output an error

Main myObj = **new** Main(); // Create an object of Main

myObj.myPublicMethod(); // Call the public method

}

}

**Abstract**

An abstract method belongs to an abstract class, and it does not have a body. The body is provided by the subclass.

**Java Encapsulation**

The meaning of Encapsulation is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* declare class variables/attributes as private.
* provide public **get** and **set** methods to access and update the value of a private variable.

**Get and Set**

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **get** and **set** methods.

The get method returns the variable value, and the set method sets the value.

Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

**public** **class** Person {

**private** String name = "Rathan Rajiv Singh"; // private = restricted access

// Getter

**public** String getName() {

**return** name;

}

// Setter

**public** **void** setName(String newName) {

**this**.name = newName;

}

#### }

#### **Example explained: -**

The get method returns the value of the variable name.

The set method takes a parameter (newName) and assigns it to the name variable. The this keyword is used to refer to the current object.

However, as the name variable is declared as private, we **cannot** access it from outside this class:

However, as we try to access a private variable, we get an error:

// Calling the above private attribute to the New class Main

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Person myObj = **new** Person();

myObj.setName("John"); // Set the value of the name variable to "John"

System.***out***.println(myObj.getName()); // Get the Name & print in the console

}

}

// Outputs "John"

If the variable was declared as public, we would expect the following output:

// John

Instead, we use the getName() and setName() methods to access and update the variable.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Person myObj = **new** Person();

myObj.setName("John"); // Set the value of the name variable to "John"

System.***out***.println(myObj.getName()); // Get the Name & print in the console

}

}

// Outputs "John"

#### **Why Encapsulation?**

* Better control of class attributes and methods
* Class attributes can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method)
* Flexible: the programmer can change one part of the code without affecting other parts
* Increased security of data

**Java Scanner**

The Scanner is mostly used to receive user input and parse them into primitive data types such as int, double or default String. It's a utility class to parse data using regular expressions by generating tokens.

**import** java.util.Scanner;

**class** Scannerclass {

**public** **static** **void** main(String[] args) {

Scanner myObj = **new** Scanner(System.***in***);

System.***out***.println("Enter username");

String userName = myObj.nextLine();

System.***out***.println("Username is: " + userName);

}}

A screenshot of a computer

Description automatically generated

**Import a Package**

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (\*). The following example will import ALL the classes in the java.util package.

import java.util.\*;

**Java Inheritance: -**

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

// Super Class (Parent)

**class** Vehicle {

**protected** String brand = "Ford"; //The code is accessible in the same package and subclasses.

**public** **void** honk() {

System.***out***.println("Tuut, tuut!");

}

}

// Sub class (Child)

**class** Car **extends** Vehicle {

**private** String modelName = "Mustang";

**public** **static** **void** main(String[] args) {

Car myFastCar = **new** Car();

myFastCar.honk();

System.***out***.println(myFastCar.brand + " " + myFastCar.modelName);

}

}

**The final Keyword**

If you don't want other classes to inherit from a class, use the final keyword:

**final** **class** Vehicle {

**protected** String brand = "Ford"; //The code is accessible in the same package and subclasses.

**public** **void** honk() {

System.***out***.println("Tuut, tuut!");

}

}

**class** Car **extends** Vehicle {

**private** String modelName = "Mustang";

**public** **static** **void** main(String[] args) {

Car myFastCar = **new** Car();

myFastCar.honk();

System.***out***.println(myFastCar.brand + " " + myFastCar.modelName);

}

}

**Java Polymorphism: -**

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; [**Inheritance**](https://www.w3schools.com/java/java_inheritance.asp) lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

**class** Animal {

**public** **void** animalSound() {

System.***out***.println("The animal makes a sound");

}

}

**class** Pig **extends** Animal {

**public** **void** animalSound() {

System.***out***.println("The pig says: wee wee");

}

}

**class** Dog **extends** Animal {

**public** **void** animalSound() {

System.***out***.println("The dog says: bow wow");

}

}

Now we can create Pig and Dog objects and call the animalSound() method on both of them

**class** Animal {

**public** **void** animalSound() {

System.***out***.println("The animal makes a sound");

}

}

**class** Pig **extends** Animal {

**public** **void** animalSound() {

System.***out***.println("The pig says: wee wee");

}

}

**class** Dog **extends** Animal {

**public** **void** animalSound() {

System.***out***.println("The dog says: bow wow");

}

}

**class** Sound {

**public** **static** **void** main(String[] args) {

Animal myAnimal = **new** Animal(); // Create a Animal object

Animal myPig = **new** Pig(); // Create a Pig object

Animal myDog = **new** Dog(); // Create a Dog object

myAnimal.animalSound();

myPig.animalSound();

myDog.animalSound();

}

}

#### Why And When To Use "Inheritance" and "Polymorphism"?

It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.

**Java Inner class: -**

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

// Outer class

**class** OuterClass {

**int** x = 10;

// Inner class

**class** InnerClass {

**int** y = 5;

}

}

**public** **class** ClassMain {

**public** **static** **void** main(String[] args) {

OuterClass myOuter = **new** OuterClass();

OuterClass.InnerClass myInner = myOuter.**new** InnerClass();

System.***out***.println(myInner.y + myOuter.x);

}

}

// Outputs 15 (5 + 10)

**Private Inner class**

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

// Outer class

**class** OuterClass {

**int** x = 10;

// Inner class

**private** **class** InnerClass {

**int** y = 5;

}

}

**public** **class** ClassMain {

**public** **static** **void** main(String[] args) {

OuterClass myOuter = **new** OuterClass();

OuterClass.InnerClass myInner = myOuter.**new** InnerClass(); // This will throw error because it Innerclass in private

System.***out***.println(myInner.y + myOuter.x);

}

}

// Outputs 15 (5 + 10)

**Static Inner class**

An inner class can also be static, which means that you can access it without creating an object of the outer class.

**class** OuterClass {

**int** x = 10;

**static** **class** InnerClass {

**int** y = 5;

}

}

**public** **class** Staticclass {

**public** **static** **void** main(String[] args) {

OuterClass.InnerClass myInner = **new** OuterClass.InnerClass();

System.***out***.println(myInner.y);

}

}

// Outputs 5

**Note:** just like static attributes and methods, a static inner class does not have access to members of the outer class.

**Access Outer Class From Inner Class**

One advantage of inner classes, is that they can access attributes and methods of the outer class.

// Outer class

**class** OuterClass {

**int** x = 10;

// Inner class

**class** InnerClass {

**int** y = 5;

}

}

**public** **class** ClassMain {

**public** **static** **void** main(String[] args) {

OuterClass myOuter = **new** OuterClass();

OuterClass.InnerClass myInner = myOuter.**new** InnerClass(); // This will throw error because it Innerclass in private

System.***out***.println(myInner.y + myOuter.x);

}

}

// Outputs 15 (5 + 10)

**Java Abstraction: -**

**Abstract Classes and Methods**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either abstract classes or [interfaces](https://www.w3schools.com/java/java_interface.asp)

The abstract keyword is a non-access modifier, used for classes and methods:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

* 1. We can’t create Object for the Abstract class in java.
  2. Abstract class can also have normal class Aswell not only abstract class.
  3. Extended abstract class should have the abstract method created in the abstract class.
  4. We can override the Attributes or Methods in abstract class in the concrete class.

// abstract class

**abstract** **class** SuperCar

{

// Abstract methods

**abstract** **void** drive();

**abstract** **void** Fly();

// Normal methods

**public** **void** sound() {

System.***out***.println("Vroom Vroom");

}

}

**class** RollsRoyce **extends** SuperCar // Concrete class

{

**public** **void** drive() {

System.***out***.println("Driving Rolls Royce");

}

**public** **void** Fly() {

System.***out***.println("Flying in Rolls Royce");

}

}

**public** **class** runClass

{

**public** **static** **void** main(String[] args) {

RollsRoyce Obj = **new** RollsRoyce();

Obj.drive();

Obj.Fly();

Obj.sound();

}

}

//Output

// Driving RollsRoyce

// Flying in RollsRoyce

**Java Interface: -**

Another way to achieve [abstraction](https://www.w3schools.com/java/java_abstract.asp) in Java, is with interfaces.

An interface is a completely "**abstract class**" that is used to group related methods with empty bodies.

// interface

**interface** InterfaceClass {

**public** **void** animalSound(); // interface method (does not have a body)

**public** **void** run(); // interface method (does not have a body)

}

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class.

**interface** InterfaceClass {

**public** **void** animalSound(); // interface method (does not have a body)

**public** **void** sleep(); // interface method (does not have a body)

}

**class** cat **implements** InterfaceClass {

**public** **void** animalSound() {

System.***out***.println("The pig says: meow");

}

**public** **void** sleep() {

System.***out***.println("Zzzz");

}

}

**class** lion **implements** InterfaceClass {

**public** **void** animalSound() {

System.***out***.println("The dog says: Roar");

}

**public** **void** sleep() {

System.***out***.println("Zzzz");

}

}

**class** Sound1 {

**public** **static** **void** main(String[] args) {

InterfaceClass mycat = **new** cat(); // Create a cat object

InterfaceClass mylion = **new** lion(); // Create a lion object

mycat.animalSound();

mylion.animalSound();

}

}

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

#### **Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

**Multiple Interface**

// Interface with Multiple Inheritance

**interface** FirstInterface {

**public** **void** myMethod(); // interface method

}

**interface** SecondInterface {

**public** **void** myOtherMethod(); // interface method

}

**class** DemoClass **implements** FirstInterface, SecondInterface {

**public** **void** myMethod() {

System.***out***.println("Some text..");

}

**public** **void** myOtherMethod() {

System.***out***.println("Some other text...");

}

}

**class** Main0 {

**public** **static** **void** main(String[] args) {

DemoClass myObj = **new** DemoClass();

myObj.myMethod();

myObj.myOtherMethod();

}

}

**JAVA ENUMS: -**

An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).

To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters: