Polymorphism:

It is the ability of an object to take on many forms. The most common use of polymorphism in OOP, occurs when a parent class reference is used to refer to a child class object.

//Animal,java

```
public class Animal{
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
//Horse.java
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
      Animal obj = new Horse();
      obj.sound();
    }
}
//Cat.java
public class Cat extends Animal{
    @Override
    public void sound(){
        System.out.println("Meow");
    }
    public static void main(String args[]){
      Animal obj = new Cat();
      obj.sound();
    }
}
```

Compile Time Polymorphism: or Static Binding

```
class Overload
{
    void demo (int a)
    {
       System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
       System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
       System.out.println("double a: " + a);
       return a*a;
    }
}
class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
        double result;
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

Static and dynamic binding
Static Binding that happens at compile time and **Dynamic Binding** that happens at runtime.

Static Binding:
The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is **compile-time**. we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overriden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it.

Dynamic Binding or Late Binding
When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. **Method Overriding** is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed.

The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens.

Static Binding Example:

```
class Human{
    public static void walk()
    {
        System.out.println("Human walks");
    }
}
class Boy extends Human{
    public static void walk(){
        System.out.println("Boy walks");
    }
    public static void main( String args[]) {
        /* Reference is of Human type and object is
         * Boy type
         */
        Human obj = new Boy();
        /* Reference is of HUman type and object is
         * of Human type.
         */
        Human obj2 = new Human();
        obj.walk();
        obj2.walk();
    }
}
```

Dynamic Binding:

```
class Human{
    //Overridden Method
    public void walk()
    {
        System.out.println("Human walks");
    }
}
class Demo extends Human{
    //Overriding Method
    public void walk(){
        System.out.println("Boy walks");
    }
    public static void main( String args[]) {
        /* Reference is of Human type and object is
         * Boy type
         */
        Human obj = new Demo();
        /* Reference is of HUman type and object is
         * of Human type.
         */
        Human obj2 = new Human();
        obj.walk();
        obj2.walk();
    }
}
```

Abstraction:

An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
        System.out.println("Woof");
   }
   public static void main(String args[]){
        Animal obj = new Dog();
        obj.sound();
   }
}
```

```
abstract class AbstractDemo{
   public void myMethod(){
      System.out.println("Hello");
   }
   abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

   public void anotherMethod() {
       System.out.print("Abstract method");
   }
   public static void main(String args[])
   {
      //error: You can't create object of it
      AbstractDemo obj = new AbstractDemo();
      obj.anotherMethod();
   }
}
```

we cannot instantiate an abstract class. This program throws a compilation error.

Output:
Unresolved compilation problem: Cannot instantiate the type AbstractDemo

```java
//Example of Abstract class and method

abstract class MyClass{
   public void disp(){
     System.out.println("Concrete method of parent class");
   }
   abstract public void disp2();
}

class Demo extends MyClass{
   /* Must Override this method while extending
    * MyClas
    */
   public void disp2()
   {
      System.out.println("overriding abstract method");
   }
   public static void main(String args[]){
      Demo obj = new Demo();
      obj.disp2();
   }
}
```

Output:

```
Output:
overriding abstract method
```

```java
//abstract class
abstract class Sum{
   /* These two are abstract methods, the child class
    * must implement these methods
    */
   public abstract int sumOfTwo(int n1, int n2);
   public abstract int sumOfThree(int n1, int n2, int n3);

   //Regular method
   public void disp(){
      System.out.println("Method of class Sum");
   }
}
//Regular class extends abstract class
class Demo extends Sum{

   /* If I don't provide the implementation of these two methods, the
    * program will throw compilation error.
    */
   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 Demo();
        System.out.println(obj.sumOfTwo(3, 7));
        System.out.println(obj.sumOfThree(4, 3, 19));
        obj.disp();
    }
}
```

Output:

```
10
26
Method of class Sum
```

Encapsulation:

The whole idea behind encapsulation is to hide the implementation details from users. Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

```
//EncapsTest.java
class EncapsulationDemo{
    private int ssn;
    private String empName;
    private int empAge;

    //Getter and Setter methods
    public int getEmpSSN(){
        return ssn;
    }

    public String getEmpName(){
        return empName;
    }

    public int getEmpAge(){
        return empAge;
    }

    public void setEmpAge(int newValue){
        empAge = newValue;
    }
```

```
public void setEmpName(String newValue){
        empName = newValue;
    }

    public void setEmpSSN(int newValue){
        ssn = newValue;
    }
}

public class EncapsTest{
    public static void main(String args[]){
        EncapsulationDemo obj = new EncapsulationDemo();
        obj.setEmpName("Mario");
        obj.setEmpAge(32);
        obj.setEmpSSN(112233);
        System.out.println("Employee Name: " + obj.getEmpName());
        System.out.println("Employee SSN: " + obj.getEmpSSN());
        System.out.println("Employee Age: " + obj.getEmpAge());
    }
}
```

Output:

```
Employee Name: Mario
Employee SSN: 112233
Employee Age: 32
```

`Interface:`

An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: Java abstract method). Also, the variables declared in an interface are public, static & final by default.

The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

```java
interface MyInterface
{
    /* compiler will treat them as:
     * public abstract void method1();
     * public abstract void method2();
     */
    public void method1();
    public void 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[])
     {
    MyInterface obj = new Demo();
    obj.method1();
     }
}
```

Interface and Inheritance

```
interface Inf1{
   public void method1();
}
interface Inf2 extends Inf1 {
   public void method2();
}
public class Demo implements Inf2{
   /* Even though this class is only implementing the
    * interface Inf2, it has to implement all the methods
    * of Inf1 as well because the interface Inf2 extends Inf1
    */
     public void method1(){
   System.out.println("method1");
     }
     public void method2(){
   System.out.println("method2");
     }
     public static void main(String args[]){
   Inf2 obj = new Demo();
  obj.method1();
   obj.method2();
     }
}
```

enum

Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester{

public static void main(String args[]){
Enumeration days;
Vector dayNames =newVector();
dayNames.add("Sunday");
 dayNames.add("Monday");
 dayNames.add("Tuesday");
 dayNames.add("Wednesday");
 dayNames.add("Thursday");
 dayNames.add("Friday");
 dayNames.add("Saturday");
 days = dayNames.elements();
while(days.hasMoreElements())
{ System.out.println(days.nextElement());
}
}
}
```