UNIT - 2 JAVA

Universal Class – Object

Universal super class

- The Object Class is known as universal super class of Java.
- This is so because Every class you create in Java automatically inherits the Object class.
- The Object class is super class of all the classes in Java either directly or Indirectly.
- You don't need to extend it manually. All the properties of Object class are already in your class.
- You can find the definition of the Object class in java.lang package And there are a few useful methods in this class which you can override in your class

The object class is the super class of all java classes. This class has following important methods.

1)Object clone() :

- The clone() method creates a copy of the invoking object and allocates memory to that object.

- For example: Banking b1 = new Banking("MM",10000);

  Banking b2 = b1.clone();

- Here b1 is created by new keyword

2) Boolean equals(Object obj):

- This method is used to compare two objects. It returns true if both object are same.

3) int hashCode():

- The hashCode() method returns the hashcode value for an object. This value is unique for every object.

4) Final Class getClass():

- This method returns the class information of a class. It returns the class name and the package name of its class.

5) String toString():

- It returns the string description of an object.
- This method should be overridden by each class to display the object information.

Access Modifiers

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

There are four types of Java access modifiers:

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

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

2. **Public**: The access level of a public modifier is everywhere. It can be accessed

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

When variables and methods are declared `private`, they cannot be accessed outside of the class. For example,

```
class Data {
    // private variable
    private String name;}
public class Main {
    public static void main(String[] main){
        // create an object of Data
        Data d = new Data();
// access private variable and field from
another class
        d.name = "Programjava";
    }}
```

In the above example, we have declared a private variable named `name`. When we run the program, we will get the following error:

```
Main.java:18: error: name has private access
in Data
        d.name = "Programjava";
          ^
```

Protected Access Modifier

When methods and data members are declared `protected`, we can access them within the same package as well as from subclasses. For example,

```
class Animal {  // protected method

    protected void display() {
        System.out.println("I am an animal"); }}
class Dog extends Animal {
    public static void main(String[] args) {
        // create an object of Dog class
        Dog dog = new Dog();
         // access protected method
        dog.display();    }}
```

Output:

```
I am an animal
```

In the above example, we have a protected method named `display()` inside the `Animal` class. The `Animal` class is inherited by the `Dog` class.

Public Access Modifier

When methods, variables, classes, and so on are declared `public`, then we can access them from anywhere. The public access modifier has no scope restriction. For example,

```
// public class
public class Animal {
    // public variable
    public int legCount;
    // public method
    public void display() {
        System.out.println("I am an animal.");
        System.out.println("I have " + legCount +
" legs.");    }}
```

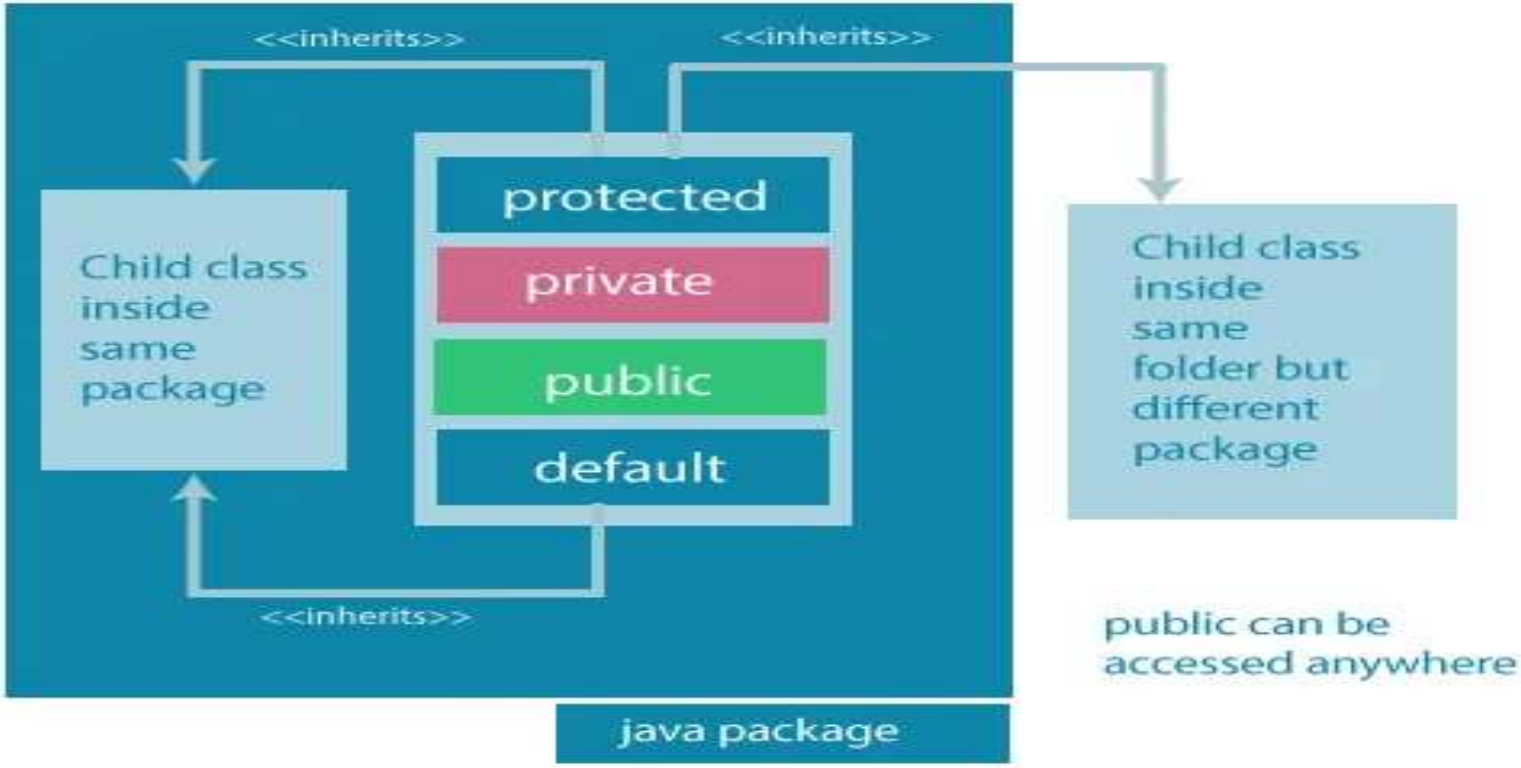
```
public class Main {
    public static void main( String[] args ) {
        // accessing the public class
        Animal animal = new Animal();

        // accessing the public variable
        animal.legCount = 4;
        // accessing the public method
        animal.display();
    }

 }
```

Output:

```
I am an animal.
 I have 4 legs.
```

Here,

- The public class `Animal` is accessed from the `Main` class.
- The public variable `legCount` is accessed from the `Main` class.
- The public method `display()` is accessed from the `Main` class.


<<inherits>>
<<inherits>>
protected
private
public
default
Child class inside same package
Child class inside same folder but different package
<<inherits>>
public can be accessed anywhere
java package
source folder

Access modifiers are mainly used for encapsulation. It can help us to control what part of a program can access the members of a class.

Constructors in Inheritance :

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

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

extends Keyword

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

Syntax

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

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

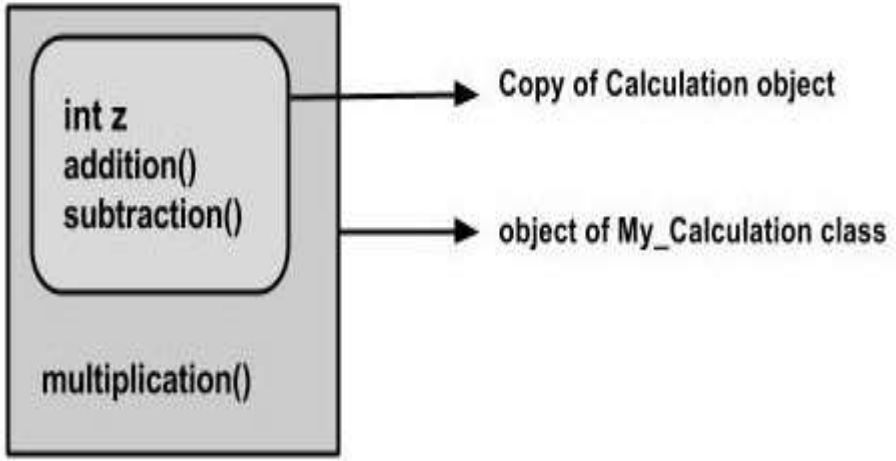
```
class Calculation {
   int z;

   public void addition(int x, int y) {
      z = x + y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void Subtraction(int x, int y) {
      z = x - y;
      System.out.println("The difference between numbers:"+z);
   }
}
```

```
public class My_Calculation extends Calculation {
  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of numbers:"+z);
  }

  public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Subtraction(a, b);
    demo.multiplication(a, b);
  }
}
```

Output

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to My_Calculation class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (cal in this case) you cannot call the method multiplication(), which belongs to the subclass My_Calculation.

```
Calculation demo = new My_Calculation();
demo.addition(a, b);
 demo.Subtraction(a, b);
```

Note − A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

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

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

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable
 super.method();
```

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

```
class Super_class {
   int num = 20;

   // display method of superclass
   public void display() {
      System.out.println("display method of superclass");
   }
}

public class Sub_class extends Super_class {
   int num = 10;

   // display method of sub class
   public void display() {
      System.out.println("display method of subclass");
    }
```

```
public void my_method() {
    // Instantiating subclass
    Sub_class sub = new Sub_class();

    // Invoking the display() method of sub class
    sub.display();

    // Invoking the display() method of superclass
    super.display();

    // printing the value of variable num of subclass
    System.out.println("variable named num in sub class:"+ sub.num);

    // printing the value of variable num of superclass
    System.out.println("variable named num in super class:"+ super.num);
  }
```

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

Output

```
display method of subclass
display method of superclass
variable named num in sub class:10
variable named num in super class:20
```

Invoking Superclass Constructor:

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

The program given in this section demonstrates how to use the super keyword to invoke the parameterized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

```
class Superclass {
    int age;

    Superclass(int age) {
        this.age = age;
    }

    public void getAge() {
        System.out.println("The value of the
variable named age in super class is: "
+age);
    }
}
```

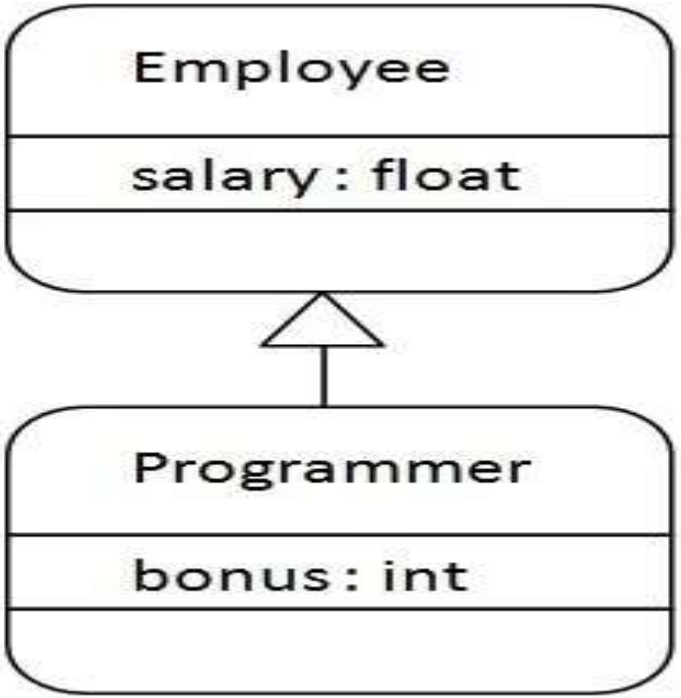
```
public class Subclass extends Superclass {
    Subclass(int age) {
        super(age);
    }

    public static void main(String args[]) {
        Subclass s = new Subclass(24);
        s.getAge();
    }
}
```

Output

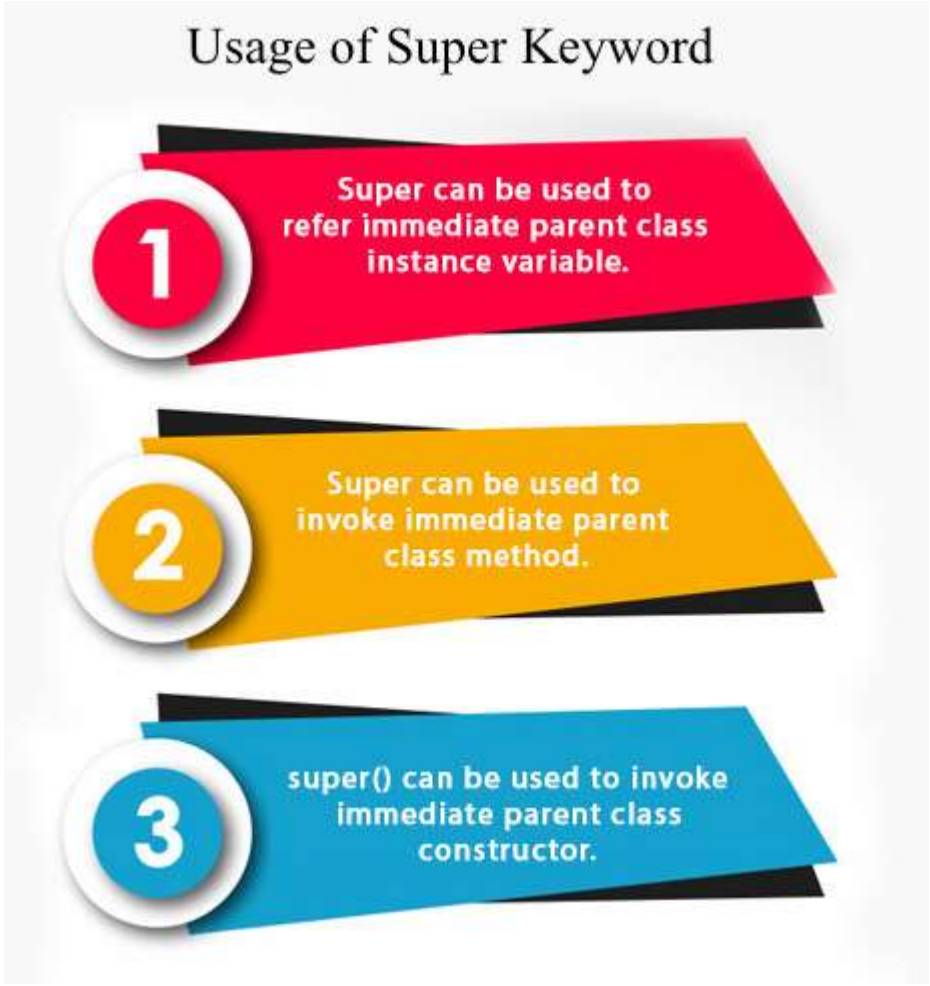
```
The value of the variable named age
in super class is: 24
```

Example : Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000; }
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```



Output :
Programmer salary is:40000.0
Bonus of programmer is:10000


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

Example :

Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{

int id;

String name;

Person(int id,String name){

this.id=id;

this.name=name;

} }
```

```
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;  }
void display(){System.out.println(id+" "+name+" "+salary);}  }
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}
```

Output:

1 ankit 45000

Default Constructor in Java

```
class Student
{
    String name;
    Double gpa;
}
public class Main
{
    public static void main(String[] args)
    {
        Student s = new Student();
        System.out.println(s.name + "\t" + s.gpa);
    }
}
```

Output:

```
null    null
```

For example, consider the `Student` class shown below with two attributes: `student name` and `GPA`. A constructor is defined for the class, which takes a string name and a double `GPA` as parameters and initializes the corresponding attributes for the new object.

Parameterized Constructor in Java

```
class Student
{
    String name;
    Double gpa;
    Student(String s, Double g)
    {
        name = s;
        gpa = g;    }}
public class Main
{
    public static void main(String[] args)
    {
        Student s = new Student("Justin", 9.75);
```

Output:

```
Justin  9.75
```

Constructor Overloading in Java

```
class Student
{
    String name;
    Double gpa;
    Student(String s, Double g)
    {
        name = s;
        this.gpa = g;     }
    Student(String s)     {
        name = s;
        gpa = null;//Setting GPA to null
    }}
public class Main{
    public static void main(String[] args)     {
        Student s1 = new Student("Justin");
        Student s2 = new Student("Jessica", 9.23);
        System.out.println(s1.name + "\t" + s1.gpa);
        System.out.println(s2.name + "\t" + s2.gpa);
    }}
```

Output:

```
Justin    null

Jessica   9.23
```

Remember that Java provides a default constructor only when no other constructor is created for a class. But for our class, constructors already exist, and so we need to create a default constructor.

```
class Student
{
    String name;
    Double gpa;
    Student(String s, Double g)
    {
        name = s;
        gpa = g;
    }
    Student(String s)
    {
        name = s;
        gpa = null;
    }
```

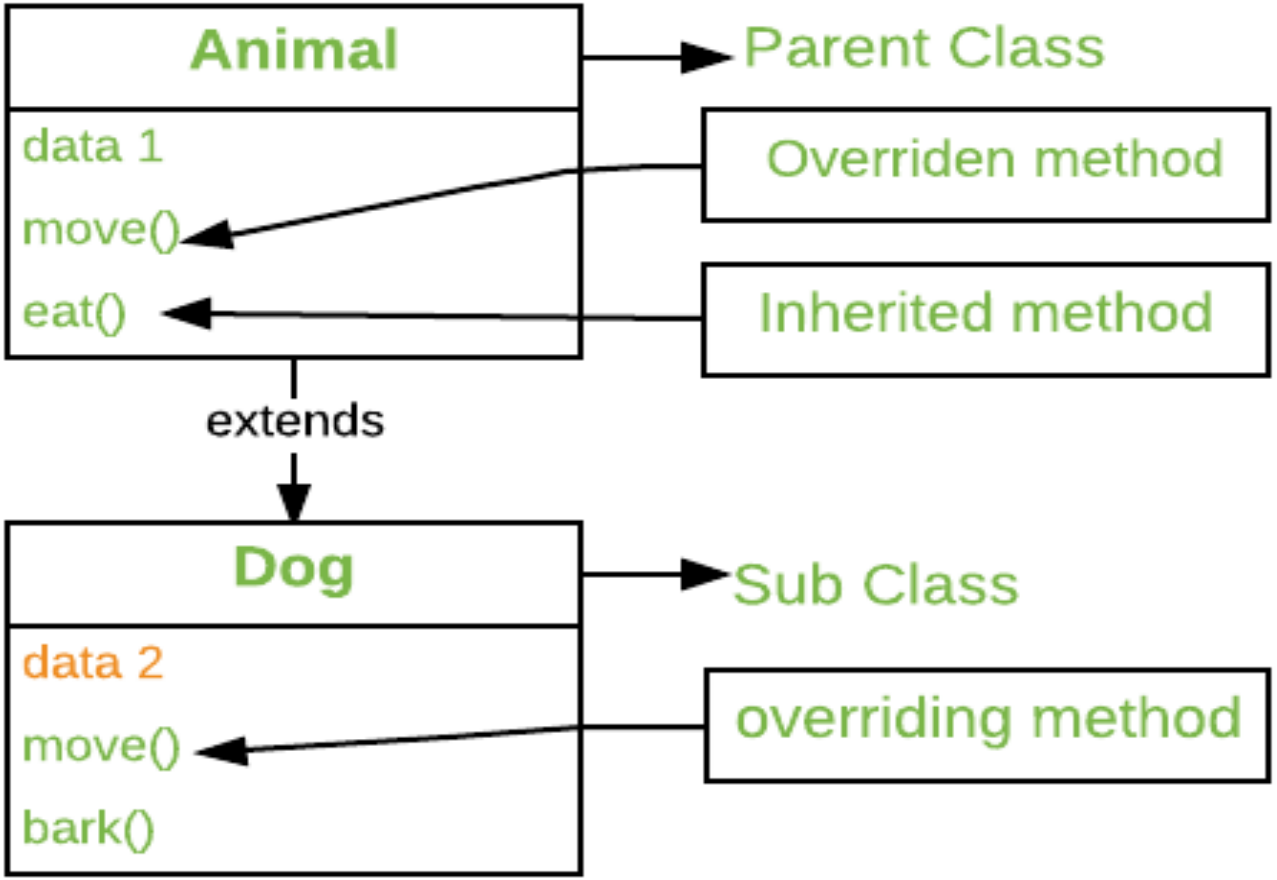
Output:

```
null                    null
Justin    null

Jessica   9.23
```

```
Student()     {
        name = null;
        gpa = null;
    }}
public class Main{
    public static void main(String[] args)
    {   Student s1 = new Student();
        Student s2 = new Student("Justin");
        Student s3 = new Student("Jessica", 9.23);
        System.out.println(s1.name + "\t" + s1.gpa);
        System.out.println(s2.name + "\t" + s2.gpa);
        System.out.println(s3.name + "\t" + s3.gpa);
    }}
```

Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.
Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.


Animal
data 1
move()
eat()
Parent Class
Overriden method
Inherited method
extends
Dog
data 2
move()
bark()
Sub Class
overriding method

Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
// A Simple Java program to demonstrate
// method overriding in java

// Base Class
class Parent {
    void show()
    {
        System.out.println("Parent's show()");
    }
}
```

```
// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show()
    {
        System.out.println("Child's show()");
    }
}
```

```
class Main {
    public static void main(String[] args){
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();
        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();    }}
```

Output:

```
Parent's show()
Child's show()
```

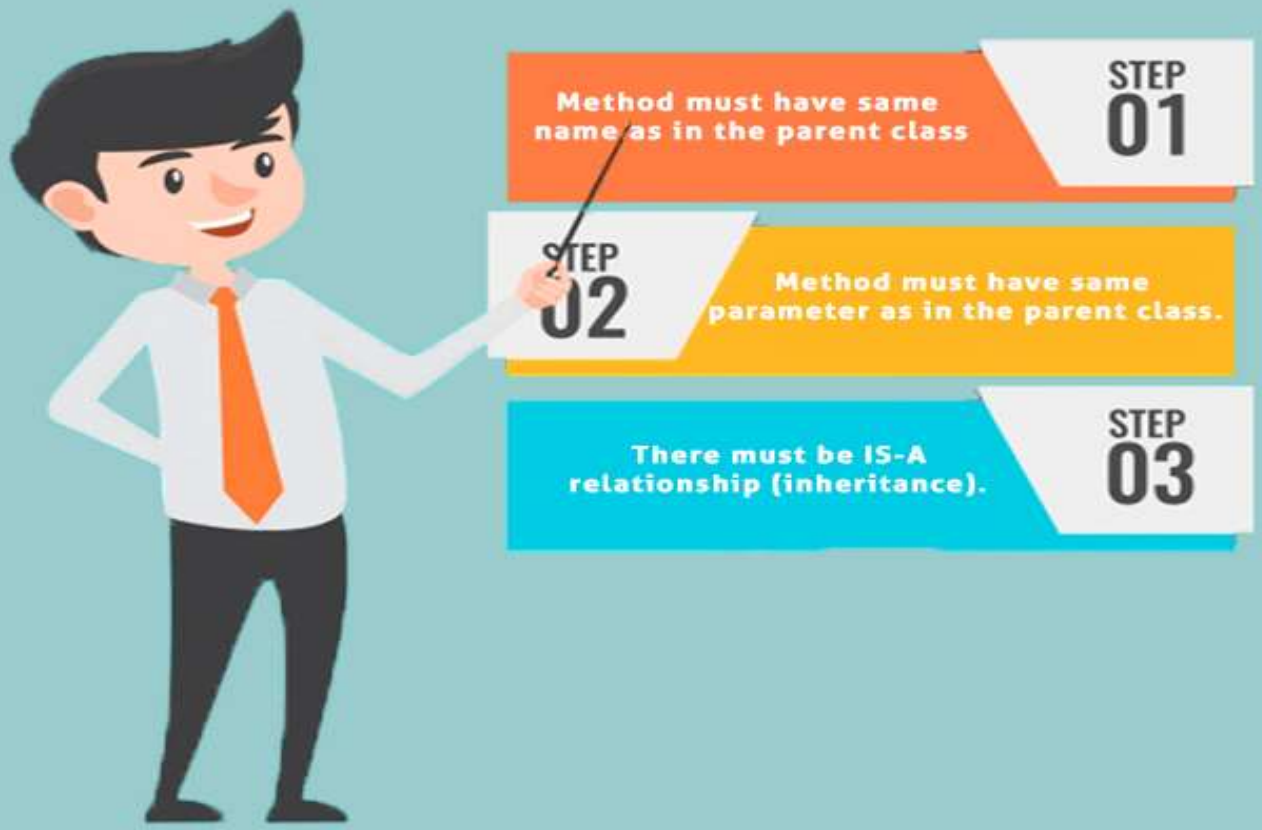
Usage of Java Method Overriding

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

Rules for Java Method Overriding

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

Rules for Java Method Overriding



Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
class Vehicle{

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

}
```

```
class Bike extends Vehicle{

  public static void main(String args[]){

  //creating an instance of child class

  Bike obj = new Bike();

  //calling the method with child class instance

  obj.run();

  }

}
```

O/P:

Vehicle is running

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation.

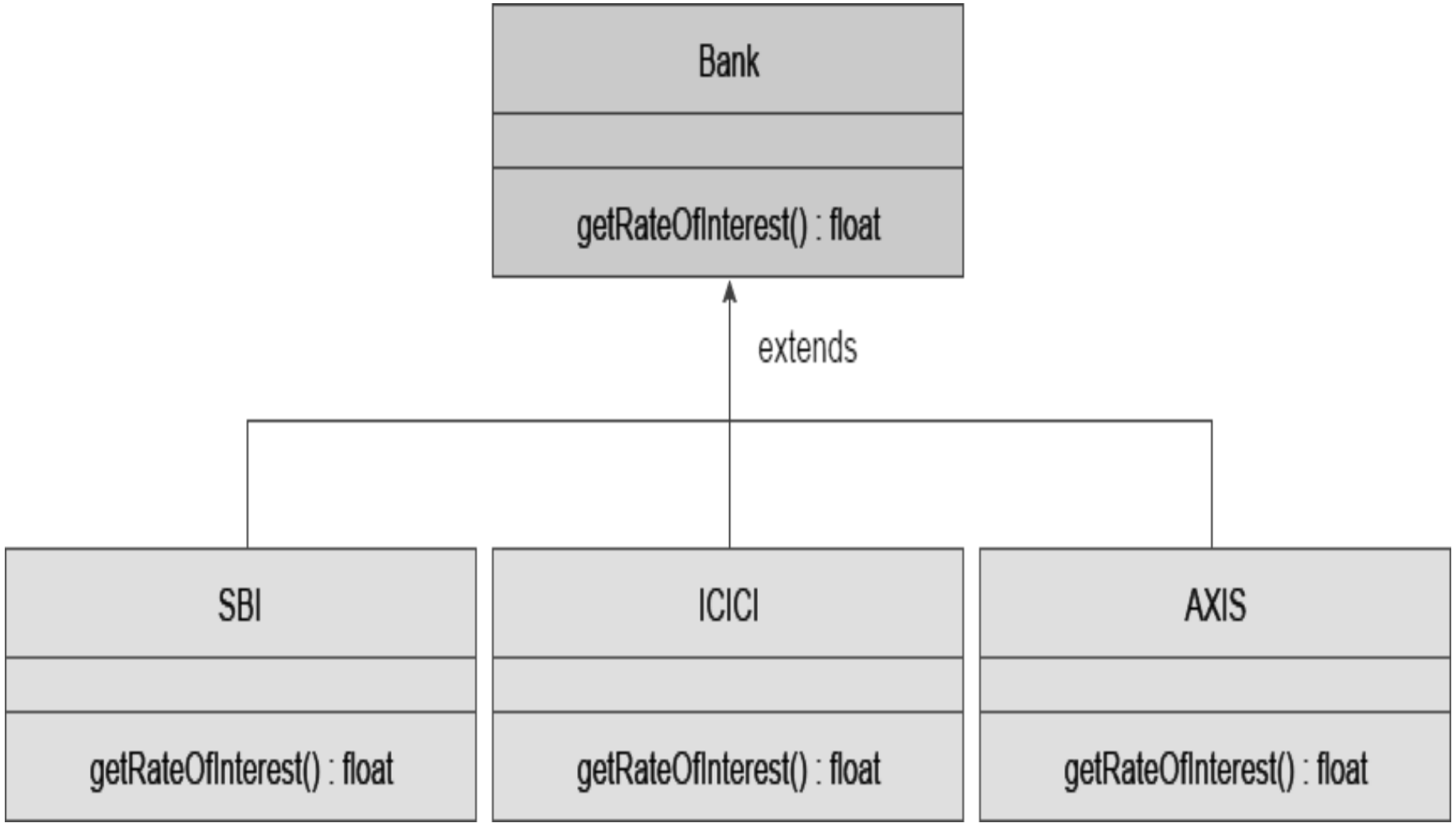
```
//Java Program to illustrate the use of Java Method Overriding
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}  }
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}
```

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

Output:

Bike is running safely

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
```

```
class AXIS extends Bank{

int getRateOfInterest(){return 9;}  }

class Test2{

public static void main(String args[]){

SBI s=new SBI();

ICICI i=new ICICI();

AXIS a=new AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}  }
```

Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch.

```
class ABC{
   //Overridden method
   public void disp()
   {
         System.out.println("disp() method of parent class");
   }
}
class Demo extends ABC{
   //Overriding method
   public void disp(){
         System.out.println("disp() method of Child class");
   }
   public void newMethod(){
         System.out.println("new method of child class");
   }
```

```
public static void main( String args[]) {
        /* When Parent class reference refers to the parent class
object
         * then in this case overridden method (the method of parent
class)
         *  is called.     */
        ABC obj = new ABC();
        obj.disp();
        /* When parent class reference refers to the child class
object
         * then the overriding method (method of child class) is
called.
         * This is called dynamic method dispatch and runtime
polymorphism
         */
```

Output:

```
disp() method of parent class

disp() method of Child class
```

Super keyword in Method Overriding

The super keyword is used for calling the parent class method/constructor. `super.myMethod()` calls the myMethod() method of base class while `super()` calls the constructor of base class.

```
class ABC{
    public void myMethod()
    {
        System.out.println("Overridden method");
    }
}
class Demo extends ABC{
    public void myMethod(){
        //This will call the myMethod() of parent class
        super.myMethod();
        System.out.println("Overriding method");
    }
    public static void main( String args[]) {
        Demo obj = new Demo();
        obj.myMethod();
    }
}
```

Output:

```
Class ABC: mymethod()
Class Test: mymethod()
```

Program to illustrate the use of method overriding in multilevel inheritance in Java:

```
class GrandFather {
 void move() {
   System.out.println("I use my stick to move!");
}}
class Father extends GrandFather {
 void move() {
   System.out.println("I can walk fast!");
 }}
```

```
class Baby extends Father {
 void move() {
   System.out.println("I crawl and have fun!");
 }
 public static void main(String[] args) {
   GrandFather ob = new Baby();
   ob.move();
 }
}
```

Output:

```
I crawl and have fun!
```

NOTE : static, final, and private access restrict the method from overriding. This means that any parent class method which has a private access specifier or is static or final in nature cannot be overridden by any of its child classes.

Abstraction in JAVA shows only the essential attributes and hides unnecessary details of the object from the user. In Java, abstraction is accomplished using Abstract class, Abstract methods, and Interfaces. Abstraction helps in reducing programming complexity and effort.

As per dictionary, abstraction is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

Abstract Class and Abstract Methods

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
 Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

A method that doesn't have its body is known as an abstract method. We use the same `abstract` keyword to create abstract methods. For example,

```
abstract void display();
```

Here, `display()` is an abstract method. The body of `display()` is replaced by `;`.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

  // abstract method
  abstract void method1();

 }
```

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java");
  }
}
```

```
abstract class Language {
  // method of abstract class
  public void display() {
    System.out.println("This is Java");  }}
class Main extends Language {
  public static void main(String[] args) {
        // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }}
```

Output

```
This is Java
```

In the above example, we have created an abstract class named `Language`. The class contains a regular method `display()`.We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```

Here, `obj` is the object of the child class `Main`. We are calling the method of the abstract class using the object `obj`.

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}
```

```
class Dog extends Animal {
  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }}
class Main {
  public static void main(String[] args) {
    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }}
```

Output

```
Bark bark
I can eat.
```

In the above example, we have created an abstract class `Animal`. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.

We have inherited a subclass `Dog` from the superclass `Animal`. Here, the subclass `Dog` provides the implementation for the abstract method `makeSound()`.

We then used the object `d1` of the `Dog` class to call methods `makeSound()` and `eat()`.

Note: If the `Dog` class doesn't provide the implementation of the abstract method `makeSound()`, `Dog` should also be declared as abstract. This is because the subclass `Dog` inherits `makeSound()` from `Animal`.

```
/* File name : Employee.java */
public abstract class Employee {
   private String name;
   private String address;
   private int number;

   public Employee(String name, String address, int number
{
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
    }
```

```
public double computePay() {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }

   public void mailCheck() {
      System.out.println("Mailing a check to " +
this.name + " " + this.address);
   }

   public String toString() {
      return name + " " + address + " " + number;
    }
```

```
public String getName() {
      return name;
   }

   public String getAddress() {
      return address;
   }

   public void setAddress(String newAddress) {
      address = newAddress;
   }

   public int getNumber() {
      return number;
   }
}
```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

```
public class Salary extends Employee {
   private double salary;   // Annual salary

    public Salary(String name, String address, int number,
double salary) {
      super(name, address, number);
      setSalary(salary);
   }
```

```
public void mailCheck() {
     System.out.println("Within mailCheck of Salary class ");
     System.out.println("Mailing check to " + getName() + " with salary "
+ salary);
   }

   public double getSalary() {
     return salary;
   }

   public void setSalary(double newSalary) {
     if(newSalary >= 0.0) {
       salary = newSalary;
     }   }

   public double computePay() {
     System.out.println("Computing salary pay for " + getName());
     return salary/52;
   }}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
public class AbstractDemo {

   public static void main(String [] args) {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);
      System.out.println("Call mailCheck using Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using Employee reference--");
      e.mailCheck();
   }
}
```

Output

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

 Call mailCheck using Employee reference--
Within mailCheck of Salary class
  Mailing check to John Adams with salary 2400.0
```

An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created. It is as shown in the program below as follows:

```
abstract class Base {

    // Constructor of class 1
    Base()
    {
        // Print statement
        System.out.println("Base
Constructor Called");    }

    // Abstract method inside class1
    abstract void fun();
}

class Derived extends Base {

    Derived()
    {
        System.out.println("Derived
Constructor Called");
    }

    void fun()
    {
        System.out.println("Derived fun()
called");
    }}
```

```
class GFG {


    public static void main(String s[])

    {

        // Creating object of class 2

        // inside main() method

        Derived d = new Derived();

    }
}
```

Output
Base Constructor Called
Derived Constructor Called

we can define static methods in an abstract class that can be called independently without an object.

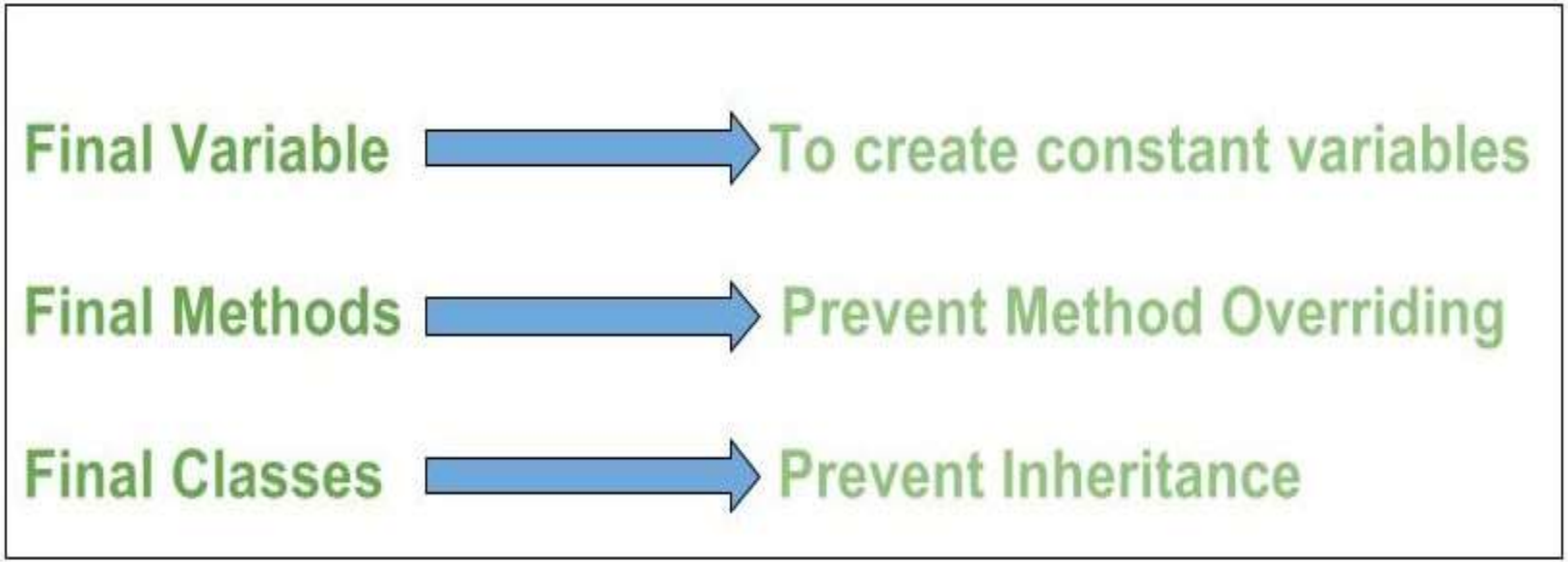
```
abstract class Helper {

    // Abstract method
    static void demofun()
    {
        System.out.println("Java is best");
    }}

public class GFG extends Helper {

    public static void main(String[] args)
    {
        // Calling method inside main()
        // as defined in above class
        Helper.demofun();
    }
}
```

Output

Java is best

final Keyword in Java

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable **only to a variable, a method or a class**. Following are different contexts where final is used.



When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited).

A class that is declared with the **final keyword** is known as the **final class**. A **final class** can't be inherited by subclasses. By use of the final class, we can restrict the **inheritance** of class. We can create a class as a **final class** only if it is complete in nature it means it must not be an **abstract class**. In java, all the wrapper classes are final class like String, Integer, etc.
If we try to inherit a final class, then the compiler throws an error at compilation time.

A class which is declared as "**final**" is called "**Final Classes**".

A class which is already has all its implementation complete so none in the world should not be able to provide additional to original class. To achieve this scenario we should declare it as "final".

For example, Java api **String class** and **System classes** are declared as final.

```
public final class Finalclass {

    public void complete() {
        System.out.println("Finalclass is complete");
    }
}
```

If final class appears in extends class anywhere than will raise compile time error. It shows that "Final class can not have any subclasses".

Note :A final class never has any subclasses as well the methods of a final class are never be overridden.

Let’s make a program where an abstract class can have a data member, constructor, abstract, final, static, and instance method (non-abstract method).

```
public abstract class AbstractClass
{
  int x = 10; // Data member.
  AbstractClass()
  {
    System.out.println("AbstractClass constructor");
  }
final void m1()
{
   System.out.println("Final method");
   }
```

```
void m2()
{
  System.out.println("Instance method");
 }
static void m3()
{
  System.out.println("Static method");
 }
  abstract void msg();
 }
public class AbsTest extends AbstractClass
{
    AbsTest()

  {

      System.out.println("AbsTest class constructor");

    }
```

```
void msg()
 {
   System.out.println("Hello Java");
 }
public static void main(String[] args)
{
  AbsTest t = new AbsTest();
   t.msg();
   t.m1();
   t.m2();
   m3();
  System.out.println("x = " +t.x);
 }

 }
```

Output:

AbstractClass constructor
AbsTest class constructor
Hello Java
Final method
Instance method
Static method

x = 10

```
class Customer
{
int account_no;
float balance_Amt;
String name;
int age;
String address;
void balance_inquiry()
{
/* to perform balance inquiry only account number
is required that means remaining properties
are hidden for balance inquiry method */
}
void fund_Transfer()
{
/* To transfer the fund account number and
balance is required and remaining properties
are hidden for fund transfer method */

  }
```

Interface

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

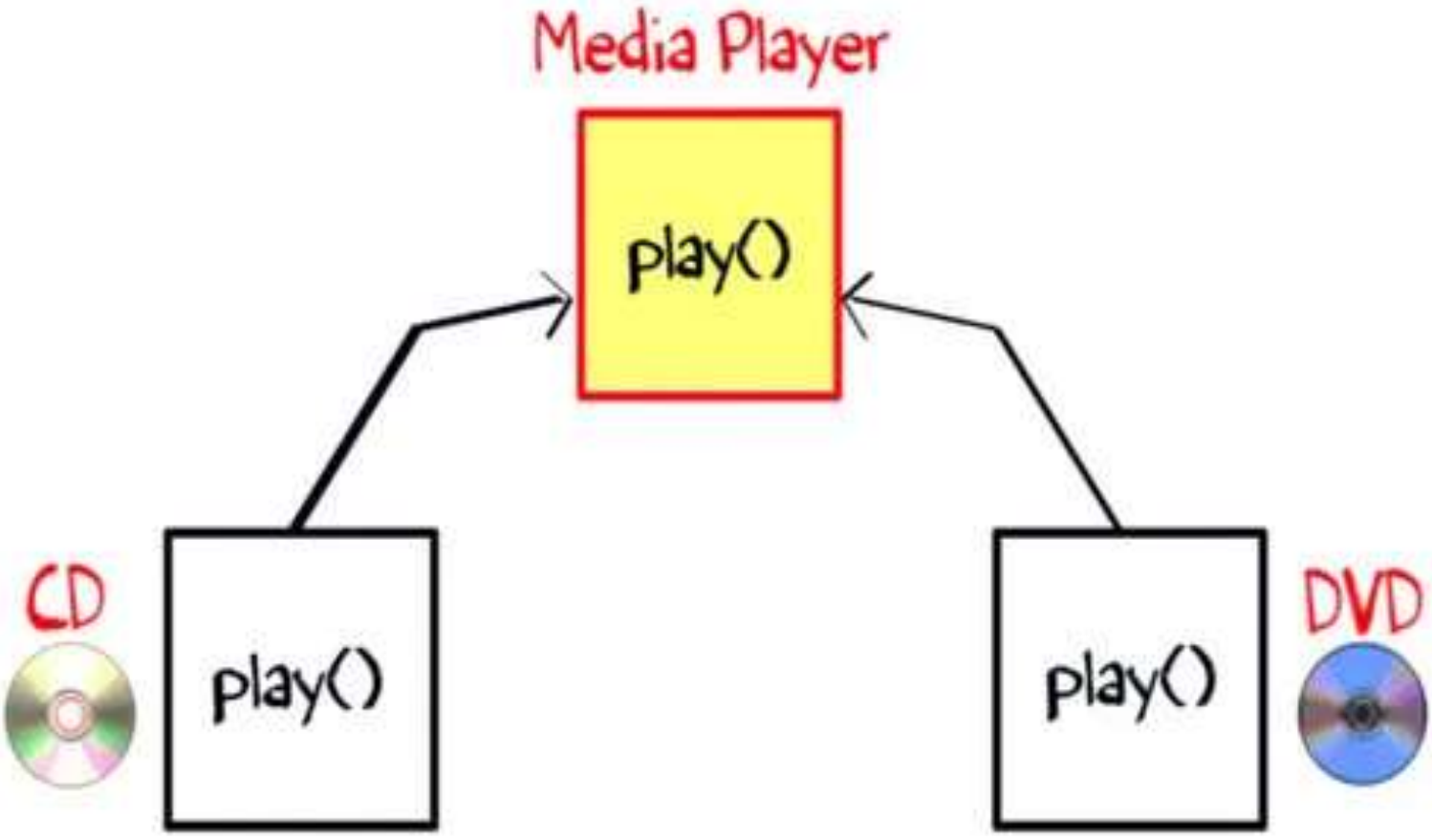
Java Interface also represents the IS-A relationship.

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.

Why use Java interface?

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

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

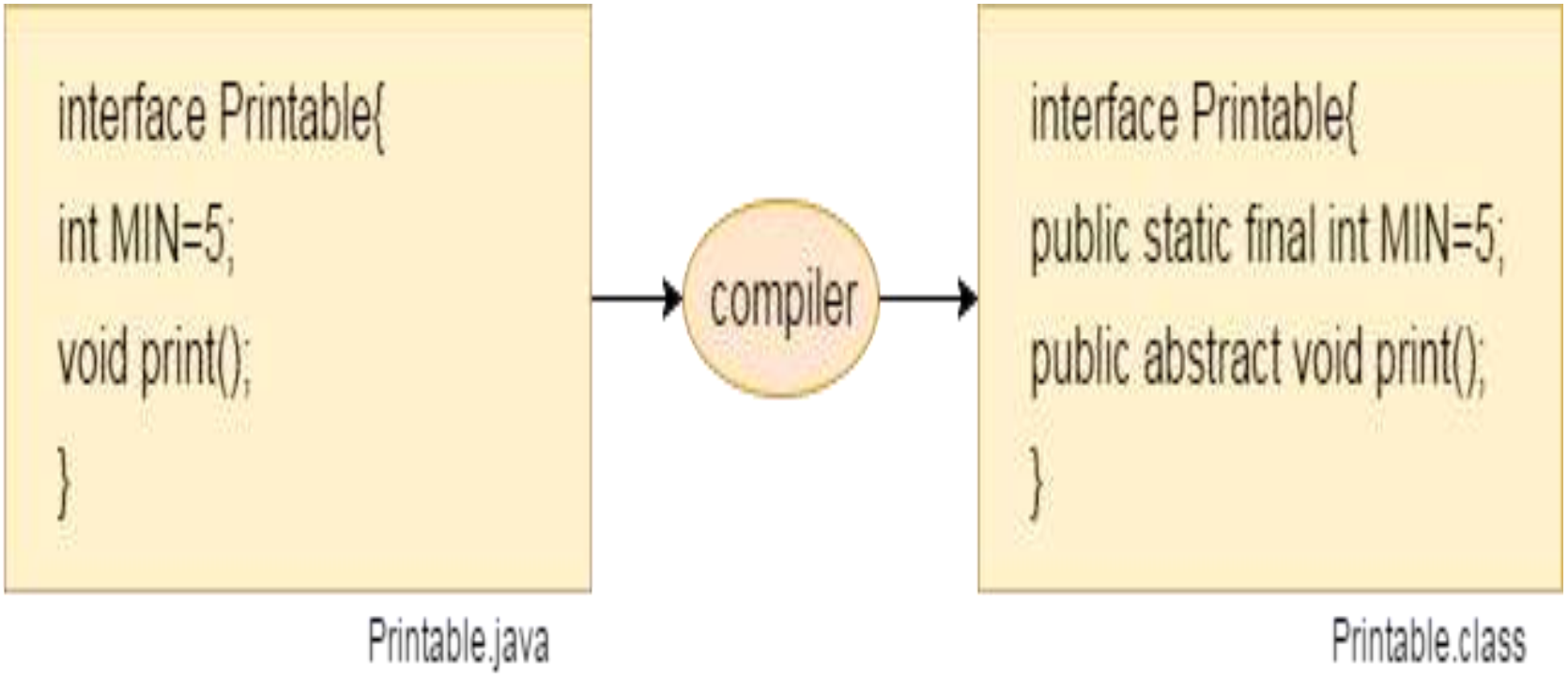

Media Player
play()
CD
play()
play()
DVD

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{

    // declare constant fields

  // declare methods that abstract

  // by default.  }
```

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



```
interface printable{

void print();

}

class A6 implements printable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

A6 obj = new A6();

obj.print();   }}
```

Output:

```
Hello
```

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void
draw(){System.out.println("drawing
rectangle");}
class Circle implements Drawable{
public void draw(){System.out.println("drawing
circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario,
object is provided by method e.g. getDrawable()
d.draw();
}}
```

Output : drawing circle

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

O/P

ROI: 9.15

Let’s consider the example of vehicles like bicycle, car, bike……, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

```
interface Vehicle {

    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
```

```
class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }
```

```
// to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed  + " gear: " +
gear);
    }
}
```

```
class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }
```

```
 // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
        System.out.println("speed: " + speed  + " gear: " + gear);
    }     }
class GFG {
     public static void main (String[] args) {
             Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();
```

```
// creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

Output;

Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface FirstInterface {
  public void myMethod(); // interface method
}
interface SecondInterface {
  public void myOtherMethod(); // interface method}
```

```
class DemoClass implements FirstInterface, SecondInterface {
  public void myMethod() {
    System.out.println("Some text..");
  }
  public void myOtherMethod() {
    System.out.println("Some other text...");
  }
}

class Main {
  public static void main(String[] args) {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }

  }
```

Realtime Example 1:

Suppose you have some rupees in your hands. You can buy from this money something from that shop where billing is done in rupees.

Suppose you go to such a shop for buying where only dollars are accepted, you cannot use your rupees there. This money is like a class. A class can fulfill only a particular requirement. It is not useful to handle different situations.

Now suppose you have a credit card. In a shop, you can easily pay in rupees by using your credit card. If you go to another shop where dollars are accepted, you can also pay in dollars. The same credit card can be used to pay in pounds also. But how is this credit card paying in different currencies? Let’s understand it.

Basically, a credit card is like an interface that performs several tasks. It is a thin plastic card that contains identification information such as your name, bank name, and perhaps some numbers.

It does not hold any money physically. But here question is that how are shop keepers able to draw money from credit card?

Behind credit card, have our bank account details from where the money is transferred to shop keepers after authentication.

This bank account can be considered as an implementation class that actually performs different tasks. See the below figure to understand the concept.

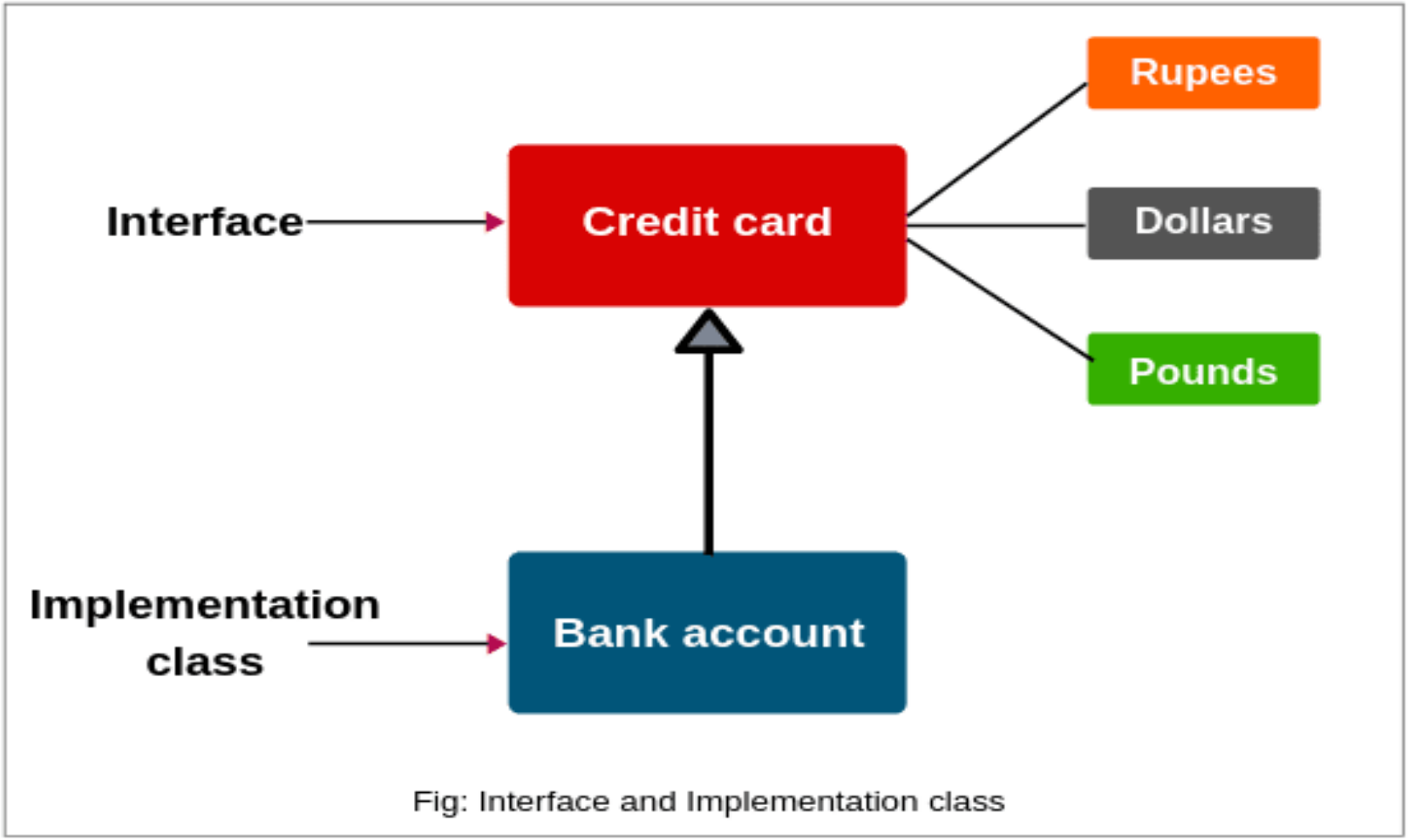


Fig: Interface and Implementation class

Example 2:

Suppose Amazon wants to integrate HDFC bank code into their shopping cart for their customer’s convenience. Amazon wants when their customers purchase any product from Amazon then they do payment through HDFC bank.

But to do payment Amazon does not have its own HDFC bank. So, they will have to take the help of the HDFC bank. Let's say HDFC developed codes like below.

```
class Transaction
{
 void withdrawAmt(int amtToWithdraw)
 {
   // logic of withdraw.
   // HDFC DB connection and updating in their DB.
  }

  }
```

Now, Amazon needs this class to integrate the payment option. So they will request HDFC bank to get this. But the problem with HDFC is that if they give the total code to Amazon then they are exposing everything of their database to them and logic will also get exposed which is a security violation.

Now what exactly Amazon needs, is method name and class name. But to compile their class they must get that class like below.

Transaction t = new Transaction();

t.withdrawAmt(500);

If the first line has to compile at Amazon end, they must have this class which HDFC cannot give.

The solution is here for HDFC that they develop an interface of Transaction class as given below.

```
Interface TransactionI
{
  void withdrawAmt(int amtToWithdraw);
   }
```

```
class TransactionImpl implements TransactionI
{
  void withdrawAmt(int amtToWithdraw)
  {
    // logic of withdraw.
    // HDFC DB connection and updating in their
DB.
  }

  }
```

Amazon now will get an interface not class, they will use it as given below.

```
TransactionI ti = new TransactionImpl();
 // right hand side may be achieve by
 // webservice or EJB


    ti.withdrawAmt(500);
```

In this case, both parties achieve their goals. Amazon only needs the method name and parameter they got it.

HDFC only want to give them name not logic so they provided. Also, it does not matter to HDFC what customers have purchased from Amazon. They just have to work for the amount deduction.

OBJECT CLONING

The object cloning is a way to create an **exact copy of an object.** For this purpose, the **clone()** method of an object class is used to clone an object. The **Cloneable** interface must be implemented by a class whose object clone to create. If we do not implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The clone() method saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed, so we can use object cloning.

Syntax

protected Object clone() throws

CloneNotSupportedException

Advantages of Cloning in Java

- Helps in reducing the lines of code.
- The most effective and efficient way of copying objects.
- Also, the clone() is considered to be the fastest method to copy an array.

```
class Student18 implements Cloneable{
int rollno;
String name;

Student18(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws CloneNotSupportedException{
return super.clone();
}
```

```
public static void main(String args[]){
try{
Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

} }
```

Output:101 amit
101 amit

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

Example

```
class Main implements Cloneable {
  String name;
  int version;
  public static void main(String[] args) {

    // create an object of Main class
    Main obj1 = new Main();

    // initialize name and version using obj1
    obj1.name = "Java";
     obj1.version = 14;
```

```
System.out.println(obj1.name);      // Java
System.out.println(obj1.version);   // 14

try {

  // create clone of obj1
  Main obj2 = (Main)obj1.clone();
  // print the variables using obj2
  System.out.println(obj2.name);     // Java
  System.out.println(obj2.version);  // 14
}
catch (Exception e) {
  System.out.println(e);
}  }}
```

Nested Classes and Java Inner Classes

In Java, just like methods, variables of a class too can have another class as its member. Writing a class within another is allowed in Java. The class written within is called the nested class, and the class that holds the inner class is called the outer class.

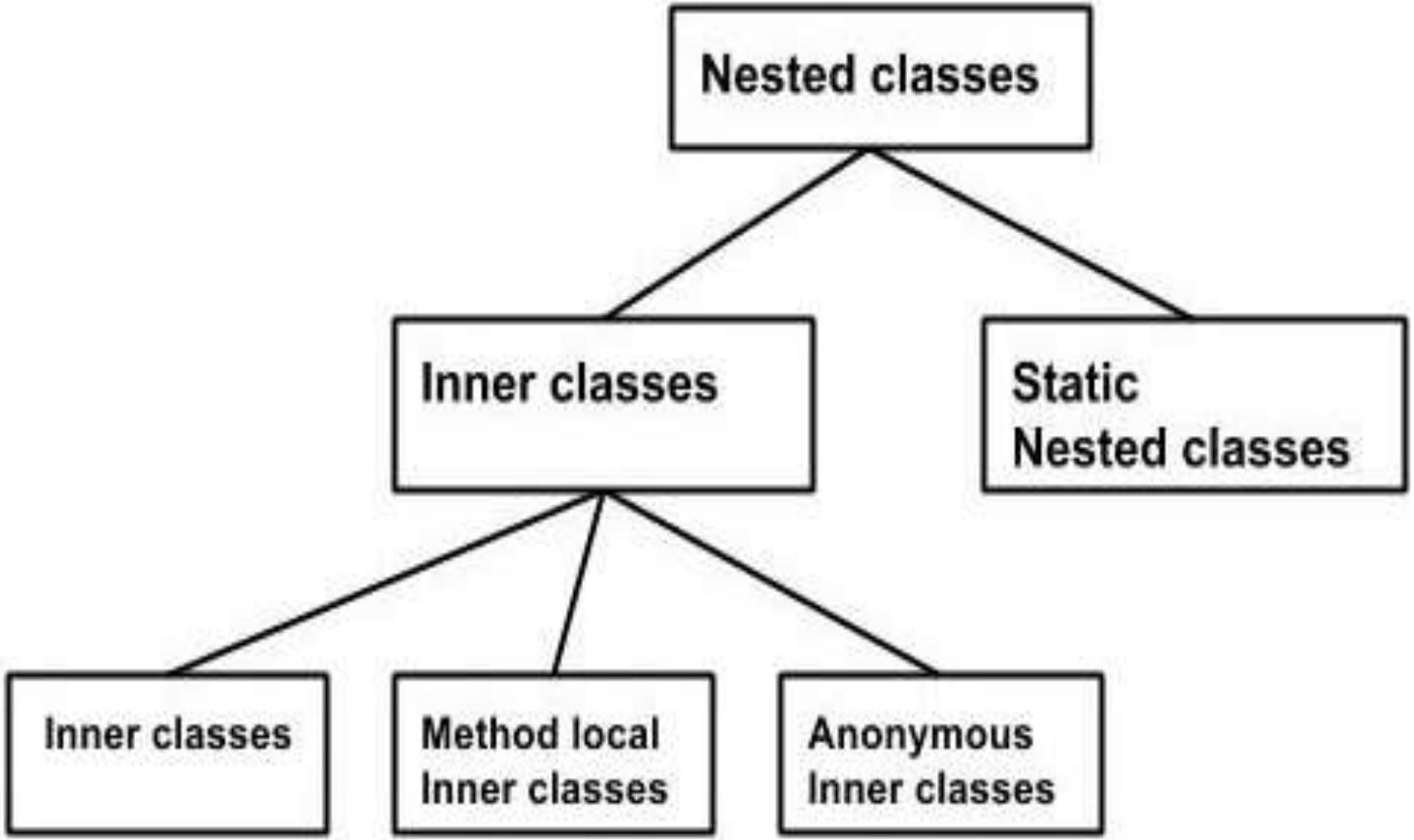
Syntax

Following is the syntax to write a nested class. Here, the class Outer_Demo is the outer class and the class Inner_Demo is the nested class.

```
class Outer_Demo
{
      class Inner_Demo
      {
      }
 }
```

Nested classes are divided into two types −

- Non-static nested classes − These are the non-static members of a class.
- Static nested classes − These are the static members of a class.


Nested classes
Inner classes
Static
Nested classes
Inner classes
Method local
Inner classes
Anonymous
Inner classes

Inner Classes

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier private, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

```
class Outer_Demo {
  int num;

  // inner class
  private class Inner_Demo {
    public void print() {
      System.out.println("This is an inner class");
    }
  }

  // Accessing the inner class from the method within
  void display_Inner() {
    Inner_Demo inner = new Inner_Demo();
    inner.print();
  }
}
```

```
public class My_class {

  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();

    // Accessing the display_Inner() method.
    outer.display_Inner();
  }
}
```

Here you can observe that Outer_Demo is the outer class, Inner_Demo is the inner class, display_Inner() is the method inside which we are instantiating the inner class, and this method is invoked from the main method.

Output

This is an inner class.

import and static import statements in Java

We can use an import statement **to import classes and interface of a particular package**. Whenever we are using import statement it is not required to use the **fully qualified name** and we can use **short name directly**. We can use **static import** to **import static member from a particular class and package**. Whenever we are using **static import** it is **not required to use the class name** to access static member and we can use directly.

import statement

- To access a class or method from another package we need to use the **fully qualified name** or we can use **import** statements.
- The class or method should also be accessible. Accessibility is based on the **access modifiers**.
- **Private** members are accessible only within the same class. So we won't be able to access a private member even with the fully qualified name or an import statement.
- The **java.lang** package is automatically imported into our code by Java.

```
import java.util.Vector;
public class ImportDemo {
   public ImportDemo() {
   //Imported using keyword, hence able to access
directly in the code without package qualification.
      Vector v = new Vector();
      v.add("Tutorials");
      v.add("Point");
      v.add("India");
      System.out.println("Vector values are: "+ v);
```

```
//Package not imported, hence referring to it using
the complete package.
      java.util.ArrayList list = new java.util.ArrayList();
      list.add("Tutorix");
      list.add("India");
      System.out.println("Array List values are: "+ list);
   }
   public static void main(String arg[]) {
      new ImportDemo();
   }
}
```

Output

```
Vector values are: [Tutorials, Point, India]

 Array List values are: [Tutorix, India]
```

Static Import Statement

- **Static imports** will import all static data so that can use **without a class name**.
- A **static import** declaration has two forms, one that imports a particular static member which is known as **single static impor**t and one that imports all **static members of a class** which is known as a **static import on demand**.
- Static imports introduced in **Java5 version**.
- One of the advantages of using static imports is **reducing keystrokes and re-usability**.

Example

```
import static java.lang.System.*;
//Using Static Import
public class StaticImportDemo {
   public static void main(String args[]) {
      //System.out is not used as it is imported using
the keyword stati.
      out.println("Welcome to Geetanjali");
   }
}
```

For Example: we always use sqrt() method of Math class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly.

Output:

```
2.0
4.0
6.3
```

```
// Java Program to illustrate
// calling of predefined methods
// without static import
class Geeks {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }}
```

```
// Java Program to illustrate
// calling of predefined methods
// with static import
import static java.lang.Math.*;
class Test2 {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

Output:

```
2.0
4.0
6.3
```

```
/ Java to illustrate calling of static member of
/ System class without Class name
mport static java.lang.Math.*;
mport static java.lang.System.*;
lass Geeks {
    public static void main(String[] args)
    {
        // We are calling static member of System
lass
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }}
```

Output:

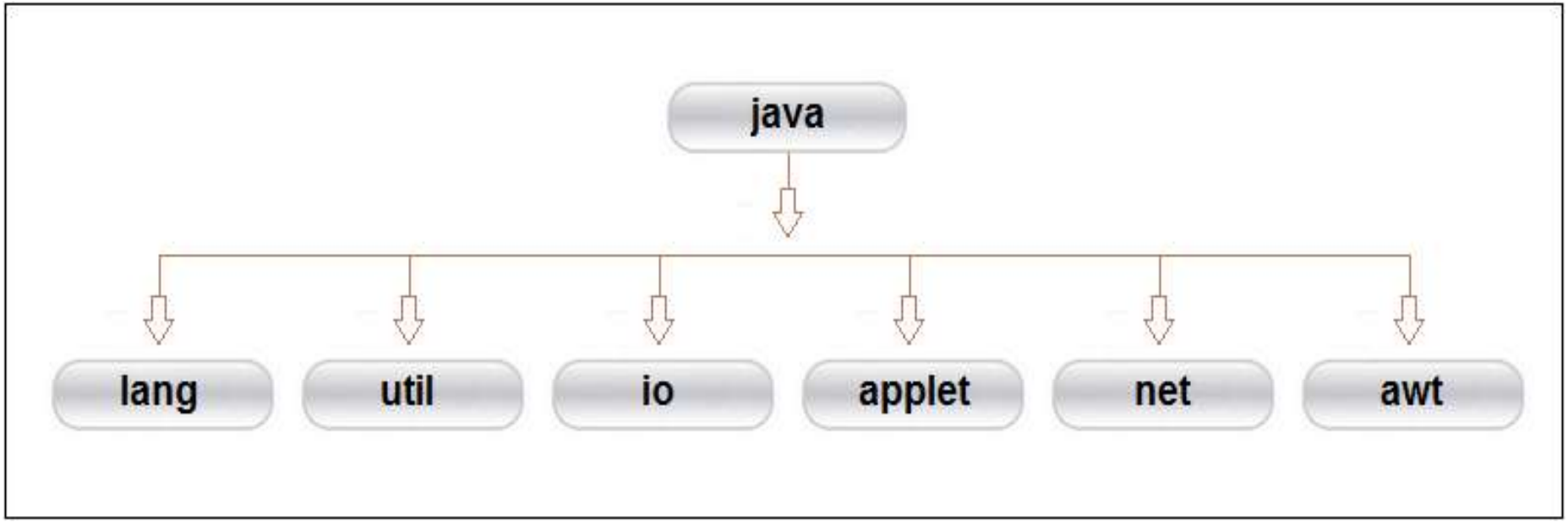
```
2.0
4.0
6.3
```

Packages & API

Java Application Programming Interface (API)

Java application programming interface (API) is a list of all classes that are part of the Java development kit (JDK). It includes all Java packages, classes, and interfaces, along with their methods, fields, and constructors. These prewritten classes provide a tremendous amount of functionality to a programmer. A programmer should be aware of these classes and should know how to use them

Java **API(Application Program Interface)** provides a large numbers of classes grouped into different packages according to functionality. Most of the time we use the packages available with the the Java API. Following figure shows the system packages that are frequently used in the programs.



A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Syntax

import *package.name.Class*; // Import a single class

import *package.name*.*; // Import the whole package

Import a Class

If you find a class you want to use, for example, the `Scanner` class, which is used to get user input, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, **java.util** is a package, while **Scanner** is a class of the **java.util** package.

To use the **Scanner** class, create an object of the class and use any of the available methods found in the **Scanner** class documentation. In our example, we will use the **nextLine()** method, which is used to read a complete line:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " +
userName);
  }

}
```

Import a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

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

java.lang	Language support classes. They include classes for primitive types, string, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, data, etc.
java.io	Input/output support classes. They provide facilities for the input and output of data.
java.applet	Classes for creating and implementing applets.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
javax.swing	Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.

Java Wrapper Classes

Wrapper classes provide a way to use primitive data types (`int`, `boolean`, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte
short	Short

Primitive Data Type	Wrapper Class
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Example 1: Primitive Types to Wrapper Objects

```
public class Wrapping {
public static void main(String[] args)
{
  int a = 50; // Primitive data type value.

 Integer i = Integer.valueOf(a); // Here, we are converting int into Integer explicitly.
 Integer j = a; // Here, Autoboxing is happening. Java compiler will write Integer.valueOf(a) internally.

 System.out.println(a+" "+i+" "+j);
  }}
```

Output:

50 50 50

The valueOf() method of Integer class is used to int number into Integer object. The return type of this method is object. This method is static in nature. Therefore, we call it using its class name.

Example 2: Wrapper Objects into Primitive Types

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

    // creates objects of wrapper class
    Integer aObj = Integer.valueOf(23);
    Double bObj = Double.valueOf(5.55);
    // converts into primitive types
    int a = aObj.intValue();
    double b = bObj.doubleValue();

    System.out.println("The value of a: " + a);
    System.out.println("The value of b: " + b);
  }}
```

Output

```
The value of a: 23
The value of b: 5.55
```

String class :

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

- ***String literal***

  ```
  String s = “Programming”;
  ```

- **Using *new* keyword**

  ```
  String s = new String (“Programming”);
  ```

```
public class Example{
    public static void main(String args[]){
    //creating a string by java string literal
    String str = "Javaprogram";
    char arrch[]={'h','e','l','l','o'};
    //converting char array arrch[] to string str2
    String str2 = new String(arrch);

    //creating another java string str3 by using new
    keyword
    String str3 = new String("Java String Example");

    //Displaying all the three strings
    System.out.println(str);
    System.out.println(str2);
    System.out.println(str3);   }}
```

String Methods:

length(): The Java String length() method tells the length of the string. It returns count of total number of characters present in the String. For example:

```
public class Example{
public static void main(String args[]{
String s1="hello";
String s2="whatsup";
System.out.println("string length is: "+s1.length());
System.out.println("string length is: "+s2.length());
}}
```

Here, String length() function will return the length 5 for s1 and 7 for s2 respectively

compareTo(): The Java String compareTo() method compares the given string with current string.

```
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="hemlo";
String s4="flag";
System.out.println(s1.compareTo(s2)); // 0 because
both are equal
System.out.println(s1.compareTo(s3)); //-1 because
"l" is only one time lower than "m"
System.out.println(s1.compareTo(s4)); // 2 because
"h" is 2 times greater than "f"}
```

This program shows the comparison between the various string. It is noticed that
if s1 > s2, it returns a positive number
if s1 < s2, it returns a negative number
if s1 == s2, it returns 0

concat() : The Java String concat() method combines a specific string at the end of another string and ultimately returns a combined string. It is like appending another string. For example:

```
public class ConcatExample{
public static void main(String args[]){
String s1="hello";
s1=s1.concat("how are you");
System.out.println(s1);
}}
```

The above code returns “hellohow are you”.

toUpperCase() and toLowerCase() :

The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
} }
```

Output:

SACHIN
sachin
Sachin

valueOf() :

The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.

```
public class Str7{
public static void main(String ar[]) {
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
```

Output:

1010

replace() :

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
public class Str8  {

public static void main(String ar[])  {

String s1="Java is a programming language. Java is a platform. Java is an

Island.";

String repString=s1.replace("Java","Kava");//replaces all occurrences of "Java"

to "Kava"

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

Output:

```
Kava is a programming language. Kava is a
```

String toString():

This method returns the String equivalent of the object that invokes it. This method does not have any parameters. Given below is the program where we will try to get the String representation of the object.

```
import java.lang.String;
import java.lang.*;
public class StringMethods {
    public static void main(String[] args) {
        Integer obj = new Integer(10);
        String str = obj.toString();
        String str2 = obj.toString(80);
        System.out.println("The String representation is " + str);
        System.out.println("The String representation is " + str2)
            }}
```

StringBuffer class:

- A string buffer is like a String, but can be modified.
- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

Constructors

StringBuffer() **This constructs a string buffer with no characters in it and an initial capacity of 16 characters.**	`StringBuffer s = new StringBuffer();`
StringBuffer(String str): It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.	`StringBuffer s = new StringBuffer("JavaforProgramming");`

Methods	Action Performed
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method
insert()	Inserts text at the specified index position
length()	Returns length of the string
reverse()	Reverse the characters within a StringBuffer object

Java.util Package:

The basic utility classes required to a programmer are present in this package. It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes .

To use any class you have to import java.util package at top of the program:-

import java.util.*;

or

import java.util.Class_name;

- Date :- This class represents a specific instant in time, with millisecond precision.

```
import java.util.Date;
//or
//import java.util.*;

public class Demo {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println("The date is : " + date);
    }
}
```

- Random :- An instance of this class is used to generate a stream of pseudorandom numbers.

```
import java.util.Random;
public class RandomNumbers{
        public static void main(String[] args) {
        Random objGenerator = new Random();
            for (int iCount = 0; iCount< 10; iCount++){
             int randomNumber = objGenerator.nextInt(100);
             System.out.println("Random No : " + randomNumber);
 } }}
```

Output:

```
Random No : 17
Random No : 57
Random No : 73
Random No : 48
Random No : 68
Random No : 86
Random No : 34
Random No : 97
Random No : 73

Random No : 18
```

Date class :

The java.util.Date class represents date and time in java. It provides constructors and methods to deal with date and time in java.

Constructors :

Date( ): This constructor initializes the object with the current date and time.

Date(long millisec): This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

```
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        Date d1 = new Date();
        System.out.println("Current date is " + d1);
        Date d2 = new Date(2323223232L);
        System.out.println("Date represented is "+ d2 );
    }
}
```

Output:

```
Current date is Tue Jul 12
18:35:37 IST 2016
Date represented is Wed Jan 28
02:50:23 IST 1970
```

```
import java.util.Date;
public class DateDemo {
 public static void main(String args[]) {
  // Instantiating a Date object
   Date date = new Date();
  // display time and date using toString() method
   System.out.println("The current date and time is:
");
   System.out.println(date.toString());
 }
}
```

Output:
The current date and time is:
Tue Apr 07 03:20:26 IST 2020

GregorianCalendar Class:

GregorianCalendar is a concrete subclass(one which has implementation of all of its inherited members either from interface or abstract class) of a **Calendar** that implements the most widely used Gregorian Calendar with which we are familiar.

Constructors :

GregorianCalendar():In order to initialize the object with the current time in the default time zone with the default locale, GregorainCalendar() is used.

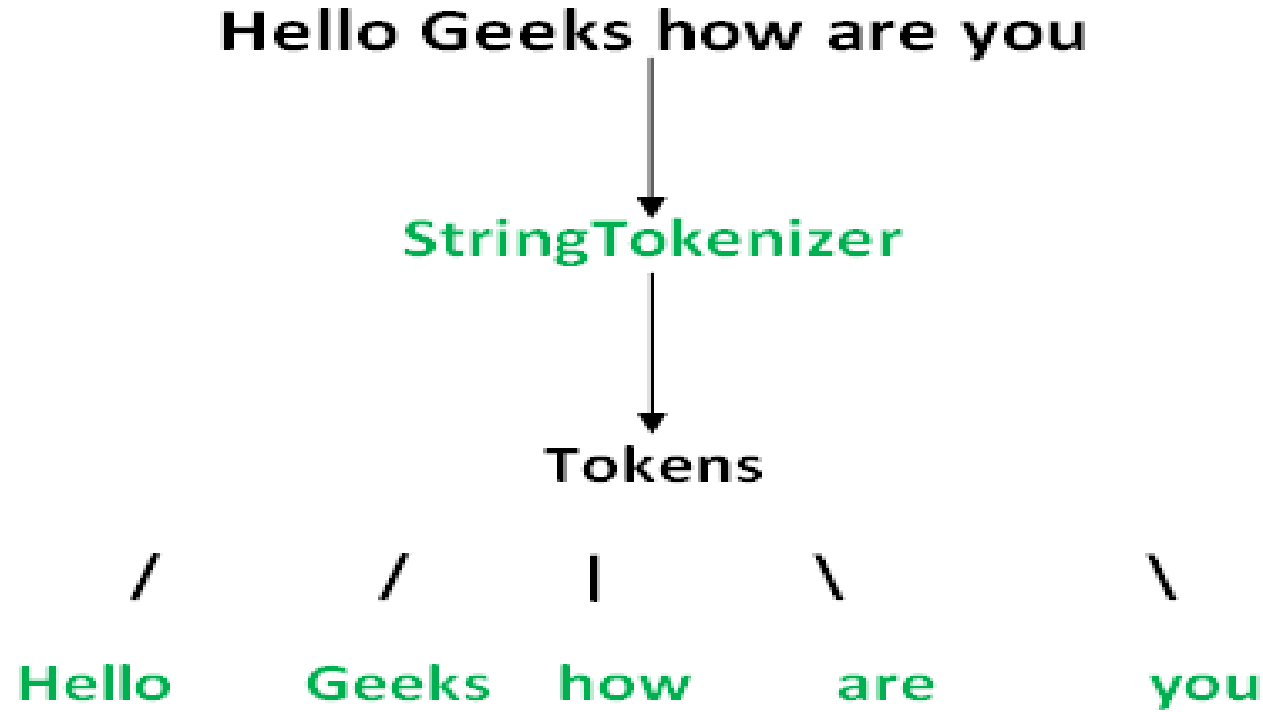
GregorianCalendar(int year, int month, int day):In order to initialize the object with the date-set defined in the default locale and time zone, GregorianCalendar(int year, int month, int day) is used.

```
import java.util.*;
public class GregorianCalendarClass{
 public static void main(String args[]) {
    String months[] = {"Jan", "Feb", "Mar", "Apr", "May",
"Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
    int year;
    GregorianCalendar gcalendar = new GregorianCalendar();
    System.out.print("Date: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + "
");
    System.out.println(year = gcalendar.get(Calendar.YEAR));
    System.out.print("Time: ");
```

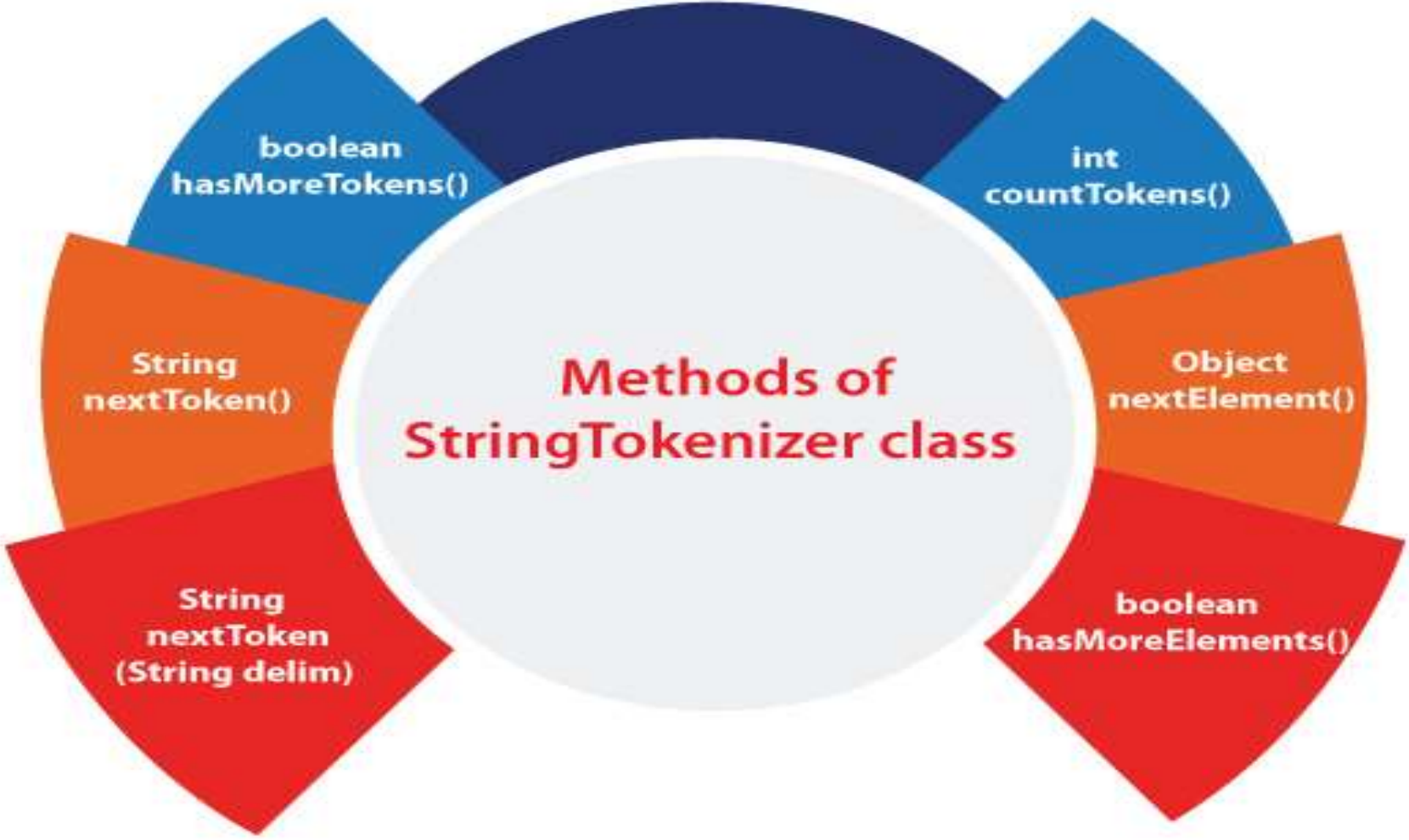
```
System.out.print(gcalendar.get(Calendar.HOUR) + ":");
   System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
   System.out.println(gcalendar.get(Calendar.SECOND));
   if(gcalendar.isLeapYear(year))
   {
     System.out.println(" current year is a leap year");
   }
   else
   {
     System.out.println("current year is not a leap year");
   } }}
```

StringTokenizer Class:

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String.



Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.


boolean
hasMoreTokens()
int
countTokens()
String
nextToken()
Object
nextElement()
Methods of
StringTokenizer class
String
nextToken
(String delim)
boolean
hasMoreElements()

Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.

```
import java.util.StringTokenizer;

public class Simple{

 public static void main(String args[]){

   StringTokenizer st = new StringTokenizer("my name is geet"," ");

     while (st.hasMoreTokens()) {

         System.out.println(st.nextToken());

     }

   }

}
```

Output:

```
my
name
is
geet
```

```
import java.util.StringTokenizer;
public class StringTokenizer1
{
 public static void main(String args[])
 {
   /* StringTokenizer object */
   StringTokenizer st = new StringTokenizer("Demonstrating methods from StringTokenizer class"," ");
     /* Checks if the String has any more tokens */
     while (st.hasMoreTokens())
     {
        System.out.println(st.nextToken());        }    }   }
```

Output:

```
Demonstrating
methods
from
StringTokenizer
class
```

Collections in Java:

Any group of individual objects which are represented as a single unit is known as the collection of the objects. **The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.**

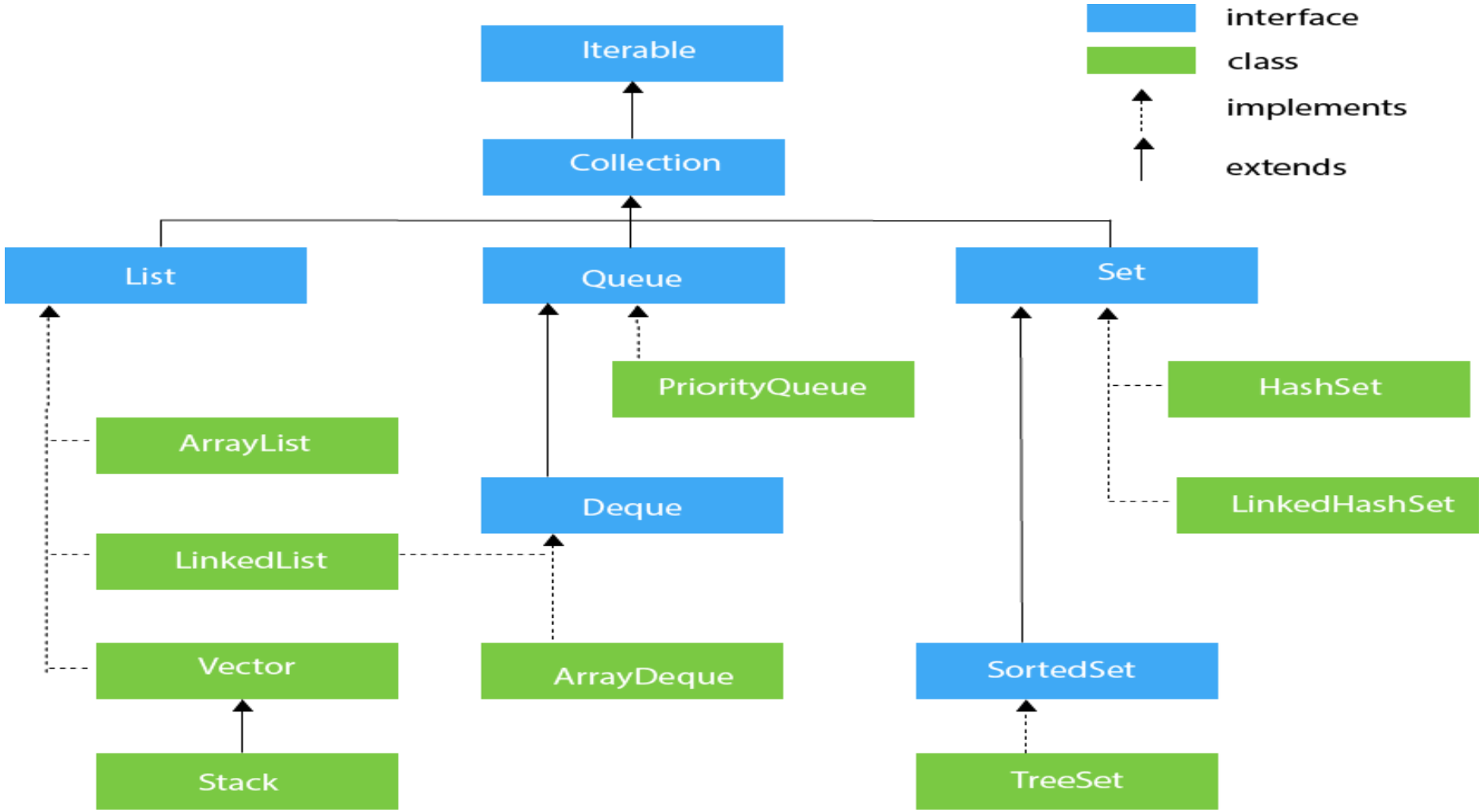
Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (Vector, HashTable, LinkedList, SortedSet,).

Why use Java collection?

There are several benefits of using Java collections such as:Reducing the effort required to write the code by providing useful data structures and algorithms

- Java collections provide high-performance and high-quality data structures and algorithms thereby increasing the speed and quality
- Unrelated APIs can pass collection interfaces back and forth
- Decreases extra effort required to learn, use, and design new API’s
- Supports reusability of standard data structures and algorithms


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

Vector Class:

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

- Vector is synchronized.
- Vector contains many legacy (inherited) methods that are not part of the collections framework.

Sr.No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c.

Methods :

- void add(int index, Object element)

Inserts the specified element at the specified position in this Vector.

- boolean add(Object o)

Appends the specified element to the end of this Vector.

- int capacity()

Returns the current capacity of this vector.

- void clear()

Removes all of the elements from this vector.

- void addElement(Object obj)

Adds the specified component to the end of this vector, increasing its size by one.

```
import java.io.*;
import java.util.*;
class GFG {
     public static void main(String[] args)
    {
        // Size of the Vector
        int n = 5;
         // Declaring the Vector with
        // initial size n
        Vector<Integer> v = new
Vector<Integer>(n);
```

```
// Appending new elements at the end of the vector
        for (int i = 1; i <= n; i++)
            v.add(i);

           System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the vector after deletion
        System.out.println(v);

        // iterating over vector elements usign for loop
        for (int i = 0; i < v.size(); i++)

            // Printing elements one by one
            System.out.print(v.get(i) + " ");    }}
```

Output

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Hashtable:

The **Hashtable** class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

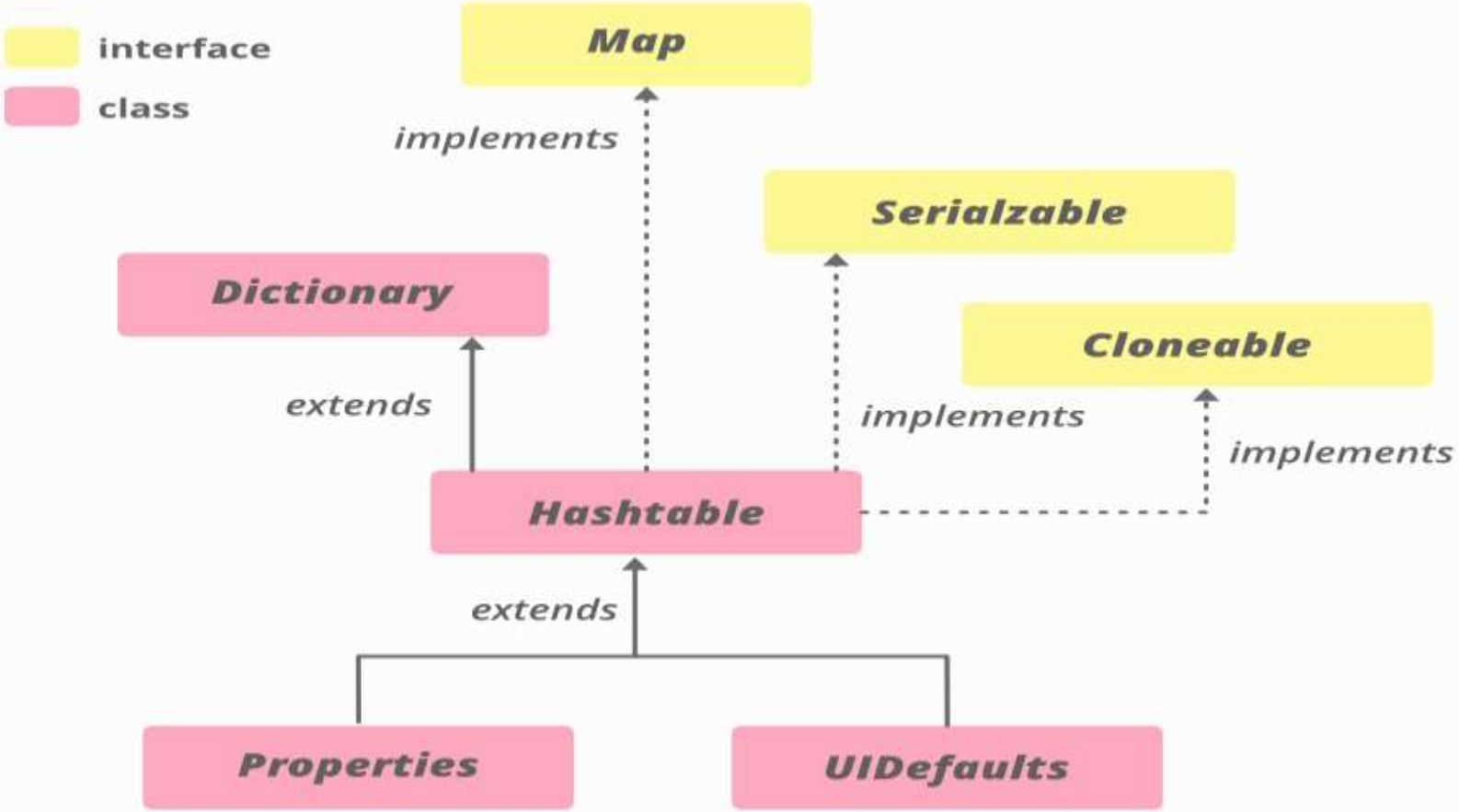
declaration for java.util.Hashtable class

public class Hashtable<K,V> **extends** Dictionary<K,V>

implements Map<K,V>, Cloneable, Serializable

the Parameters for java.util.Hashtable class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.


interface
class
Map
implements
Serialzable
Dictionary
Cloneable
extends
implements
implements
Hashtable
extends
Properties
UIDefaults

Constructor	Description
Hashtable()	It creates an empty hashtable having the initial default capacity and load factor.
Hashtable(int capacity)	It accepts an integer parameter and creates a hash table that contains a specified initial capacity.

```
// adding elements to Hashtable
import java.io.*;
import java.util.*;
class AddElementsToHashtable {
    public static void main(String args[])
    {
        // No need to mention the Generic type twice
        Hashtable<Integer, String> ht1 = new
Hashtable<>();
        // Initialization of a Hashtable using Generics
        Hashtable<Integer, String> ht2
            = new Hashtable<Integer, String>();
```

```
// Inserting the Elements using put() method
        ht1.put(1, "one");
        ht1.put(2, "two");
        ht1.put(3, "three");

        ht2.put(4, "four");
        ht2.put(5, "five");
        ht2.put(6, "six");

        // Print mappings to the console
        System.out.println("Mappings of ht1 : " + ht1);
        System.out.println("Mappings of ht2 : " + ht2);
    }}
```

Output

```
Mappings of ht1 : {3=three, 2=two, 1=one}
Mappings of ht2 : {6=six, 5=five, 4=four}
```

```
import java.util.*;
class Hashtable1{
 public static void main(String args[]){
  Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");

  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }  } }
```

LinkedList:

Linked List is a part of the Collection framework present in java.util package. This class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.

- The elements are linked using pointers and addresses.
- Each element is known as a node.
- Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.

Constructors in the LinkedList:

1. LinkedList(): This constructor is used to create an empty linked list. If we wish to create an empty LinkedList with the name ll, then, it can be created as:

```
LinkedList ll = new LinkedList();
```

2. LinkedList(Collection C): This constructor is used to create an ordered list that contains all the elements of a specified collection, as returned by the collection's iterator. If we wish to create a LinkedList with the name ll, then, it can be created as:

```
LinkedList ll = new LinkedList(C);
```

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){
  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }   }  }
```

Output: Ravi
Vijay
Ravi
Ajay

SortedSet Interface:

The SortedSet interface present in java.util package extends the Set interface present in the collection framework. It is an interface that implements the mathematical set. This interface contains the methods inherited from the Set interface and adds a feature that stores all the elements in this interface to be stored in a sorted manner.

The SortedSet interface is declared as:

public interface SortedSet extends Set

```
// Java program to demonstrate the Sorted Set
import java.util.*;
class SortedSetExample{
      public static void main(String[] args)
    {
        SortedSet<String> ts   = new TreeSet<String>();
          // Adding elements into the TreeSet   using add()
        ts.add("India");
        ts.add("Australia");
        ts.add("South Africa");

        // Adding the duplicate element
        ts.add("India");

        // Displaying the TreeSet
        System.out.println(ts);
```

```
// Removing items from TreeSet  using remove()
    ts.remove("Australia");
    System.out.println("Set after removing "+ "Australia:" + ts);

    // Iterating over Tree set items
    System.out.println("Iterating over set:");
    Iterator<String> i = ts.iterator();
    while (i.hasNext())
        System.out.println(i.next());
}}
```

Output:

```
[Australia, India, South
Africa]
Set after removing
Australia:[India, South Africa]
Iterating over set:
India
South Africa
```

Stack:

Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of last-in-first-out.

The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Declaration:

```
public class Stack<E> extends Vector<E>
```

Sr. No.	Method & Description
1	boolean empty() Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	Object peek( ) Returns the element on the top of the stack, but does not remove it.

3	Object pop( ) Returns the element on the top of the stack, removing it in the process.
4	Object push(Object element) Pushes the element onto the stack. Element is also returned.
5	int search(Object element) Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

```
import java.util.*;
public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);   }
   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);   }
```

```
public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("empty stack");
    }  }}
```

Output

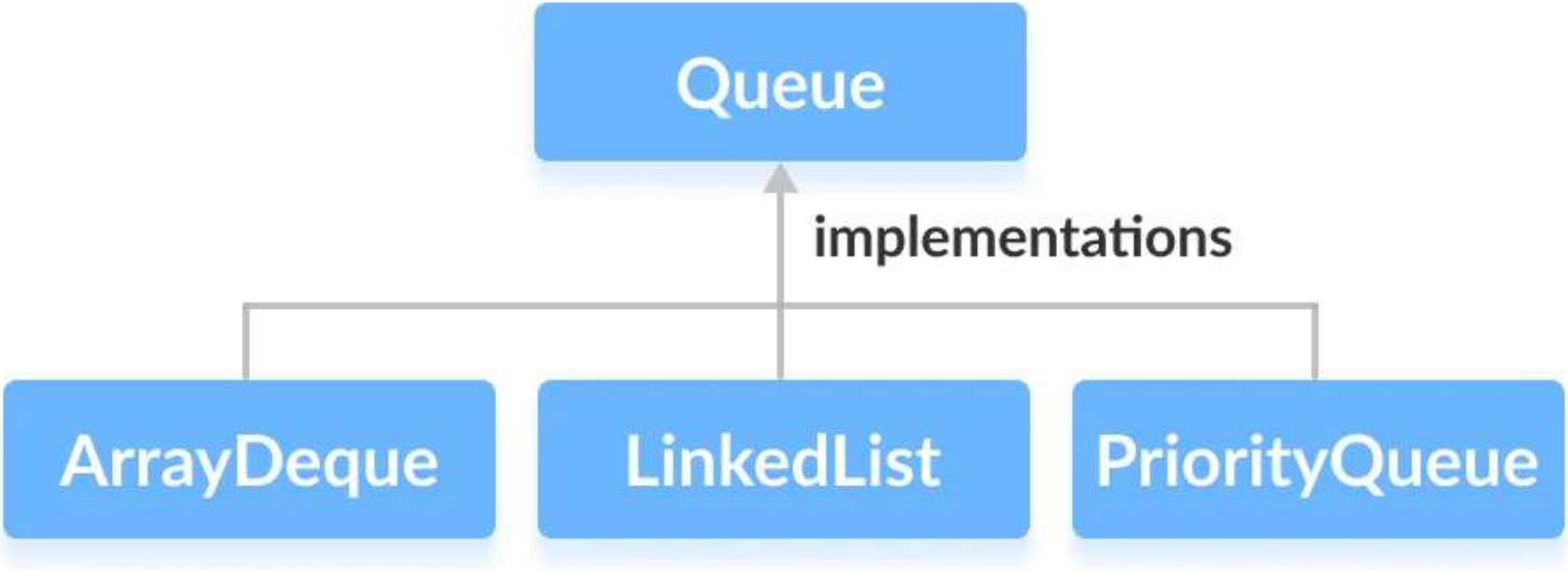
```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

Queue Interface:

The Queue interface present in the java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.

Queue Interface declaration

```
public interface Queue<E> extends Collection<E>
```



```
// Java program to demonstrate a Queue

import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {

    public static void main(String[] args)
    {
        Queue<Integer> q
            = new LinkedList<>();

        // Adds elements {0, 1, 2, 3, 4} to
        // the queue
        for (int i = 0; i < 5; i++)
            q.add(i);
```

```
// Display contents of the queue.
        System.out.println("Elements of
queue "   + q);

        // To remove the head of queue.
        int removedele = q.remove();
        System.out.println("removed element-
"   + removedele);

        System.out.println(q);
```

```
 // To view the head of queue
        int head = q.peek();
        System.out.println("head of queue-"  + head);

        // Rest all methods of collection
        // interface like size and contains
        // can be used with this
        // implementation.
        int size = q.size();
        System.out.println("Size of queue-" + size);
    }
}
```

Output:

```
Elements of queue [0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
Size of queue-4
```

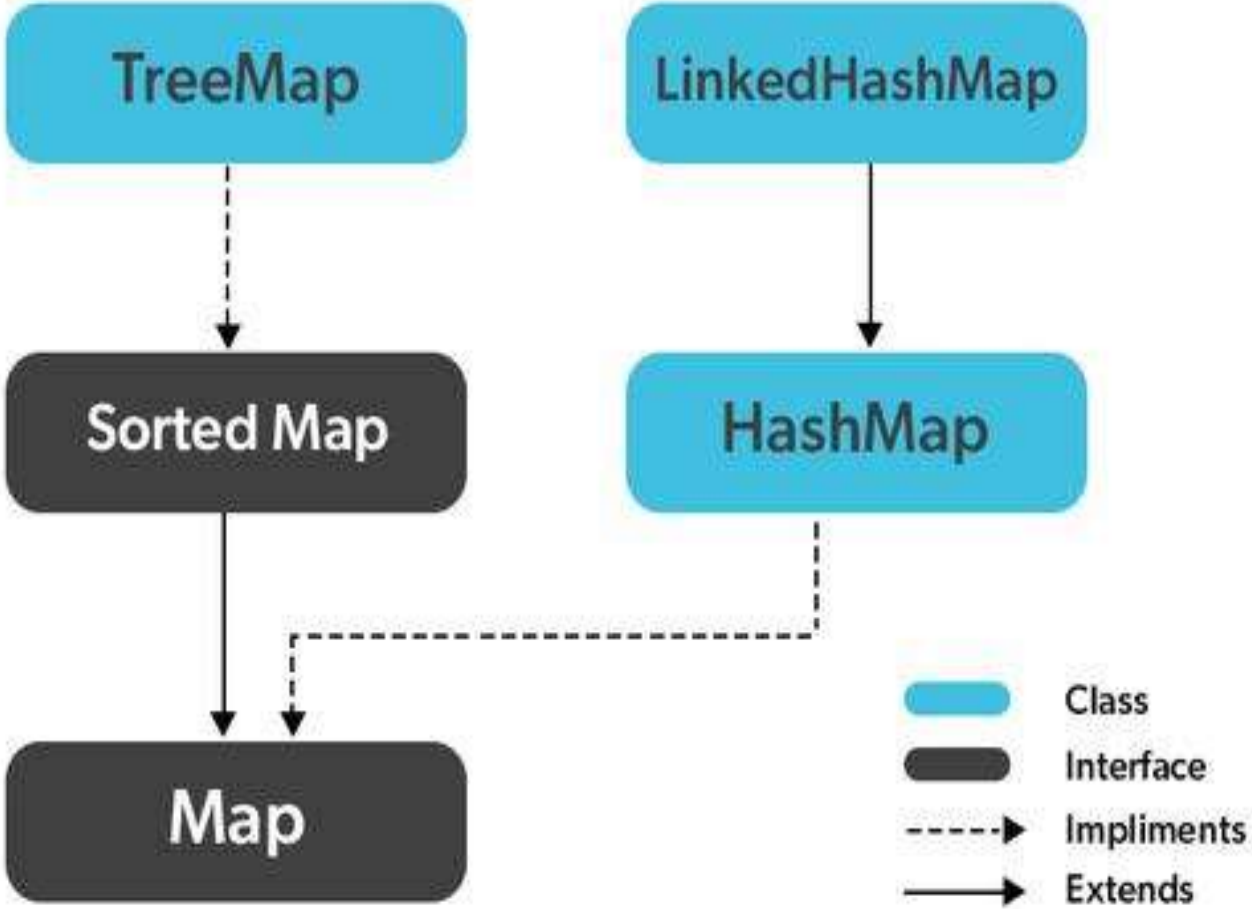
Map Interface:

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

Maps are perfect to use for key-value association mapping such as dictionaries.

Some common scenarios are as follows:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).


TreeMap
LinkedHashMap
Sorted Map
HashMap
Map
Class
Interface
Impliments
Extends

Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object.

Syntax:

```
Map hm = new HashMap();

  // Obj is the type of the object to be
  stored in Map
```

```
// Java Program to Demonstrate Working of Map interface
import java.util.*;
class GFG {
    public static void main(String args[])
    {
        // Creating an empty HashMap
        Map<String, Integer> hm
            = new HashMap<String, Integer>();

        // Inserting pairs in above Map    // using put() method
        hm.put("a", new Integer(100));
        hm.put("b", new Integer(200));
        hm.put("c", new Integer(300));
        hm.put("d", new Integer(400));
```

```
// Traversing through Map using for-each loop
        for (Map.Entry<String, Integer> me :
             hm.entrySet()) {

            // Printing keys
            System.out.print(me.getKey() + ":");
            System.out.println(me.getValue());
        }
    }
}
```

Output:

```
a:100
b:200
c:300
d:400
```

User-Defined Packages:

Conceptually, you can think of java packages as being similar to different folders on your computer.Java package is a mechanism of grouping similar type of classes, interfaces, and sub-classes collectively based on functionality. In other words, we can say a package is a container of a group of related classes where some of the classes are accessible are exposed, and others are kept for internal purposes.

Using packages while coding offers a lot of advantages like:

- **Re-usability: The classes contained in the packages of another program can be easily reused**
- **Name Conflicts: Packages help us to uniquely identify a class, for example, we can have *company.sales.Employee* and *company.marketing.Employee* classes**
- **Controlled Access: Offers access protection such as protected classes, default classes and private class**
- **Data Encapsulation: They provide a way to hide classes, preventing other programs from accessing classes that are meant for internal use only**
- **Maintainance: With packages, you can organize your project better and easily locate related classes**

Creating a Package in Java

Creating a package in Java is a very easy task. Choose a name for the package and include a *package* command as the first statement in the Java source file. The java source file can contain the classes, interfaces, enumerations, and annotation types that you want to include in the package. For example, the following statement creates a package named ***MyPackage***.

Syntax :

```
package MyPackage;
```

The package statement simply specifies to which package the classes defined belongs to..

Note: If you omit the package statement, the class names are put into the default package, which has no name. Though the default package is fine for short programs, it is inadequate for real applications.

To create a class inside a package, you should declare the package name as the first statement of your program. Then include the class as part of the package. But, remember that, a class can have only one package declaration. Here's a simple program to understand the concept.

```
package MyPackage;
public class gfg {
     public void show()
    {
        System.out.println("Hello students!!”);
    }
      public static void main(String args[])
    {
        gfg obj = new gfg();
        obj.show();     }}
```

Output :

```
Hello students!!
```

we'll import the user-defined package "example" created in the above example. And use the function to print messages.

```
import MyPackage.gfg;

public class GFG {
    public static void main(String args[])
    {
        gfg obj = new gfg();
        System.out.println(obj.show());
    }
}
```

Output :

```
Hello students!!
```

```
package mypack;

public class Simple{

 public static void main(String args[]){

   System.out.println("Welcome to package");

  } }
```

How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

1. javac -d directory javafilename

For example

Note :

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

How to Create a package?

Creating a package is a simple task as follows

- Choose the name of the package
- Include the package command as the first line of code in your Java Source File.
- The Source file contains the classes, interfaces, etc you want to include in the package
- Compile to create the Java packages

Step 1) Consider the following package program in Java:

```
package p1;

class c1(){
public void m1(){
System.out.println("m1 of c1");
}
public static void main(string args[]){
c1 obj = new c1();
obj.m1();
}

  }
```

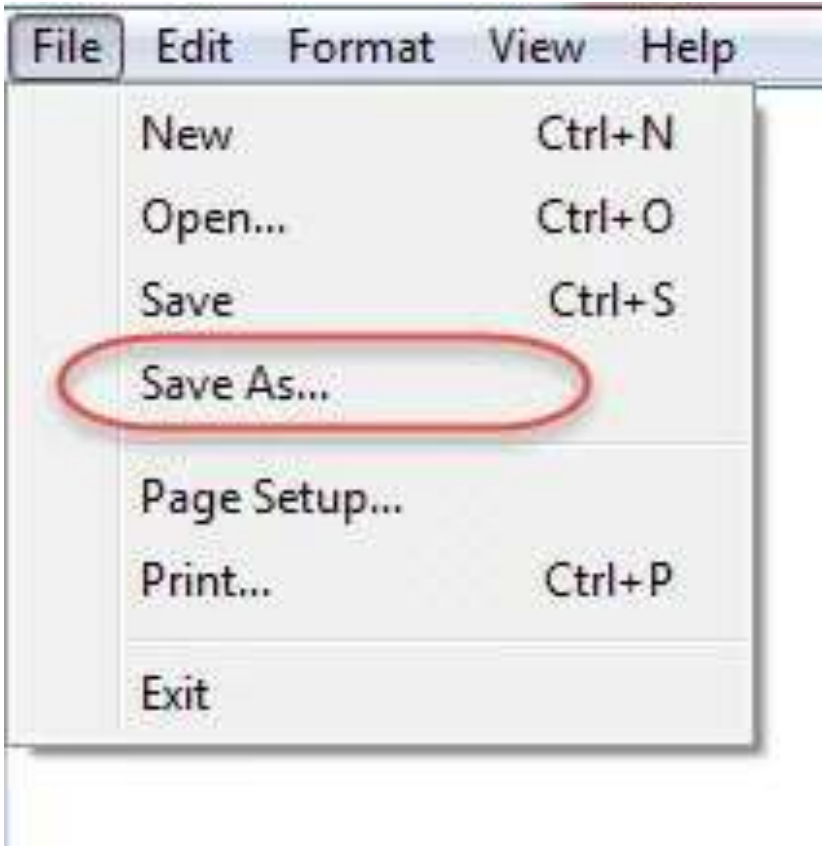
File Edit Format View Help

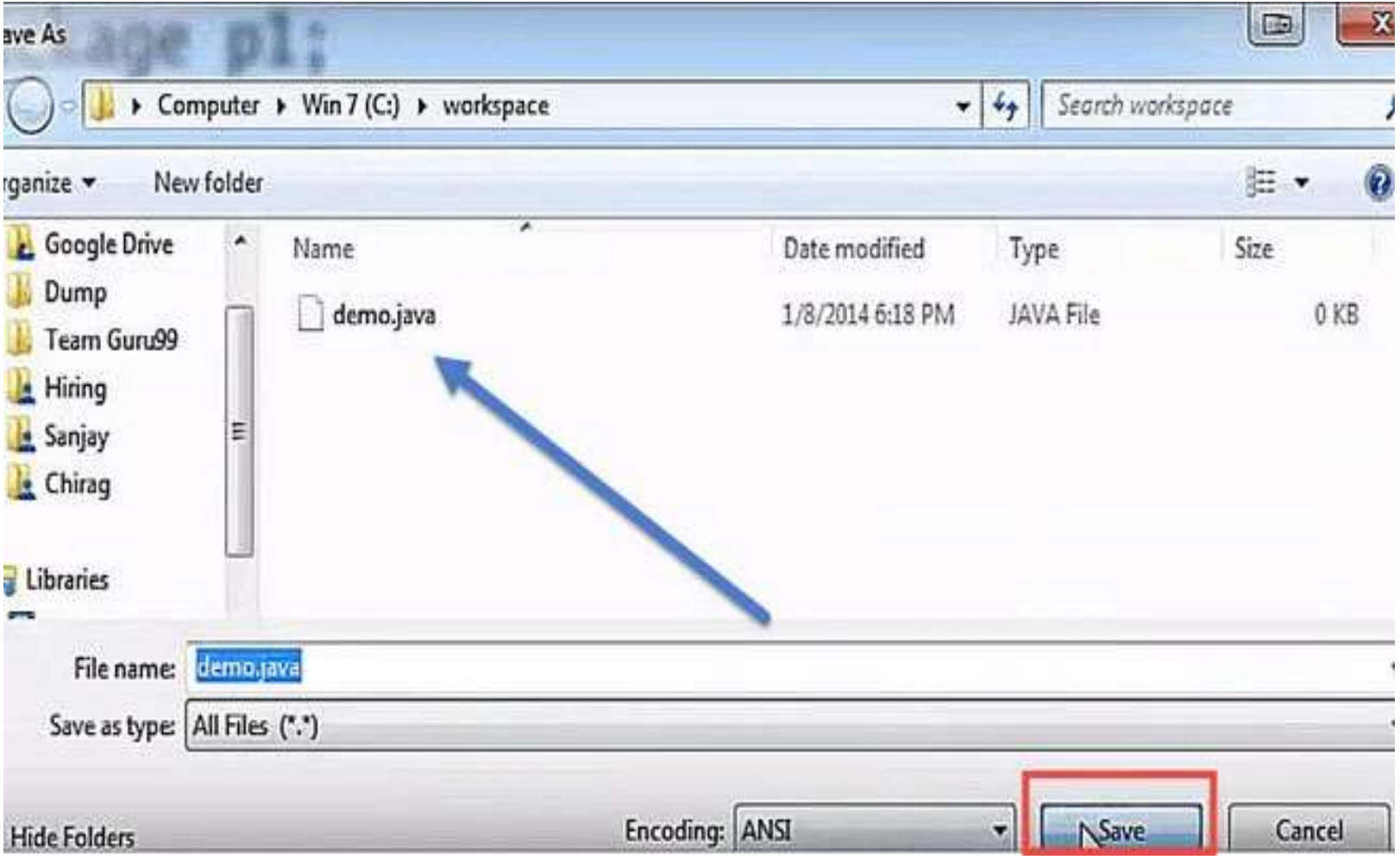
```
package p1;

class c1(){
public void m1(){
System.out.println("m1 of c1");
}
public static void main(String args[]){
c1 obj = new c1();
obj.m1();
}

}
```

Here,

1. To put a class into a package, at the first line of code define package p1
2. Create a class c1
3. Defining a method m1 which prints a line.
4. Defining the main method
5. Creating an object of class c1
6. Calling method m1

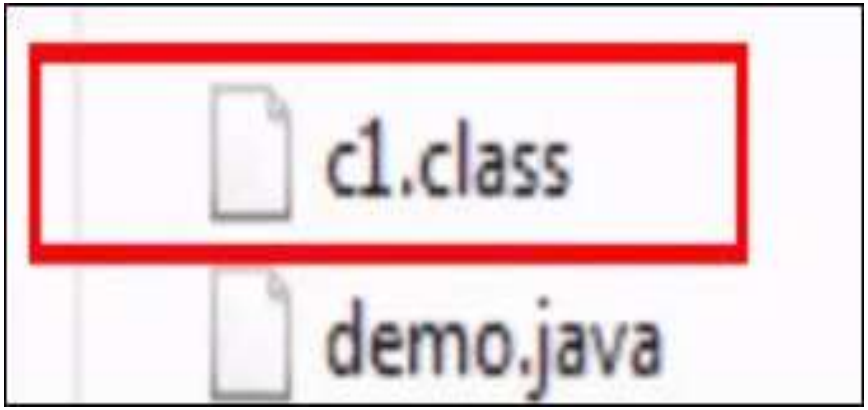
Step 2) In next step, save this file as demo.java




ave As
Computer ▸ Win 7 (C:) ▸ workspace
Search workspace
ganize
New folder
Google Drive
Dump
Team Guru99
Hiring
Sanjay
Chirag
Libraries
Name
Date modified
Type
Size
demo.java
1/8/2014 6:18 PM
JAVA File
0 KB
File name:
demo.java
Save as type:
All Files (*.*)
Hide Folders
Encoding:
ANSI
Save
Cancel

Step 3) In this step, we compile the file.

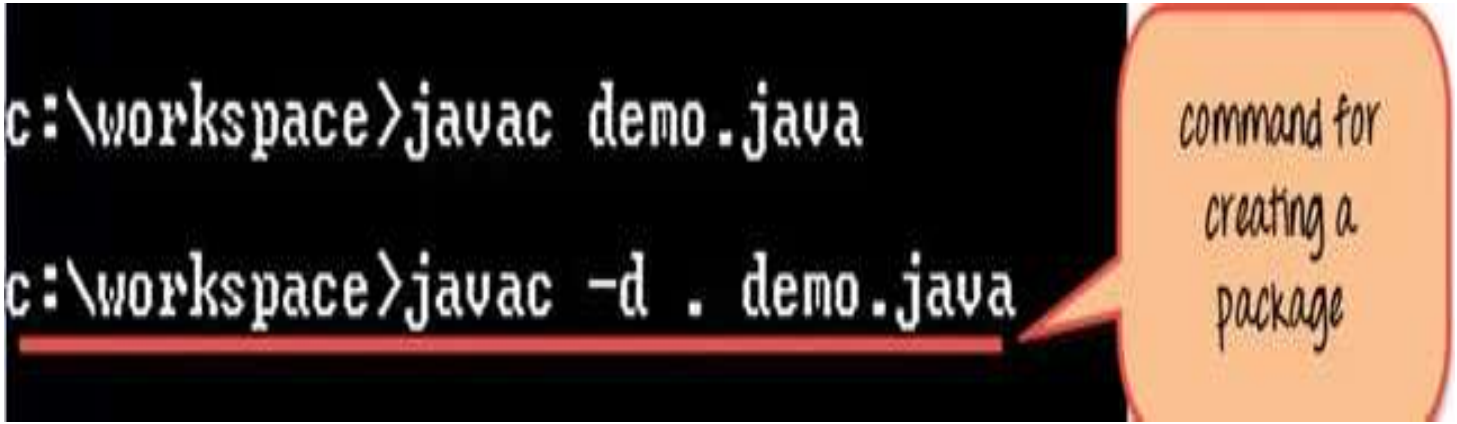
The compilation is completed. A class file c1 is created. However, no package is created? Next step has the solution



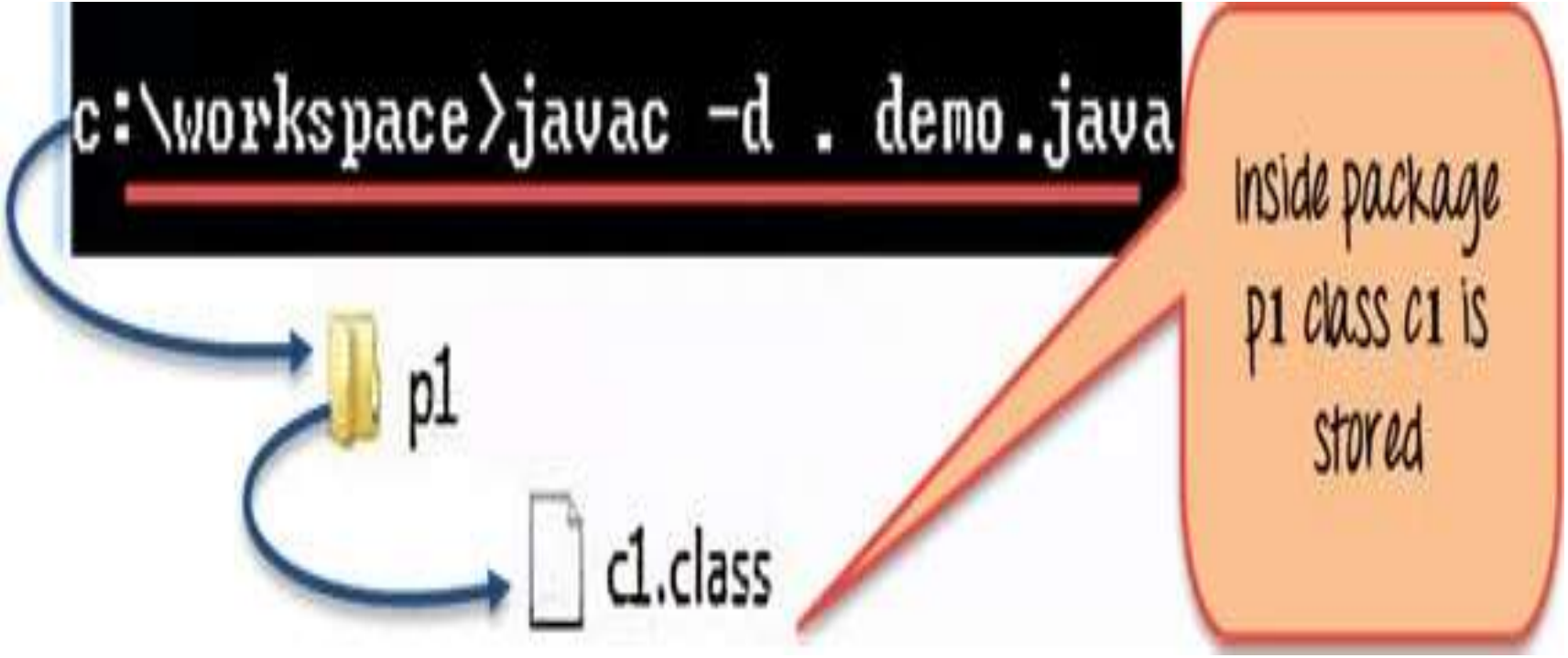
Step 4) Now we have to create a package, use the command

```
javac -d . demo.java
```

This command forces the compiler to create a package.
The **“.”** operator represents the current working directory.



Step 5) When you execute the code, it creates a package p1. When you open the java package p1 inside you will see the c1.class file.

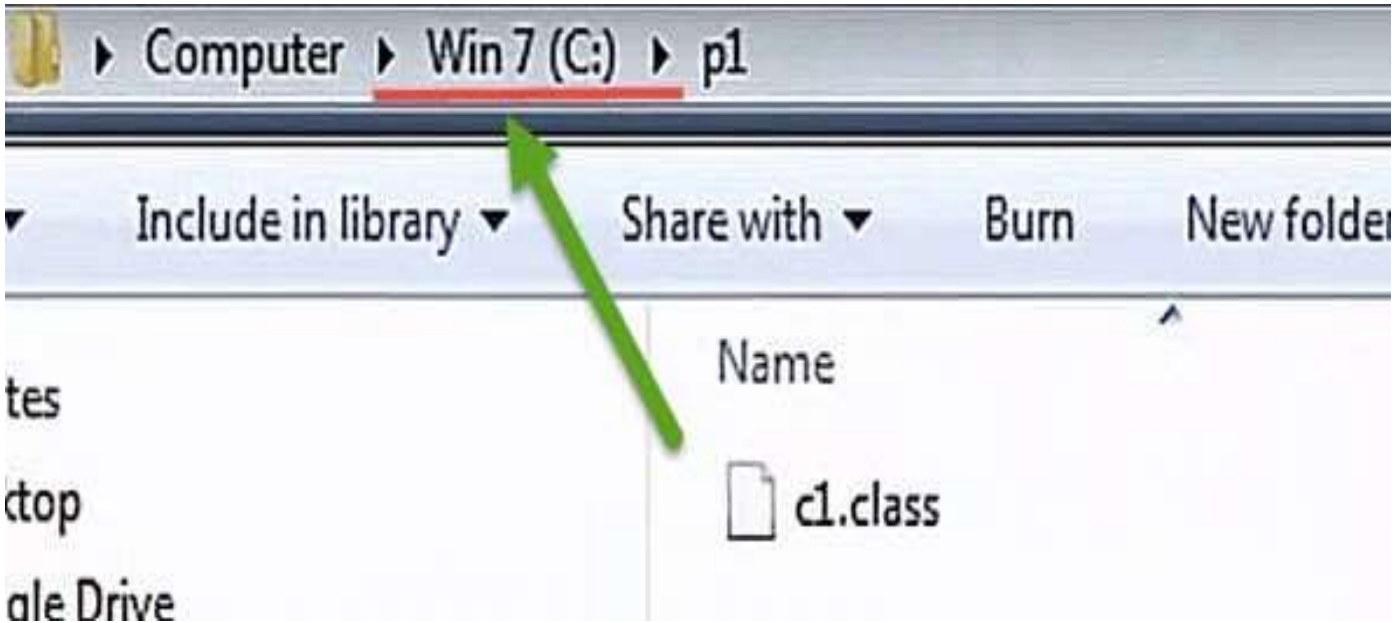


Step 6) Compile the same file using the following code

```
javac -d .. demo.java
```

Here ".." indicates the parent directory. In our case file will be saved in parent directory which is C Drive

File saved in parent directory when above code is executed.



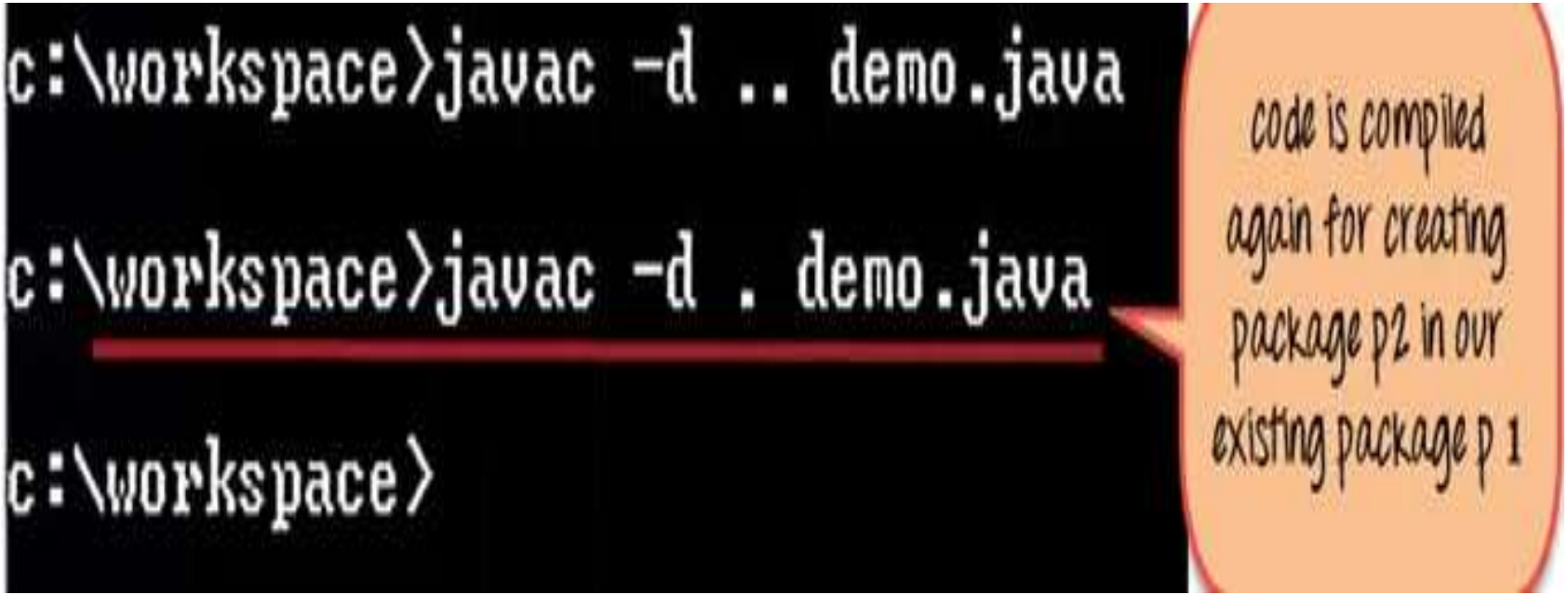
Step 7) Now let's say you want to create a sub package p2 within our existing java package p1. Then we will modify our code as

```
package p1.p2;

class c1{
public void m1() {
System.out.println("m1 of c1");
}

  }
```

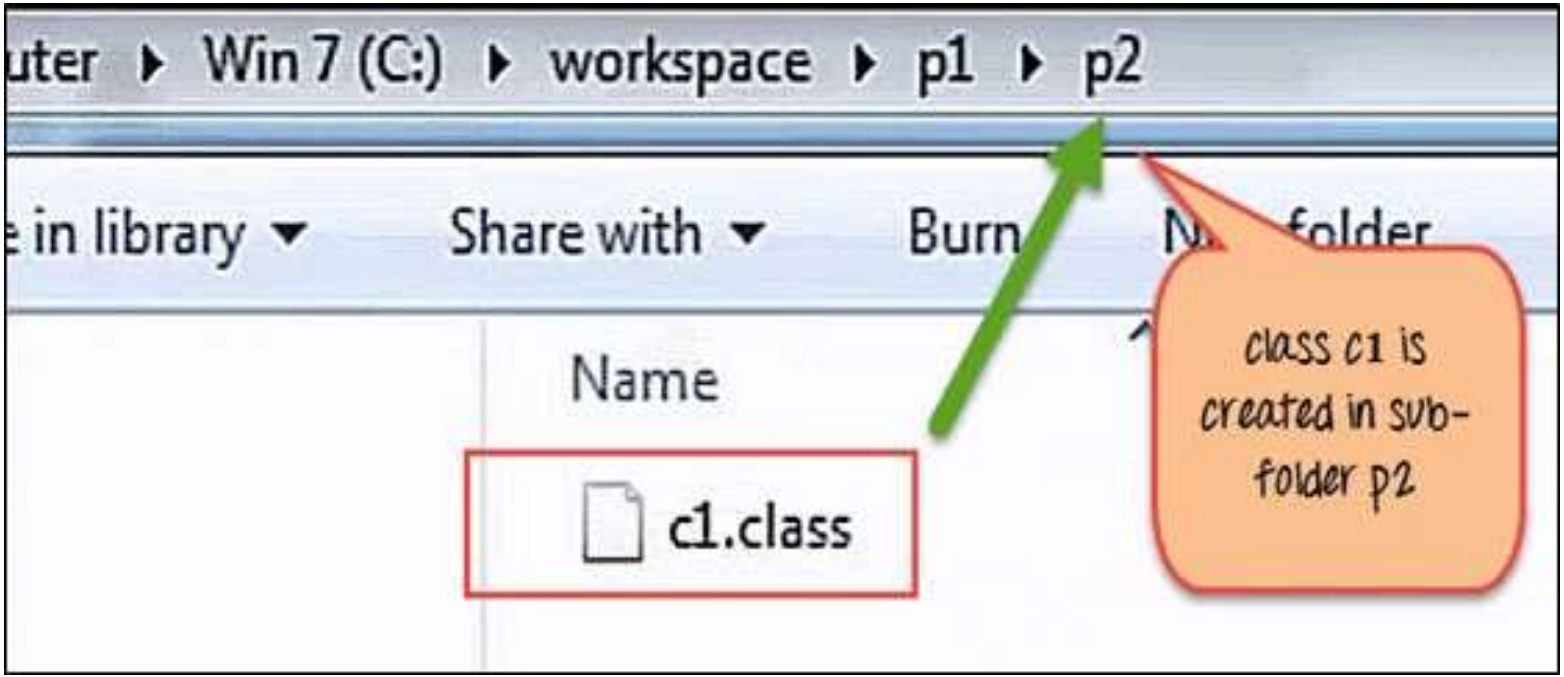
```
package p1.p2;

class c1{
public void m1(){
System.out.println("m1
}
```

code changed in order to add a sub-package p2 to our existing package p1

Step 8) Compile the file

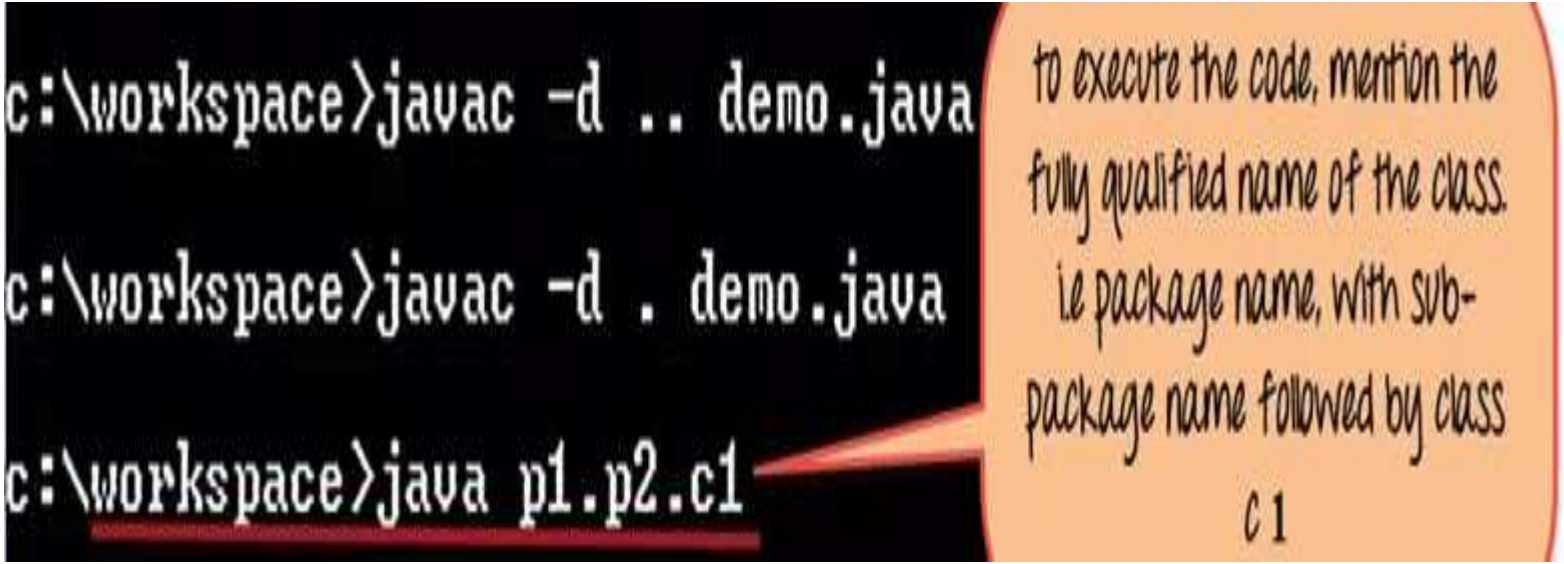


As seen in below screenshot, it creates a sub-package p2 having class c1 inside the package.

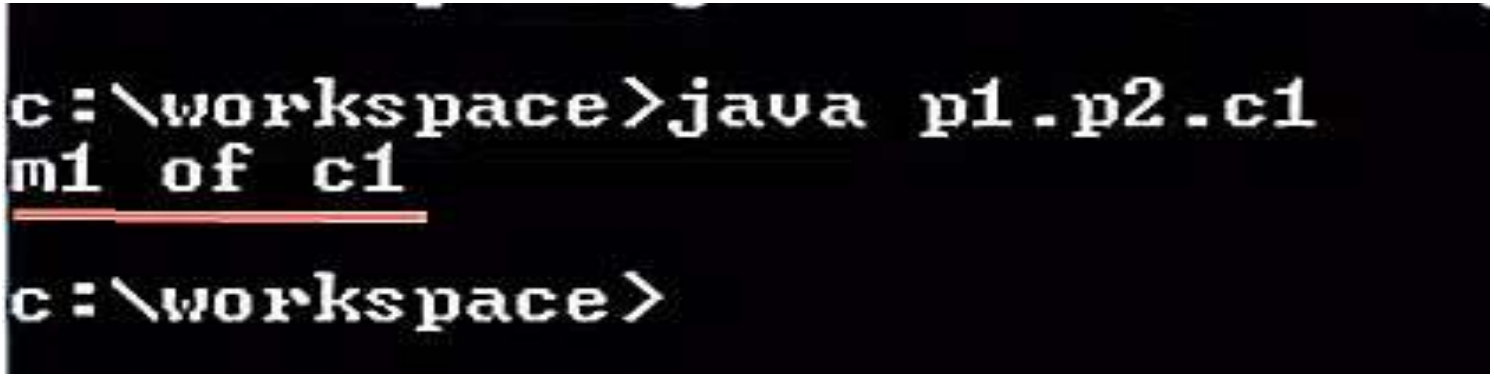


Step 9) To execute the code mention the fully qualified name of the class i.e. the package name followed by the sub-package name followed by the class name –

```
java p1.p2.c1
```



This is how the package is executed and gives the output as “m1 of c1” from the code file.



UNIT- 2 COMPLETED