Object Oriented Programming

Procedural (or Structural) Programming

- The basic **principle** of the structured programming approach is to divide a program into functions and modules that perform specific tasks. The use of modules and functions makes the program code cleaner , more understandable and readable.
- This approach is also known as the **top-down approach**. A top-down approach begins with high level design and ends with low level design or development.
- Data is global, and all the functions can access global data i.e. data move openly around the system from function to function. The basic **drawback** of the procedural programming approach is that data is not secured because data is global and can be accessed by any function.
- This approach gives importance to functions rather than data. It focuses on the development of medium to large software applications, for example, C was used for modern operating system development.
- It does not model real world problems and its applications really well as there are no OOP features like Inheritance , Data hiding and encapsulation , etc.

Object Oriented Programming

*Object Oriented programming (OOP) is a programming paradigm that relies on the concept of **classes and objects**. It is used to structure a software program into simple, reusable pieces of code blueprints (called classes), which are used to create individual instances (called objects).*

This approach is very close to the real-world because the state and behavior of these classes and objects are almost the same as real-world entities. OOP programming languages: C++ and JAVA.

***AIM** :* "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 operate on them so that no other part of the code can access this data except that function ."

Note: **Object-based programming** approach is the approach which primarily supports only few OOP concepts like data encapsulation , data hiding, function overloading and objects. This approach don't implement inheritance and dynamic binding. Eg of OBP language : Ada programming language.
Thus Object Oriented Programming = Object based Programming + Inheritance + Dynamic Binding.

Note: **Purely Object Oriented Programming** languages are those languages in which everything accessible(both predefined types and user defined types and even functions) are objects. Languages like C++ and Java are thus not pure OOP languages because they have primitive data types like int , double which are not objects. Example of Pure OOP languages are python, ruby, dart. Etc.

Key Points

1. OOP treats data as a critical element.
2. Emphasis is on data rather than procedure.
3. Decomposition of the problem into simpler modules.
4. Doesn’t allow data to freely flow in the entire system, i.e. localized control flow.
5. Programs are divided into classes and their objects.
6. Data is hidden/protected from external functions.
7. Objects may communicate with each other by functions.
8. It is an Bottom Up Approach. A bottom-up approach begins with low level design or development and ends with high level design.

Advantages of OOP

- *Real World Model & Applications*: It models the real world very well.

- *Maintainability*: With OOP, programs are easy to test, debug, understand and maintain , thus complex software programs are easy to manage.
- *Data Security*: Principle of Data Hiding helps to build secure programs that cannot be invaded by code in other parts of programs.
- *Reusability*: OOP offers code reusability. Already created classes or their features can be reused using inheritance without having to write them again.
- *Fast Development*: OOP facilitates the quick development of programs where parallel development of classes and partition of work is possible and thus higher productivity.
- *Scalability*: Object Oriented Systems can be easily upgraded from small to large systems.

Disadvantages of OOP

1. **Larger program size:** Object-oriented programs typically involve more lines of code than procedural programs.
2. **Slower programs:** Object-oriented programs are typically slower than procedurebased programs, as they typically require more instructions to be executed.
3. **Complex Programs and High Skills:** It is very complex to create programs based on the interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be a big challenging to comprehend initially.

Features (or Pillars) of OOP

a. **Data Abstraction**
b. **Data Encapsulation**
c. **Inheritance**
d. **Polymorphism**
e. **Dynamic Binding**
f. **Message Passing**

Applications of OOP

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext, hypermedia and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

COMPARISON BETWEEN POP & OOP

BASIS	**Procedural**	**Object-Oriented**
Approach	Top-down.	Bottom-up.
Basis	Main focus is on the procedure or structure of a program .	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Entity accessing	No access specifier observed.	Access specifier are "public", "private", "protected".

mode		
Overloading or Polymorphism	Neither it overload functions nor operators.	It overloads functions, constructors, and operators.
Inheritance	There is no provision of inheritance.	Various types of Inheritance can be implemented.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected, hence data security increases.
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Abstract Class & Interface	No concept of abstract classes or interface.	Abstract Classes & Interface available in Java, (pure virtual function in C++).
Example	C, VB, FORTRAN, Pascal	C++, Java, C#, Python.

Class & Object

Class
"Class is an user-defined data type, which holds its own data members (or instance variables) and member functions (or methods), which can be accessed and used by creating an instance of that class."

A class is like a blueprint for an object. The primary purpose of the class is to store data and information. Data members & member functions of a class define the properties & behavior of the objects in a class.

In general, class declarations can include these components, in order:

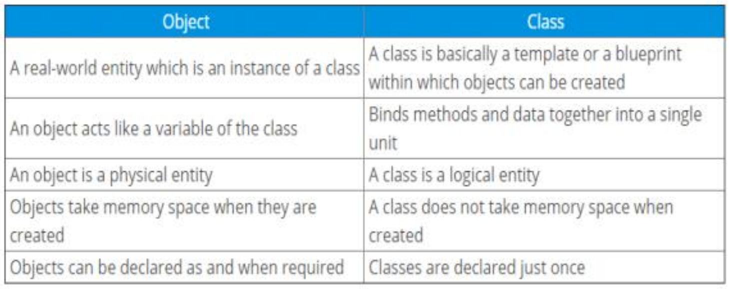
- **Modifiers**: A class can be public or has default access.
- **class keyword**: class keyword is used to create a class.
- **Class name**: The name should begin with a capital letter.
- **Superclass**: The name of the class's parent (superclass), if any, preceded by the keyword ***extends***. A class can only extend (subclass) one parent.
- **Interfaces**: A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword ***implements***. A class can implement more than one interface.
- **Body**: The class body surrounded by braces, { }.

<u>Note</u>: *"Memory space for objects is allocated when they are declared and not when class is specified"*.
This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of class specification. Thus only space for data members is allocated separately every time object is declared (and memory for code of member functions gets already allocated only once for one class).

Special Characteristics of member functions of a class :
1. Several different classes can use the same function name.
2. Member functions can access the private data of the class. A non-member function cannot do so.
3. A member function can call another member function directly without using dot (.) operator.

Object
" An Object is an identifiable entity with some characteristics and behavior. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated."

A class is the blueprint of the object, but also, we can say the implementation of the class is the object. The class is not visible to the world, but the object is.



Object	Class
A real-world entity which is an instance of a class	A class is basically a template or a blueprint within which objects can be created
An object acts like a variable of the class	Binds methods and data together into a single unit
An object is a physical entity	A class is a logical entity
Objects take memory space when they are created	A class does not take memory space when created
Objects can be declared as and when required	Classes are declared just once

<u>Real World Example of Class & Object</u>
- Consider an *ATM*. An ATM is a class. It's machine which is pretty much useless until you insert your debit card. After you insert your debit card, the machine has information about you and your bank account and the balance in it, so at this point it is an object.
- You have used a class and created an object, now you can perform operations on it like withdrawal of money or checking you balance or getting statement of your account, these operations will be methods belonging to that class (ATM) but you cannot use them until you create an object out of it.
- And when you did perform whatever operation you wanted to perform and clicked exit/cancel and removed your card, you just destroyed the object. Now it is not an object, it has methods and all (the functions an ATM can perform) but you cannot use them until you insert your card again and create an object.

<u>Creating Objects of a Class</u>
There are three steps involved in creating a object:
1. Declaration of an object reference (by default, it will hold ***null*** reference, until it is not initialized)
2. Memory allocation of object (instance variables) in heap memory
3. Initializing reference with the object.

3. Initializing reference with the object.

$$\overbrace{\text{ClassName objectName}}^{1} \underbrace{=}_{3} \overbrace{\text{new ClassName();}}^{2}$$

The **new** operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

objectName is just a reference to the object memory in heap, not the actual object itself (unlike primitive datatype). Naming convention for object is same as naming convention for any variable.

ClassName should start with a capital letter(according to the naming conventions). Example) System, String, Integer, etc.

In Java, memory for an object is always allocated on heap. Only primitive datatypes (int, boolean, char, float, double, etc) are allocated space on stack memory. Only the reference to the object, which holds the address of the actual object, is stored in stack memory.

Java Inner Class

Java inner class or nested class is a class that is declared inside the class or interface. We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable. Additionally, it can access all the members of the outer class, including private data members and methods.

Advantages or Need of Inner Class

1. Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
2. Code Optimization: Nested classes are used to develop more **readable and maintainable code** because it logically group classes and interfaces in one place only.

Types of Inner Class

1. **Member Inner Class:**
   A non-static class that is created inside a class but outside a method is called member inner class. It is also known as a regular inner class. It can be declared with access modifiers like public, default, private, and protected. Nested Inner class can access any private instance variable of outer class.

   Example:
   ```
   class Outer {
     int outerVar = 5;
     class Inner {
        int innerVar = 10;
     }
   }
   class Main {
     public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
     }
   }
   ```

   Note: We can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself.

2. **Local Inner Class:**
   Inner class can be declared within a method of an outer class. Local inner class can't be marked as private, protected, static but can be marked as abstract and final, but not both at the same time.

   Example:
   ```
   class Outer {
      void outerMethod() {
         class Inner {...}
   ```

```
        Inner y = new Inner();
    }
}
class Main{
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Note: Method Local inner classes can't use local variable of outer method until that local variable is not declared as final. The main reason we need to declare a local variable as a final is that local variable lives on stack till method is on the stack but there might be a case the object of inner class still lives on the heap.

3. **Anonymous Inner Class:**
   Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

   It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

   a. **Class (may be abstract or concrete)**
      Example:
```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
    void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

   a. **Interface**
      Example:
```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

4. **Static Nested Class:**
   Static nested classes are not technically an inner class. They are like a static member of outer class.

   Example:
```
class Outer {
   private static void outerMethod() {
     System.out.println("inside outerMethod");
   }

   static class Inner {
     public static void main(String[] args) {
        System.out.println("inside inner class Method");
        outerMethod();
     }
```

```
  }
}
```

Constructor

A constructor is a special method for every class, which is used to *construct* the values of instance variables at the time of object creation. It is called when an object instance of the class is created. At the time of calling constructor, memory for the object is allocated in the heap memory.

Every time an object is created using the new keyword, a constructor is called. It calls a *default constructor* (no-arg constructor) if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default. 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 of Defining a 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.
4. Constructor should be public. (Although constructor can be made private, but we will not be able to create an object using new keyword in that case. Private constructors are used in specific situations to limit the number of instances of a class).

Difference between Constructor and Method

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

Constructor Overloading

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Types of Constructors
There are three types of constructors in Java:

1. **Default constructor** (no-arg constructor):
   A constructor with zero arguments is known as default constructor. If no constructor is written explicitly, then compiler provides a default constructor only.
   Although, we can define our own default constructors also. Custom default constructor is used to provide the default values to the instance variables of the object like 0, null, etc., depending on the type.

   Example:
```
import java.lang.*;
import java.util.*;

class Pepcoding{
    public int students;
    public Pepcoding()
    { students = 0; }
}

class Program{
    public static void main(String args[])
    {
        Pepcoding obj = new Pepcoding();
        // Calling Custom Default Constructor

        System.out.println(obj.students);
    }
}
```

2. 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.

Example:

```
import java.lang.*;
import java.util.*;

class Pepcoding{
    public int students;
    public Pepcoding(int students)
    { this.students = students; }
}

class Program{
    public static void main(String args[])
    {
        Pepcoding obj = new Pepcoding(50);
        // Calling Parameterized Constructor

        System.out.println(obj.students);
    }
}
```

3. Copy constructor

A copy constructor is a special type of parameterized constructor where there is only one argument in the argument list, which is of the type of same class only. It is used to copy the values from another object (argument) into the current object (this).

Example:

```
import java.lang.*;
import java.util.*;

class Pepcoding{
    public int students;
    public Pepcoding(int students)
    { this.students = students; }
    public Pepcoding(Pepcoding obj)
    { this.students = obj.students; }
}

class Program{
    public static void main(String args[])
    {
        Pepcoding obj = new Pepcoding(50);
        // Calling Parameterized Constructor

        Pepcoding copy = new Pepcoding(obj);
        // Calling Copy Constructor
```

```
            System.out.println(copy.students);
        }
    }
```

Very Important Note: If the programmer provides atleast one custom constructor in the class, whether it is default or parameterized, compiler will *not create* it's own default constructor in that case.

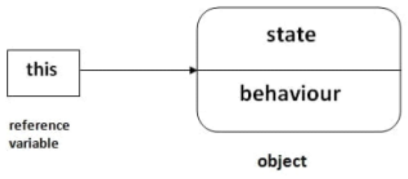
Hence, if we are defining a parameterized constructor, then it is a good practice to also define a default constructor, otherwise, there will be no constructor available with 0 arguments.

Example:

```
import java.lang.*;
import java.util.*;

class Pepcoding{
    public int students;
    public Pepcoding(int students)
    { this.students = students; }
}

class Program{
    public static void main(String args[])
    {
        Pepcoding obj = new Pepcoding();
        // Trying to Call Default Constructor, but the constructor provided by Compiler
        // will not be created since, there is a parameterized constructor.
        // It will give: Constructor Undefined Exception

        System.out.println(copy.students);
    }
}
```

this keyword

In Java, this is a **reference variable** that refers to the current object.



Uses of this keyword

- this can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

  Example:
  ```
  import java.lang.*;
  import java.util.*;
  class Pepcoding{
      public String student;
      public int rollNo;
      Pepcoding(String student, int rollNo){
          student = student;
          rollNo = rollNo;
      }
  }
  class Program{
      public static void main(String args[])
      {
          Pepcoding obj = new Pepcoding("Ram", 25);
          System.out.println(obj.student + ',' + obj.rollNo);
      }
  }
  ```

  It will print null, 0. To resolve the ambiguity, we can modify the constructor by using this keyword:

  ```
  import java.lang.*;
  import java.util.*;
  class Pepcoding{
      public String student;
      public int rollNo;
      Pepcoding(String student, int rollNo){
          this.student = student;
          this.rollNo = rollNo;
      }
  }
  class Program{
      public static void main(String args[])
      {
          Pepcoding obj = new Pepcoding("Ram", 25);
          System.out.println(obj.student + ',' + obj.rollNo);
      }
  }
  ```

  Now, it will print Ram, 25.

- this can be used to invoke current class method or constructor (implicitly). If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

  Example:
  ```
  import java.lang.*;
  import java.util.*;
  class Pepcoding{
      public String student;
      public int rollNo;
      Pepcoding(String student, int rollNo){
          this.setRollNo(rollNo);
          this.setStudent(student);
      }
      void setStudent(String student)
      {
          this.student = student;
  ```

```
    }
    void setRollNo(int rollNo)
    {
        this.rollNo = rollNo;
    }
}
class Program{
    public static void main(String args[])
    {
        Pepcoding obj = new Pepcoding("Ram", 25);
        System.out.println(obj.student + ',' + obj.rollNo);
    }
}
```

- this() can be used to invoke current class constructor. It can be used in constructor chaining. For Eg, it can be used to call parameterized constructor from default constructor.

  Example:

```
import java.lang.*;
import java.util.*;
class Pepcoding{
    public int rollNo;
    Pepcoding()
    { this(1); }
    Pepcoding(int rollNo)
    {
        this.rollNo = rollNo;
    }
}
class Program{
    public static void main(String args[])
    {
        Pepcoding obj = new Pepcoding();
        System.out.println(obj.rollNo);
    }
}
```

- this can be passed as an argument in the method or constructor call.
- this can be used to return the current class instance from the method.

static keyword

static keyword is used for memory management in Java. The static keyword belongs to the class itself rather than an instance of the class. The static keywords can be applied to instance variables, methods, a section/block of code in class, or nested class.

Static variable

- The static variable can be used to refer to the *common property* of all objects (which is not unique for each object).
- Hence, all instances (objects) of the class share the same static variable.
- The static variable gets memory only once in the class area at the time of *class loading* by JVM.
- Advantages of static variable is that it makes your program memory efficient (i.e., it saves memory).

Example:
```
import java.lang.*;
import java.util.*;
class Pepcoding{
    public static int students = 50;
}
class Program{
    public static void main(String args[])
    {
        Pepcoding obj1 = new Pepcoding();
        Pepcoding obj2 = new Pepcoding();
        System.out.println(Pepcoding.students);
        System.out.println(obj1.students);
        System.out.println(obj2.students);
    }
}
```

All three print statements will give 50, as all instances will share same students variable. Also, we can access the static variable using dot (.) operator on class itself.

Static method

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example:
```
import java.lang.*;
import java.util.*;
class Pepcoding{
    public static int students = 50;
    public static int calcBatches()
    { return students/10; }
}
class Program{
    public static void main(String args[])
    {
        Pepcoding obj1 = new Pepcoding();
        Pepcoding obj2 = new Pepcoding();
        System.out.println(Pepcoding.calcBatches());
        System.out.println(obj1.calcBatches());
        System.out.println(obj2.calcBatches());
    }
}
```

Q) Why is the Java *main* method static?
Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

Static block

- Is used to initialize the static data members and data members marked as *final*.
- It is executed before the main method at the time of class loading.

Example:
```
import java.lang.*;
import java.util.*;
class Pepcoding{
    public static final int students;
    static{
        students = 50;
    }
}
class Program{
    public static void main(String args[])
    {
        Pepcoding obj1 = new Pepcoding();
        Pepcoding obj2 = new Pepcoding();
        System.out.println(Pepcoding.students);
        System.out.println(obj1.students);
        System.out.println(obj2.students);
    }
}
```

Inheritance

"The capability of a class (child class) to derive properties and characteristics from another class (parent class) is called Inheritance."
When we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class which possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

Real world example of inheritance can be *humans*. We inherit certain properties from the class 'Human' such as the ability to speak, breathe, eat, drink, etc.
We can also take the example of *cars*. The class 'Car' inherits its properties from the class 'Automobiles' which inherits some of its properties from another class 'Vehicles'.

Limitations or Disadvantages of inheritance
- Increases the time and effort required to execute a program as it requires jumping back and forth between different classes
- Any modifications to the program would require changes both in the parent as well as the child class
- Needs careful implementation else would lead to incorrect results
- Inherited functions work slower than normal function as there is indirection.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child cla sses.

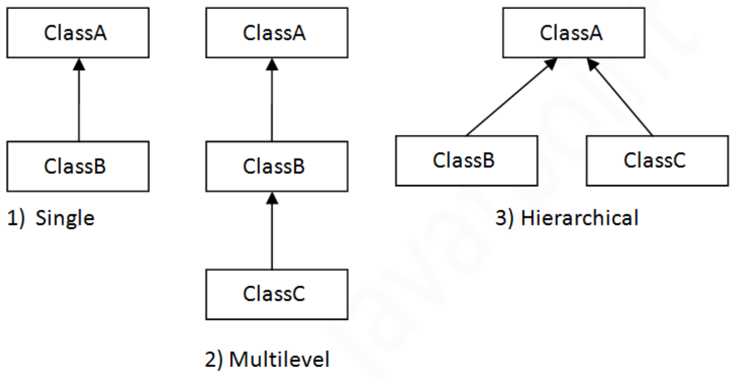
Advantages or Need of Inheritance

- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of inhe rited class.
- Reusability enhanced reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.

The **extends** keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Terms used in Inheritance
- **Sub Class:** The class that inherits properties from another class is called **Derived Class** or Sub class. It can be defined by specifying its relationship with the base class(es) in addition to its own details.
- **Super Class:** The class whose properties are inherited by sub class is called **Base Class** or Super class.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and m ethods of the existing class.

Types of Inheritance in Java



1. **Single Level Inheritance**: When a class inherits another class, it is known as a single inheritance.
   Example: Pepcoder **IS-A** Programmer. Hence, class Pepcoder can extend class Programmer, with some extra functionality like batch details, attendance, etc.

```
import java.lang.*;
import java.util.*;
class Programmer{
    public String language = "Java";
}
class Pepcoder extends Programmer{
    public String batch = "NADOS1";
};
```

```
class Program{
    public static void main(String args[])
    {
        Pepcoder obj = new Pepcoder();
        System.out.println(obj.language);
        System.out.println(obj.batch);
    }
}
```

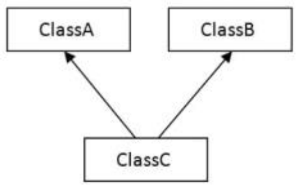
2. **Multilevel Inheritance**: When there is a chain of inheritance, it is known as multilevel inheritance.
   Example: Teaching Assistant can extend Pepcoder, because teaching assistant is a pepcoder, and Pepcoder was already extending Programmer.

```
import java.lang.*;
import java.util.*;
class Programmer{
    public String language = "Java";
}
class Pepcoder extends Programmer{
    public String batch = "JSP1";
};
class TeachingAssistant extends Pepcoder{
    public int doubtsTaken = 0;
    TeachingAssistant(int doubtsTaken)
    { this.doubtsTaken = doubtsTaken; }
};
class Program{
    public static void main(String args[])
    {
        TeachingAssistant obj = new TeachingAssistant(20);
        System.out.println(obj.language);
        System.out.println(obj.batch);
        System.out.println(obj.doubtsTaken);
    }
}
```

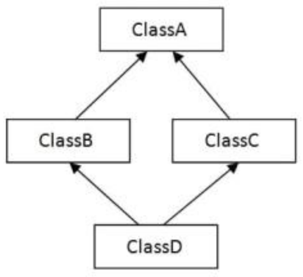
3. **Hierarchial Inheritance**: When two or more classes inherits a single class, it is known as hierarchical inheritance.
   Example: Animal can extended by Dog, Cat, Horse, etc.

```
import java.lang.*;
import java.util.*;
class Animal{
    public int size = 100;
}
class Dog extends Animal{
    void bark() {System.out.println("Barking.."); }
};
class Cat extends Animal{
    void meow() {System.out.println("Meow.."); }
};
class Program{
    public static void main(String args[])
    {
        Dog myDog = new Dog();
        Cat myCat = new Cat();
        System.out.println(myDog.size);
        myDog.bark();
        System.out.println(myCat.size);
        myCat.meow();

    }
}
```

Note: Java does not support ***Multiple inheritance***. However, there is a workaround to acieve similar functionality, using the power of *Interfaces.* Hence, in Java, one class is allowed to extend only one parent class. There cannot be more than one super classes of a sub class. Hence, inheritance of following kinds is **not** possible in Java(it is not allowed intentionally to avoid the ***Deadly Diamond of Death*** Problem):



4) Multiple



5) Hybrid

.

Constructors in Inheritance
Order of execution of constructors in multilevel inheritance is always from topmost class (superclass) to the bottom-most class (childclass). Please note that, even if we make an object of child class using new keyword, i.e. even if we call constructor of derived class first, this derived class constructor itself calls the base class constructor, in the first statement implicitly.

Hence, though first constructor called is of derived class, first constructor executed is of base class. This concept can be understood by the help of an example of building of a construction. Always, first, the foundation is laid, i.e. the base is created, then only the sky-touching floors are constructed. You cannot directly construct the floors on top without building all the floors below it.

If we want to pass arguments to base class constructor, i.e. call parameterized constructor of base class, then we can write super(argument list); as the first statement in the child class constructor. This will tell the compiler, that programmer have explicitly mentioned, which base class constructor to be called. If this statement is not written, then compiler always add super(); statement impliclty, thereby calling the default constructor of base class. Also, please remember that the super(); statement should be the first statement inside the constructor body. There must be no statement before it.

super Keyword
The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable. It is used if parent class and child class have same fie lds.
   Example:
```
import java.lang.*;
import java.util.*;
class Animal{
    String color = "Red";
}
class Dog extends Animal{
    String color = "Red";
    void getColor() { System.out.println(super.color); }
};
class Program{
    public static void main(String args[])
    {
        Dog myDog = new Dog();
        myDog.getColor();
    }
}
```

2. super can be used to invoke immediate parent class method. It should be used if subclass contains the same method as parent c lass. In other words, it is used if method is overridden.
   Example:
```
import java.lang.*;
import java.util.*;
class Animal{
    String color = "Red";
    void getColor() {System.out.println(this.color); }
}
class Dog extends Animal{
    String color = "Red";
    void getColor() { super.getColor(); }
};
class Program{
    public static void main(String args[])
    {
        Dog myDog = new Dog();
        myDog.getColor();
    }
}
```

3. super() can be used to invoke immediate parent class constructor.
   Example:
```
import java.lang.*;
import java.util.*;
class Animal{
    String color;
    public Animal() { color = "Red"; }
}
class Dog extends Animal{
    int size;
    public Dog()
    {
        super();
        size = 10;
    }
};
class Program{
    public static void main(String args[])
    {
        Dog myDog = new Dog();
        System.out.println(myDog.color);
    }
}
```

Java Compiler provides super() call in the constructor of subclass, implicitly, to call for the constructor of super class (constructor chaining), if not done explicitly by the programmer.

Important Note: Call to super(); must be the first statement of the subclass constructor. We cannot write any statement before calling super().

Polymorphism

"The word polymorphism means having many forms. polymorphism is the ability of a message to be displayed in more than one form."
Polymorphism is the ability of data to be processed in more than one form. It allows the performance of the same task in various ways. It consists of method overloading and method overriding, i.e., writing the method once and performing a number of tasks using the same method name.

Real world example of polymorphism can be a *girl*. She can be a daughter, mother, sister and and in all a human being.
Another example can be a *mobile phone*. The same mobile phone is used to take calls, click pictures & videos, run calculator and other applications, etc.

Advantages of Polymorphism:

- It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required. Saves a lot of time.
- Single variable can be used to store multiple data types.
- Easy to debug the codes.

Disadvantages of Polymorphism:

- Run time polymorphism can lead to the performance issue as machine needs to decide which method or variable to invoke so it basically degrades the performances as decisions are taken at run time.
- Polymorphism reduces the readability of the program. One needs to identify the runtime behavior of the program to identify actual execution time.

Polymorphism is extensively used in implementing inheritance. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. Polymorphism can be implemented using **Method Overloading** and **Method Overriding**.

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.

Method overloading is a technique to implement **compile-time polymorphism** in Java. A call invocation is binded with appropriate method at the compile-time only. This kind of method binding is known as static binding.

Example) Overloading addition method to handle addition of two integers versus two doubles, we can use same function name, thus improving readibility.

Rules for Writing Overloaded Methods

1. By changing number of arguments:

Example:
```
    int add(int a,int b){return a+b;}
    int add(int a,int b,int c){return a+b+c;}
```

1. By changing the data type

Example:
```
    int add(int a, int b){return a+b;}
    double add(double a, double b) {return a+b;}
```

Note: Method Overloading is not possible by changing the return type of the method only. Hence, two methods with same argument list, but different return type are not overloaded methods. Hence, writing such methods in same class will lead to ambiguity, thus method redefined compile-time error will be displayed.

Example:
```
    int add(int a, int b){return a+b;}
    long add(int a, int b) {return a+b;}
```

It will lead to add method redefined error, as both methods have exactly same argument list, but just different return type.

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. Method overriding is used to provide the specific implementation of a method which is already provided by its superclass. Method overriding can be used to implement **run-time polymorphism** by using the concept of dynamic/run-time binding.

Rules for Writing Overrided Method

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

Example:
```
import java.lang.*;
import java.util.*;
class Animal{
    void eat() { System.out.println("Eating"); }
}
class Dog extends Animal{
    void eat() { System.out.println("Eating Dog Food"); }
};
class Program{
    public static void main(String args[])
    {
        Dog myDog = new Dog();
        myDog.eat();
    }
}
```

It will print "Eating Dog Food" as overrided method eat() of class Dog will be called.

Note: We cannot override *static* methods because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area. Due to this reason, main method cannot be overrided.

Method Overloading vs Method Overriding

No.	**Method Overloading**	**Method Overriding**
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

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.

If the reference variable of Parent class refers to the object of Child class, it is known as ***upcasting***. Note that, parent need not be class, it can be interface also.

Example

class A{}

class B extends A{}

A a=new B();//upcasting

<u>Example of Run Time Polymorphism</u>

We are creating two classes Car and Alto. Alto class extends Car class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime. Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
import java.lang.*;
import java.util.*;
class Car{
    void run()
    { System.out.println("running"); }
  }
class Alto extends Car{
    void run()
    { System.out.println("run at maximum 100kmph"); }
}

class Program{
    public static void main(String args[])
    {
        Car b = new Alto();//upcasting
        b.run();  // Run Time Polymorphism or Dynamic Method Dispatch
    }
}
```

final keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context:

1. **Final Variable**
   If you make any variable as final, you **cannot change the value** of final variable(It will be constant).
   Example:
```
import java.lang.*;
import java.util.*;
class MyScience{
    public static final long speedOfLight = 3e8;
}
class Program{
    public static void main(String args[])
    {
        System.out.println(MyScience.speedOfLight);
    }
}
```

2. **Final Method**
   If you make any method as final, you **cannot override it**. (Final method can be inherited, but not overrided).
   Example:
```
import java.lang.*;
import java.util.*;
class MyMath{
    public long radius = 10;
    public final double getArea()
    {
        return Math.PI * radius * radius;
    }
}
class Program{
    public static void main(String args[])
    {
        MyMath obj = new MyMath();
        System.out.println(obj.getArea());
    }
}
```

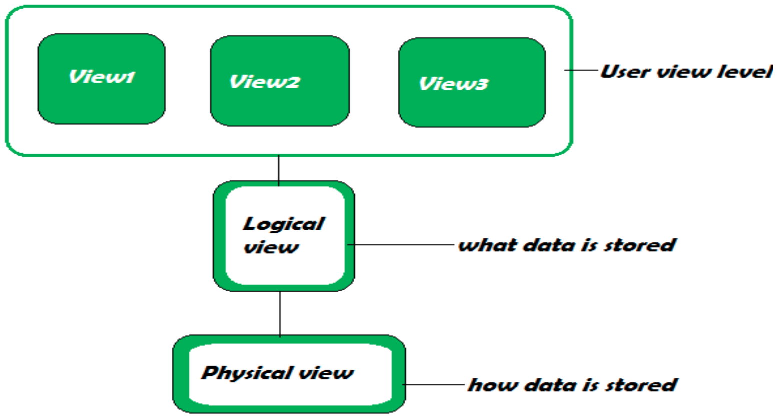
3. **Final Class**
   A final class **cannot be extended(inherited)**. Hence, final class is used to prevent inheritance. For example, all Wrapper Classes like Integer,Float etc. are final classes. We can not extend them.

   Example:
```
import java.lang.*;
import java.util.*;
final class MyMath{
    public long radius = 10;
    public final double getArea()
    {
        return Math.PI * radius * radius;
    }
}
class Program{
    public static void main(String args[])
    {
        MyMath obj = new MyMath();
        System.out.println(obj.getArea());
    }
}
```

Data Abstraction

"Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation."

Abstraction refers to the act of representing important and special features without including the background details or explanation about that feature. Data abstraction simplifies database design. Since classes implement Data Abstraction, hence classes are called *ABSTRACT DATATYPE (ADT)*.



a. **Physical Level:**
   It describes how the records are stored, which are often hidden from the user. It can be described with the phrase, "block of storage."

b. **Logical Level:**
   It describes data stored in the database and the relationships between the data. The programmers generally work at this level as they are aware of the functions needed to maintain the relationships between the data.

c. **View Level:**
   Application programs hide details of data types and information for security purposes. This level is generally implemented with the help of GUI, and details that are meant for the user are shown.

Advantages or Need of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

Real world example of Data Abstraction is *ATM Machine*: All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement…etc. but we can't know internal details about ATM.

Abstraction in Java
In Java, abstraction is mainly achieved using ***abstract classes and interfaces***.

Abstract Class
- An abstract class is a class that is declared with ***abstract*** keyword. Any class that contains one or more abstract methods must also be declared with abstract keyword.
- An abstract method is a method that is declared without implementation. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
- A method defined abstract must always be redefined in the subclass, thus making overriding compulsory. If we do not override abstract methods in subclass, then the subclass will also become abstract.
- There can be no object of an abstract class. That is, an **abstract class can not be directly instantiated with the new operator**.
- An abstract class can have parameterized constructors and default constructor is always present in an abstract class.

- Abstract classes are made to achieve **generalization**. For eg, if we want to create a super class Animals for all the animal types like dog, cat, etc. then we can make animal as abstract, because Animal itself is not a real world entity. Hence, we do not need any objects of class Animal.

Example:

```
import java.lang.*;
import java.util.*;
abstract class Animal{
    String color;
    Animal(String color)
    { this.color = color; }
    public abstract void makeSound();
}
class Dog extends Animal{
    Dog(String color)
    { super(color); }

    @Override
    public void makeSound()
    { System.out.println("Woof Woof .. "); }
}
class Cat extends Animal{
    Cat(String color)
    { super(color); }
    @Override
    public void makeSound()
    { System.out.println("Meow Meow .. "); }
}
class Program{
    public static void main(String args[])
    {
        Cat myCat = new Cat("Black");
        myCat.makeSound();
        Dog myDog = new Dog("White");
        myDog.makeSound();
    }
}
```

Interfaces

- Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
- All methods in an interface are abstract. Hence, interface acts as an abstract class with no concrete method.
- Interfaces specify what a class must do and not how. It is the blueprint of the class. So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- There are many interfaces present in Java Collections framework like Queue Interface, Comparator Interface, etc.
- To implement interface use ***implements*** keyword, instead of extends in inheritance. Apart from **total abstraction**, interfaces is also used to achieve behavior similar to ***multiple inheritance***, as more than one interfaces can be implemented in java, but more than one classes cannot be extended.

Example:

Consider interface *Shape*. We can implement Shape in different concrete classes like Rectangle, Circle, Triangle, etc.

```
import java.lang.*;
import java.util.*;
interface Shape
{
    void input(int param);
    void area();
}
class Circle implements Shape
{
    int radius = 0;
    final double PI = 3.14;
    @Override
    public void input(int radius)
    {
        this.radius = radius;
    }
    @Override
    public void area()
    {
        double area = PI * radius * radius;
        System.out.println("Area of circle : " + area);
    }
}
class Square implements Shape
{
    int side = 0;
    public void input(int side)
    {
        this.side = side;
    }
    public void area()
    {
        double area = side * side;
        System.out.println("Area of Square : " + area);
    }
}
class Program{
    public static void main(String args[])
    {
        Square obj = new Square();
        obj.input(5); obj.area();
        Circle obj2 = new Circle();
        obj2.input(5); obj2.area();
    }
}
```

Differences between Abstract Class and Interface

Abstract class	Interface
Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. (Note, it can have **default and static methods** also.)
Concrete class can inherit/extend only one class. Hence, abstract class **doesn't support multiple inheritance**.	Concrete class can implement multiple interfaces. Hence, Interface **supports multiple inheritance**.
Abstract class **can have final, non-final, static and non-static**	Interface has **only static and final variables**.

variables.	
The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Encapsulation

"Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them."

Encapsulation also leads to data abstraction and data hiding (as using encapsulation also hides the data). This concept is often used to hide the internal state representation of an object from the outside.

Encapsulation => Data Abstraction + Data Hiding

Q) How to achieve Encapsulation in Java ?
In Java, it can be implemented using Class Body and access specifiers. If you are creating class, you are doing encapsulation. Access specifiers plays an important role in implementing encapsulation in Java.

The process of implementing encapsulation can be sub-divided into two steps:
1. The data members should be labeled as private using the private access specifiers.
2. The member function which manipulates the data members (getters and setters) should be labeled as public using the public access specifier.

Note: Setters can be made public, as we can do *validations* on data or user inside the method body. So, there will be no risk to data security by making setters as public.

Real world example of Encapsulation is *medicine capsule*. All medicine ingredients are encapsulated inside a single capsule.

Methods
Variables
Class

Example
```
import java.lang.*;
import java.util.*;

class Pepcoder{
    private String language = "Java";
    private int batch = 0;
    public String getLanguage()
    { return language; }

    public int getBatch()
    { return batch; }
    public void setLanguage(String language)
    { this.language = language; }

    public void setBatch(int batch)
    { this.batch = batch; }
}
class Program{
    public static void main(String args[])
    {
        Pepcoder stud = new Pepcoder();
        stud.setBatch(10);
        System.out.println(stud.getBatch());
        stud.setLanguage("C++");
        System.out.println(stud.getLanguage());
    }
}
```

Need or Advantages of Encapsulation
- Encapsulation protects an object from unwanted access by clients.
- Encapsulation allows access to a level without revealing the complex details below that level.
- It reduces human errors and Makes the application easier to understand.
- Simplifies the maintenance of the application by organizing the code better.

Differences between Abstraction & Encapsulation

S.NO	Abstraction	Encapsulation
1.	Abstraction is the process or method of gaining the information.	While encapsulation is the process or method to contain the information.
2.	In abstraction, problems are solved at the design or interface level.	While in encapsulation, problems are solved at the implementation level.
3.	Abstraction is the method of hiding the unwanted information.	Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
4.	We can implement abstraction using abstract class and interfaces.	Whereas encapsulation can be implemented using by access modifier i.e.

		private, protected and public.
5.	In abstraction, implementation complexities are hidden using abstract classes and interfaces.	While in encapsulation, the data is hidden using methods of getters and setters.
6.	The objects that help to perform abstraction are encapsulated.	Whereas the objects that result in encapsulation need not be abstracted.

Multithreading

Thread: A single thread in Java is basically a lightweight and the smallest unit of processing. There are two types of thread – ***user thread and daemon thread*** (daemon threads are used when we want to clean the application and are used in the background, for eg, garbage collector in Java). When an application first begins, user thread is created, and afterwards, we can create many user threads and daemon threads. Advantage of using only single thread is that it reduces the overhead cost of the program, also reducing it's maintenance cost.

Multithreading: Multithreading in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications execute two or more threads run concurrently. Hence, it is also known as **Concurrency** in Java. Each thread runs parallel to each other. Mulitple threads don't allocate separate memory area, hence they save memory. Also, context switching between threads takes less time.

Benefits or Advantages of Multithreading

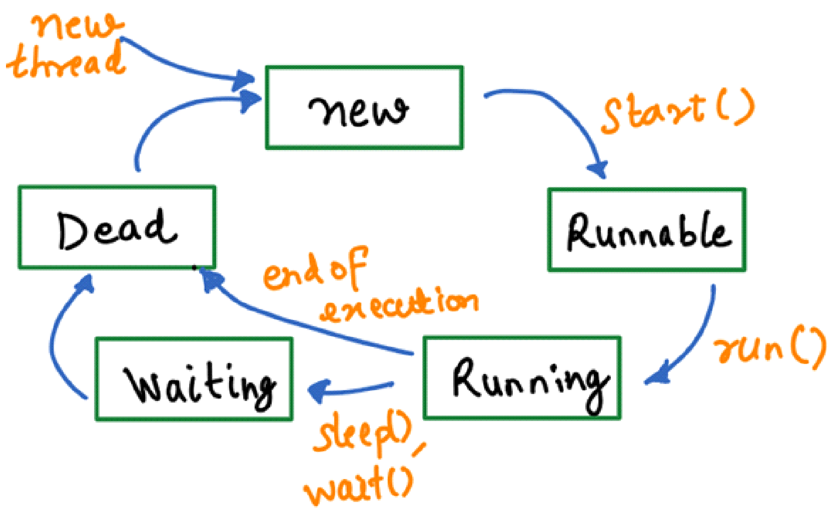
- Improved throughput. Many concurrent compute operations and I/O requests within a single process.
- Simultaneous and fully symmetric use of multiple processors for computation and I/O
- Superior application responsiveness. If a request can be launched on its own thread, applications do not freeze or show the "hourglass". An entire application will not block, or otherwise wait, pending the completion of another request.
- Improved server responsiveness. Large or complex requests or slow clients don't block other requests for service. The overall throughput of the server is much greater.
- Minimized system resource usage. Threads impose minimal impact on system resources. Threads require less overhead to create, maintain, and manage than a traditional process.
- Program structure simplification. Threads can be used to simplify the structure of complex applications, such as server-class and multimedia applications. Simple routines can be written for each activity, making complex programs easier to design and code, and more adaptive to a wide variation in user demands.
- Better communication. Thread synchronization functions can be used to provide enhanced process-to-process communication. In addition, sharing large amounts of data through separate threads of execution within the same address space provides extremely high-bandwidth, low-latency communication between separate tasks within an application.

Life Cycle of a Thread

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

There are various stages of life cycle of thread:

1. **New:** In this phase, the thread is created using class "Thread class".It remains in this state till the program **starts** the thread. It is also known as born thread.
2. **Runnable:** In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
3. **Running:** When the thread starts executing, then the state is changed to "running" state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
4. **Waiting:** This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
5. **Dead:** This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in "dead state".



LIFE CYCLE OF A THREAD

Threads can be created by using two mechanisms :

- **Extending the Thread class**

We can create a thread by creating a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1: We need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method –

public void run()

Step 2: Once Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method.

void start();

Example:

```
import java.lang.*;
import java.util.*;
class myThread extends Thread {
    private Thread t;
    private String threadName;

    myThread( String name) {
       threadName = name;
       System.out.println("Creating " +  threadName );
    }

    public void run() {
       System.out.println("Running " +  threadName );
    }

    public void start () {
       System.out.println("Starting " +  threadName );
       if (t == null) {
          t = new Thread (this, threadName);
          t.start ();
```

```
        }
    }
}

public class Program {

    public static void main(String args[]) {
       myThread T1 = new myThread( "Thread-1");
       T1.start();

       myThread T2 = new myThread( "Thread-2");
       T2.start();
    }
}
```

- **Implementing the Runnable Interface**

Step 1: We need to implement a ***run()*** method provided by a Runnable interface. This method provides an entry point for the thread and you will put your complete business logic inside this method.

public void run()

Step 2: Now, instantiate a **Thread** object using the following constructor:

Thread (Runnable threadObj , String threadName)

Here, *threadObj* is an instance of class that implements Runnable interface and *threadName* is the name given to the new thread.

Step 3: Once a Thread object is created, you can start it by calling ***start()*** method, which executes a call to run( ) method. Following is a simple syntax of start() method:

void start();

Example

```
import java.lang.*;
import java.util.*;
class myThread implements Runnable {
    private Thread t;
    private String threadName;

    myThread( String name) {
       threadName = name;
       System.out.println("Creating " +  threadName );
    }

    public void run() {
       System.out.println("Running " + threadName);
    }

    public void start () {
       System.out.println("Starting " +  threadName );
       if (t == null) {
          t = new Thread (this, threadName);
          t.start ();
```

```
      }
   }
}

public class Program {

   public static void main(String args[]) {
      myThread R1 = new myThread( "Thread-1");
      R1.start();

      myThread R2 = new myThread( "Thread-2");
      R2.start();
   }
}
```

Thread Methods

Following is the list of important methods available in the Thread class.

Method & Description
public void **start**() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
public void **run**() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
public final void **setName**(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
public final void **setPriority**(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
public final void **setDaemon**(boolean on) A parameter of true denotes this Thread as a daemon thread.
public final void **join**(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
public void **interrupt**() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
public final boolean **isAlive**() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

Method & Description
public static void **yield**() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
public static void **sleep**(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds.
public static boolean **holdsLock**(Object x) Returns true if the current thread holds the lock on the given Object.
public static Thread **currentThread**() Returns a reference to the currently running thread, which is the thread that invokes this method.
public static void **dumpStack**() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Thread Synchronization

- In multithreading, there is the asynchronous behavior of the programs. If one thread is writing some data and another thread which is reading data at the same time, might create inconsistency in the application.
- When there is a need to access the shared resources by two or more threads, then synchronization approach is utilized.
- Java has provided synchronized methods to implement synchronized behavior.
- In this approach, once the thread reaches inside the synchronized block, then no other thread can call that method on the same object. All threads have to wait till that thread finishes the synchronized block and comes out of that.
- In this way, the synchronization helps in a multithreaded application. One thread has to wait till other thread finishes its execution only then the other threads are allowed for execution.

```
synchronized (object)
{
    // block of statements to
    // be synchronized

}
```