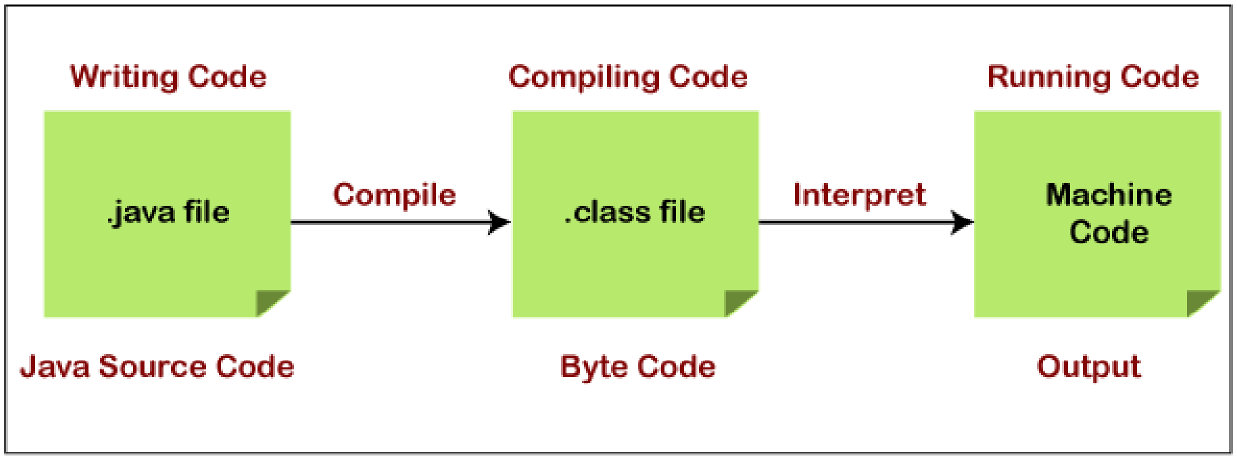
Chapter 1 : Java Introduction

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

How JAVA Works?

Java is compiled into the bytecode and then it is interpreted to machine code.
Java works by first compiling the source code into bytecode. Then, the bytecode can be compiled into machine code with the Java Virtual Machine (JVM). Java's bytecode can run on any device with the JVM which is why Java is known as a "write once, run anywhere" language.



Why to Learn java Programming?

Java is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Software Development Domain. I will list down some of the key advantages of learning Java Programming:

Object Oriented –

In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Platform Independent –

Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

Simple –

Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.

Secure –

With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architecture-neutral –

Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Portable –

Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

Robust –

Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

JAVA Installation

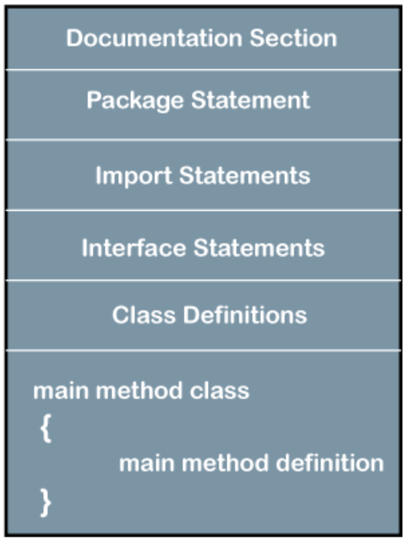
Go to Google and type “Install JDK” => Installs JAVA JDK
Go to Google and Type “Install Intellij Idea” => Installs JAVA IDE

JDK (JAVA Development kit) = collection of tools used for developing and running java programs

JRE (JAVA runtime environment) = Helps in executing programming developed in JAVA

Basic Structure of a Java Program

Java is an object-oriented programming, platform-independent, and secure programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications. So, before diving in depth, it is necessary to understand the basic structure of Java program in detail. In this section, we have discussed the basic structure of a Java program. At the end of this section, you will able to develop the Hello world Java program, easily.



Structure of Java Program

Let's see which elements are included in the structure of a Java program. A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviors

Documentation Section

The documentation section is an important section but optional for a Java program. It includes basic information about a Java program. The information includes the author's name, date of creation, version, program name, company name, and description of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use comments. The comments may be single-line, multi-line, and documentation comments.

- **Single-line Comment**: It starts with a pair of forwarding slash (//). For example:
  ```
  //First Java Program
  ```
- **Multi-line Comment**: It starts with a /* and ends with */. We write between these two symbols. For example:
  ```
  /*It is an example of
  multiline comment*/
  ```
- **Documentation Comment**: It starts with the delimiter (/**) and ends with */. For example:
  ```
  /**It is an example of documentation comment*/
  ```

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the package name in which the class is placed. Note that there can be only one package statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used. For all these classes package belongs to a single parent directory. We use the keyword package to declare the package name. For example:

- package javatpoint; //where javatpoint is the package name
- package com.javatpoint; //where com is the root directory and javatpoint is the subdirectory

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the import keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements. For example:

- import java.util.Scanner; //it imports the Scanner class only
- import java.util.*; //it imports all the class of the java.util package

Interface Section

It is an optional section. We can create an interface in this section if required. We use the interface keyword to create an interface. An interface is a slightly different from the class. It contains only constants and method declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the implements keyword. An interface can also be used with other interfaces by using the extends keyword. For example:

```
interface car
{
void start();
void stop();
}
```

Class Definition

In this section, we define the class. It is vital part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the class keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method. For example:

```
class Student //class definition
{
}
```

Class Variables and Constants

In this section, we define variables and constants that are to be used later in the program. In a Java program, the variables and constants are defined just after the class definition. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It defines the life of the variables. For example:

```
class Student //class definition
{
String sname;  //variable
int id;
double percentage;
}
```

Main Method Class

In this section, we define the main() method. It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the main() method:

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

For example:

```
public class Student //class definition
{
public static void main(String args[])
{
//statements
}
}
```

You can read more about the Java main() method here.

Methods and behavior

In this section, we define the functionality of the program by using the methods. The methods are the set of instructions that we want to perform. These instructions execute at runtime and perform the specified task. For example:

```
public class Demo //class definition
{
public static void main(String args[])
{
void display()
{
System.out.println("Welcome to javatpoint");
}
//statements
}
```

```
    }
```

Working of the "Hello World" program shown above :

1. **package com.company :**
   - Packages are used to group the related classes.
   - The "Package" keyword is used to create packages in Java.
   - Here, com.company is the name of our package.
2. **public class Main :**
   - In Java, every program must contain a class.
   - The filename and name of the class should be the same.
   - Here, we've created a class named "Main".
   - It is the entry point to the application.
3. **public static void main(String[]args){..} :**
   - This is the main() method of our Java program.
   - Every Java program must contain the main() method.
4. **System.out.println("Hello World"):**
   - The above code is used to display the output on the screen.
   - Anything passed inside the inverted commas is printed on the screen as plain text.

Naming Conventions

- For classes, we use Pascal Convention. The first and Subsequent characters from a word are capital letters (uppercase).
  Example: Main, MyScanner, MyEmployee
- For functions and variables, we use camelCaseConvention. Here the first character is lowercase, and the subsequent characters are uppercase like myScanner, myMarks

Data Types in Java

Variables

- A variable is a container that stores a value.
- This value can be changed during the execution of the program.
- Example: int number = 8; (Here, int is a data type, the number is the variable name, and 8 is the value it contains/stores).

Rules for declaring a variable name

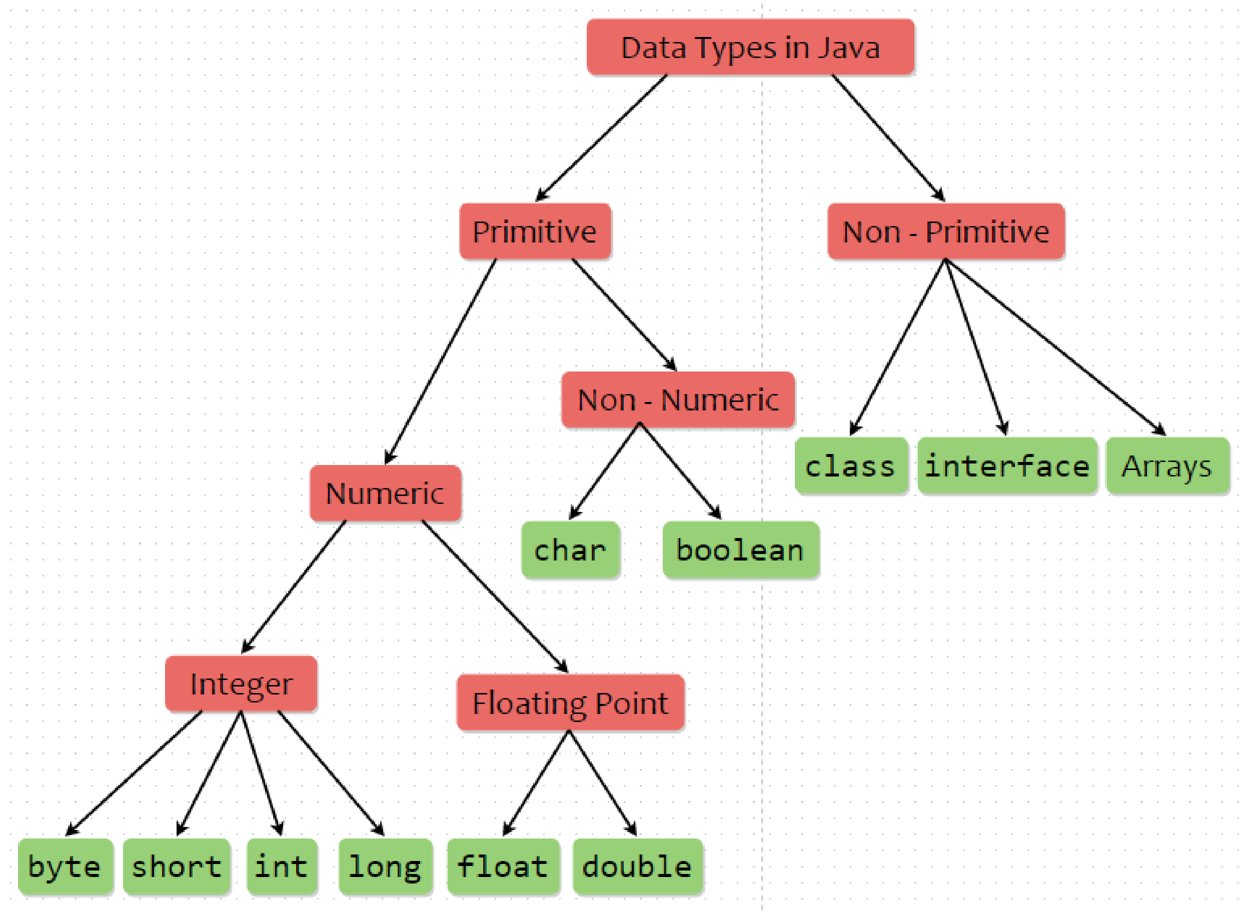
We can choose a name while declaring a Java variable if the following rules are followed:

- Must not begin with a digit. (E.g., 1arry is an invalid variable)
- Name is case sensitive. (Harry and harry are different)
- Should not be a keyword (like Void).

- White space is not allowed. (int Code With Harry is invalid)
- Can contain alphabets, $character, _character, and digits if the other conditions are met.

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.



Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type

- float data type
- double data type

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

Boolean one = false

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

```
int a = 100000, int b = -200000
```

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between - 9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

```
long a = 100000L, long b = -200000L
```

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example:

```
float f1 = 234.5f
```

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 12.3
```

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

```
char letterA = 'A'
```

Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system.

Literals

A constant value that can be assigned to the variable is called a literal.

- 101 – Integer literal
- 10.1f – float literal
- 10.1 – double literal (default type for decimals)
- ‘A’ – character literal
- true – Boolean literal
- “Sunil” – String literal

Example:

```
package com.company;
        public class JT_2_literals{
          public static void main(String[] args) {
            byte age = 21;
            int age2 = 56;
            short age3 = 87;
            long ageDino = 5666666666666L;
            char ch = 'A';
            float f1 = 5.6f;
            double d1 = 4.66;

            boolean a = true;
            System.out. println(age);
            String str = "Sunil";
            System.out.println(str);

        }
}
```

Output:

```
21
Sunil
```

Getting User Input in Java

Scanner class of java.util package is used to take input from the user's keyboard.The Scanner class has many methods for taking input from the user depending upon the type of input. To use any of the methods of the Scanner class, first, we need to create an object of the Scanner class as shown in the below example :

```
import java.util.Scanner;  // Importing  the Scanner class
Scanner scan = new Scanner(System.in);
//Creating an object named "scan" of the Scanner class.
```

Taking an integer input from the keyboard :

```
Scanner S = new Scanner(System.in);  //(Read from the keyboard)
int a = S.nextInt();  //(Method to read from the keyboard)
```

Example 1:

```
package com.company;
import  java.util.Scanner;

public class JT_3_TakingInput {
   public static void main(String[] args) {
      System.out.println("Taking Input From the User");
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter First Number ");
      int a = scan.nextInt();
      System.out.println("Enter Second Number ");
      int b = scan.nextInt();
      int sum = a + b;
      System.out.println("The Sum of these number is");
      System.out.println(sum);
   }
}
```

Output:

```
Taking Input From the User
Enter First Number
20
Enter Second Number
30
The Sum of these number is
50
```

Example 2:

```
package com.company;
import  java.util.Scanner;

public class JT_3_TakingInput {
   public static void main(String[] args) {
```

```
            System.out.println("Taking Input From the User");
            Scanner scan  = new Scanner(System.in);
            System.out.println("Enter First Number");
            float a = scan.nextFloat();
            System.out.println("Enter Second Number");
            float b = scan.nextFloat();
            float mul = a* b;
            System.out.println("The Multiple of these number is");
            System.out.println(mul);
        }
    }
```

Output:

```
    Taking Input From the User
    Enter First Number
    3.45
    Enter Second Number
    6.88
    The Multiple of these number is
    23.736
```

Example: 3

```
    package com.company;
    import  java.util.Scanner;

    public class JT_3_TakingInput {
        public static void main(String[] args) {

            System.out.println("Taking Input From the User");
            Scanner scan  = new Scanner(System.in);
            System.out.println("Enter First Number");
            boolean b1 = scan.hasNextFloat();
            System.out.println(b1);
        }
    }
```

Output:

```
    Taking Input From the User
    Enter First Number
    355.222
    true
```

Operators

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.

Example: a + b = c

- In the above example, 'a' and 'b' are operands on which the '+' operator is applied.

There are many types of operators in Java which are given below:

1. Arithmetic Operator
2. Assignment Operator
3. Comparison Operators
4. Logical Operator
5. Bitwise Operator

1. Arithmetic Operators:

- Arithmetic operators are used to perform mathematical operations such as addition, division, etc on expressions.
- Arithmetic operators cannot work with Booleans.
- % operator can work on floats and doubles.
- Let x=7 and y=2

Operator	Name	Description	Example
+	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

Example: 1

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 2;
    System.out.println(x % y);
  }
}
```

Output:

```
1
```

Example: 2

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    ++x;
    System.out.println(x);
  }
}
```

Output:

```
6
```

2. Assignment Operator

Assignment operators are used to assign values to variables.

In the example below, we use the assignment operator (=) to assign the value 10 to a variable called x:

Operator	Example	Same As
=	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

Example: 1

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    System.out.println(x);
  }
}
```

Output:

```
5
```

Example: 2

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x += 3;
    System.out.println(x);
  }
}
```

Output:

```
8
```

Example: 3

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x -= 3;
    System.out.println(x);
  }
}
```

Output:

```
2
```

Example: 4

```
public class Main {
  public static void main(String[] args) {
    double x = 5;
    x /= 3;
    System.out.println(x);
  }
}
```

Output:

```
1.6666666666666667
```

Example: 5
```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x %= 3;
    System.out.println(x);
  }
}
```
Output:
```
        2
```

Example: 6
```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x &= 3;
    System.out.println(x);
  }
}
```
Output:
```
        1
```

Example: 7
```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x |= 3;
    System.out.println(x);
  }
}
```
Output:
```
        7
```

Example: 8
```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x ^= 3;
    System.out.println(x);
  }
}
```
Output:
```
        6
```

Example: 9
```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x >>= 3;
    System.out.println(x);
  }
}
```
Output:
```
        0
```

Example: 10
```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    x <<= 3;
    System.out.println(x);
  }
}
```
Output:
```
        40
```

3. Comparison Operators:
- As the name suggests, these operators are used to compare two operands.
- Let x=7 and y=2

Operator	Name	Example
==	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

Example:

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 3;
    System.out.println(x == y); // returns false because 5 is not equal to 3
  }
}
```

Output:

```
False
```

4. Logical Operators

- These operators determine the logic in an expression containing two or more values or variables.
- Let x = 8 and y =2

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Example: 1

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    System.out.println(x > 3 && x < 10);
// returns true because 5 is greater than
3 AND 5 is less than 10
  }
}
```

Output:

```
True
```

Example: 2

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    System.out.println(x > 3 || x < 4);
// returns true because one of the
conditions are true (5 is greater than 3,
but 5 is not less than 4)
```

```
  }
}
```

Output:

```
True
```

Example: 3

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    System.out.println(!(x > 3 && x < 10)); // returns false because ! (not) is used to reverse
the result
  }
}
```

Output:

```
False
```

5. Bitwise Operators

Operator	Description	Example
& (bitwise and)	1&1 =1, 0&1=0,1&0=0,1&1=1, 0&0 =0	A & B) = (100 & 011) = 000
| (bitwise or)	1&0 =1, 0&1=1,1&1=1, 0&0=0	(A | B) = (100 | 011) = 111
^ (bitwise XOR)	1&0 =1, 0&1=1,1&1=0, 0&0=0	(A ^ B) = (100 ^ 011) = 111
<< (left shift)	This operator moves the value left by the number of bits specified.	
>> (right shift)	This operator moves the value left by the number of bits specified.	

Precedence & associativity in java

Precedence and associativity are two features of Java operators. When there are two or more operators in an expression, the operator with the highest priority will be executed first.

For example, consider the equation, 1 + 2 * 5. Here, the multiplication (*) operator is executed first, followed by addition. Because multiplication operator takes precedence over the addition operator.

Alternatively, when an operand is shared by two operators (2 in the example above is shared by + and *), the higher priority operator processes the shared operand. You should have grasped the significance of precedence or priority in the execution of operators from the preceding example.

However, the situation may not always be as obvious as in the example above. What if the precedence of all operators in an expression is the same? In that instance, the second quality associated with an operator, associativity, comes into existence.

Associativity specifies the order in which operators are executed, which can be left to right or right to left. For example, in the phrase a = b = c = 8, the assignment operator is used from right to left. It means that the value 8 is assigned to c, then c is assigned to b, and at last b is assigned to a. This phrase can be parenthesized as (a = (b = (c = 8)).

The priority of a Java operator can be modified by putting parenthesis around the lower order priority operator, but not the associativity. In the equation (1 + 2) * 3, for example, the addition will be performed first since parentheses take precedence over the multiplication operator.

Operator Precedence in Java (Highest to Lowest)

Category	**Operators**	**Associativity**
Postfix	++ - -	Left to right
Unary	+ - ! ~ ++ - -	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left

Java Operator Associativity
Operators with the same precedence follow the operator group's operator associativity. Operators in Java can be left-associative, right-associative, or have no associativity at all. Left-associative operators are assessed from left to right, right-associative operators are reviewed from right to left, and operators with no associativity are evaluated in any order.

Operator Precedence Vs Operator Associativity
The operator's precedence refers to the order in which operators are evaluated within an expression whereas associativity refers to the order in which the consecutive operators within the same group are carried out.
Precedence rules specify the priority (which operators will be evaluated first) of operators.

Data Type of Expressions

Resulting data type after arithmetic operation

- Result = byte + short -> integer
- Result = short + integer -> integer
- Result = long + float -> float
- Result = integer + float -> float
- Result = character + integer -> integer
- Result = character + short -> integer
- Result = long + double -> double
- Result = float + double -> double

Example:

```
package com.company;

    public class JT_6_resulting_datatype {
            public static void main(String[] args) {
                    int y = 6;
                    short z = 8;
                    byte x = 5;
                    int a = y + z ;

                    int b = 45 + 4;
                    float c = 6.54f + 6;
                    System.out.println(a);
                    System.out.println(b);
                    System.out.println(c);
            }
    }
```

Output:

```
14
49
12.54
```

Increment/Decrement Operators

Java provides two operators namely ++ and --, to increment and decrement values by 1 respectively.

There are two variants of these operators −

Pre-increment/decrement − This form, increments/decrements the value first, and then performs the specified operation.

Example

In the following example, the initial value of the variable i is 5. We are printing the incremented value of it using the pre increment operator.

Since we are using the pre increment operator, the value of i is incremented then printed.

```
public class ForLoopExample {
  public static void main(String args[]) {
    int i = 5;
    System.out.println(++i);
    System.out.println(i);
  }
}
```

Output:

```
6
```

Post-increment/decrement – This form, performs the specified operation first, and then increments/decrements the value.

Example

In the following example, the initial value of the variable i is 5. We are printing the incremented value of it using the post increment operator and, we are printing the i value again.

Since we are using the post increment operator, the value of i is printed and then incremented.

```
public class ForLoopExample {
  public static void main(String args[]) {
    int i = 5;
    System.out.println(i++);
    System.out.println(i);
  }
}
```

Output

```
5
6
```

Example

```
public class ForLoopExample {
  public static void main(String args[]) {
    int i = 5;
    System.out.println(i--);
    System.out.println(i);
    int j =5;
```

```
            System.out.println(--j);
          }
        }
```
Output

```
        5
        4
        4
```

Introduction to strings

Strings in Java are Objects that are backed internally by a char array. Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

Syntax:

```
        <String_Type> <string_variable> = "<sequence_of_string>";
```
Example:

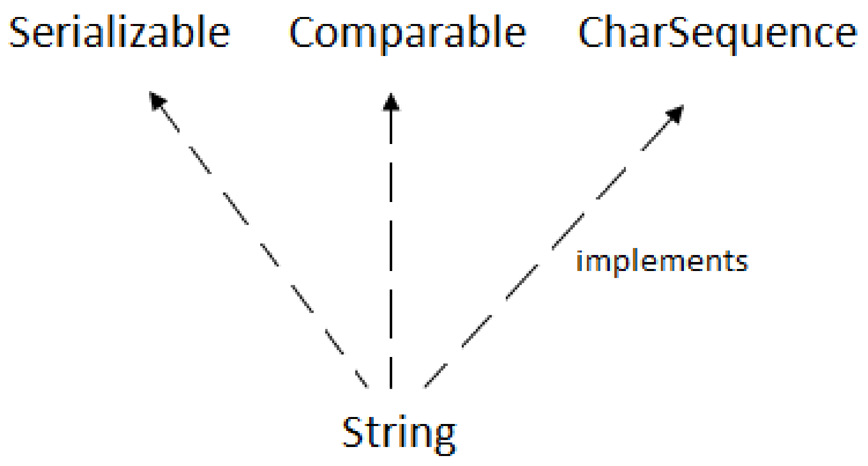
```
        String str = " javatpoint ";
```
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
        char[] ch={'j','a','v','a','t','p','o','i','n','t'};
        String s=new String(ch);
```
is same as:

```
        String s="javatpoint";
```

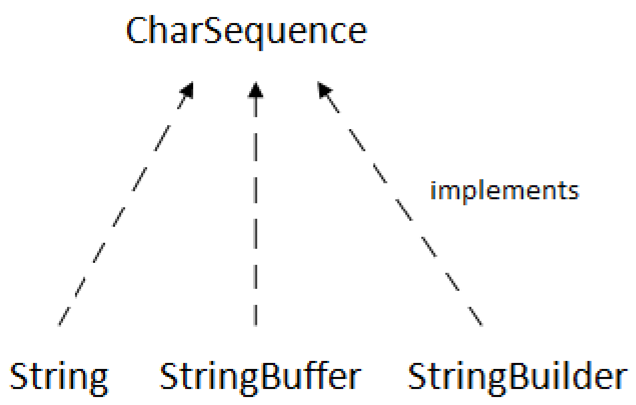
Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

By string literal

By new keyword

1) String Literal

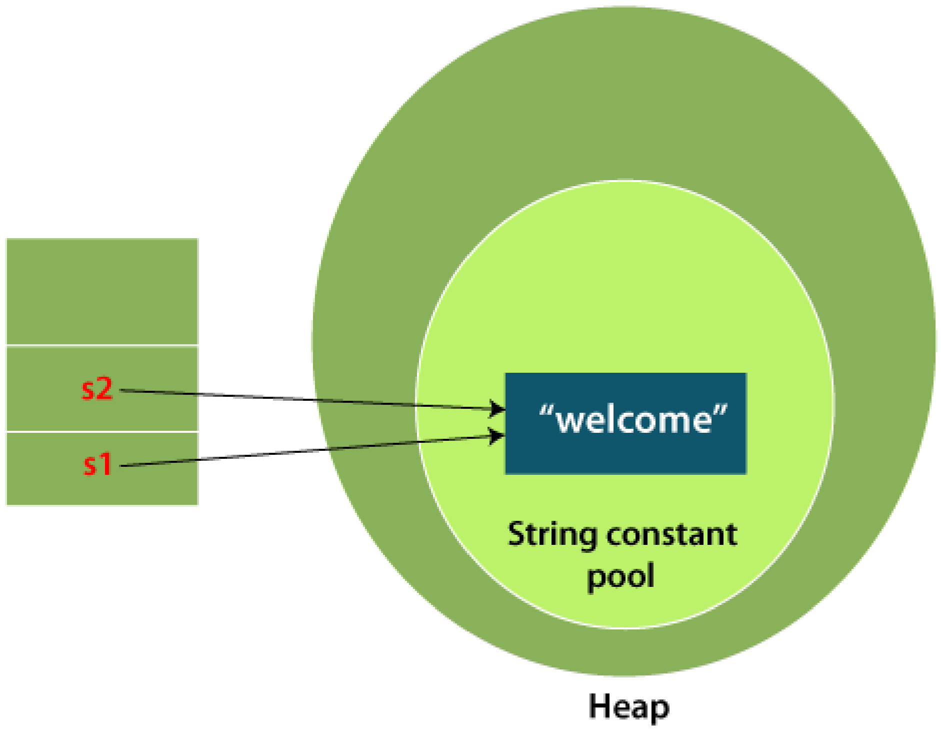
Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

String s=new String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Example

```
public class StringExample{
        public static void main(String args[]){
                String s1="java";//creating string by Java string literal
                char ch[]={'s','t','r','i','n','g','s'};
                String s2=new String(ch);//converting char array to string
                String s3=new String("example");//creating Java string by new
                keyword
                System.out.println(s1);
                System.out.println(s2);
                System.out.println(s3);
            }
        }
```

Output:

```
        java
        strings
        example
```

The above code, converts a char array into a String object. And displays the String objects s1, s2, and s3 on console using println() method.

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	It returns char value for the particular index
2	int length()	It returns string length
3	static String format(String format, Object... args)	It returns a formatted string.
4	static String format(Locale l, String format, Object... args)	It returns formatted string with given locale.
5	String substring(int beginIndex)	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	It returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	It returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	It returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	It returns a joined string.
10	boolean equals(Object another)	It checks the equality of string with the given object.

11	boolean isEmpty()	It checks if string is empty.
12	String concat(String str)	It concatenates the specified string.
13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	It compares another string. It doesn't check case.
16	String[] split(String regex)	It returns a split string matching regex.
17	String[] split(String regex, int limit)	It returns a split string matching regex and limit.
18	String intern()	It returns an interned string.
19	int indexOf(int ch)	It returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.
21	int indexOf(String substring)	It returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.
23	String toLowerCase()	It returns a string in lowercase.
24	String toLowerCase(Locale l)	It returns a string in lowercase using specified locale.
25	String toUpperCase()	It returns a string in uppercase.
26	String toUpperCase(Locale l)	It returns a string in uppercase using specified locale.
27	String trim()	It removes beginning and ending spaces of this string.
28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

Control Statements

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.
Java provides three types of control flow statements.

1. **Decision Making statements**
   - if statements
   - switch statement
2. **Loop statements**
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. **Jump statements**
   - break statement
   - continue statement

Decision-Making statements

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.
1) If Statement:
In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:
It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {
    statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the if statement in the java code.

Student.java

```
public class Student {
        public static void main(String[] args) {
                int x = 10;
                int y = 12;
                if(x+y > 20) {
                    System.out.println("x + y is greater than 20");
                }
        }
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
  statement 1; //executes when condition is true
}
else{
  statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
        public static void main(String[] args) {
                int x = 10;
                int y = 12;
                if(x+y < 10) {
                System.out.println("x + y is less than      10");
                }   else {
                System.out.println("x + y is greater than 20");
                }
            }
        }
```
Output:

```
        x + y is greater than 20
```

3) if-else-if ladder

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax

```
        if(condition 1) {
          statement 1; //executes when condition 1 is true
        }
        else if(condition 2) {
          statement 2; //executes when condition 2 is true
        }
        else {
          statement 2; //executes when all the conditions are false
        }
```

Consider the following example.

Student.java

```
        public class Student {
            public static void main(String[] args) {
                String city = "Delhi";
                if(city == "Meerut") {
                  System.out.println("city is meerut");
                }else if (city == "Noida") {
                   System.out.println("city is noida");
                }else if(city == "Agra") {
                   System.out.println("city is agra");
                }else {
                  System.out.println(city);
                }
            }
        }
```

Output: **Delhi**

4. Nested if-statement

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax

```
if(condition 1) {
        statement 1; //executes when condition 1 is true
    if(condition 2) {
      statement 2; //executes when condition 2 is true
    }
    else{
      statement 2; //executes when condition 2 is false
    }
}
```

Example:

```
public class Student {
    public static void main(String[] args) {
      String address = "Delhi, India";

      if(address.endsWith("India")) {
        if(address.contains("Meerut")) {
          System.out.println("Your city is Meerut");
        }else if(address.contains("Noida")) {
          System.out.println("Your city is Noida");
        }else {
          System.out.println(address.split(",")[0]);
        }
      }else {
        System.out.println("You are not living in India");
      }
    }
  }
}
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied.
- It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

Syntax

```
switch (expression){
   case value1:
    statement1;
    break;
   .
   .
   .
   case valueN:
    statementN;
    break;
   default:
    default statement;
}
```

Example:

Student.java

```
public class Student implements Cloneable {
   public static void main(String[] args) {
      int num = 2;
      switch (num){
         case 0:
            System.out.println("number is 0");
            break;
         case 1:
            System.out.println("number is 1");
            break;
         default:
            System.out.println(num);
      }
   }
}
```
Output:

```
      2
```
While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

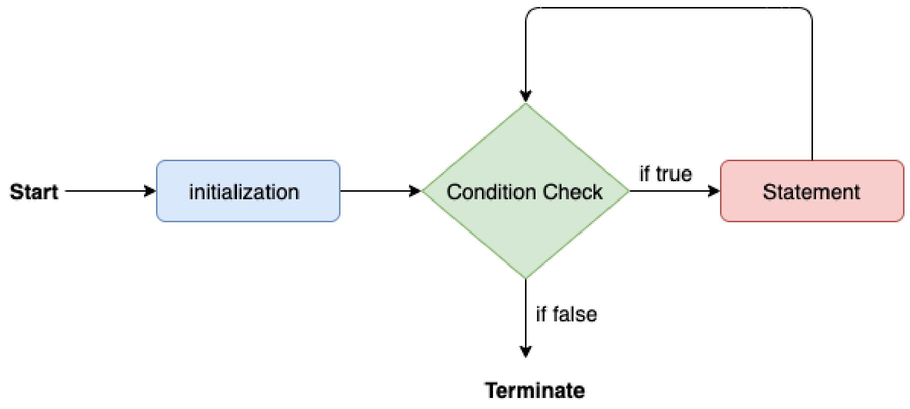
- for loop
- while loop
- do-while loop

Let's understand the loop statements one by one.

1. for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```
The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java
```
public class Calculattion {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int sum = 0;
        for(int j = 1; j<=10; j++) {
        sum = sum + j;
        }
        System.out.println("The sum of first 10 natural numbers is " + sum);
    }
}
```
Output:
```
The sum of first 10 natural numbers is 55
```

2. for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.
```
for(data_type var : array_name/collection_name){
  //statements
}
```
Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String[] names = {"Java","C","C++","Python","JavaScript"};
        System.out.println("Printing the content of the array names:\n");
        for(String name:names) {
        System.out.println(name);
        }
    }
}
```
Output:
```
Printing the content of the array names:
```

Java
C
C++
Python
JavaScript

3. while loop

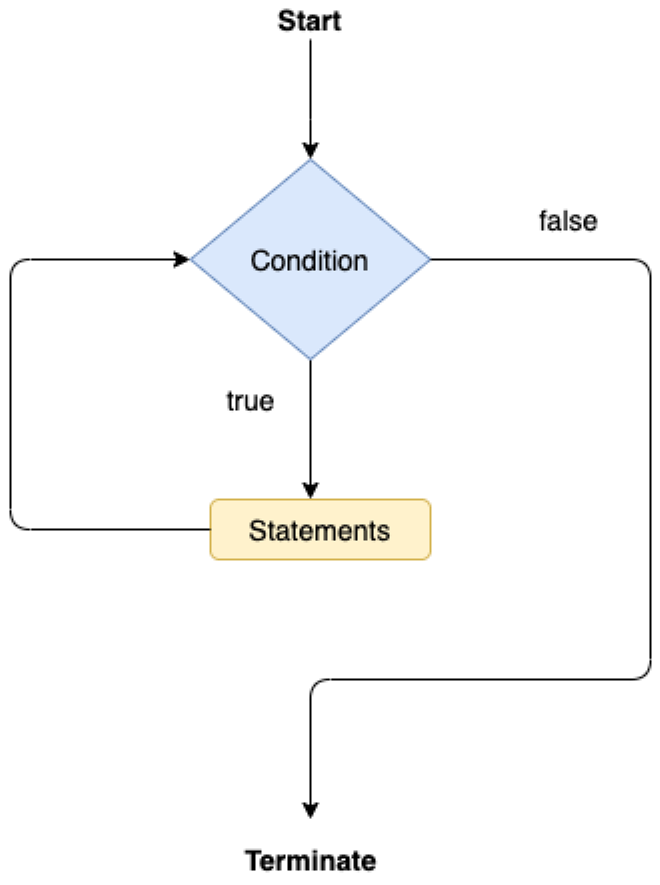
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

```
while(condition){
//looping statements
}
```

The flow chart for the while loop is given in the following image.

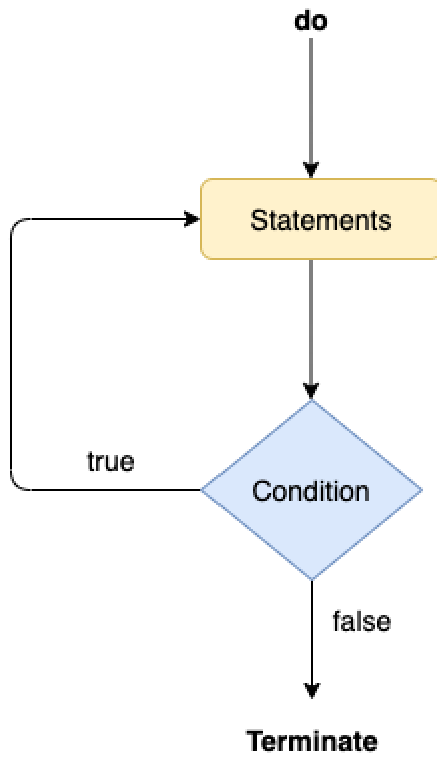


Consider the following example.

Calculation .java
```
public class Calculation {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                int i = 0;
                System.out.println("Printing the list of first 10 even numbers \n");
                while(i<=10) {
                System.out.println(i);
                i = i + 2;
                }
        }
}
```
Output:
```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```
4. do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.
```
do
{
//statements
} while (condition);
```
The flow chart of the do-while loop is given in the following image.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
	public class Calculation {
		public static void main(String[] args) {
			// TODO Auto-generated method stub
			int i = 0;
			System.out.println("Printing the list of first 10 even numbers \n");
			do {
			System.out.println(i);
			i = i + 2;
			}while(i<=10);
		}
	}
```

Output:

```
	Printing the list of first 10 even numbers
	0
	2
	4
	6
	8
	10
```

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

1. break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.
The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.
The break statement example with for loop

2. continue statement

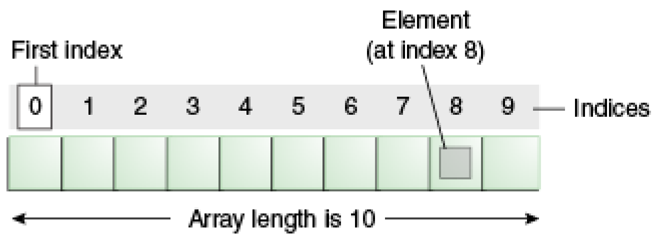
Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.
Consider the following example to understand the functioning of the continue statement in Java.

Example:

```
public class JT_16_break_and_continue {
    public static void main(String[] args) {
        // Break and continue using loop

        System.out.println("Using For Loop");

        for (int i = 0; i<5;i++){
            System.out.println(i);
            System.out.println("Java is great");

            if (i == 2){
                System.out.println("Ending the loop");
                break;
            }
        }


        System.out.println("\n"+"Using For while");
        int i = 0;
        while (i<5){
            System.out.println(i);
            System.out.println("Java is great");

            if (i == 2){
                System.out.println("Ending the loop");
                break;
            }
            i++;
        }


        System.out.println("\n"+"Using do while");
        int a = 0;
         do{
            System.out.println(a);
            System.out.println("Java is great");

            if (a == 2){
                System.out.println("Ending the loop");
                break;
            }
            a++;
        }while (a<5);




        System.out.println("\n"+"continue");
        for (int b = 0; b < 5; b++){
            if (b == 2) {
                System.out.println("Ending the loop");
                continue;
            }
        System.out.println(b);
        System.out.println("Java is great");
      }
    }
}
```

Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.
Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.
Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.
In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.
Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Creating Arrays

1. First Methods:
```
int [] marks = new int[5];   //declaration + memory allocation
```

2. Second Methods:
```
int [] marks ;  // declaration
marks  = new int[5]; // memory allocation
```

3. Third Methods:
```
int [] age = {24,45,78,56,12};
```

Example:
```
public class JT_18_array {
   public static void main(String[] args) {
      /*-------------- Method 1 -------------*/
      // int [] marks = new int[5]; //declaration + memory allocation
      /*-------------- Method 2 -------------*/
      int [] marks ;  // declaration
      marks  = new int[5]; // memory allocation
```

```
          marks[0] = 100;
          marks[1]= 60;
          marks[2]= 70;
          marks[3]= 40;
          marks[4]=50;
          System.out.println(marks[0]);

          /*-------------- Method 3 -------------*/
          int [] age = {24,45,78,56,12};
          System.out.println(age[2]);

        }
    }
```

Advantages

- Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.
- Random access: We can get any data located at an index position.

Disadvantages

- Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

1. Single Dimensional Array
2. Multidimensional Array

Single Dimensional Array

Syntax to Declare an Array in Java

```
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
        class Testarray{
                public static void main(String args[]){
                        int a[]=new int[5];//declaration and instantiation
                        a[0]=10;//initialization
                        a[1]=20;
                        a[2]=70;
                        a[3]=40;
                        a[4]=50;
                        //traversing array
                        for(int i=0;i<a.length;i++)//length is the property of array
                        System.out.println(a[i]);
                }
        }
```

Output:

```
        10
        20
        70
        40
        50
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

int a[]={33,3,4,5};//declaration, instantiation and initialization

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
        class Testarray1{
                        public static void main(String args[]){
                        int a[]={33,3,4,5};//declaration, instantiation and initialization
                        //printing array
                        for(int i=0;i<a.length;i++)//length is the property of array
                        System.out.println(a[i]);
                }
        }
```

Output:

```
        33
        3
        4
        5
```

For-each Loop for Java Array

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

Example:

```
public class JT_19_For_Each_loop {
   public static void main(String[] args) {
      int [] age = {24,45,78,56,12};

      for (int element: age){
         System.out.println(element);
      }
   }
}
```

Output:

```
24
45
78
56
12
```

Multidimensional Arrays

A multidimensional array is an array of arrays.
To create a two-dimensional array, add each array within its own set of curly braces:
Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];
```

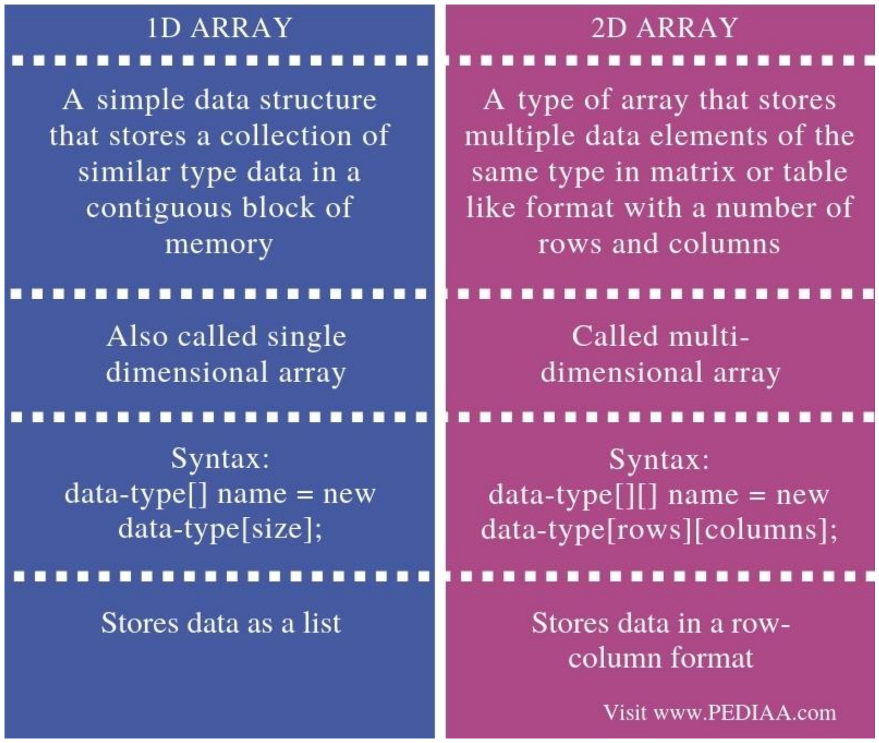
Example:

```
public class Jt_20_Multidim_Array {
   public static void main(String[] args) {
      int [] marks; //  1-D array
```

```
        int [][] flats; // 2-D array
        flats = new int [2][3];// 2row ---  3column |
        flats[0][0] = 101;
        flats[0][1] = 102;
        flats[0][2] = 103;
        flats[1][0] = 201;
        flats[1][1] = 202;
        flats[1][2] = 203;

        System.out.println("Printing array ");
        for (int i = 0 ; i < flats.length; i++) {
          for (int j = 0; j < flats[i].length; j++) {
            System.out.print(flats[i][j]);
            System.out.print(" ");
          }
          System.out.println("\n");
         }

    }
  }
```

Output:

```
101 102 103
201 202 203
```



1D ARRAY	2D ARRAY
A simple data structure that stores a collection of similar type data in a contiguous block of memory	A type of array that stores multiple data elements of the same type in matrix or table like format with a number of rows and columns
Also called single dimensional array	Called multi-dimensional array
Syntax: data-type[] name = new data-type[size];	Syntax: data-type[][] name = new data-type[rows][columns];
Stores data as a list	Stores data in a row-column format

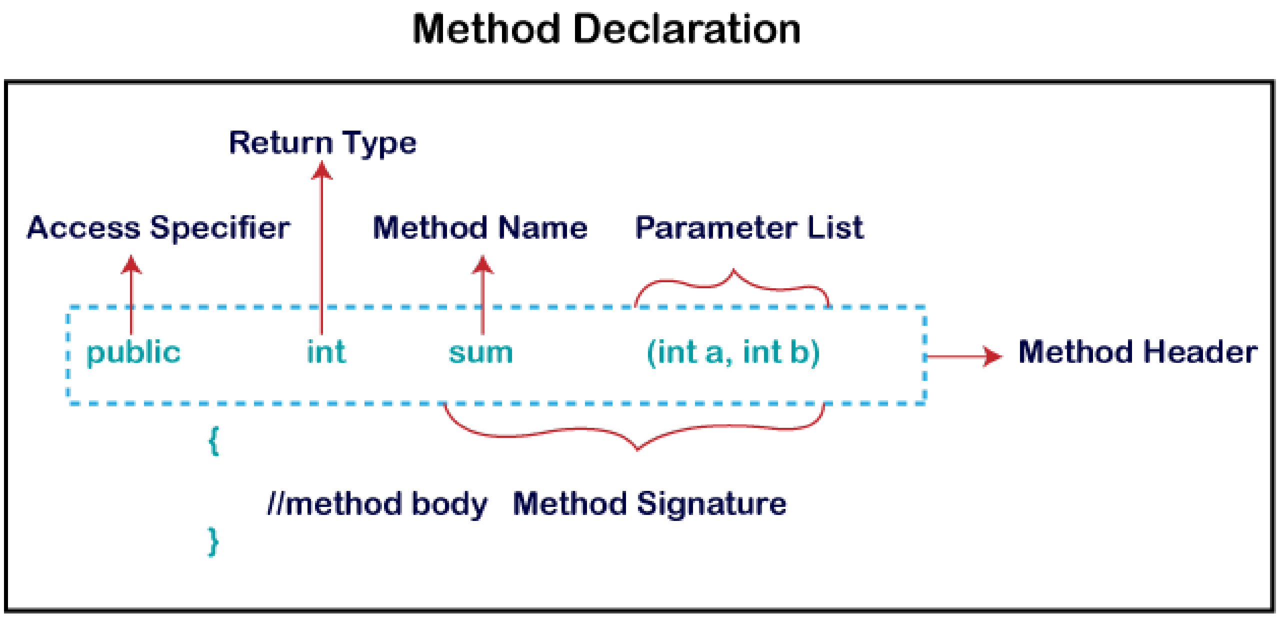
In general, a method is a way to perform some task. Similarly, the method in Java is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using methods. In this section, we will learn what is a method in Java, types of methods, method declaration, and how to call a method in Java.

What is a method in Java?
A method in Java or Java Method is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything. Methods in Java allow us to reuse the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.

Note: Methods are time savers and help us to reuse the code without retyping the code.

Creating Method
Considering the following example to explain the syntax of a method −

Syntax
```
public static int methodName(int a, int b) {
  // body
}
```



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

- **Public**: The method is accessible by all classes when we use public specifier in our application.
- **Private**: When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected**: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default**: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

While defining a method, remember that the method name must be a verb and start with a lowercase letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in uppercase except the first word. For example:

Single-word method name: sum(), area()
Multi-word method name: areaOfCircle(), stringComparision()

It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

There are two types of methods in Java:

1. Predefined Method
2. User-defined Method

1. Predefined Method: In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point.

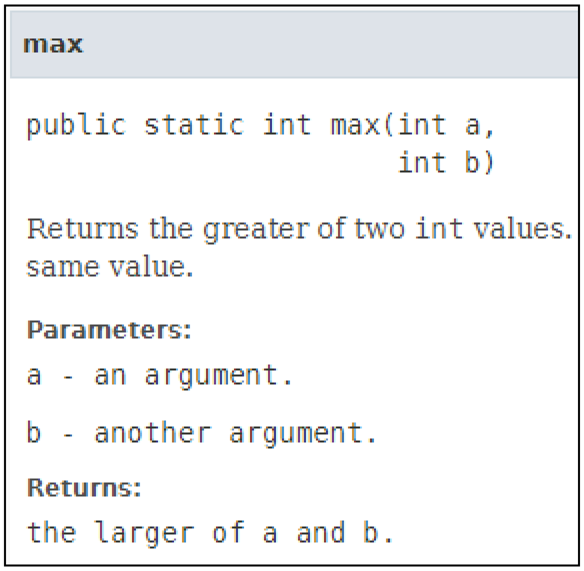
Example:

```
public class Demo
        {
        public static void main(String[] args)
        {
        // using the max() method of Math class
        System.out.print("The maximum number is: " + Math.max(9,7));
        }
}
```

Output:

```
The maximum number is: 9
```

In the above example, we have used three predefined methods main(), print(), and max(). We have used these methods directly without declaration because they are predefined. The print() method is a method of PrintStream class that prints the result on the console. The max() method is a method of the Math class that returns the greater of two numbers. We can also see the method signature of any predefined method by using the link https://docs.oracle.com/. When we go through the link and see the max() method signature, we find the following:



max

```
public static int max(int a,
                      int b)
```

Returns the greater of two int values.
same value.

Parameters:

a - an argument.

b - another argument.

Returns:

the larger of a and b.

n the above method signature, we see that the method signature has access specifier public, non-access modifier static, return type int, method name max(), parameter list (int a, int b). In the above example, instead of defining the method, we have just invoked the method. This is the advantage of a predefined method. It makes programming less complicated.
Similarly, we can also see the method signature of the print() method.

2. User-defined Method: The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

How to Create a User-defined Method
Let's create a user defined method that checks the number is even or odd. First, we will define the method.

```
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

We have defined the above method named findevenodd(). It has a parameter num of type int. The method does not return any value that's why we have used void. The method body contains the steps to check the number is even or odd. If the number is even, it prints the number is even, else prints the number is odd.

How to Call or Invoke a User-defined Method
Once we have defined a method, it should be called. The calling of a method in a program is simple. When we call or invoke a user-defined method, the program control transfer to the called method.

```
import java.util.Scanner;
public class EvenOdd
        {
        public static void main (String args[])
        {
        //creating Scanner class object
        Scanner scan=new Scanner(System.in);
        System.out.print("Enter the number: ");
        //reading value from the user
```

```
            int num=scan.nextInt();
            //method calling
            findEvenOdd(num);
    }
```
In the above code snippet, as soon as the compiler reaches at line findEvenOdd(num), the control transfer to the method and gives the output accordingly.

Let's combine both snippets of codes in a single program and execute it.
```
import java.util.Scanner;
public class EvenOdd
{
  public static void main (String args[])
  {
    //creating Scanner class object
    Scanner scan=new Scanner(System.in);
    System.out.print("Enter the number: ");
    //reading value from user
    int num=scan.nextInt();
    //method calling
    findEvenOdd(num);
  }
  //user defined method
  public static void findEvenOdd(int num)
  {
    //method body
    if(num%2==0)
      System.out.println(num+" is even");
    else
      System.out.println(num+" is odd");
  }
}
```
Output 1:
```
Enter the number: 12
12 is even
Output 2:
Enter the number: 99
99 is odd
```
Let's see another program that return a value to the calling method.

In the following program, we have defined a method named add() that sum up the two numbers. It has two parameters n1 and n2 of integer type. The values of n1 and n2 correspond to the value of a and b, respectively. Therefore, the method adds the value of a and b and store it in the variable s and returns the sum.

Example:

```
public class Addition
{
   public static void main(String[] args)
   {
      int a = 19;
      int b = 5;
      //method calling
      int c = add(a, b);   //a and b are actual parameters
      System.out.println("The sum of a and b is= " + c);
   }
   //user defined method
   public static int add(int n1, int n2)   //n1 and n2 are formal parameters
   {
      int s;
      s=n1+n2;
      return s; //returning the sum
   }
}
```

Output:

```
The sum of a and b is= 24
```

Method Overloading

When a class has two or more methods by the same name but different parameters, it is known as method overloading. It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.
Let's consider the example discussed earlier for finding minimum numbers of integer type. If, let's say we want to find the minimum number of double type. Then the concept of overloading will be introduced to create two or more methods with the same name but different parameters.

The following example explains the same –

```
package com.company;
public class JT_21_Method_Overloading {
   static void foo(){
      System.out.println("Hello");
   }
```

```
    static  void foo(int c){
       System.out.println("Good Moring " + c + " Bro");
    }
    static  void foo(int d,int e){
      System.out.println("Good Moring " + d + " Bro");
      System.out.println("Good Moring " + e + " Bro");
    }
    public static void main(String[] args) {
       foo();
       foo(4000 );
       foo(6000, 5000 );  // argument are actual!
    }
}
```

Output:

```
Hello
Good Moring 4000 Bro
Good Moring 6000 Bro
Good Moring 5000 Bro
```

Variable Argument (Varargs)

The varrags allows the method to accept zero or muliple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

- We don't have to provide overloaded methods so less code.

Syntax of varargs:

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

```
return_type method_name(data_type... variableName){
       //code
}
```

First, let’s look at the example without using varargs:

```
package com.company;
public class JT_22_Varagrs {
   static int sum(int a,int b){
      return  a+ b;
   }
   static int sum(int a,int b ,int c ){
        return  a+ b+c;
    }static int sum(int a,int b ,int c ,int d){
        return  a+ b +c+d;
    }
    public static void main(String[] args) {
        System.out.println("The sum of 4 and 5 is: " + sum(4,5));
        System.out.println("The sum of 6, 8 and 9 is: " + sum(6,8, 9 ));
        System.out.println("The sum of 10, 20, 30 and 40 is: " + sum(10,20,30,40));
    }
}
```

Output:

```
The sum of 4 and 5 is: 9
The sum of 6, 8 and 9 is: 23
The sum of 10, 20, 30 and 40 is: 100
```

As you can clearly see, you had to overload sum() method to make it work for 3 arguments.

What if the user wants to add 5 numbers or 10 or 100? So let see example

let's look at the example using varargs:

```
package com.company;
public class JT_22_Varagrs {
    static int sum(int ...arr){
        int result = 0;
        for (int a: arr) {
            result +=a;
        }
        return  result;
    }
    public static void main(String[] args) {
        System.out.println("The sum of 4 and 5 is: " + sum(4,5));
        System.out.println("The sum of 6, 8 and 9 is: " + sum(6,8, 9 ));
        System.out.println("The sum of 10, 20, 30 and 40 is: " + sum(10,20,30,40));
    }
}
```

Output:

```
The sum of 4 and 5 is: 9
The sum of 6, 8 and 9 is: 23
The sum of 10, 20, 30 and 40 is: 100
```

What happen if we didn't pass the value at sum() let's look the example using varargs:

```
package com.company;
public class JT_22_Varagrs {
   static int sum(int ...arr){
      int result = 0;
      for (int a: arr) {
         result +=a;
      }
      return  result;
   }
   public static void main(String[] args) {
      System.out.println (sum());
      System.out.println("The sum of 4 and 5 is: " + sum(4,5));
      System.out.println("The sum of 6, 8 and 9 is: " + sum(6,8, 9 ));
      System.out.println("The sum of 10, 20, 30 and 40 is: " + sum(10,20,30,40));
   }
}
```

Output:

```
0
The sum of 4 and 5 is: 9
The sum of 6, 8 and 9 is: 23
The sum of 10, 20, 30 and 40 is: 100
```

What happen when we make sum(int x …arr) let's look the example using varargs:

```
package com.company;
public class JT_22_Varagrs {
   static int sum(int x ...arr){
      int result = 0;
      for (int a: arr) {
         result +=a;
      }
      return  result;
   }
   public static void main(String[] args) {
      System.out.println (sum());
      System.out.println("The sum of 4 and 5 is: " + sum(4,5));
      System.out.println("The sum of 6, 8 and 9 is: " + sum(6,8, 9 ));
      System.out.println("The sum of 10, 20, 30 and 40 is: " + sum(10,20,30,40));
   }
}
```

Output:

```
Error
```

Here your output will be show error because sum(int x ...arr) will say that there must be the value at x let look example:

```
package com.company;
public class JT_22_Varagrs {
   static int sum(int x ...arr){
      int result = 0;
      for (int a: arr) {
         result +=a;
      }
      return  result;
   }
   public static void main(String[] args) {
      System.out.println (sum(5,10));
      System.out.println("The sum of 4 and 5 is: " + sum(4,5));
      System.out.println("The sum of 6, 8 and 9 is: " + sum(6,8, 9 ));
      System.out.println("The sum of 10, 20, 30 and 40 is: " + sum(10,20,30,40));
   }
}
```

Output:

```
15
The sum of 4 and 5 is: 9
The sum of 6, 8 and 9 is: 23
The sum of 10, 20, 30 and 40 is: 100
```

Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.
It makes the code compact but complex to understand.

Syntax:

```
returntype methodname(){
      //code to be executed
      methodname();//calling same method
}
```

Java Recursion Example: Factorial Number
Methods: 1

```
public class JT_23_Recursion {
    static int factorial(int n){
        if(n == 0 || n == 1){
            return  1;
        }else {
            return  n * factorial(n-1);
        }
    }
    public static void main(String[] args) {
        int n = 5;
        System.out.println("The value of factorial n is: " + factorial(n));
    }
}
```

Output:

```
Enter the value of n
5
The value of factorial n is: 120
```

Methods: 2

```
public class JT_23_Recursion {
    static int factorial(int n){
        if(n == 0 || n == 1){
            return  1;
        }else {
            return  n * factorial(n-1);
        }
    }
    public static void main(String[] args) {
        System.out.println("Enter the value of n");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        System.out.println("The value of factorial n is: " + factorial(n));
}
```

Output:

```
The value of factorial n is: 120
```

Factorial Using iterative Methods in java

```
package com.company;
import java.util.Scanner;

public class JT_23_Recursion {
    static int factorial_iterative(int n){
        if(n == 0 || n == 1){
            return  1;
        }else {
            int product  = 1;
            for (int i = 1; i <= n; i++) {
                product *= i;
            }
            return  product;
        }
    }
    public static void main(String[] args) {
        System.out.println("Enter the value of n");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        System.out.println("The value of factorial iterative n is: " + factorial_iterative(n));
    }
}
```

Output:

```
Enter the value of n
5
The value of factorial iterative n is: 120
```

Java Recursion Example: Fibonacci Series

```
package com.company;
import java.util.Scanner;

public class JT_24_recursive_Fibonacci {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        System.out.println("Enter the value of n");
        int n = in.nextInt();

        for(int i=1; i<=n; i++){
            //call recursive function for the ith term
            System.out.print(fibonacci(i)+" ");
        }
    }
    private static int fibonacci(int n){
        if(n==1){
            return 0;
        }
        else if(n == 2){
            return 1;
        } else {
            return fibonacci(n-2)+fibonacci(n-1);
        }
    }
}
```

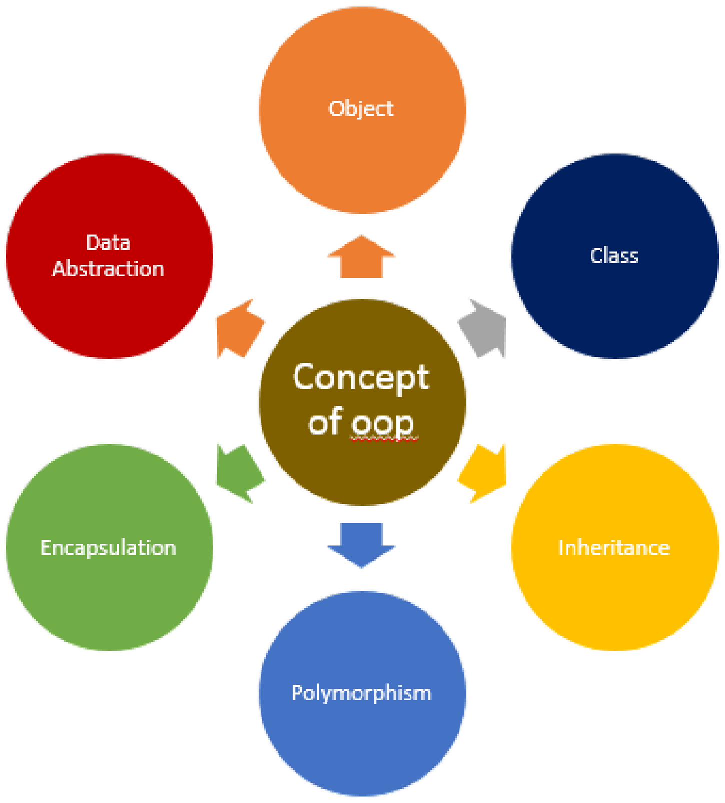
Output:

```
Enter the value of n
12
0 1 1 2 3 5 8 13 21 34 55 89
```

OOPs Concepts in Java

Object-Oriented Programming System (OOPs) is a programming concept that works on the principles of abstraction, encapsulation, inheritance, and polymorphism. It allows users to create objects they want and create methods to handle those objects. The basic concept of OOPs is to create objects, re-use them throughout the program, and manipulate these objects to get results.
OOP meaning "Object Oriented Programming" is a popularly known and widely used concept in modern programming languages like Java.

Features of OOP in java



1. Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

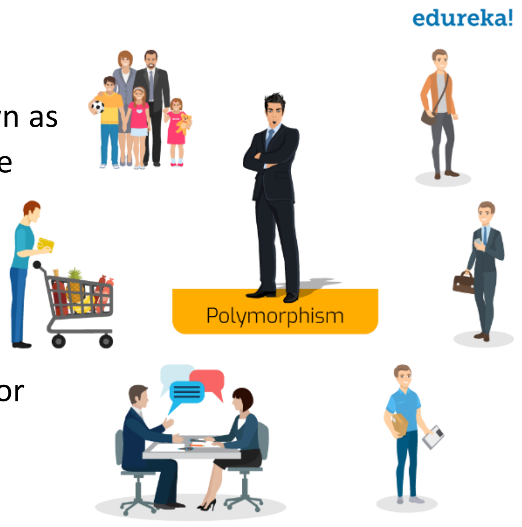
2. Class

The class is one of the Basic concepts of OOPs which is a group of similar entities. It is only a logical component and not the physical entity. Lets understand this one of the OOPs Concepts with example, if you had a class called "Expensive Cars" it could have objects like Mercedes, BMW, Toyota, etc. Its properties(data) can be price or speed of these cars. While the methods may be performed with these cars are driving, reverse, braking etc.

3. Inheritance

Inheritance is one of the Basic Concepts of OOPs in which one object acquires the properties and behaviors of the parent object. It's creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.

4. Polymorphism

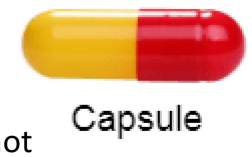


If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
In Java, we use method overloading and method overriding to achieve polymorphism.
Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

5. Abstraction

Abstraction is one of the OOP Concepts in Java which is an act of representing essential features without including background details. It is a technique of creating a new data type that is suited for a specific application. Lets understand this one of the OOPs Concepts with example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

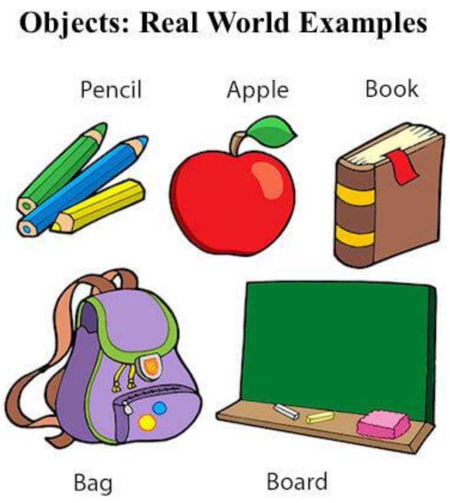
6. Encapsulation



Encapsulation is one of the best Java OOPs concepts of wrapping the data and code. In this OOPs concept, the variables of a class are always hidden from other classes. It can only be accessed using the methods of their current class. For example – in school, a student cannot exist without a class.

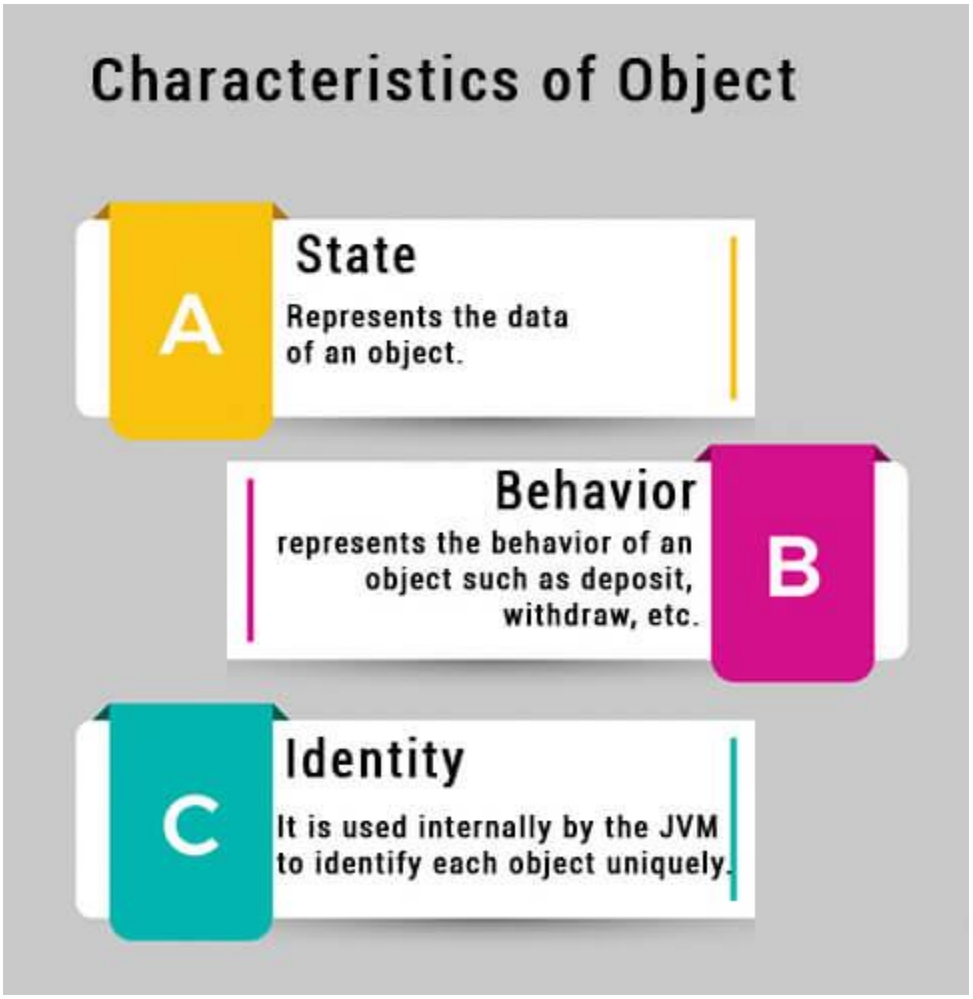
What is an object in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.



An object has three characteristics:

- **State**: represents the data (value) of an object.
- **Behavior**: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity**: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.


Characteristics of Object
A
State
Represents the data of an object.
Behavior
represents the behavior of an object such as deposit, withdraw, etc.
B
C
Identity
It is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.
An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behavior.
- The object is an instance of a class.

What is a class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>{
   field;
   method;
}
```

Creating Our Own Java Class

```
package com.company;
class Employee{
   int id;
   String name;
   public  void  printDetails(){
      System.out.print("My name " + name);
      System.out.println(" and my id " + id);
   }
}

public class JT_25_Custom_Class {
   public static void main(String[] args) {
      System.out.println("This the custom class");
      Employee details = new Employee(); // Instantiating a new Employee object
      Employee students = new Employee(); // Instantiating a new Employee object

      // Setting properties or attributes
      details.id = 1;
      details.name = "Gita";
      students.id = 2;
      students.name = "Esha";

      details.printDetails();
      students.printDetails();
   }
}
```

Output:

```
This the custom class
My name Gita and my id 1
My name Esha and my id 2
```

Access Modifiers

There are two types of modifiers in Java: access modifiers and non-access modifiers.
The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

Getters and Setters

Getter and Setter are methods used to protect your data and make your code more secure. Getter returns the value (accessors), it returns the value of data type int, String, double, float, etc. For the convenience of the program, getter starts with the word "get" followed by the variable name.

Getter ➻ Returns the value [accessors]

Defining a getter

We can define the getters method by using the get keyword with no parameter a valid return type.

Syntax:

return_type get field_name{
}

While Setter sets or updates the value (mutators). It sets the value for any variable which is used in the programs of a class. and starts with the word "set" followed by the variable name. Getter and Setter make the programmer convenient in setting and getting the value for a particular data type. In both getter and setter, the first letter of the variable should be capital.

setter ➻ Sets / updates the value [mutators]

Defining a setter

We can declare the setter method using the set keyword with one parameter and without return type.

Syntax:

set field_name {
}

In the below code, we've created total 4 methods:
setName(): The argument passed to this method is assigned to the private variable name.
getName(): The method returns the value set by the setName() method.
setId(): The integer argument passed to this method is assigned to the private variable id.
getId): This method returns the value set by the setId() method.

Example:

```
package com.company;
class  myEmployee{
  private int id;
  private String name;

   public  String getName(){
      return  name;
   }
  public void setName(String n){
     name = n;
  }
  public void setId( int i){
     id = i;
  }
   public  int getId(){
      return  id;
   }
}

public class JT_27_Access_Modifer {
   public static void main(String[] args) {
      myEmployee details = new myEmployee();

      details.setId(1);
      System.out.println(details. getId());

      details.setName("Sunil");
      System.out.println(details. getName());
   }
}
```

Output:

```
1
Sunil
```

Constructors

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

***Note**: It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.*

Rules for creating Java constructor

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Syntax:

```
class A
{
  .......

  // A Constructor
  new A() {}

  .......
}
// We can create an object of the above class
// using the below statement. This statement
// calls above constructor.
Geek obj = new Geek();
```

Types of Java constructors

There are two types of constructors in Java:

1. **Default constructor (no-arg constructor)**
2. **Parameterized constructor**

1. No-argument constructor

A constructor that has no parameter is known as the default constructor. If we don't define a constructor in a class, then the compiler creates a default constructor(with no arguments) for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.

***Note**: Default constructor provides the default values to the object like 0, null, etc. depending on the type.*

Example:

```
package com.company;
class  myEmployeeMain{
   private int id;
   private String name;
   public myEmployeeMain (){
        id   = 45;
        name  = "Sunil";
   }
   public  String getName(){
      return  name;
   }
   public void setName(String n){
      name = n;
   }
   public void setId( int i){
      id = i;
   }
   public  int getId(){
      return  id;
   }
}

public class JT_28_Constructor {
   public static void main(String[] args) {
      myEmployeeMain details = new myEmployeeMain();
      System.out.println(details.getId());
      System.out.println(details.getName());
   }
}
```

Output:

```
45
Sunil
```

2. Parameterized Constructor

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Example:

```
package com.company;
class  myEmployeeMain{
   private int id;
   private String name;

   public myEmployeeMain(String myName , int myId){
      id   = myId;
      name  = myName;
   }
   public  String getName(){
      return  name;
   }
   public void setName(String n){
      name = n;
   }
   public void setId( int i){
      id = i;
   }
   public  int getId(){
      return  id;
   }
}

public class JT_28_Constructor {
   public static void main(String[] args) {
      myEmployeeMain details = new myEmployeeMain( "GorkCoder", 12);
      System.out.println(details.getId());
      System.out.println(details.getName());
   }
}
```

Output:

```
12
GorkCoder
```

Constructor overloading

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.
Consider the following Java program, in which we have used different constructors in the class.

Example: 1

```
package com.company;
class  myEmployeeCons{
   private int id;
   private String name;
   //  First Methods
   public myEmployeeCons (){
        id   = 45;
        name  = "Sunil";
   }

   // Seconds Methods
   public myEmployeeCons(String myName , int myId){
      id   = myId;
      name  = myName;
   }
   public  String getName(){
      return  name;
   }
   public void setName(String n){
      name = n;
   }
   public void setId( int i){
      id = i;
   }
   public  int getId(){
      return  id;
   }
}

public class JT_29_Constructor_methods_overloading {
   public static void main(String[] args) {
      //call for "Seconds Methods"
      // myEmployeeCons details = new myEmployeeCons( "GorkCoder", 12);

      // call  for "First Methods"
      myEmployeeCons details = new myEmployeeCons( );

      System.out.println(details.getId());
      System.out.println(details.getName());
   }
}
```

Output:

```
45
Sunil
```

Example: 2

```
//call for "Seconds Methods"
  myEmployeeCons details = new myEmployeeCons( "GorkCoder", 12);
```

Output:

```
12
GorkCoder
```

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
  .....
  .....
}
class Sub extends Super {
  .....
  .....
}
```

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

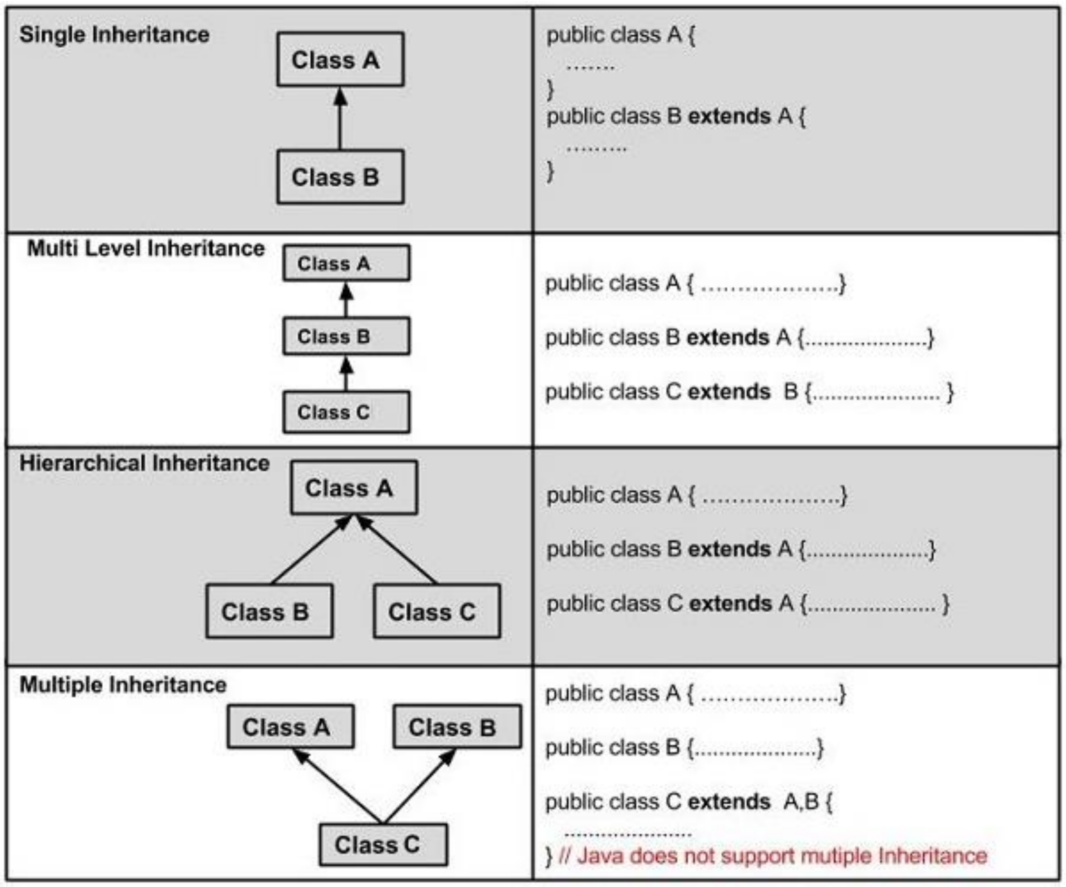
Example:

```
package com.company;
class Base{
   int x ;
   public  int getX(){
      return  x;
   }
   public  void setX(int x){
      System.out.println("I am in base and setting X now.");
      this.x = x;
   }
}
class Derived extends Base{
   int y;
   public int getY() {
      return y;
   }
   public void setY(int y) {
      System.out.println("I am in Derived and setting Y now.");
      this.y = y;
   }
}
public class JT_32_Inheritance {
   public static void main(String[] args) {
      //Creating an object of Base class
      Base b = new Base();
      b.setX(4);
      System.out.println(b.getX());

      //Creating an object of Derived class
```

```
            Derived d = new Derived();
            d.setX(43);
            System.out.println(d.getX());

            d.setY(23);
            System.out.println(d.getY());
        }
    }
```
Output:
```
    I am in base and setting X now.
    4
    I am in base and setting X now.
    43
    I am in Derived and setting Y now.
    23
```
Types of Inheritance

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



```
public class A {
  .......
}
public class B extends A {
  .........
}
```
```
public class A { ...................}
public class B extends A {...................}
public class C extends  B {.................... }
```
```
public class A { ...................}
public class B extends A {...................}
public class C extends A {.................... }
```
```
public class A { ...................}
public class B {...................}
public class C extends  A,B {
  ....................
} // Java does not support mutiple Inheritance
```

1. **Single Inheritance Example**

When a class inherits another class, it is known as a single inheritance. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal{
        void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
        void bark(){System.out.println("barking...");}
}
class TestInheritance{
        public static void main(String args[]){
        Dog d=new Dog();
        d.bark();
        d.eat();
}}
```

Output:

```
barking...
eating...
```

2. **Multilevel Inheritance Example**

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal{
        void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
        void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
        void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
        public static void main(String args[]){
        BabyDog d=new BabyDog();
        d.weep();
        d.bark();
        d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

3. **Hierarchical Inheritance Example**

When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating...
```

4. **Why multiple inheritance is not supported in java?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
	void msg(){System.out.println("Hello");}
}
```

```
class B{
        void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
          C obj=new C();
          obj.msg();//Now which msg() method would be invoked?
} }
```

Output: Compile Time Error

Inheritance and constructors

In Java, constructor of base class with no argument gets automatically called in derived class constructor. For example, output of following program is:
Base Class Constructor Called
Derived Class Constructor Called

```
package com.company;
class Base1{
   Base1(){
      System.out.println("I am Constructor");
   }
}
class Derived1 extends Base1 {
   Derived1(){
      System.out.println("I am Derived1 class Constructor");
   }
}
public class JT_33_Inheritance_Constructor {
   public static void main(String[] args) {
      Base1 b1 = new Base1();
      Derived1 d1 = new Derived1();
   }
}
```

Output:

```
I am Constructor
I am Constructor
I am Derived1 class Constructor
```

Constructors Overloading in Inheritance

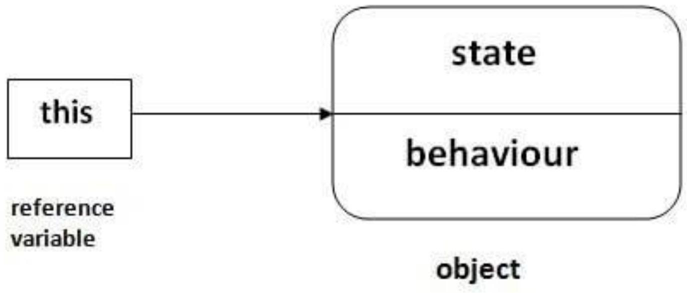
```
package com.company;
class Base2{
   Base2(){
      System.out.println("I am Constructor");
   }
   Base2(int x){
      System.out.println("I am Overloaded Constructor of Base2 with value of x : " + x);
   }
}
class Derived2 extends Base2 {
   Derived2(){
      //super(1);
      System.out.println("I am Derived2 class Constructor");
   }

   Derived2(int x ,int y){
       super(x);
      System.out.println("I am Overloaded Constructor of Derived2  with value of y: " + y);
   }
}
class ChildOfDerives extends Derived2 {
   ChildOfDerives(){
      System.out.println("I am a child of derived Constructor");
   }
   ChildOfDerives(int x, int y, int z){
      super(x, y);
      System.out.println("I am Overloaded Constructor of ChildOfDerives  with value of z:
" + z);
   }
}
public class JT_34_Inheri_Constr_Overloading {
   public static void main(String[] args) {
     //Derived2 d1 = new Derived2();
     //Derived2 d1 = new Derived2(78,15);
      //ChildOfDerives d1 = new ChildOfDerives( );
       ChildOfDerives d1 = new ChildOfDerives(78,15,27);
   }}
```

Output:

```
I am Overloaded Constructor of Base2 with value of x : 78
I am Overloaded Constructor of Derived2  with value of y: 15
I am Overloaded Constructor of ChildOfDerives  with value of z: 27
```

this Keyword

this keyword in Java is a reference variable that refers to the current object of a method or a constructor. The main purpose of using this keyword in Java is to remove the confusion between class attributes and parameters that have same names.



Following are various uses of 'this' keyword in Java:

- It can be used to refer instance variable of current class
- It can be used to invoke or initiate current class constructor
- It can be passed as an argument in the method call
- It can be passed as argument in the constructor call
- It can be used to return the current class instance

Example:

```
package com.company;
class basic{
   int a;
    basic(int a){
      this.a = a;
   }
   public  int getA(){
      return  a;
   }
   public int returnone(){
      return 1;
   }
}
public class JT_35_this_keyword {
   public static void main(String[] args) {
      basic e = new basic(5);
      System.out.println(e.getA());
   }
}
```

Output: 5

Super keyword

The super keyword in java is a reference variable that is used to refer parent class objects. The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Example:

```
package com.company;
class superKey{
    int a;
    superKey(int a){
        this.a = a;
    }
    public  int getA(){
        return  a;
    }
    public int returnone(){
        return 1;
    }
}
class superDerived extends  superKey{
    superDerived(int c){
         super(c);
        System.out.println("I am Constructor");
    }
}
public class JT_36_Super_key {
    public static void main(String[] args) {
        superKey e = new superKey(5);
        superDerived d = new superDerived(80);
        System.out.println(e.getA());
    }
}
```

Output:

```
I am Constructor
5
```

Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Example:

```
package com.company;
class A{
   public int a;
   public int method1(){
      return 4;
   }
   public  void method2(){
      System.out.println("I am a methods 2 of class A.");
   }
}
class B extends A{
   public  void method2(){
      System.out.println("I am a methods 2 of class B.");
   }
   public  void method3(){
      System.out.println("I am a methods 3 of class B.");
   }
}
public class JT_37_method_overriding {
   public static void main(String[] args) {
      A a= new A();
      a.method2();
      B b = new B();
      b.method2();
   }}
```

Output:

```
I am a methods 2 of class A.
I am a methods 2 of class B.
```

Polymorphism in Java

Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

Types of polymorphism

In Java polymorphism is mainly divided into two types:

1. Compile-time Polymorphism
2. Runtime Polymorphism

Compile-time Polymorphism

A polymorphism which is exists at the time of compilation is called compile time or early binding or static polymorphism.

Example: Method overloading

Whenever a class contain more than one method with same name and different types of parameters called method overloading.

Dynamic Method Dispatch / Runtime polymorphism

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

- Dynamic method dispatch is also known as run time polymorphism.
- It is the process through which a call to an overridden method is resolved at runtime.
- This technique is used to resolve a call to an overridden method at runtime rather than compile time.
- To properly understand Dynamic method dispatch in Java, it is important to understand the concept of upcasting because dynamic method dispatch is based on upcasting.

Example:

```
package com.company;
class One{
    public void name(){
        System.out.println("My name is Java");
    }
}
class Two extends  One{
  public void name(){
        System.out.println("My name is java in Class Two.");
  }
 }
public class JT_38_Dynamic_methods_dispatch {
    public static void main(String[] args) {
//          One obj1 = new One(); // Allowed
//          Two obj2 = new Two(); // Allowed
//          obj1.name();

        One obj = new Two(); // Yes it is allowed
//          Two obj = new One(); // not allowed
        obj.name();

    }
}
```

Output:

```
    My name is java in Class Two.
```

Abstract class

A class which contains the abstract keyword in its declaration is called abstract class.
A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.
Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
Abstraction lets you focus on what the object does instead of how it does it.
Ways to achieve Abstraction
There are two ways to achieve abstraction in java

Abstract class (0 to 100%)
Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example:

```
package com.company;
abstract  class animal{
   public abstract void sound();
}
class Dog extends animal {
   public void sound(){
      System.out.println("Dog is Barking.");
   }
}
class Lion extends animal{
   public void sound(){
      System.out.println("Lion is Roar.");
   }
}
public class JT_39_abstract {
   public static void main(String[] args) {
      Dog d = new Dog();
      Lion l = new Lion();
      d.sound();
      l.sound();
   }
}
```

Output:

```
Dog is Barking.
Lion is Roar.
```

Interface

Interface is just like a class which contains only abstract method. To achieve interface java provides a keyword called implements.

- Interface method are by default public and abstract.
- Interface variable are by default public + static + final.
- Interface method must be overridden inside the implementing classes
- Interface nothing but deals between client and developer.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Syntax:

```
interface <interface_name>{
     // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Example:

```
package com.company;
import java.util.Scanner;

interface client {
   void input();
   void output();
}
class Sunil implements  client {
   String name;
   double salary;

   public void  input(){
      Scanner r = new Scanner(System.in);
      System.out.println("Enter Username: ");
      name = r.nextLine();

      Scanner r1 = new Scanner(System.in);
      System.out.println("Enter Salary: ");
      salary = r1.nextDouble();
```

```
    }
    public void output(){
        System.out.println(name+" "+ salary);
    }
}
public class JT_40_interface {
    public static void main(String[] args) {
        client c = new Sunil();
        c.input();
        c.output();
    }
}
```

Output:

```
Enter Username:
Sunil
Enter Salary:
500000
Sunil 500000.0
```

Package

A packages arrange number of classes interfaces and sub-package of same type into a particular group.

Note: package is nothing but folder in windows.

A java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

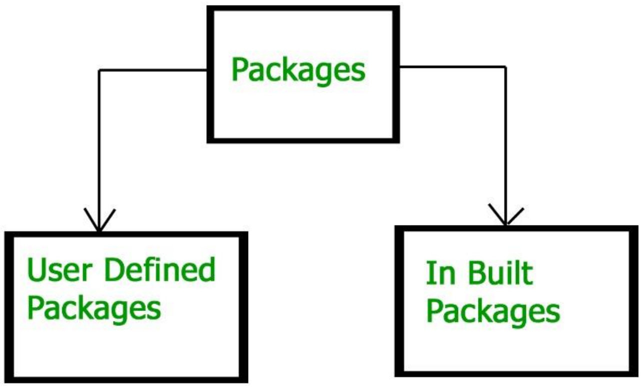
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.

Types of packages:



Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:

1) java.lang: Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) java.io: Contains classed for supporting input / output operations.

3) java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) java.applet: Contains classes for creating Applets.

5) java.awt: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) java.net: Contain classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory myPackage (name should be same as the name of the package). Then create the MyClass inside the directory with the first statement being the package names.

```
// Name of the package must be same as the directory
// under which this file is saved
        package myPackage;
        public class MyClass
        {
           public void getNames(String s)
           {
              System.out.println(s);
           }
        }
```

Now we can use the MyClass class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;
public class PrintName
{
  public static void main(String args[])
  {
    // Initializing the String variable
    // with a value
    String name = "GeeksforGeeks";

    // Creating an instance of class MyClass in
    // the package.
    MyClass obj = new MyClass();

    obj.getNames(name);
  }
}
```

Note : MyClass.java must be saved inside the myPackage directory since it is a part of the package.

Exception handling

What is exception?

An exception is unexpected/ unwanted/abnormal situation that occurred at runtime called exception.

Exception handling

In exception handling, we should have an alternate source through which we can handle the exception.

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Blocks & Keywords used for exception handling

1. try: The try block contains set of statements where an exception can occur.

```
try
{
   // statement(s) that might cause exception
}
```

2.catch : Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

```
catch
{
  // statement(s) that handle an exception
  // examples, closing a connection, closing
  // file, exiting the process after writing
  // details to a log file.
}
```

3.throw: Throw keyword is used to transfer control from try block to catch block.
4.throws: Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.
5.finally: It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks.

Example:

```
class Division {
   public static void main(String[] args)
   {
      int a = 10, b = 5, c = 5, result;
      try {
         result = a / (b - c);
         System.out.println("result" + result);
      }

      catch (ArithmeticException e) {
         System.out.println("Exception caught:Division by zero");
      }

      finally {
         System.out.println("I am in final block");
      }
   }
}
```

Output:

```
Exception caught:Division by zero
I am in final block
```

Multithreading

Multithreading in Java is a process of executing multiple threads simultaneously.
A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
2) You **can perform many operations together, so it saves time**.
3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

Process-based Multitasking (Multiprocessing)
Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Thread

Thread is a pre-defined class which is available in java.lang package. Thread is a basic units of CPU and it is well known for independent execution.

How to crate thread in java?

1. By extending thread class
2. By implementing runnable interface

By extending thread class

```
package com.company;
class MyThread1 extends  Thread {
   public void run(){
      int i = 0;
      while (i< 40){
         System.out.println("My Thread is Running");
         System.out.println("I am Happy!");
         i++;
      }
   }
}
class MyThread2 extends  Thread {
   public void run(){
      int i = 0;
      while (i< 40){
         System.out.println("Thread 2 is good");
         System.out.println("I am Sad!");
         i++;
      }
   }
}
public class JT_42_Multi_thread {
   public static void main(String[] args) {
      MyThread1 t1 = new MyThread1();
      MyThread2 t2 = new MyThread2();
      t1.start();
      t2.start();
   }
}
```

By implementing runnable interface

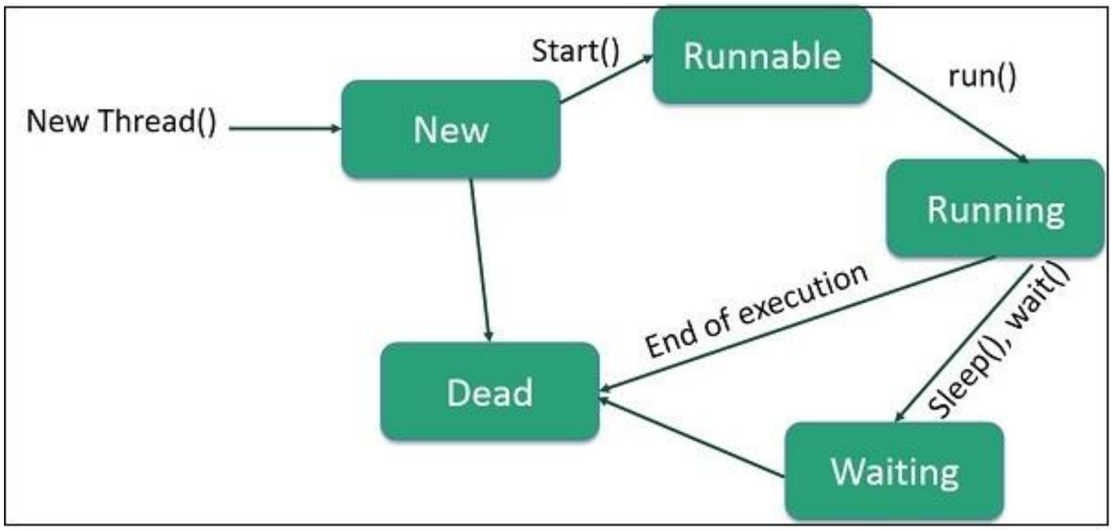
```
package com.company;
class  MyThreadRunnable1 implements Runnable {
   public  void  run(){
      System.out.println("I am Thread not a Thread 1");
      System.out.println("I am Thread not a Thread 1");
   }
}
class  MyThreadRunnable2 implements Runnable {
   public  void  run(){
      System.out.println("I am Thread not a Thread 2");
      System.out.println("I am Thread not a Thread 2");
   }
}
public class JT_43_By_implementing_runnable_interface {
   public static void main(String[] args) {
      MyThreadRunnable1 bullet1 = new MyThreadRunnable1();
      Thread gun1 = new Thread(bullet1);

      MyThreadRunnable2 bullet2 = new MyThreadRunnable2();
      Thread gun2 = new Thread(bullet2);

      gun1.start();
      gun2.start();
   }
}
```

Output:

```
I am Thread not a Thread 2
I am Thread not a Thread 2
I am Thread not a Thread 1
I am Thread not a Thread 1
```

life cycle of a thread

A thread goes through various stages in its lifecycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle −

New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. Thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Implementing the Thread States in Java

In Java, to get the current state of the thread, use Thread.getState() method to get the current state of the thread. Java provides java.lang.Thread.State class that defines the ENUM constants for the state of a thread, as a summary of which is given below:

1. New

Declaration: public static final Thread.State NEW

Description: Thread state for a thread that has not yet started.

2. Runnable

Declaration: public static final Thread.State RUNNABLE

Description: Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as a processor.

3. Blocked

Declaration: public static final Thread.State BLOCKED

Description: Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling Object.wait().

4. Waiting

Declaration: public static final Thread.State WAITING

Description: Thread state for a waiting thread. Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

Object.wait with no timeout

Thread.join with no timeout

LockSupport.park

5. Timed Waiting

Declaration: public static final Thread.State TIMED_WAITING

Description: Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:
Thread.sleep
Object.wait with timeout
Thread.join with timeout
LockSupport.parkNanos
LockSupport.parkUntil

6. Terminated

Declaration: public static final Thread.State TERMINATED

Description: Thread state for a terminated thread. The thread has completed execution.