UNIT - I

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING (OOP) AND CLASSES

Introduction to Object Oriented Programming (OOP)

- OOPs refers to languages that use objects in programming, they use objects as a primary source to implement what is to happen in the code.
- OOP paradigms that use Objects, **which are instances of classes to model real world** entities and their interactions in software development.
- 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.

Object-Oriented Programming (OOP) Principles

- Object-oriented programming (OOP) is at the core of Java.
- OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.
- **Two Paradigms**
  - Process-Oriented Model
  - Object Oriented Programming

Class Fundamentals

- In the object-oriented programming technique, we design a program using **objects and classes.**
- A class is a blueprint or template for creating objects.
- It defines the attributes (properties) and methods (functions) that the objects will have.
- **An object** in Java is a **physical as well as a logical entity**, whereas, a **class** in Java is a **logical entity** only.
- **Object in Java**
- An object is a real-world entity which is the basic unit of OOPs for example Person, PC etc.
- **Different objects** have different states or attributes, and behaviors.
- An object is an instance of a class. It is a self-contained unit that encapsulates data and behavior.
-

An object has three characteristics:

- State: represents the data (value) of an object.
- Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- Identity: An object identity is typically implemented via a unique ID.

- An **object is an instance of a class**. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

- Object Definitions:

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

About sample programs descriptions

- First and foremost, Java is case sensitive. If you made any mistakes in capitalization (such as typing Main instead of main), the program will not run.
- The keyword **public** is called an access modifier; these modifiers control the level of access other parts of a program have to this code.
- The keyword **class** reminds you that everything in a Java program lives inside a class.
- Classes are the building blocks with which all Java applications and applets are built. Everything in a Java program must be inside a class.
- The standard naming convention (which we follow in the name **FirstSample**) is that class names are nouns that start with an uppercase letter.
- If a name consists of multiple words, use an initial uppercase letter in each of the words

About sample programs descriptions - Contd

- You need to make the file name for the source code the same as the name of the public class, with the extension .java appended. Thus, you must store this code in a file called FirstSample.java. (Again, case is important—don't use firstsample.java.)
- Then when you compile this source code, you end up with a file containing the bytecodes for this class.
- The Java compiler automatically names the bytecode file **Firstpgm.class** and stores it in the same directory as the source file.
- Finally, launch the program by issuing the following command:

  java Firstpgm

Simple Program

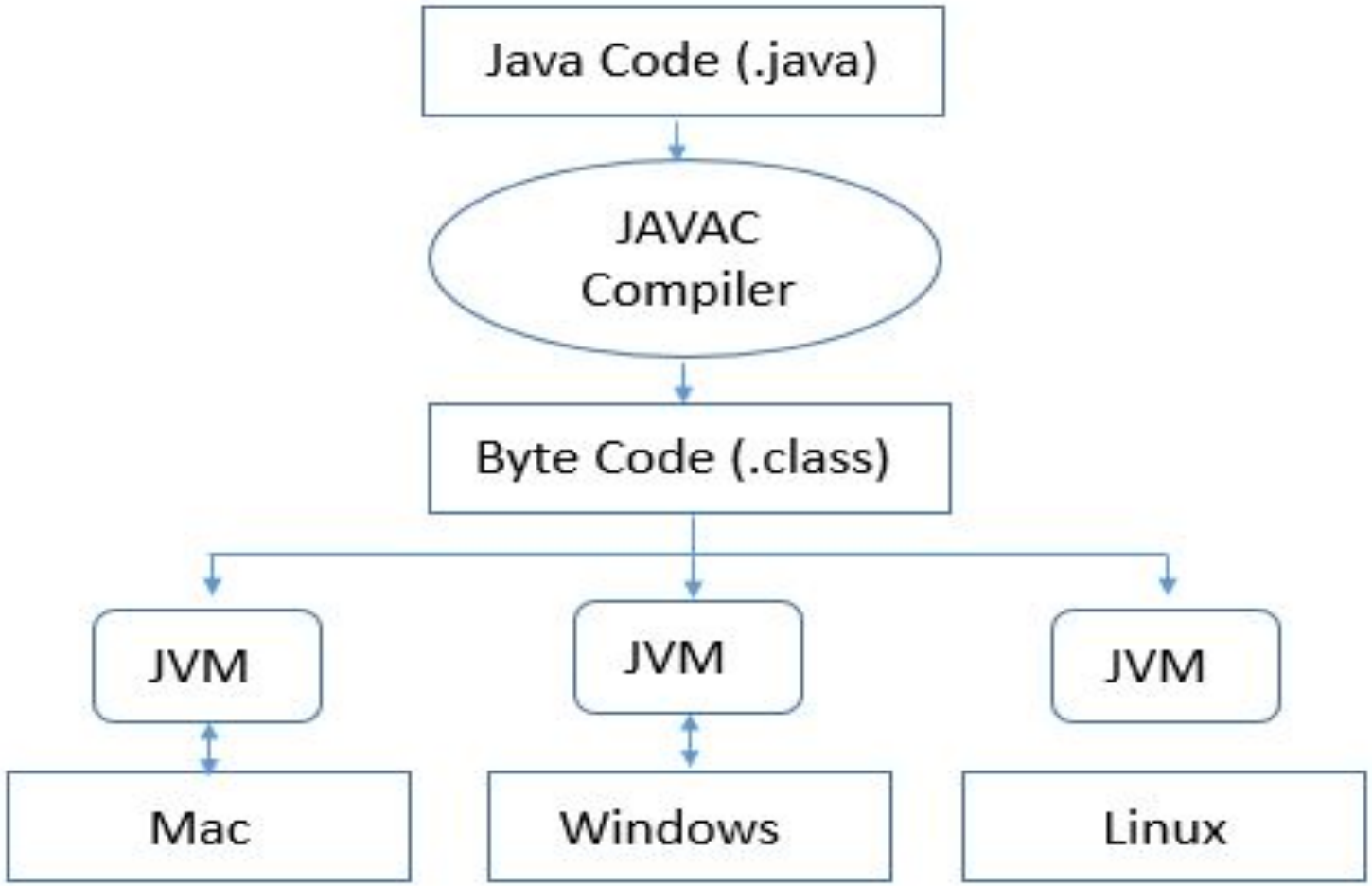
Let’s look more closely at one of the simplest Java programs you can have—one that merely prints a message to console:

```
class Firstpgm
{
public static void main(String[] args)
{
System.out.println("Hello MCA Students!");
}
}
```

Parameters used in First Java Program

- **Class** keyword is used to declare a class in Java.
- **Public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **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 creating 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.
- **Main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

JVM


Java Code (.java)
JAVAC Compiler
Byte Code (.class)
JVM
JVM
JVM
Mac
Windows
Linux

Class pgm without Object

```
class Main {

public static void main(String[] args)
{

  int x = 5;

    System.out.println(x);
  }
}
```

Class pgm with Object

```
public class Main1 {
  int x = 5;

  public static void main(String[] args) {
    Main1 myobj = new Main1();
    System.out.println(myobj.x);
  }
}
```

Sample Pgm

```
class Sample1 {
public static void main(String args[])
    {
    int num;
    num = 25;
    System.out.println("This is Number" + num);
    num = num * 2;
    System.out.println("The value of num * 2 is");
    System.out.println(num);
    }

}
```

IF Statement

```
public class TimeOfDayGreeting {
    public static void main(String[] args) {
        // Get the current time
        // Calendar calendar = Calendar.getInstance();
        int hourOfDay = 9;

        // Determine the part of the day
        String partOfDay;
        if (hourOfDay >= 4 && hourOfDay < 12) {
            partOfDay = "morning";
        } else if (hourOfDay >= 12 && hourOfDay < 17) {
            partOfDay = "afternoon";
        } else if (hourOfDay >= 17 && hourOfDay < 21) {
            partOfDay = "evening";
        } else {
            partOfDay = "night";
        }

        // Display the greeting
        System.out.println("Good " + partOfDay + "!");
    }
}
```

For statement program

File Edit View

```
public class Forpgm {
  public static void main(String[] args) {
    for (int i = 0; i <= 10; i = i + 2) {
      System.out.println(i);
    }
  }
}
```

Multiple For statement

```
import java.util.Scanner;

public class HospitalManagementSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numberOfPatients = 1;

        // Create an array to store patient records
        String[] patientNames = new String[numberOfPatients];
        int[] patientAges = new int[numberOfPatients];

        // Populate patient records
        for (int i = 0; i < numberOfPatients; i++) {
            System.out.println("Enter details for patient " + (i + 1) + ":");
            System.out.print("Name: ");
            patientNames[i] = scanner.nextLine();
            System.out.print("Age: ");
            patientAges[i] = scanner.nextInt();
            scanner.nextLine(); // Consume the newline character
        }
```

Multiple For statement

```
        // Display patient information
        System.out.println("\nPatient Information:");
        for (int i = 0; i < numberOfPatients; i++) {
            System.out.println("Patient " + (i + 1) + ":");
            System.out.println("Name: " + patientNames[i]);
            System.out.println("Age: " + patientAges[i]);
            System.out.println();
        }

        // Simulate treatment for each patient
        for (int i = 0; i < numberOfPatients; i++) {
            System.out.println("Treating patient " + patientNames[i]);
            // Perform medical procedures or treatments here
            System.out.println("Treatment completed for " + patientNames[i] + "\n");
        }

        // Release hospital resources, e.g., hospital beds
        for (int i = 0; i < numberOfPatients; i++) {
            System.out.println("Releasing bed for patient " + patientNames[i]);
            // Release resources and update hospital records
        }

        // Close the scanner
        scanner.close();
    }
}
```

Class Fundamentals

- All Java codes are defined in a Class.
- It has **variables and methods.**
- **Variables** are attributes that define the state of a class.
- **Methods** are the place where the exact business logic has to be done.
- It contains a set of statements (or) instructions to satisfy a particular requirement.

Syntax to declare a class:

```
access_modifier class<class_name>
{
    data member;
    method;
    constructor;
    nested class;
    interface;
}
```

Example

```
class Student
{
    int id=25;//data member (also instance variable)
    String name = "Dev"; //data member (also instance variable)

    public static void main(String args[])
    {
        Student s1=new Student();//creating an object of Student
        System.out.println(s1.id);
        System.out.println(s1.name);
     }
}
```

Multiple Objects

```
public class MulObj {
  int x = 5;

  public static void main(String[] args) {
    MulObj Obj1 = new MulObj();  // Object 1
    MulObj Obj2 = new MulObj();  // Object 2
    System.out.println(Obj1.x);
    System.out.println(Obj2.x);
  }
}
```

Using Multiple Classes

Whether this program is Right?

```
class One {
  int x = 5;
}

public class Second {
  public static void main(String[] args) {
    Second myObj = new Second();
    System.out.println(myObj.x);
  }
}
```

Right Program

```
class One {
  int x = 5;
}

public class Second {
  public static void main(String[] args) {
    One myObj = new One();
    System.out.println(myObj.x);
  }
}
```

Java Variables

- A variable is a container which holds the value while the Java program is executed.
- A variable is assigned with a data type.
- Variable is a name of memory location.
- There are three types of variables in java: local, instance and static.

- There are two types of data types in Java
- : primitive and non-primitive.

Variable

- A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.
- **Types of Variables**
- There are three types of variables in Java:
  - local variable - A variable declared **inside the body of the method** is called local variable
  - instance variable - A variable **declared inside the class but outside the body of the method**. It is not declared as static.
  - static variable - A variable that is declared as static is called a static variable. It cannot be local.

Example to understand the types of variables in java

```
1.  public class A
2.  {
3.      static int m=100;//static variable
4.      void method()
5.      {
6.          int n=90;//local variable
7.      }
8.      public static void main(String args[])
9.      {
10.         int data=50;//instance variable
11.     }
12. }//end of class
```

Programs based on Variable

```
public class Vari1
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
        System.out.println(n);
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
        System.out.println(m);
        System.out.println(data);

    }
}//end of class
```

Programs based on Variable

```
public class Vari1
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
        System.out.println(n);
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
        System.out.println(m);
        System.out.println(data);
        Vari1 obj1 = new Vari1();
        obj1.method();
    }
}//end of class
```

Declaring Objects

- There are 3 ways to initialize object in Java.

- By reference variable
- By method
- By constructor

Employee Class

```
import java.io.*;
public class Employee {

    String name;
    int age;
    String designation;
    double salary;

    // This is the constructor of the class Employee
    public Employee(String name) {
        this.name = name;
    }

    // Assign the age of the Employee  to the variable age.
    public void empAge(int empAge) {
        age = empAge;
    }

    /* Assign the designation to the variable designation.*/
    public void empDesignation(String empDesig) {
        designation = empDesig;
    }

    /* Assign the salary to the variable     salary.*/
    public void empSalary(double empSalary) {
        salary = empSalary;
    }

    /* Print the Employee details */
    public void printEmployee() {
        System.out.println("Name:"+ name );
        System.out.println("Age:" + age );
        System.out.println("Designation:" + designation );
        System.out.println("Salary:" + salary);
    }
}
```

EmployeeTest Class

```
import java.io.*;
public class EmployeeTest {

   public static void main(String args[]) {
      /* Create two objects using constructor */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");

      // Invoking methods for each object created
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
      empTwo.empSalary(500);
      empTwo.printEmployee();
   }
}
```

Compile Both Files and Run the Main class

```
C:\Java Programs>javac Employee.java

C:\Java Programs>javac EmployeeTest.java

C:\Java Programs>java EmployeeTest
```

Using new Keyword

```
public class Example1
{
void show()
{
System.out.println("Introduction to Java object Concepts");
}
public static void main(String[] args)
{
//creating an object using new keyword
Example1 obj = new Example1();
//invoking method using the object
obj.show();
}
}
```

Introducing Methods

- Classes usually consist of two things: instance variables and methods.
- The topic of methods is a large one because Java gives them so much power and flexibility.

Sample programs based on Method concepts

```
class Box {
double width;
double height;
double depth;
     void volume()
     {
          System.out.print("Volume is");
          System.out.println(width * height * depth);
     }
}
class BoxDemo3
     {
          public static void main(String args[])
          {
               Box mybox1 = new Box();
               Box mybox2 = new Box();

               mybox1.width =10;
               mybox1.height = 20;
               mybox1.depth = 15;
               mybox1.volume();

               mybox2.width =3;
               mybox2.height =6;
               mybox2.depth = 9;
               mybox2.volume();
          }
     }
```

Program for Method Calling

```
public class MinimumNumber {

    public static void main(String[] args) {
        int a = 11;
        int b = 6;
        int c = minFunction(a, b);
        System.out.println("Minimum Value = " + c);
    }

    /** returns the minimum of two numbers */
    public static int minFunction(int n1, int n2) {
        int min;
        if (n1 > n2)
            min = n2;
        else
            min = n1;

        return min;
    }
}
```

Method using the void Keyword

- The void keyword allows us to create methods which do not return a value.
- Here, in the following example we're considering a void method methodRankPoints.
- This method is a void method, which does not return any value.
- Call to a void method must be a statement i.e. methodRankPoints(255.7);.
- It is a Java statement that ends with a semicolon as shown in the following example.

```
public class ExampleVoid {

    public static void main(String[] args) {
        methodRankPoints(255.7);
    }

    public static void methodRankPoints(double points) {
        if (points >= 202.5) {
            System.out.println("Rank:A1");
        }else if (points >= 122.4) {
            System.out.println("Rank:A2");
        }else {
            System.out.println("Rank:A3");
        }
    }
}
```

Passing Parameters by Value

```
public class swappingExample {

   public static void main(String[] args) {
      int a = 30;
      int b = 45;
      System.out.println("Before swapping, a = " + a + " and b = " + b);

      // Invoke the swap method
      swapFunction(a, b);
      System.out.println("\n**Now, Before and After swapping values will be same here**:");
      System.out.println("After swapping, a = " + a + " and b is " + b);
   }

   public static void swapFunction(int a, int b) {
      System.out.println("Before swapping(Inside), a = " + a + " b = " + b);

      // Swap n1 with n2
      int c = a;
      a = b;
      b = c;
      System.out.println("After swapping(Inside), a = " + a + " b = " + b);
   }
}
```

Domain Example based on Method concept and Control Statement

```
class PatientInfo
{
String Pname;
int Pid;
int Page;
     void painfo() // Patient Information
     {
          System.out.print("The Patient Information is");
          System.out.println("Patient Name:"+Pname);
          System.out.println("Patient ID:"+ Pid);
          System.out.println("Patient Age:"+ Page);
     }
}


class MedicalDomain
{
     int patientid=1;
public static void main(String[] args)
     {
          MedicalDomain obj1= new MedicalDomain();
          System.out.println("Patient ID is:"+obj1.patientid);

          PatientInfo obj2 = new PatientInfo();
          obj2.Pname = "Anil";
          obj2.Pid = 10;
          obj2.Page = 25;
          obj2.painfo();
```

Domain Example based on Method concept and Control Statement

```
//Control Statement
        int Time =14;
        if (Time < 10)
        {
        System.out.println("First slot");
        } else if (Time < 12)
        {
        System.out.println("Second slot");
        }  else {
         System.out.println("Third slot");
        }

    }
}
```

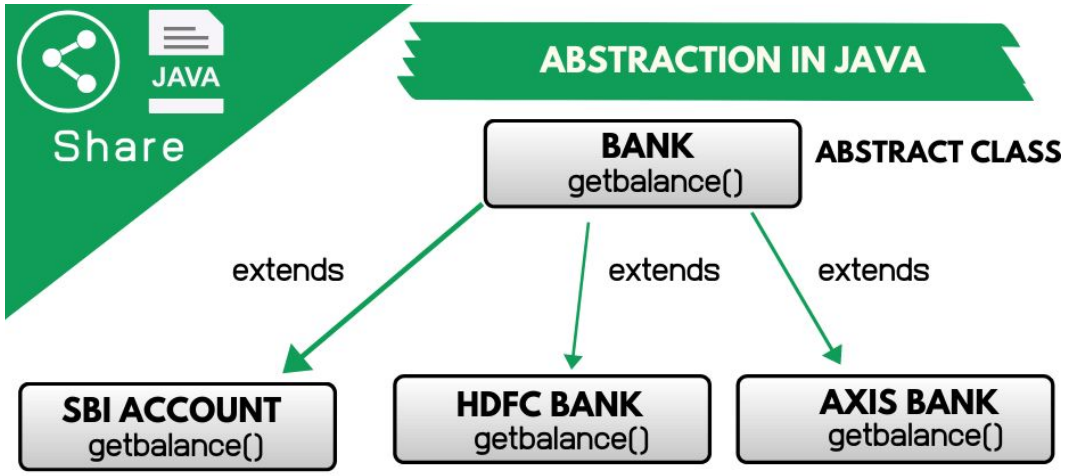
The four pillars of OOP Principles are

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction

- An essential element of object-oriented programming is an abstraction.
- Humans manage complexity through abstraction.
- It **hide the complex implementation details** of an object and expose only the essential features.
- Data Abstraction is the property by virtue of which only **the essential details** are displayed to the user.
- **Non-essential units** are not displayed to the user.
- The process of **identifying** only the **required characteristics of an object**, **ignoring the irrelevant details.**
- The **properties and behaviors of an object differentiate** it from other objects of similar type and also help in **classifying/grouping the object**.

Abstraction



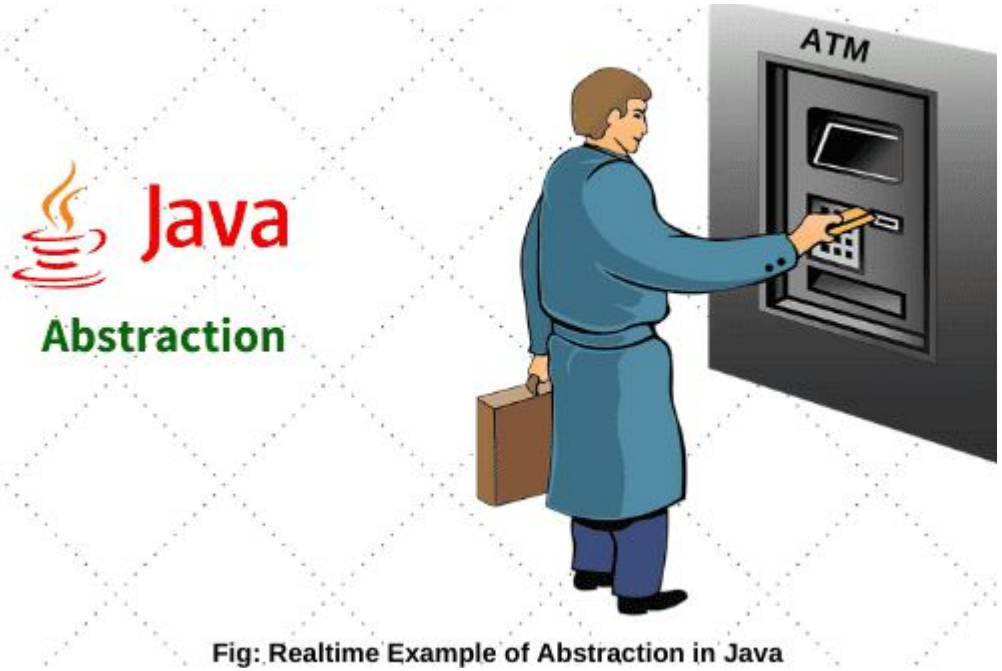


Fig: Realtime Example of Abstraction in Java

Abstraction Program 1

```
// Parent Class
abstract class Patient {
        abstract void PatientInfo1();

}

// Child Class
class PatientDetails extends Patient {
        void PatientInfo1()
        {
                String Pname = "Hemanth";
                int Page = 21;
                String Plocation = "Bangalore";

                System.out.println(Pname);
                System.out.println(Page);
                System.out.println(Plocation);
        }
}

// Driver Class
class AbstractMedical1 {
        // main function
        public static void main(String args[])
        {
                // object created
                Patient P = new PatientDetails();
                P.PatientInfo1();
        }
}
```

Abstraction Program2 without Extends Keyword

```
// Parent Class
abstract class Patient {
        abstract void PatientInfo1();
        abstract void PatientInfo2();

}

// Child Class
class PatientDetails1  {
        void PatientInfo1()
        {       String Pname = "Hemanth";
                int Page = 21;
                String Plocation = "Bangalore";

                System.out.println("Patient Name:"+Pname);
                System.out.println("Patient Age:"+Page);
                System.out.println("Patient Location:"+Plocation);
        }

}
```

```
// Sub Child Class extends Patient
class PatientDetails2  {
        void PatientInfo2()
        {       String P_status = "Good";
                int P_Sugar = 110;
                int P_BP = 120;

                System.out.println("Patient Status:"+P_status);
                System.out.println("Patient Sugar Level:"+P_Sugar);
                System.out.println("Patient BP Level:"+P_BP);
        }
}
// Driver Class
class AbstractMedical2 {
        // main function
        public static void main(String args[])
        {       // object created
                PatientDetails1 P = new PatientDetails1();
                P.PatientInfo1();
                PatientDetails2 P2 = new PatientDetails2();
                P2.PatientInfo2();
        }
}
```

Abstraction Program2 without Extends Keyword Cont..

- Slide 23 pgm will show Error which is mentioned below.
- To eliminate this error we need to define both abstract method in all the child classes.

```
c:\Java Programs>javac AbstractMedical2.java
AbstractMedical2.java:9: error: PatientDetails1 is not abstract and does not override abstract method PatientInfo2() in
Patient
class PatientDetails1 extends Patient
^
AbstractMedical2.java:22: error: PatientDetails2 is not abstract and does not override abstract method PatientInfo1() in
 Patient
class PatientDetails2 extends Patient  {
^
2 errors
```

Abstraction Program2 with Extends Keyword

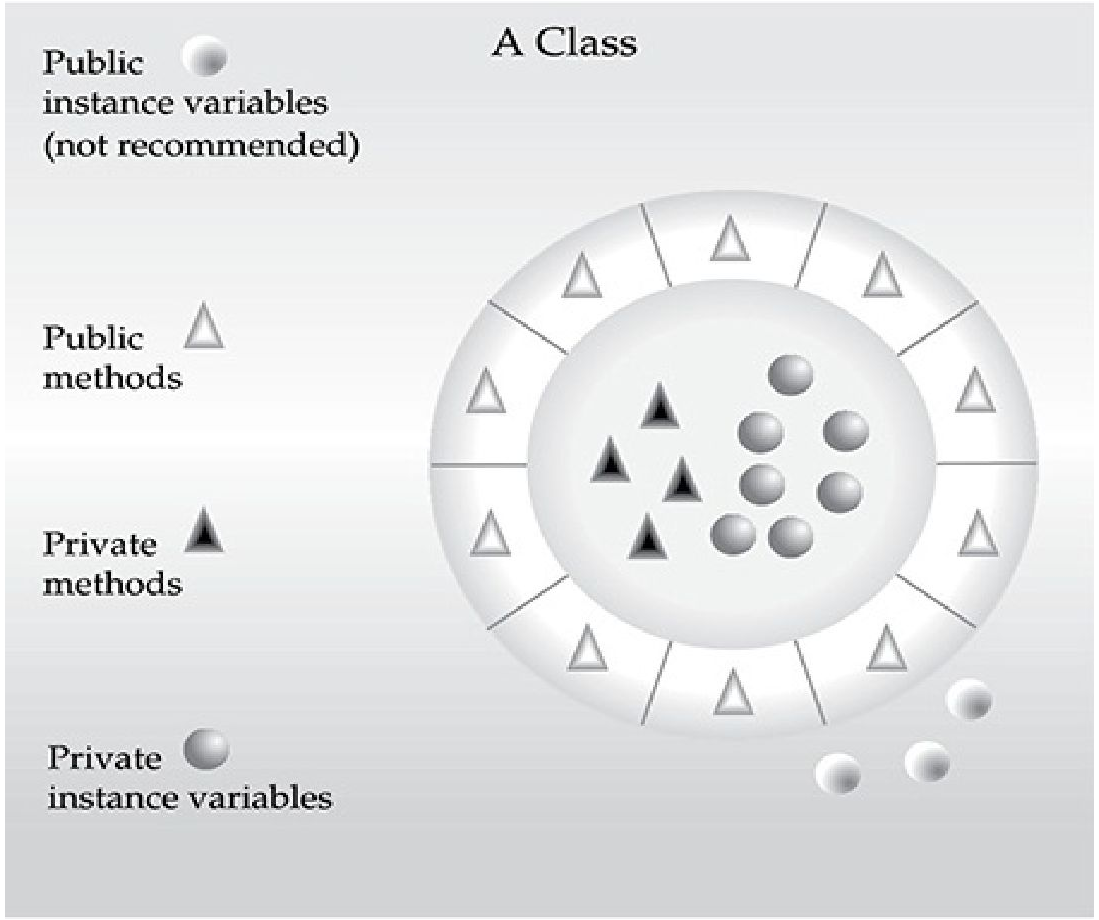
```
// Parent Class
abstract class Patient {
        abstract void PatientInfo1();
        abstract void PatientInfo2();

}

// Child Class
class PatientDetails1 extends Patient
{
        void PatientInfo1()
        {       String Pname = "Hemanth";
                int Page = 21;
                String Plocation = "Bangalore";

                System.out.println("Patient Name:"+Pname);
                System.out.println("Patient Age:"+Page);
                System.out.println("Patient Location:"+Plocation);
        }

void PatientInfo2()
        {       String P_status = "Good";
                int P_Sugar = 110;
                int P_BP = 120;

                System.out.println("Patient Status:"+P_status);
                System.out.println("Patient Sugar Level:"+P_Sugar);
                System.out.println("Patient BP Level:"+P_BP);
        }

}
```

```
// Driver Class
class AbstractMedical2 {
        // main function
        public static void main(String args[])
        {       // object created
                PatientDetails1 P = new PatientDetails1();
                P.PatientInfo1();
                P.PatientInfo2();
                //PatientDetails2 P2 = new PatientDetails2();
                //P2.PatientInfo2();
        }
}
```

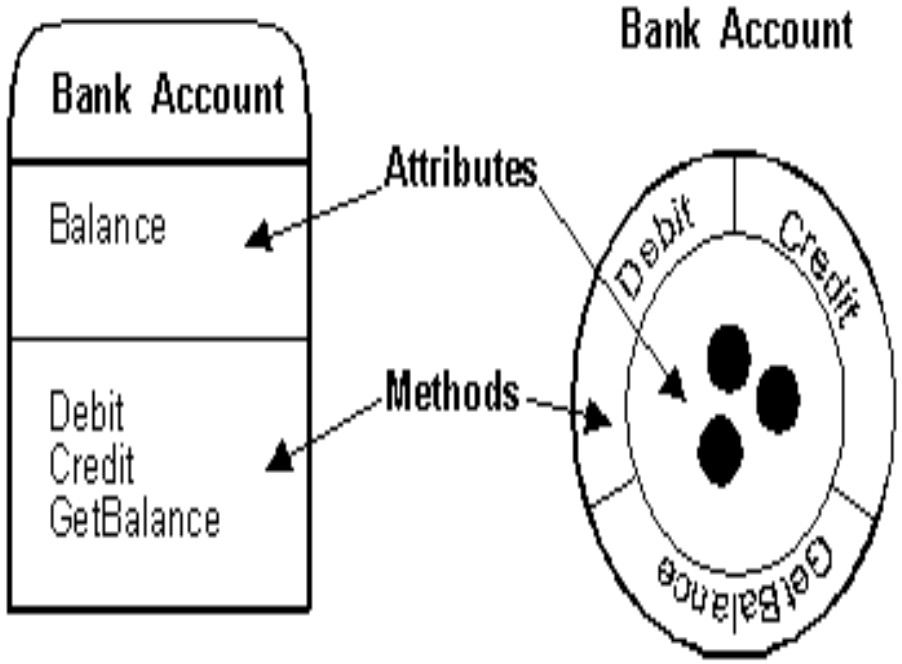
Encapsulation

- Encapsulation is the mechanism that **binds together code** and the data it manipulates, and keeps both **safe from outside interference and misuse.**
- One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
- Access to **the code and data inside** the wrapper is tightly controlled through a well-defined **interface.**
- Declare class **variables/attributes as private**
- provide public **get and set methods** to access and update the value of a private variable

Encapsulation: public methods can be used to protect private data.



Encapsulation



Encapsulation Pgm

```
class Encapsulate
        {

        // private variables declared
        // these can only be accessed by public methods of class
        private String PName;
        private int PAge;
        private int PWardNo;

        // get method for name to access
        public String getName() { return PName; }

        // get method for age to access
        public int getAge() { return PAge; }

        // get method for roll to access
        public int getWardNo() { return PWardNo; }

        // set method for name to access
        public void setName(String newPName){ PName = newPName; }

        // set method for age to access
        public void setAge(int newPAge) { PAge = newPAge; }

        // set method for Ward number to access
        public void setWardNo(int newPWardNo) { PWardNo = newPWardNo; }
        }
```

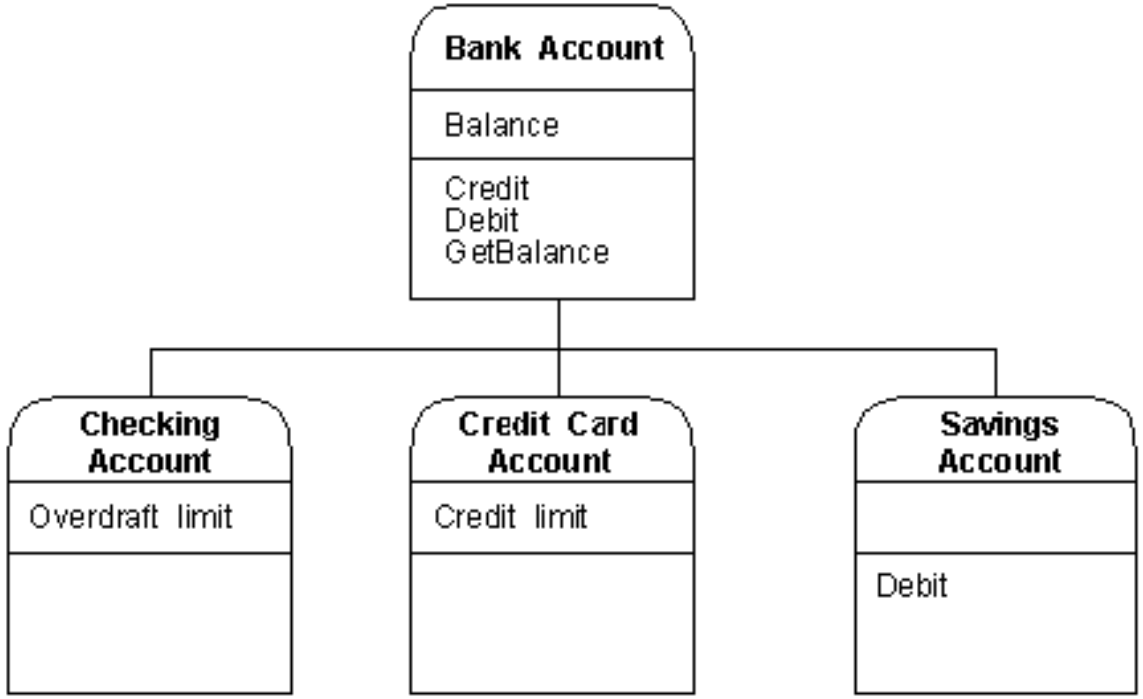
Encapsulation Pgm Cont..

```
// Class to access variables of the class Encapsulate

public class EncapsulationMedical1 {
        public static void main(String[] args)
        {
                Encapsulate obj = new Encapsulate();

                // setting values of the variables
                obj.setName("Sam");
                obj.setAge(25);
                obj.setWardNo(10);

                // Displaying values of the variables
                System.out.println("Patient name: " + obj.getName());
                System.out.println("Patient age: " + obj.getAge());
                System.out.println("Patient Ward Number: " + obj.getWardNo());

                // Direct access of PName is not possible due to encapsulation
                // System.out.println("Patient name: " +obj.PName);
        }
}
```

Inheritance

- Inheritance is the process by which one object acquires the properties of another object.
- This is important because it supports the concept of **hierarchical classification.**
- The most of the knowledge is made manageable by hierarchical (that is, top-down) classifications.
  - Super class
  - Sub class
  - Reusability

Inheritance



Inheritance

```
class Employee{
 float salary=40000;
}
class BankProgrammer extends Employee{
 int increment=10000;
 float updatedsalary=salary+increment;

 public static void main(String args[]){
   BankProgrammer p=new BankProgrammer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Increment of Programmer is:"+ p.increment);
    System.out.println("Updated salary of Programmer is:"+ p.updatedsalary);
}
}
```

```
// Java Program to Illustrate & Invocation of Constructor.
// Class 1_Super class
class Doctor {

        // Constructor of super class
        Doctor()
        {
                // Print statement
                System.out.println("Doctor Class Constructor Called ");
        }
}

// Class 2_Sub class
class Nurse extends Doctor {

        // Constructor of sub class
        Nurse()
        {

                // Print statement
                System.out.println("Nurse Class Constructor Called ");
        }
}

// Class 3
// Main class
class InheritanceMedicalDomain {

        // Main driver method
        public static void main(String[] args)
        {
                //Doctor d1 = new Doctor();
                Nurse d = new Nurse();
// Note: Here first super class constructor will be
//called there after derived(sub class) constructor will be called
        }
}
```

Inheritance Pgm-2

```
// the parent class
class Person {
  // general member variables
  String patientname;
  String patientlocation;

  // general methods
  public void setPatientName(String patientname) {
    this.patientname = patientname;
  }

  public void setPatientLocation(String patientlocation) {
    this.patientlocation = patientlocation;
  }
}
// the child class inheriting the parent class
class Patient extends Person {
  // specific member variables
  String patientid;
  int bill;
  String joiningDate;
```

```
  // constructor
  Patient(String patientname, String patientlocation, String patientid, int
bill, String joiningDate) {

    // set the patientname and patientlocation using the
    // setpatientname() and patientlocation() methods
    // of the parent class Person
    // that is inherited by this child class Employee
    this.setPatientName(patientname);
    this.setPatientLocation(patientlocation);

    // now set the member variables of this class
    this.patientid = patientid;
    this.bill = bill;
    this.joiningDate = joiningDate;
  }

  // show Patient details
  public void showDetail() {
    System.out.println("Employee details:");
    System.out.println("PatientID: " + this.patientid);
    System.out.println("patient name: " + this.patientname);
    System.out.println("patient location: " + this.patientlocation);
    System.out.println("Patient Bill: " + this.bill);
    System.out.println("Joining Date: " + this.joiningDate);
  }
}
```

```
// Main class
class InheritanceMedicalDomain1 {

        // Main driver method
        public static void main(String[] args)
        {
                String patientname = "Bipin";
                String patientlocation = "Kerala";
                String patientid = "E01";
                int bill = 3000;
                String joiningDate = "2023-10-25";

    // creating an object of the Patient class
    Patient patObj = new Patient(patientname, patientlocation, patientid, bill,
joiningDate);

    // show detail
    patObj.showDetail();

  }
}
```

Using new keyword, invoke the constructor (default or parametrized) of the class.

```
public class Example2
{
Example2 ()
{
System.out.println("Introduction to Java object Concepts");
}
public static void main(String[] args)
{
//creating an object using new keyword
Example2 obj = new Example2();
}
}
```

public no-arg constructors

```
class Apollo {
  int Doctor_ID;
  String specialization;
  int License_Information;

  // public constructor
  public Apollo() {
    Doctor_ID = 200;
    specialization = "cardiologist";
    License_Information = 510;
  }
}

class Constructor1 {
  public static void main(String[] args) {

    // object is created in another class
    Apollo obj = new Apollo();
    System.out.println("Doctor ID = " + obj.Doctor_ID);
    System.out.println("Doctor specialization = " + obj.specialization);
    System.out.println("License_Information = " + obj.License_Information);
  }
}
```

Parameterized Constructor

```
class Appointment
{
        String AppointmentID;
        String PurposeOfAppointment;
        String Feedback;

        Appointment(String App, String Purpose)
        {
          AppointmentID = App;
          PurposeOfAppointment = Purpose;
          System.out.println("AppointmentID is:" + AppointmentID);
          System.out.println("Purpose Of Appointment is:" + PurposeOfAppointment);
        }


}

class ParameterizedConstructor1 {
  public static void main(String[] args) {

    // call constructor by passing a two values
      Appointment obj1 = new Appointment("25","Fever");


  }
}
```

Constructor Overloading

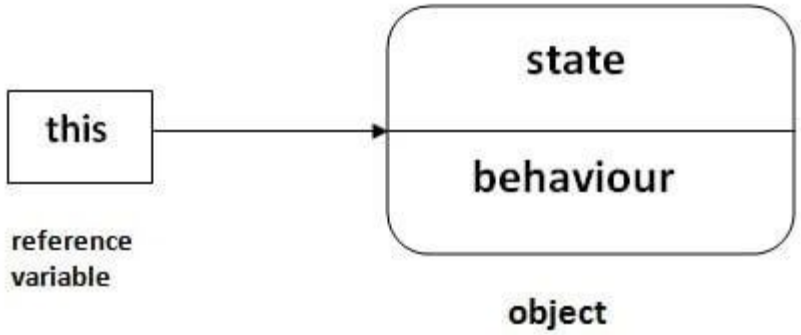
```
class Appointment
{
        String AppointmentID;
        String PurposeOfAppointment;
        String Feedback;

        Appointment(String App, String Purpose)
        {
          AppointmentID = App;
          PurposeOfAppointment = Purpose;
          System.out.println("AppointmentID is:" + AppointmentID);
          System.out.println("Purpose Of Appointment is:" + PurposeOfAppointment);
        }

        Appointment(String App, String Purpose, String Review)
        {
          AppointmentID = App;
          PurposeOfAppointment = Purpose;
          Feedback = Review;
          System.out.println("AppointmentID is:" + AppointmentID);
          System.out.println("Purpose Of Appointment is:" + PurposeOfAppointment);
         System.out.println("Feedback of Doctor:" + Feedback);
        }
}

class ParameterizedConstructor2 {
  public static void main(String[] args) {

    // call constructor by passing a two values
      Appointment obj1 = new Appointment("25","Fever");
      Appointment obj2 = new Appointment("26","Headache","Happy");

  }
}
```

constructor overloading

```
class Demo{
        int   value1;
        int   value2;
        /*Demo(){
          value1 = 10;
          value2 = 20;
          System.out.println("Inside 1st Constructor");
       }*/
       Demo(int a){
         value1 = a;
         System.out.println("Inside 2nd Constructor");
      }
      Demo(int a,int b){
      value1 = a;
      value2 = b;
      System.out.println("Inside 3rd Constructor");
     }
     public void display(){
        System.out.println("Value1 === "+value1);
        System.out.println("Value2 === "+value2);
   }
   public static void main(String args[]){
      Demo d1 = new Demo();
      Demo d2 = new Demo(30);
      Demo d3 = new Demo(30,40);
      d1.display();
      d2.display();
      d3.display();
  }
}
```

this Keyword

- Sometimes a method will need to refer to the object that invoked it.
- Java defines this keyword. **this can be used inside any method to refer to the current object.**
- That is, this is always a reference to the object on which the method was invoked.
- We can use this **anywhere a reference to an object of the current class** type is permitted.



In the below example, parameters and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

```
class MedInfo1{
int Patientid;
String Patientname;
float age;
MedInfo1(int Patientid,String Patientname,float age){
Patientid=Patientid;
Patientname=Patientname;
age=age;
}
void display(){System.out.println(Patientid+" "+Patientname+" "+age);}
}
class TestThis1{
public static void main(String args[]){
MedInfo1 m1=new MedInfo1(25,"Curz",50f);
MedInfo1 m2=new MedInfo1(26,"sham",28f);
m1.display();
m2.display();
}}
```

Solution for previous slide program

```
class MedInfo1{
int Patientid;
String Patientname;
float age;
MedInfo1(int Patientid,String Patientname,float age){
this.Patientid=Patientid;
this.Patientname=Patientname;
this.age=age;
}
void display(){System.out.println(Patientid+" "+Patientname+" "+age);}
}
class TestThis1{
public static void main(String args[]){
MedInfo1 m1=new MedInfo1(25,"Curz",50f);
MedInfo1 m2=new MedInfo1(26,"sham",28f);
m1.display();
m2.display();
}}
```

this: to invoke current class method

```
class A{
void m(){System.out.println("About Health Care");}
void n(){
System.out.println("Healthy Life");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Overridden

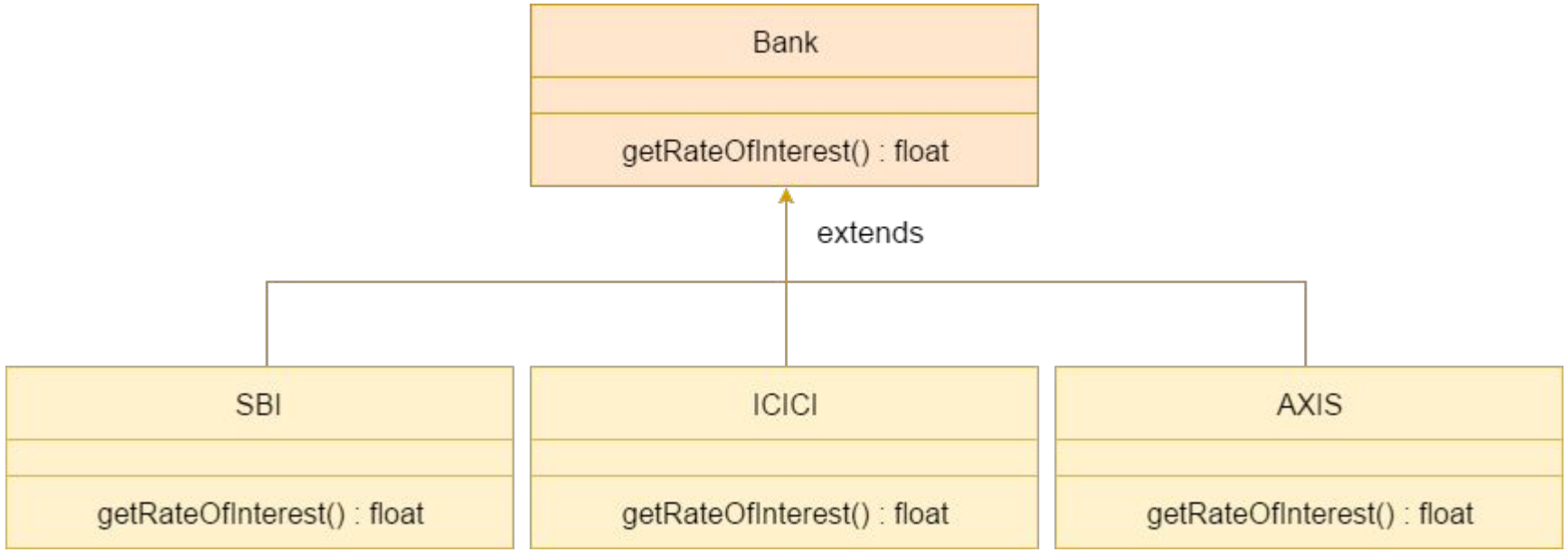
- In Java, constructors cannot be overridden in the same way that methods can be overridden.

- Overriding a method means providing a different implementation of the method in a subclass, while maintaining the same method signature (name, return type, and parameters).

Polymorphism

- Polymorphism is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation.
- Consider a stack (which is a lastin,first-out list). You might have a program that requires three types of stacks.
- One stack is used for **integer values**, **one for floating-point values**, and **one for characters**.
- The algorithm that implements each stack is the same, even though the data being stored differs.

Polymorphism



Polymorphism

- he word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- **Example -** A person at the same time can have different characteristics. Like a man at the same time is a family member, Sports person, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
- In Java polymorphism is mainly divided into two types:
  - Compile-time Polymorphism
  - Runtime Polymorphism

Compile-time Polymorphism (Method overloading)

- When a class has **two or more methods by the same name but different parameters**, it is known as method overloading.
- It is different from overriding. In overriding, a method has the same method name, type, number of parameters, etc.

- 1. changing number of arguments
- 2. changing data type of arguments
- 3. Changing the sequence of Data type of parameters

1. Method Overloading: changing number of arguments

```
class Medicine{
static int add(int medicine1,int medicine2){return medicine1+medicine2;}
static int add(int medicine1,int medicine2,int medicine3 )
{return medicine1+medicine2+medicine3;}
}
class MedicineInfo{
public static void main(String[] args){
System.out.println(Medicine.add(12,13));
System.out.println(Medicine.add(12,13,14));
}}
```

2. Method Overloading: changing data type of arguments

```
class Medicine{
static int add(int medicine1,int medicine2)
{return medicine1+medicine2;}
static double add(double medicine1,double medicine2)
{return medicine1+medicine2;}
}
class MedicineInfo{
public static void main(String[] args){
System.out.println(Medicine.add(12,13));
System.out.println(Medicine.add(14.4,16.2));
}}
```

3. Changing the sequence of Data type of parameters

```
class Medicine{
static double add(int medicine1,double medicine2)
{return medicine1+medicine2;}
static double add(double medicine1,int medicine2)
{return medicine1+medicine2;}
}
class MedicineInfo2{
public static void main(String[] args){
System.out.println(Medicine.add(12,13.1));
System.out.println(Medicine.add(14.4,16));
}}
```

Runtime Polymorphism Pgm(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 Pgm

```
// A base class to represent a medical procedure
class MedicalProcedure {
    // Common properties and methods

    // A method to perform the procedure
    public void performProcedure() {
        System.out.println("Performing a generic medical procedure.");
    }
}

// Subclass for a specific medical procedure
class SurgeryProcedure extends MedicalProcedure {
    // Specific properties and methods related to surgery

    // Override the performProcedure method to provide a specific implementation
    @Override
    public void performProcedure() {
        System.out.println("Performing a surgery procedure.");
        // Specific surgery code here
    }
}
```

Method overriding Pgm Cont..

```
// Subclass for another medical procedure
class XRayProcedure extends MedicalProcedure {
    // Specific properties and methods related to X-ray

    // Override the performProcedure method to provide a specific implementation
    @Override
    public void performProcedure() {
        System.out.println("Performing an X-ray procedure.");
        // Specific X-ray code here
    }
}

public class MethodOverriding11 {
    public static void main(String[] args) {
        MedicalProcedure procedure1 = new SurgeryProcedure();
        MedicalProcedure procedure2 = new XRayProcedure();

        // Polymorphism: Call the overridden methods based on the actual object type
        procedure1.performProcedure();
        procedure2.performProcedure();
    }
}
```

Note

- Private methods can not be overridden
- Final methods can not be overridden in Java

Polymorphism, Encapsulation, and Inheritance Work Together

- When properly **applied, polymorphism, encapsulation, and inheritance combine** to produce a programming environment that supports the development of far more **robust and scaleable** programs than does the **process-oriented model.**

Buzzwords of Java

- The key considerations were summed up by the Java team in the following list of buzzwords:
  - Simple
  - Secure
  - Portable
  - Object-oriented
  - Robust
  - Multithreaded
  - Architecture-neutral
  - Interpreted
  - High performance
  - Distributed
  - Dynamic

Simple

- Java was designed to be easy for the professional programmer to learn and use effectively.
- Assuming that you have some programming experience, you will not find Java hard to master.
- If you already understand the **basic concepts of object oriented programming**, learning Java will be even easier.

Object-Oriented

- Although influenced by its predecessors, Java was not designed to be source code compatible with any other language.
- This allowed the Java team the freedom to design with a blank slate.
- One outcome of this was a clean, usable, pragmatic approach to objects.

Robust

- The **multi platformed environment** of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems.
- Thus, the ability to create robust programs was given a **high priority in the design of Java.**
- To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development.
- At the same time, Java frees you from having to worry about many of the most common causes of programming errors.
- Because Java is a strictly typed language, it checks your code at compile time.
- However, it also checks your code at run time.

Robust

- To better understand how Java is robust, consider two of the main reasons for program failure: **memory management mistakes and mishandled exceptional conditions** (that is, run-time errors).
- Memory management can be a difficult, tedious task in traditional programming environments.
- For example, in C/C++, the programmer will often **manually allocate and free dynamic memory**.
- This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using.
- **Java virtually eliminates these problems by managing memory allocation and deallocation** for you. (In fact, deallocation is completely automatic, because **Java provides garbage collection** for unused objects.)

Multithreaded

- Java was designed to meet the real-world requirement of **creating interactive, networked programs.**
- To accomplish this, Java supports **multithreaded programming**, which allows you to write programs that do **many things simultaneously**.
- The Java run-time system comes with an elegant yet sophisticated solution for **multiprocess synchronization** that enables you to construct smoothly running interactive systems.

Architecture-Neutral

- **A central issue** for the Java designers was **that of code longevity and portability.**
- At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine.
- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.
- The Java designers made several hard decisions in the Java language and the **Java Virtual Machine in an attempt to alter this situation.**
- Their goal was "write once; run anywhere, any time, forever."
- To a great extent, this goal was accomplished.

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an **intermediate representation called Java bytecode.**
- This code can be executed on any system that **implements the Java Virtual Machine.**
- Most previous attempts at cross-platform solutions have done so at the expense of performance.
- As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
- Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

- Java is designed for **the distributed environment of the Internet because it handles TCP/IP protocols.**
- In fact, accessing a resource using a URL is not much different from accessing a file.
- Java also supports **Remote Method Invocation (RMI).**
- This feature enables a program to invoke methods across a network.

Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
- This makes it possible to dynamically link code in a safe and expedient manner.
- This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.