Object Oriented Programming in JAVA

Lecture 1
Introduction to Java

UNIT - 1

JAVA Basics

Why Java is Important

- Two reasons :
  - The problem with the C / C ++ language is that they are not portable and are not independent platform languages.
  - The emergence of the World Wide Web, which required mobile applications
  - Portability and security have necessitated the establishment of Java

History of CORE JAVA

- James Gosling - Sun Microsystems
- Co founder – Vinod Khosla
- Oak - Java, May 20, 1995, Sun World
- JDK Evolutions
  - JDK 1.0 (January 23, 1996)
  - JDK 1.1 (February 19, 1997)
  - J2SE 1.2 (December 8, 1998)
  - J2SE 1.3 (May 8, 2000)
  - J2SE 1.4 (February 6, 2002)
  - J2SE 5.0 (September 30, 2004)
  - Java SE 6 (December 11, 2006)
  - Java SE 7 (July 28, 2011
  - Java SE 8 (18th Mar 2014)
  - Java SE 9 (21st Sep 2017)
  - Java SE 10 (20th Mar 2018)

Cont..

- Java Editions.
  - **J2SE**(Java 2 Standard Edition) - to develop client-side standalone applications or applets.
  - **J2ME**(Java 2 Micro Edition) - to develop applications for mobile devices such as cell phones.
  - **J2EE**(Java 2 Enterprise Edition) - to develop server-side applications such as Java servlets and Java ServerPages.

Introduction to CORE JAVA

- A **universal object-oriented** language.
- Write **Once,** Run Anywhere (WORA).
- Designed for **simple Internet and Internet** applications.
- **Wide** acceptance

Introduction to CORE JAVA

Introduction:

- **Java** was developed by ***Sun Microsystems*** (which is now the subsidiary of Oracle) in the year **1995**.
- ***James Gosling*** is known as the father of Java.
- Before Java, its name was ***Oak***.
- Java is a High level **programming language** and a **platform independent.**
- **Java** is a **robust**, **object-oriented** and **secure** programming language.

Introduction to CORE JAVA

Introduction: Java Platforms and Editors

There are 4 platforms or editions of Java:

- **Java SE (Java Standard Edition)**
  - It is a Java programming platform and it includes Core topics like OOPs, String, Exception, etc..
- **Java EE (Java Enterprise Edition)**
  - It is an enterprise platform which is mainly used to develop web and enterprise applications
- **Java ME (Java Micro Edition)**
  - It is a micro platform which is mainly used to develop mobile applications.
- **JavaFX**
  - It is used to develop rich internet applications. It uses a light-weight user interface API.

Introduction to CORE JAVA

Introduction:

- **Java** was developed by ***Sun Microsystems*** (which is now the subsidiary of Oracle) in the year **1995**.
- ***James Gosling*** is known as the father of Java.
- Before Java, its name was ***Oak***.
- Java is a High level **programming language** and a **platform independent.**
- **Java** is a **robust**, **object-oriented** and **secure** programming language.

Introduction to CORE JAVA

Java is an object-oriented, class-based programming language.

The language is designed to have as few dependencies implementations as possible.

The term WORA, write once and run everywhere is often associated with this language. It means whenever we compile a Java code, we get the byte code (.class file), and that can be executed (without compiling it again) on different platforms provided they support Java.

Introduction to CORE JAVA

It is used for:

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has huge community support (tens of millions of developers)
- Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs

Terminologies in Java

JVM (Java Virtual Machine):

- JVM is the specification that facilitates the runtime environment in which the execution of the Java bytecode takes place.
- Whenever one uses the command *java,* an instance of the JVM is created.
- JVM facilitates the definition of the memory area, register set, class file format, and fatal error reporting. Note that the JVM is platform dependent.

Byte Code:

The Java compiler compiles the Java code to generate the .class file or the byte code. One has to use the *javac* command to invoke the Java compiler.

Classpath:

As the name suggests, classpath is the path where the Java compiler and the Java runtime search the .class file to load.

Terminologies in Java

Java Development Kit (JDK):
It is the complete Java Development Kit that encompasses everything, including JRE(Java Runtime Environment), compiler, java docs, debuggers, etc.
JDK must be installed on the computer for the creation, compilation, and execution of a Java program.

Java Runtime Environment (JRE):
JRE is part of the JDK. If a system has only JRE installed, then the user can only run the program.
In other words, only the *java* command works. The compilation of a Java program will not be possible (the *javac* command will not work).

Garbage Collector:
Programmers are not able to delete objects in Java. In order to do so, JVM has a program known as Garbage Collector.
Garbage Collectors recollect or delete unreferenced objects. Garbage Collector makes the life of a developer/ programmer easy as they do not have to worry about memory management.

Difference between Java and C

- **C Language:**
  - **The main** difference is that C is a **structured** language **whereas** Java is an **object-oriented** language **with a** mechanism **for defining** classes and objects.
  - Java does not support explicit pointer **types.**
  - Java does not have **a** preprocessor, so #define,
  - #include and #ifdef **statements cannot be used.**
  - Java does not **contain** structures, unions and enum **datatypes.**
  - Java does not include keywords like goto, sizeof and typedef.
  - Java adds labeled break and continue statements.
  - Java adds many features **needed** for **object-oriented** programming.

Difference between Java and C++

- C++ language
- Features removed in java:
  - Java **does not** support pointers to **prevent**
  - unauthorized access **to** memory locations.
  - Java does not **contain** structures, unions and enum
  - **datatypes.**
  - Java does not support operator **overloading. The preprocessor is completely deprecated in Java as it** plays **a** less important role in **C++.**
  - Java does not perform automatic type **conversion, so precision is lost.**

Cont…

- Java does not support **globals. All methods** and **variables are** declared within a class and **are** part of that class.
- Java does not **accept** default arguments.
- Java does not support multiple superclass subclass inheritance(i.e. multiple inheritance).

  This is **achieved** using **the "interface"** concept.
- **Unsigned integer cannot be declared** in **Java.**
- In **Java,** objects are passed by reference only. In **C++,** objects **can** be passed by value or **by** reference.

Cont …

New features added in Java:

- Multithreading, **which** allows two or more **parts** of the same program to **run simultaneously.**
- C++ has a set of library functions that **share** a common header file. **However, Java** replaces **this** with its own set of API classes. **Add**
- packages and interfaces.
- Java supports automatic garbage collection.
- **Improvements in Java to allow** break and continue statements to **target labels. Uses**
- **Unicode** characters **to provide** portability.

Cont …

Features that differ:

- C++ and **Java support the boolean datatype, but** C++ **accepts** any **non-zero** value as true and **0** as false. **In Java,** True and **False** are predefined literals that are values **for** boolean **expressions.**
- Java replaced the destructor function with **finalize().**
- C++ supports **Java-like** exception **handling.** However, in C++ there is no **need to catch the** thrown **exception.**

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

COMPILER	INTERPRETER
Compiler scans the entire program and translates the whole of it into machine code at once.	Interpreter translates just one statement of the program at a time into machine code.
A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.	An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower.
A compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed.	An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory.
A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder while working with a compiler.	Keeps translating the program continuously till the first error is confronted. If any error is spotted, it stops working and hence debugging becomes easy.
Compliers are used by programming languages like C and C++ for example.	Interpreters are used by programming languages like Ruby and Python for example.

DIFFERENCE BETWEEN APPLET & APPLICATION

APPLET	**APPLICATION**
Applets are small Java programs that are designed to be included with the HTML web document. They require a Java-enabled web browser for execution.	Applications are just like a Java programs that can be execute independently without using the web browser.
Applet does not require a main function for its execution.	Application program requires a main function for its execution.
Applets don't have local disk and network access.	Java application programs have the full access to the local file system and network.
Applets can only access the browser specific services. They don't have access to the local system.	Applications can access all kinds of resources available on the system.
Applets cannot execute programs from the local machine.	Applications can executes the programs from the local system.
An applet program is needed to perform small tasks or the part of it.	An application program is needed to perform some task directly for the user.

Characteristics of Java

- Java is simple
- Java is object-oriented
- Java is distributed
- Java is interpreted
- Java is robust
- Java is architecture-neutral
- Java is portable
- Java's performance
- Java is multithreaded
- Java is dynamic
- Java is secure

Implementation of Application Programs in Java

1. The program creation (writing the code)
2. The program compilation.
3. Executing the compiled code.

1. The program Creation-

- The Java program can be written using a Text Editor (Notepad++ or NotePad or other editors will also do the job.) or IDE (Eclipse, NetBeans, etc.).

FileName: TestClass.java

```
public class TestClass
{
// main method
public static void main(String []args)
{
// print statement
System.out.println("Hello World is my first Java Program.");
}
}
```

Write the above code and save the file with the name TestClass. The file should have the .java extension.

2. The program Compilation

- Open the command prompt, and type *javac TestClass.java*. *javac* is the command that makes the Java compiler come to action to compile the Java program.
- After the command, put the name of the file that needs to be compiled. In our case, it is *TestClass.java*. After typing, press the enter button.
- a TestClass.class file will be generated that contains the byte code. If there is some error in the program, the compiler will point it out, and *TestClass.class* will not be created.

3. Running / Executing the program

After the .class file is created, type *java TestClass* to run the program. The output of the program will be shown on the console, which is mentioned below.

Output:

Hello World is my first Java Program.

Various Keywords Used in Java

- **main() method:** The most important method of the program where the execution begins. Therefore, all the logic must reside in the main method. If the main() method is not containing the logic, then it will be there in some other method, but that method must be invoked from the main() method directly or indirectly.
- **class:** The keyword class is used for declaring class in the Java language.
- **void:** it means that the function or method will not be returning anything.
- **System.out.println():** It is used to print statements, patterns, etc., on the console.
- **String argvs[]:** It is a command line argument that is used for taking input.
- **public:** It is an access specifier keyword. When it is applied to a method, then that method is visible to all. Other access specifier keywords are, private, protected, and default.

Various Keywords Used in Java

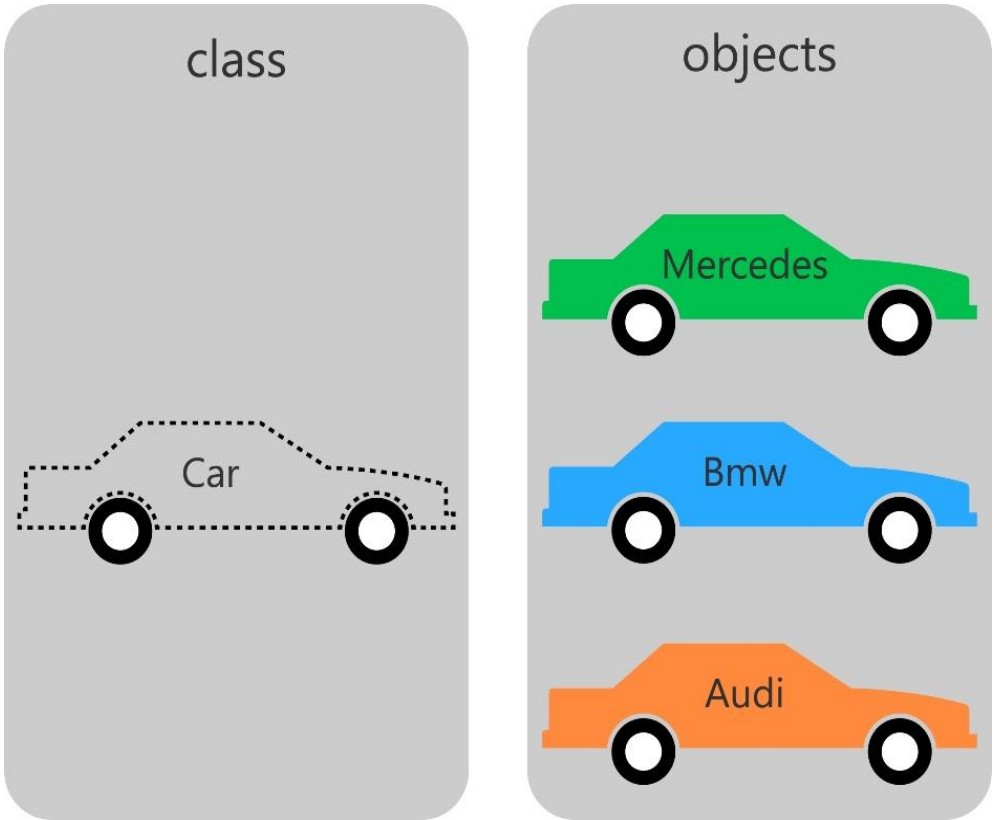
- **import java.io.*:** It means that all of the classes present in the package **java.io** is imported. The java.io package facilitates the output and input streams for writing and reading data to files. * means all. If one wants to import only a specific class, then replace the * with the name of the class.
- **System.in:** It is the input stream that is utilized for reading characters from the input-giving device, which is usually a keyboard in our case.
- **static void main():** The static keyword tells that the method can be accessed without doing the instantiation of the class.
- **System.out:** As System.in is used for reading the characters, System.out is used to give the result of the program on an output device such as the computer screen.
- **double, int:** The different data types, int for the integers, double for double. Other data types are char, boolean, float, etc.
- **println():** The method shows the texts on the console. The method prints the text to the screen and then moves to the next line. For the next line, ln is used.

Difference between JDK, JRE, and JVM

Class and Objects in OOPs

A Class is a kind of blue print or template that refer to the methods and attributes that will be in each object.

An Objects are the basic unit of OOP. They are instances of class, which have data members and uses various member functions to perform tasks.



Class Definition

Circle
-radius:double=1.0 -color:String="red"
+getRadius():double +getColor():String +getArea():double

Instances

c1:Circle	c2:Circle	c3:Circle
-radius=2.0 -color="blue"	-radius=2.0 -color="red"	-radius=1.0 -color="red"
+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()

Introduction to CORE JAVA

Basic Structure of Java program:

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

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility. It means it is visible to all.

Introduction to CORE JAVA

Basic Structure of Java program:

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

- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.

Introduction to CORE JAVA

Basic Structure of Java program:

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

- **main** represents the starting point of the program.
- **String[] args** is used for command line argument.
- **System.out.println()** is used to print statement. Here, **System** is a class, **out** is the object of **PrintStream** class, **println()** is the method of **PrintStream class**.

Introduction to CORE JAVA

Basic Structure of Java program:

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

To compile:javac Simple.java

To execute:java Simple

Introduction to CORE JAVA

Variable:

A variable is a container which holds the value while the program is executed. It is name of ***reserved area allocated*** *in memory*.

- There are three types of variables in Java:
  - local variable
  - instance variable
  - static variable

```
class A{
int data=50;                 //instance variable
static int m=100;            //static variable
void method()
{
int n=90;                    //local variable
}
}//end of class
```

Introduction to CORE JAVA

Variable:

Local Variable

- A variable declared inside the body of the method is called local variable.
- You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
- A local variable cannot be defined with "static" keyword.

Introduction to CORE JAVA

Variable:

Instance Variable

- A variable declared inside the class but outside the body of the method, is called instance variable. It is not declared as static.
- It is called instance variable because its value is instance specific and is not shared among instances.

Introduction to CORE JAVA

Variable:

Static variable

- A variable which is declared as static is called static variable.
- It cannot be local.
- You can create a single copy of static variable and share among all the instances of the class.
- Memory allocation for static variable happens only once when the class is loaded in the memory.

Introduction to CORE JAVA

Data Types in Java:

Data types specify the different sizes and values that can be stored in the variable.

There are two types of data types in Java:

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

Introduction to CORE JAVA

Data Types in Java:

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

Introduction to CORE JAVA

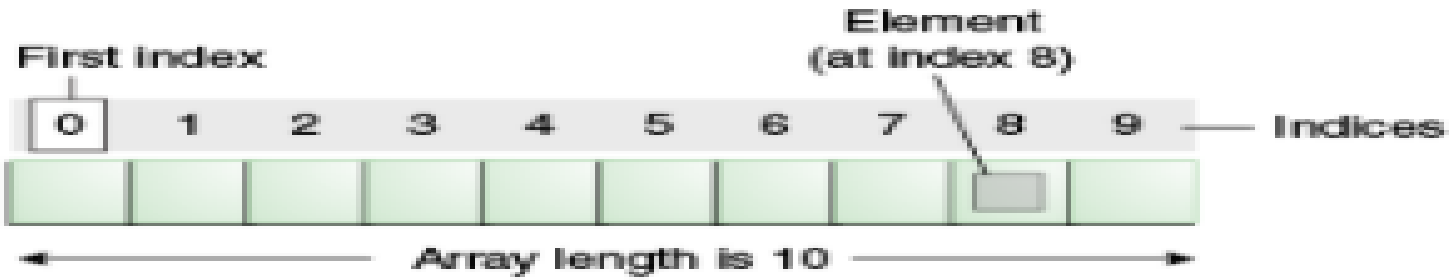
Operators in Java: Operator in Java is a symbol which is used to perform operations.

Operator Type	Category	Precedence
Unary	postfix	*expr++ expr--*
	prefix	*++expr --expr +expr -expr ~ !*
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Introduction to CORE JAVA

Arrays in Java:

- **Java array** is an object which contains elements of a similar data type.
- The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements.
- Array in Java is index-based,
  - 1st element of the array is stored at the 0th index,
  - 2nd element of the is stored on 1st index and so on.



Introduction to CORE JAVA

Types of Arrays in Java:

There are two types of array.

- Single Dimensional Array
  - Declaration of Array
    - dataType[] arr; ➔ int[] arr;
    - dataType []arr; ➔ char []arr;
    - dataType arr[]; ➔ double arr[];
  - Initialization of Array
    - array Var=new datatype[size]; ➔ array x=new int [5];

Introduction to CORE JAVA

Types of Arrays in Java:

- Multi-Dimensional Array
  - Declaration of Array
    - dataType[][] arrayRefVar; (or) ➔ int[][] X;
    - dataType [][]arrayRefVar; (or) ➔ char [][]Y;
    - dataType arrayRefVar[][]; (or) ➔ double z[][];
    - dataType []arrayRefVar[];
  - Initialization of Array
    - **int**[][] arr=**new int**[3][3];
- Jagged Array
  - If we are creating odd number of columns in a 2D array, it is known as a jagged array.
  - In other words, it is an array of arrays with different number of columns.

Introduction to CORE JAVA

Types of Arrays in Java:

Example of Jagged Array:

```
class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
int count = 0;
for (int i=0; i<arr.length; i++)
    for(int j=0; j<arr[i].length; j++)
        arr[i][j] = count++;
        //printing the data of a jagged array
        for (int i=0; i<arr.length; i++){
            for (int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}
```

Output:

```
0 1 2
3 4 5 6
7 8
```

Object Oriented Concepts

Object oriented programming aims to implement real world entities like inheritance, hiding, polymorphism etc in programming.

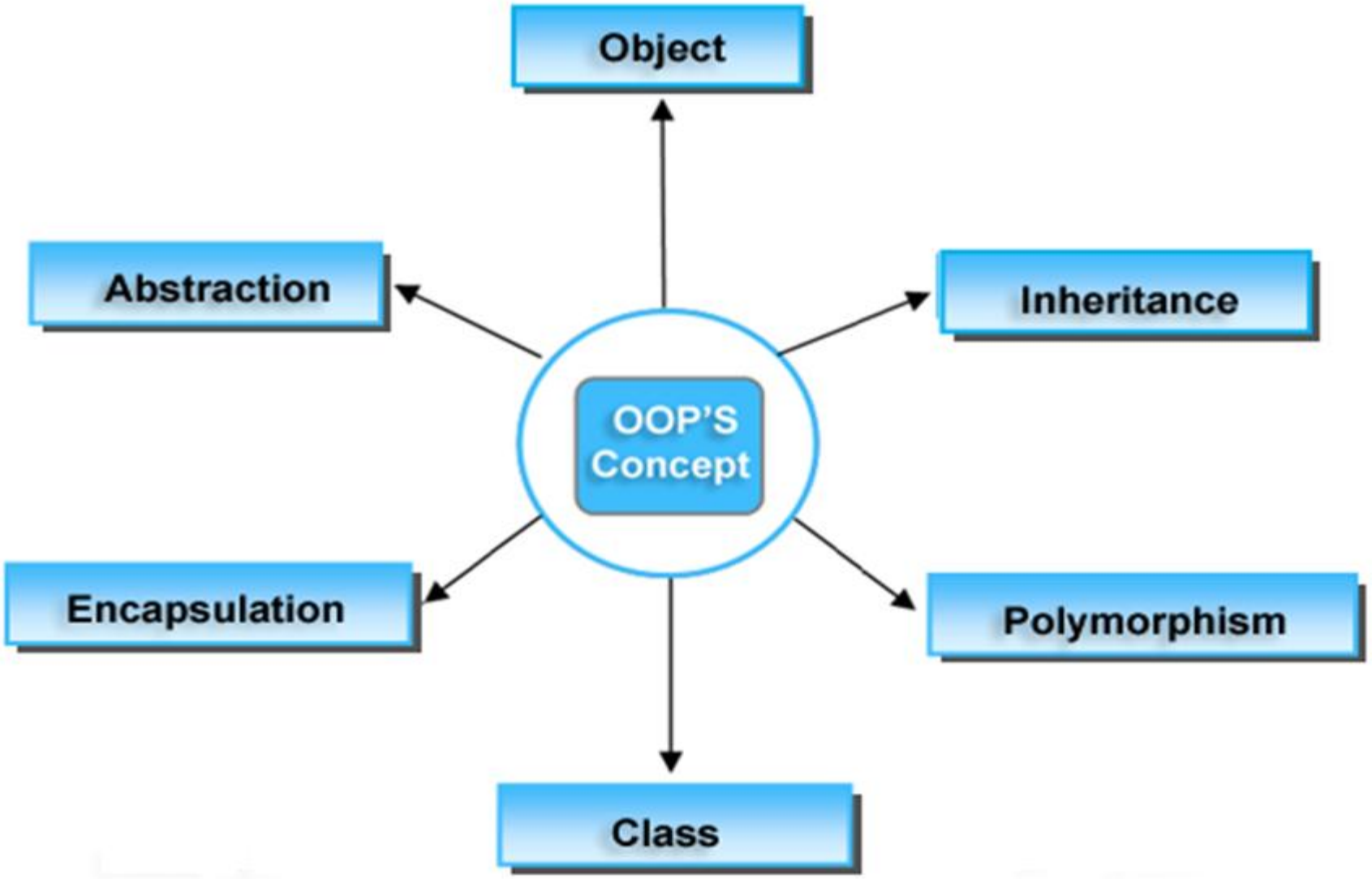
The main aim of OOP is to bind together the data and the functions that operates on them so that no other part of code can access this data except that function.

Technique used to develop programs revolving around the real world.

In OOPs every real life object has properties and behavior.

Cont…

Object Oriented Concepts



ACCESS MODIFIERS

- **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.

- **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.

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

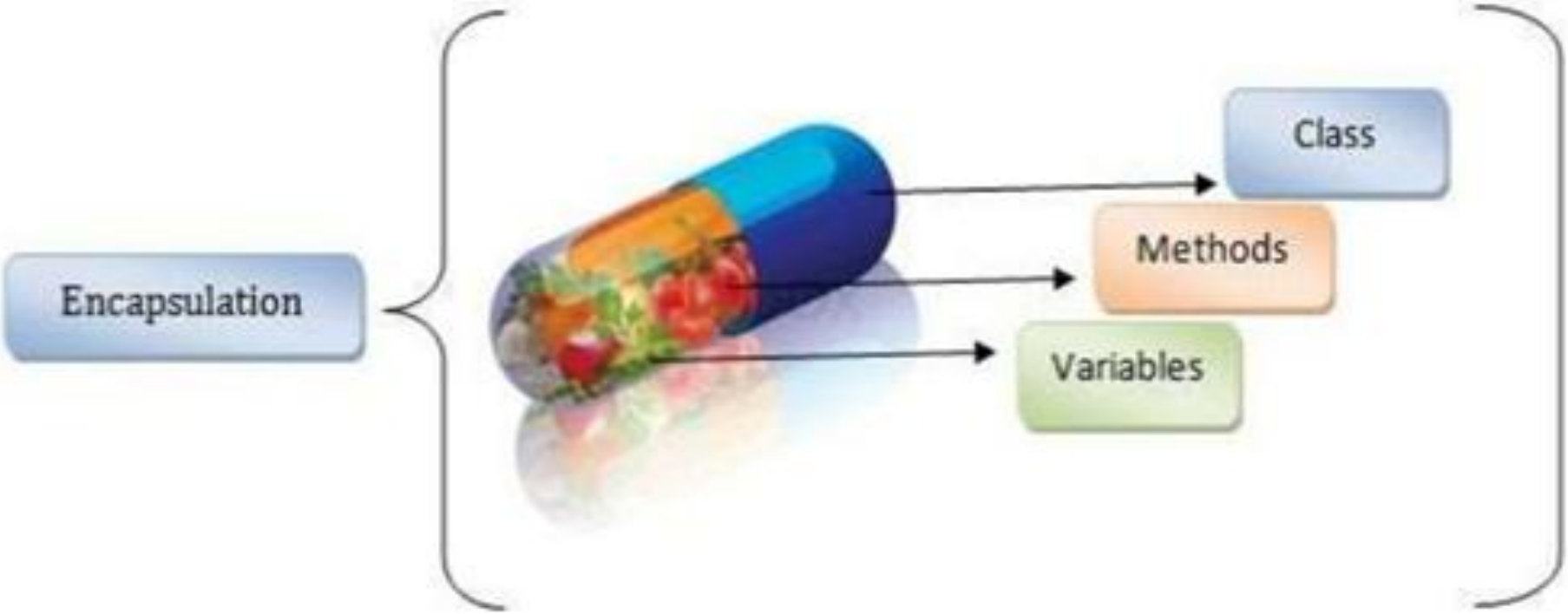
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Cont…

Encapsulation in OOPs

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

The **Java Bean** class is the example of a fully encapsulated class.



Encapsulation in OOPs

Example:-

```
public class Student{
//private data member
private String name;
//getter method for name
public String getName(){
return name;
}
//setter method for name
public void setName(String name){
this.name=name
}
}
```

Inheritance in OOPs

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

```
class Super
{
   ------
   ------
}

class Sub extends Super
{
  -------
  -------
  -------
}
```

Inheritance in OOPs

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.

It is used to **invoke the superclass** constructor from subclass.

Inheritance in OOPs

```
class Superclass
{
 int num = 100;
}
class Subclass extends Superclass {
int num = 110;
 void printNumber()
{
System.out.println(num);
}
public static void main(String args[])
{
Subclass obj= new Subclass();
obj.printNumber();
}
}
```

Output : 110

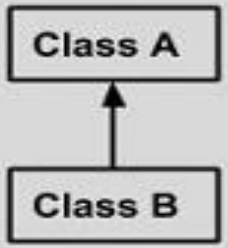
```
class Superclass
{
 int num = 100;
}
class Subclass extends Superclass {
int num = 110;
 void printNumber()
{
System.out.println(super.num);
}
public static void main(String args[])
{
Subclass obj= new Subclass();
obj.printNumber();
}
}
```

Output : 100

Inheritance in OOPs

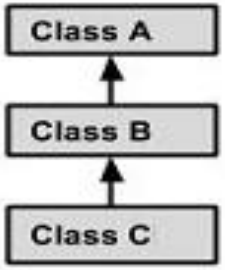
Types of Inheritance:-

Single Inheritance



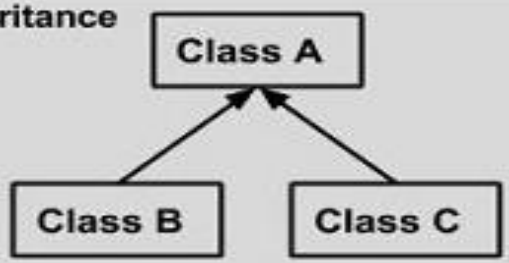

```
public class A {
   .....
}
public class B extends A {
   .....
}
```

Multi Level Inheritance



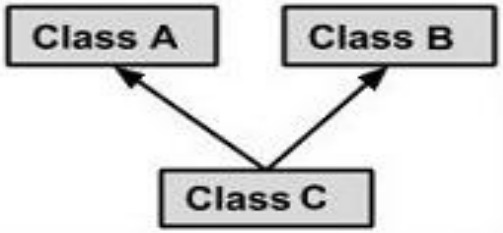

```
public class A { .....}
public class B extends A {.....}
public class C extends  B {..... }
```

Hierarchical Inheritance




```
public class A { .....}
public class B extends A {.....}
public class C extends A {..... }
```

Multiple Inheritance

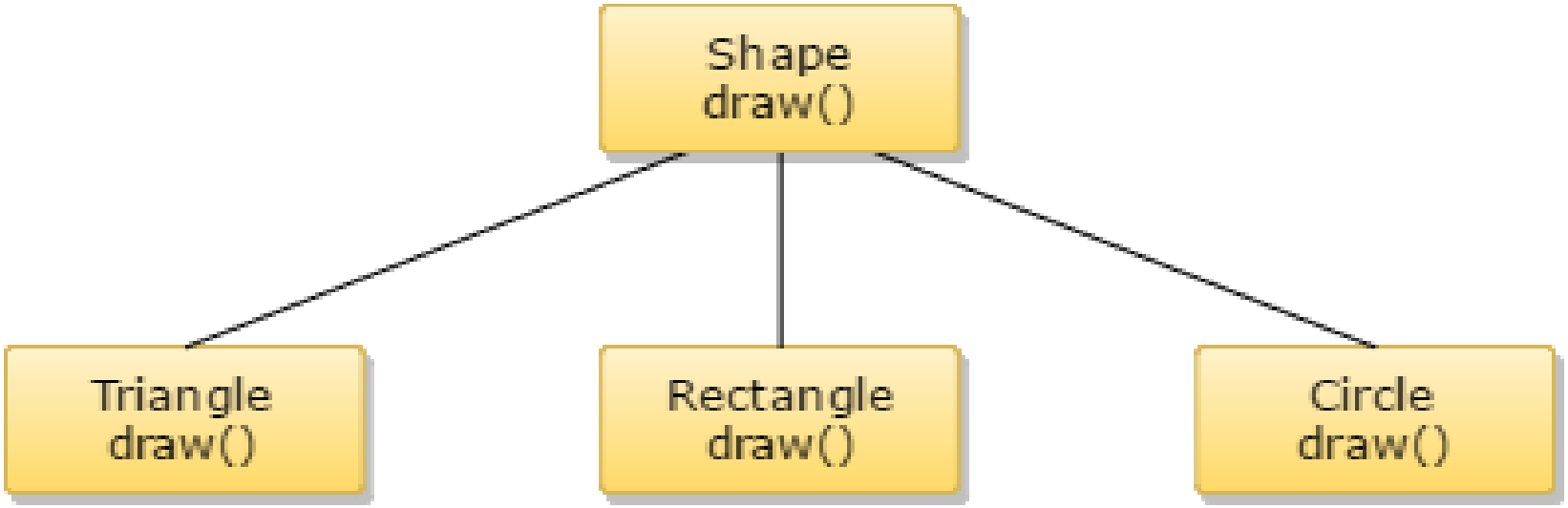



```
public class A { .....}
public class B {.....}
public class C extends  A,B {
   .....
} // Java does not support mutiple Inheritance
```

Polymorphism in OOPs

The ability of different objects to respond, each in its own way, to identical messages is called **Polymorphism.**

The most common use of polymorphism in OOP occurs when a parent class reference is used to refer a child class object.



Polymorphism

Polymorphism in OOPs

Method Overloading:

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type

```
class Adder{

static int add(int a,int b){return a+b;}

static int add(int a,int b,int c){return a+b+c;}

}

class TestOverloading1{

public static void main(String[] args){

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}}
```

Output:

```
22

33
```

Polymorphism in OOPs

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.

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).

Polymorphism in OOPs

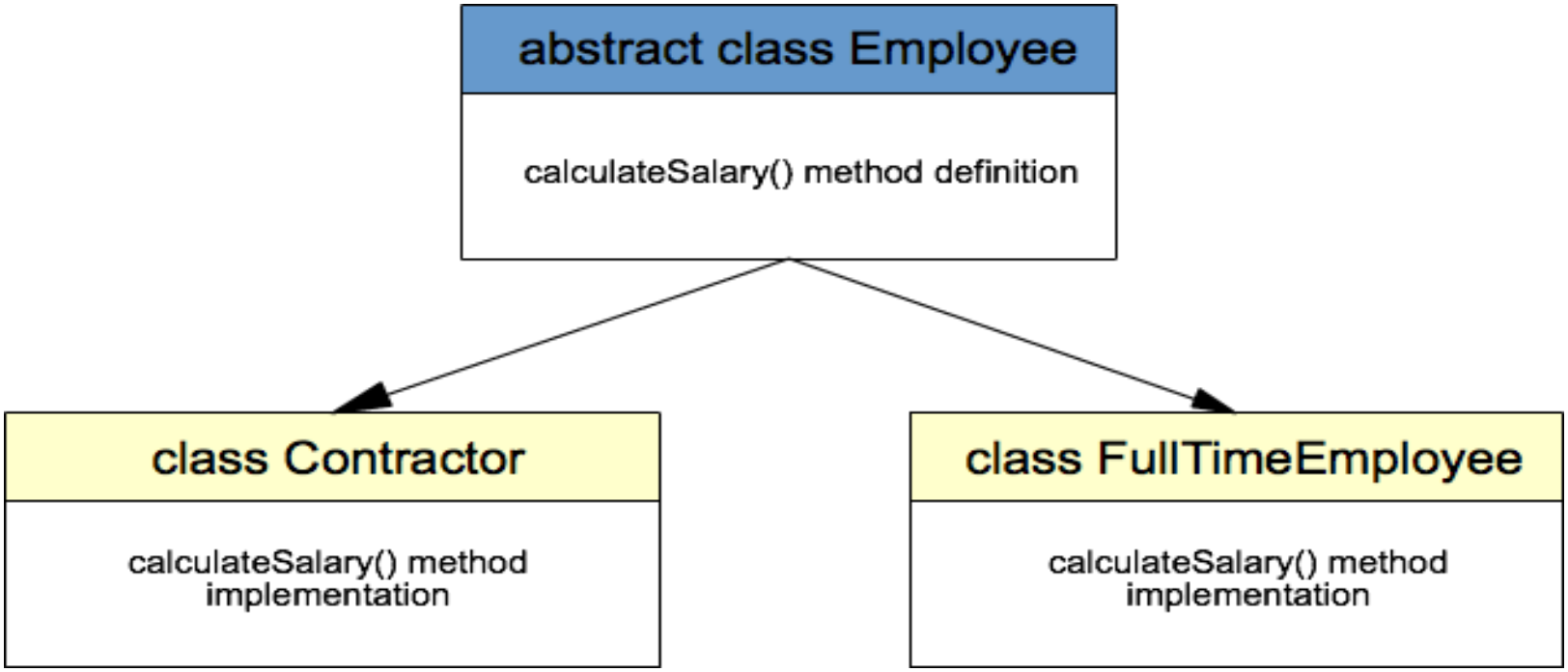
```
class Vehicle{

  void run(){
System.out.println("Vehicle is running");
}
}

class Bike2 extends Vehicle{
  void run()
{
System.out.println("Bike is running safely");
}

  public static void main(String args[]){
  Bike2 obj = new Bike2();
  obj.run();
  }
}
```

Output:

```
Bike is running safely
```

Abstraction in OOPs

Abstraction refers to the act of showing only those features that are relevant to the specific user.



Cont…

Abstraction in OOPs

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods.*
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Cont…

Abstraction in OOPs

Abstract Method

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semi colon (;) at the end.

Cont…

Abstraction in OOPs

```
abstract class Sum
{
public abstract int sumOfTwo(int n1, int n2);
public abstract int sumOfThree(int n1, int n2, int n3);
public void disp()
  {
   System.out.println("Method of    class Sum");
   }
}
class Value extends Sum
 {
  public int sumOfTwo(int num1, int num2)
  {
   return num1+num2;
  }
public int sumOfThree(int num1, int num2, int num3)
{
return num1+num2+num3;
}
public static void main(String args[])
{
 Sum obj = new Value();
System.out.println(obj.sumOfTwo(3, 7));
System.out.println(obj.sumOfThree(4, 3, 19));
 obj.disp();
}
}
```

Interfaces

- An **interface** is a reference type in **Java**.
- It is a collection of methods (abstract and public in default) and variables (public, static and final).
- A class implements an **interface**, thereby inheriting the abstract methods of the **interface**.
- The **interface** can not be instantiated.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
- A class can implement any number of interfaces.
- A interface can inherit multiple interfaces.
- A class cannot implement two interfaces that have methods with same name but different return type.
- An interface cannot contain a constructor (as it cannot be used to create objects)

Cont…

Interfaces

Syntax :

interface <interface_name> {

// declare constant fields

// declare methods that abstract by default.

}

Note: *All the methods in interface are declared with empty body and are public and all fields are public, static and final by default.*

Cont…

Interfaces

Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction.

Cont…

Interfaces

Accessing interfaces:

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

Cont…

```
interface MyInterface {
public  void method1();   //default abstract method
}
Interface I2{
Method2();
}

 class Demo implements MyInterface
{
public void method1()
{
System.out.println("implementation of method1");
 }
 public void method2()
{
 System.out.println("implementation of method2");
}
 public static void main(String arg[])
{
Demo obj = new Demo(); //object instantiated
MyInterface obj1 = new Demo(); // reference variable
obj.method1();
obj.method2();
obj1.method1();
obj1.method2(); //error
}
 }
```

Cont…

Abstract Class v/s Interfaces

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Exception Handling

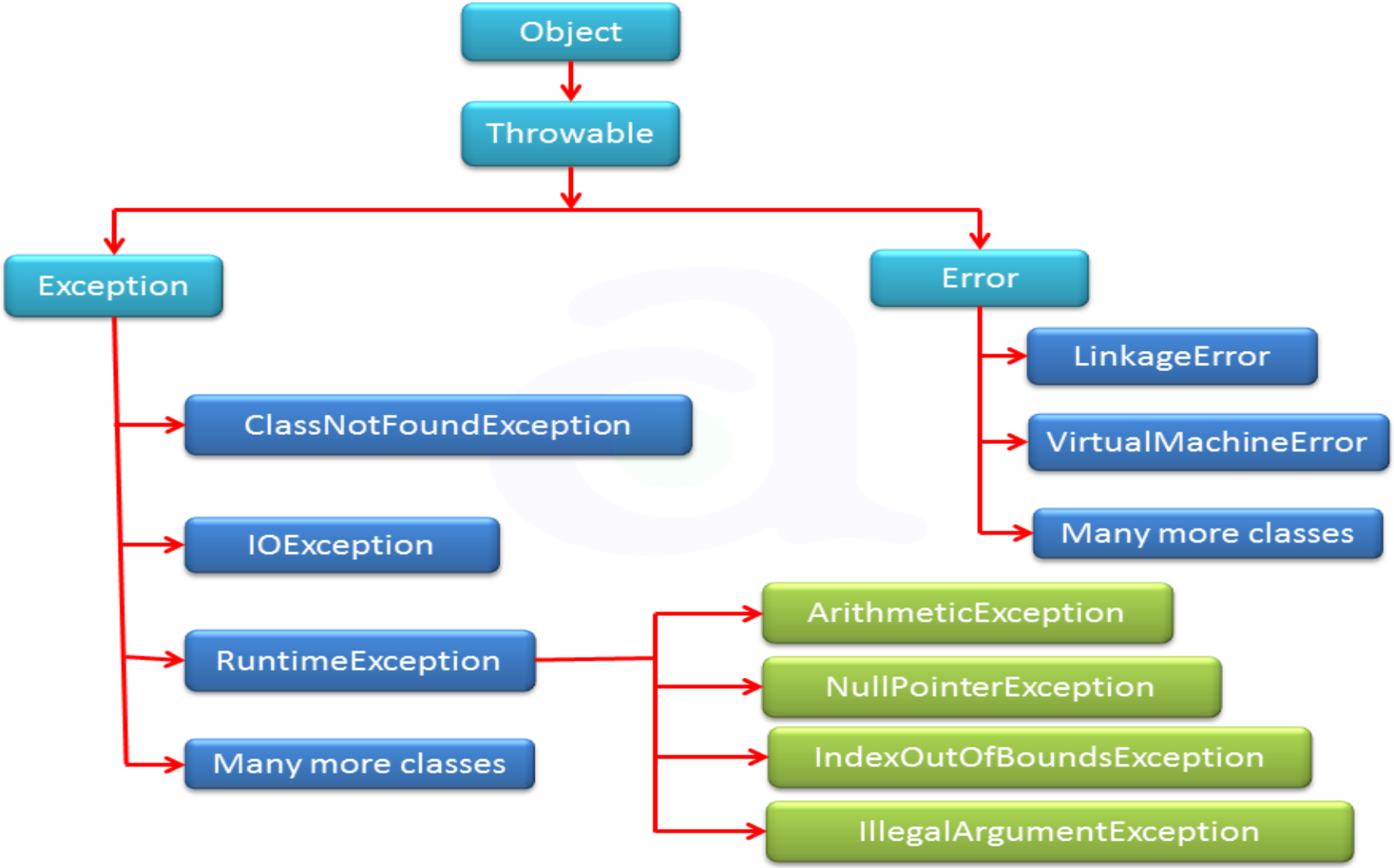
Exception in Java are any abnormal, unexpected events or extraordinary conditions that may occur at runtime.

Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing – often changing the normal flow of program execution.

Java Exception handling is used to handle error conditions in a program systematically by taking the necessary action.

Exception Handling

Exception Handling Hierarchy



Exception Handling

Exception Handling Categories:-

- **Checked:** **Checked** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using ***throws*** keyword.

- **Unchecked:** **Unchecked** are the exceptions that are not checked at compiled time. In Java exceptions under ***Error*** and ***RuntimeException*** classes are unchecked exceptions, everything else under **throwable** is checked.

Exception Handling

Java Exception Keywords:-

- **try:** The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.

- **catch:** The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

- **finally:** The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.

- **throw:** The "throw" keyword is used to throw an exception.

- **throws:** The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Example: throw

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
    validate(13);
    System.out.println("rest of the code...");
  }
}
```

Example: throws

```
import java.io.IOException;

class Testthrows1
{
  void m()throws IOException
{
    throw new IOException("device error");
//checked exception
  }

 void n()throws IOException
{
    m();
  }
  void p()
{
   try
     {
       n();
      }
catch(Exception e)
  {
System.out.println("exception handled");
  }
}
  public static void main(String args[])
{
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Example: finally

```
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
}
catch(Exception e)
{
System.out.println(e);
}
finally{System.out.println("finally block is executed");}
}
}
```

Exception Handling

Difference between “throw” and “throws”

Throw	Throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Ex:
void m(){
throw new ArithmeticException("sorry");
}
	Ex:
	void m()**throws** ArithmeticException{
	//method code
	}

Exception Handling

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException

Threads

A thread is a:

- Facility to allow multiple activities within a single process
- Referred as lightweight process
- A thread is a series of executed statements
- Each thread has its own program counter, stack and local variables
- A thread is a nested sequence of method calls
- Its shares memory, files and per-process state

Threads

Need of thread and Use:-

- To perform asynchronous or background processing
- Increases the responsiveness of GUI applications
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities

Threads

Threads LifeCycle

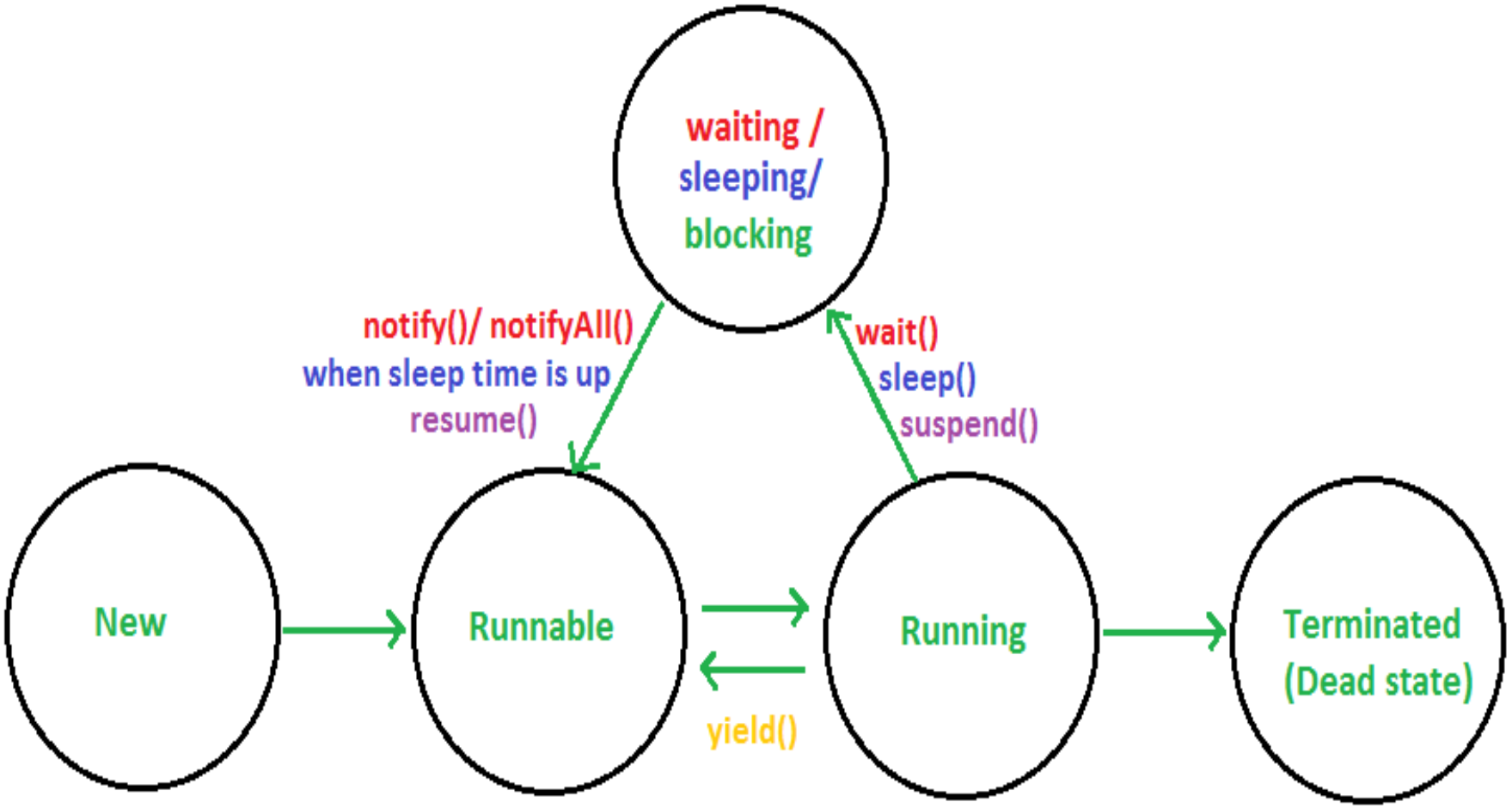


Fig. THREAD STATES

Threads (Life Cycle)

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Threads

Thread creation in Java

- Thread implementation in java can be achieved in two ways:
- Extending the **java.lang.Thread** class
- Implementing the **java.lang.Runnable** Interface

Note: The Thread and Runnable are available in the java.lang.* package

Threads

By extending thread class

- The class should extend Java **Thread** class.
- The class should override the **run()** method.
- The functionality that is expected by the Thread to be executed is written in the **run()** method.

void start(): Creates a new thread and makes it runnable.

void run(): The new thread begins its life inside this method.

Threads

By Implementing Runnable interface

- The class should implement the **Runnable** interface.
- The class should implement the **run()** method in the Runnable interface.
- The functionality that is expected by the Thread to be executed is put in the **run()** method.

Threads

By extending thread class

```
public class MyThread extends Thread
{
 public void run()
{
System.out.println("thread is running...");
}
 public static void main(String[] args)
 {
MyThread obj = new MyThread();
obj.start();
}
```

By Implementing Runnable interface

```
public class MyThread implements Runnable
{
 public void run()
{
System.out.println("thread is running..");
}
public static void main(String[] args)
{
Thread t = new Thread(new MyThread());
t.start();
}
```

Threads

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException

- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Threads

```
class TestSleepMethod1 extends
Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{
            Thread.sleep(500);
        }
     catch(InterruptedException e)
       {
         System.out.println(e);
        }
         System.out.println(i);
 }
}
```

```
public static void main(String args[])
{
  TestSleepMethod1 t1=new TestSleep
Method1();

  TestSleepMethod1 t2=new TestSleep
Method1();

  t1.start();
  t2.start();
 }
}
```

Threads

Extends Thread class vs Implements Runnable Interface?

- Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
- If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

Threads

Commonly used methods in thread class:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void suspend():** is used to suspend the thread(deprecated).
- **public void resume():** is used to resume the suspended thread(deprecated).
- **public void stop():** is used to stop the thread(deprecated).

public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

Threads

Can we start a thread twice ?

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

OutPut:

Exception in thread "main" java.lang.IllegalThreadStateException

Constructors in Java

Constructor is a block of code that initializes the newly created object.

A **constructor** resembles an instance method in **java** but it's not a method as it doesn't have a return type.

Constructor has same name as the class and looks like this in a **java** code.

Every time an object is created using the new() keyword, at least one **constructor** is called.

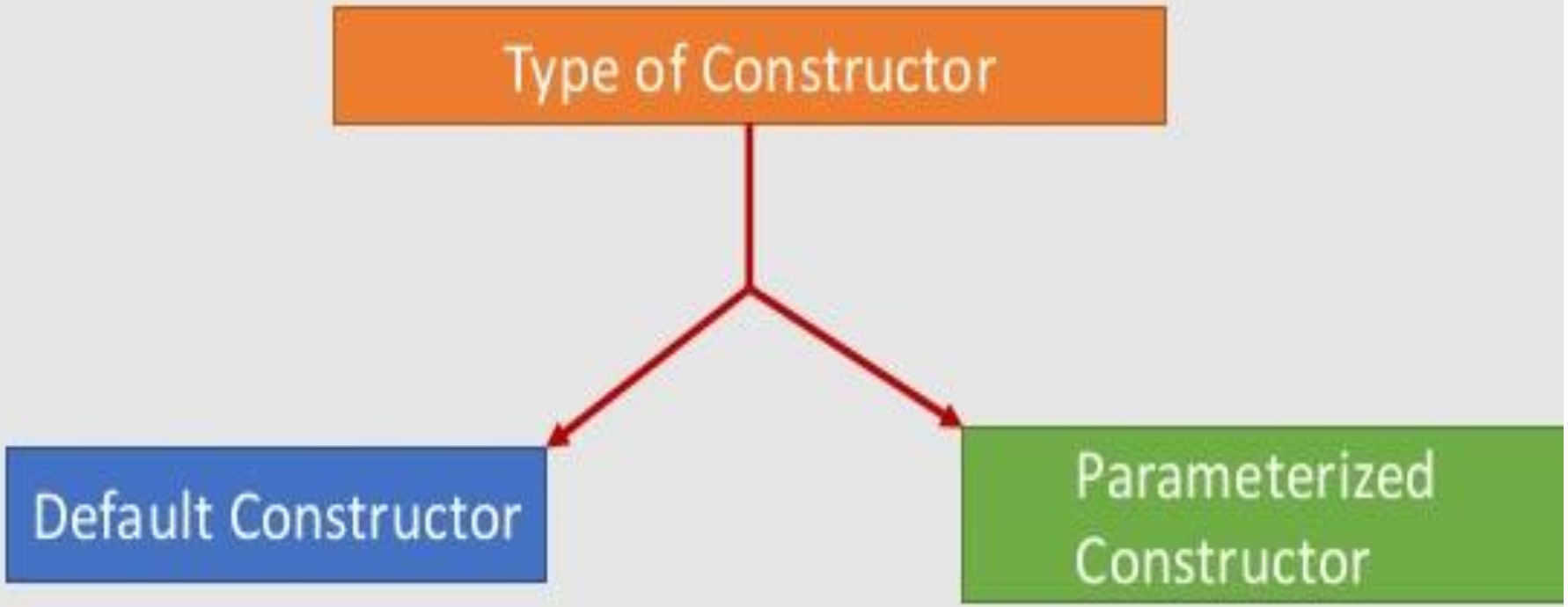
Rules for Constructor

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

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Constructors

- There are two Types of Constructors:
  - Default constructor (no-arg constructor)
  - Parameterized constructor



Default Constructor

A constructor is called "**Default Constructor**" when it doesn't have any parameter.

Syntax of default constructor:

<Constructor Name Same as Class name>()
{ }

```
class Myconstructor
{
 Myconstructor() //creating a default constructor
{
System.out.println(“Constructor created");
}
public static void main(String args[]){
Myconstructor  b=new  Myconstructor();  //calling a default constructor
}
}
```

Default Constructor

```
class Student{
int id;
String name;
void display(){
System.out.println(id+" "+name);
}
public static void main(String args[]){
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
} }
```

NOTE: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Parameterized Constructor

- A constructor which has a specific number of parameters is called a parameterized constructor.
- The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Parameterized Constructor

```
class Student{
    int id;
    String name;
    Student(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Constructor v/s Method

Constructor	Method
Constructor is used to initialize the state of an object	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor	Method is not provided by compiler in any case.
Constructor name must be same as the class name	Method name may or may not be same as class name.

THANK YOU