Java Regex

- The Java Regex or Regular Expression is an API to define pattern for searching or manipulating strings
- It is used to define constraint on strings such as password and email validation.

It provides following classes and interface for regular expressions. The Matcher and Pattern classes are widely used in java regular expression.

- MatchResult interface
- Matcher class
- Pattern class
- PatternSyntaxException class

Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

No.	Method	Description
1	static Pattern compile(String regex)	compiles the given regex and return the instance of pattern.
2	Matcher matcher(CharSequence input)	creates a matcher that matches the given input with pattern.
3	static boolean matches(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] split(CharSequence input)	splits the given input string around matches of given pattern.
5	String pattern()	returns the regex pattern.

Matcher class

It implements MatchResult interface. It is a regex engine i.e. used to perform match operations on a character sequence.

No.	Method	Description
1	boolean matches()	test whether the regular expression matches the pattern.
2	boolean find()	finds the next expression that matches the pattern.
3	boolean find(int start)	finds the next expression that matches the pattern from the given start number

Program N0. 95

RegexDemo_95

```
import java.util.Scanner;
import java.util.regex.*;

public class RegexDemo_95
{
	public static void main(String[] args)
	{
```

```
            Scanner scan = new Scanner(System.in);
            System.out.println(Pattern.matches("[0-9]", "23434"));
       }
}
```

Program NO. 96

RegexDemo_96

```
import java.util.regex.*;
public class RegexExample3
{
      public static void main(String args[])
      {
            boolean isValidMob;
            isValidMob = Pattern.matches("[0-9]{1,10}", "9ds0810100");
            System.out.println(isValidMob);
      }
}
```

Regex Character class

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[0-9]	0 through 9

Classes and Objects

- Java is an object-oriented programming language therefore the underlying structure of all java programs is classes
- Everything to be represented in a java program must be encapsulated in a class that defines the state and behavior of objects
- A class is a user defined data type. Once the class type has been defined, we can create "variables" of that type. These "variables" are called instances or objects of classes
- An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.
- An object has three characteristics:
  - **state:** represents data (value) of an object.
  - **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
  - **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely
- For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.
- **Object is an instance of a class**
  - Class is a template or blueprint from which objects are created. So object is the instance(result) of a class
- A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.
- A class in java can contain:
  - **Data member**
  - **Method**
  - **Constructor**
  - **Block**
  - **Class and interface**
- Classes create objects and objects use methods.
- Classes provide a convenient method for packing together a group of logically related data items and functions.
- In java the data items are called fields and the functions are called methods.

```
Syntax to declare a class:
class <class_name>
{
    datamember;
    method();
}
```

- **Object -** Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.

Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

Program N0. 67

Class Student

```
class Student
{
	private int roll; //fields/data members/instance memberes
	private  String name;

	public void createStudent(int r, String n)
	{
		roll  = r;
		name = n;
	}
	public void showStudent()
	{
		System.out.println("\n\t Roll No. :" + roll);
		System.out.println("\n\t Name : " + name);
	}
}

public class ClassDemo1_67
{
	public static void main(String[] args)
	{
		Student stud1 = new Student();
		stud1.createStudent(101, "Kavita");
		stud1.showStudent();
	}
}
```

Methods

- A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println() method, for example, the system actually executes several statements in order to display a message on the console.

Creating Method

- Method definition consists of a method header and a method body

```
modifier returnType nameOfMethod (Parameter List)
{
 // method body
}
```

- **Modifier**: It defines the access type of the method and it is optional to use.
- **Return Type**: Method may return a value.
- **Name of method**: This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List**: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **Method body**: The method body defines what the method does with statements.

Program N0. 68

Class Employee

```
class Employee
{
       private int empId;
       private String empName;
       private double sal;

       public void getEmp(int id, String ename, double amt)
       {
             empId = id;
             empName = ename;
             sal = amt;
       }
       public void showEmp()
       {
             System.out.println("\n\t Emp Id = " + empId);
             System.out.println("\n\t Emp Name = " + empName);
             System.out.println("\n\t Salary = " + sal);
       }
}
public class ClassEmployee68
{
      public static void main(String[] args)
      {
            Employee emp1 = new Employee();
            emp1.getEmp();
            emp1.showEmp();

      }
}
```

Passing Parameters by Value

- Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter
- While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference

Program N0. 69

Class Product

```
class Product
{
	private int pid;
	private double price, qty, bamt;

	public void getProduct(int id, double pr, double q)
	{
		pid = id;
		price = pr;
		qty = q;
	}

	public void calcBamt()
	{
		bamt = price*qty;
		System.out.println("Product Id = " + pid);
		System.out.println("Price  = " + price);
		System.out.println("Quantity = " + qty);
		System.out.println("Bill Amount = " + bamt);
	}

}

public class ProductDemo
{
	public static void main(String[] args)
	{
		Product toy1 = new Product();
		toy1.getProduct(101, 350.50, 2.00);
		toy1.calcBamt();
	}
}
```

Program N0. 70

Class Rectangle

```
class Rectangle
```

```
{
	public int length, breadth, height;

	public void getData(int l, int b, int h)
	{
		length = l;
		breadth = b;
		height = h;
	}

	public int getArea()
	{
		int area;
		area = length*breadth;
		return area;
	}

	public int getVol()
	{
		return (length*breadth*height);
	}
}

public class ClassRectangle70
{
	public static void main(String[] args)
	{
		Rectangle rect1 = new Rectangle();
		rect1.getData(10,3,2);

		int area;
		area = rect1.getArea();
		System.out.println("\n\t Area of Rect1 = " + area);
		System.out.println("\n\t Vol of Rect1 = " +
rect1.getVol());
	}
}
```

assignments

Program N0. 71

Define a class BOOK with the following specifications

Private members of the class BOOK are

BOOK NO	: integer type

BOOKTITLE	: 20 characters
PRICE	: float (price per copy)
TOTAL_COST()	: A function to calculate the total cost for N number of copies where N is passed to the function as argument

Public members of the class BOOK are

INPUT()	: function to read BOOK_NO. BOOKTITLE, PRICE
PURCHASE()	: function to ask the user to input the number of copies to be purchased. It invokes TOTAL_COST() and prints the total cost to be paid by the user.

Note : You are also required to give detailed function definitions

Program N0. 72

Define a class in Java with following description:

Private Members

A data member Flight number of type integer
A data member Destination of type string
A data member Distance of type float
A data member Fuel of type float
A member function CALFUEL() to calculate the value of Fuel as per the following criteria

Distance	Fuel
<=1000	500
more than 1000 and <=2000	1100
more than 2000	2200

Public Members

A function FEEDINFO() to allow user to enter values for Flight Number, Destination, Distance & call function CALFUEL() to calculate the quantity of Fuel
A function SHOWINFO() to allow user to view the content of all the data members

Program N0. 73

Define a class in Java with following description:

Private members:

bcode	: 4 digits code number
bname	: String
innings, notout, runs	: integer type
batavg	: it is calculated according to the formula : batavg =runs/(innings-notout)
calcavg()	: Function to compute batavg

Public members:

readdata()	: Function to accept value from bcode, name, innings, notout and invoke the function calcavg()
displaydata()	: Function to display the data members on the screen

What is Overloading?

Overloading refers to the process of having multiple methods or constructors using a common name but with different argument list. The methods can be in the same class or in both super class and sub class.

- When a class has two or more methods by same name but different parameters, it is known as method overloading
- Method overloading allows to define same name for two or more methods, providing that the number of parameters or the data type of parameters are different.
- Method overloading is one way to realize polymorphism, which means to allow one interface to be used with multiple methods.
- To declare more than one methods, all methods must be declared with different argument list.
- The methods would perform different operations depending on the argument list in the method call.
- The correct method to be invoked is determined by checking the number and type of arguments.

Example #1:

```
public class Dog {
    private String name;

    public Dog() {
        this.name = "Puppy";
    }

    public Dog(String name) {
        this.name = name;
    }
}
```

Here, the class `Dog` has two overloaded constructors: the first one with no argument; and the second one with a `String` argument (for name). Then we have two ways for creating `Dog` objects:

```
Dog dog1 = new Dog();    // name is 'Puppy'
Dog dog2 = new Dog("Rex");    // name is 'Rex'
```

Example #2:

```
public class Cat {
    public void catches(Mouse mouse) {
        System.out.print("Catching a mouse...");
    }

    public void catches(Fish fish) {
        System.out.print("Catching a fish...");
    }
}
```

Here, the `Cat` class has two overloaded methods: one for catching a mouse; and another for catching a fish. Both has same name (`catches`) but argument type is different (`Mouse` and `Fish`). So we can write:

```
Cat myCat = new Cat();
myCat.catches(new Mouse());
myCat.catches(new Fish());
```

Example #3:

```
public class Lion extends Cat {

    public void catches(Buffalo buffalo) {
        System.out.print("Catching a buffalo...");
    }
}
```

As you can see, this `Lion` class is a sub type of the `Cat` class. The `Lion` class overloads the `catches()` method in order to catch a buffalo. In this case, the method in the super class is called overloaded method, and the method in the sub class is called overloading method.

We can write:

```
Lion lion = new Lion();
lion.catches(new Mouse());
lion.catches(new Buffalo());
```

Why is Overloading?

At low level, overloading is for re-using method names and constructor names depending on needs. For example, you may want to create a `Dog` with default name 'Puppy', or sometimes you want to create a `Dog` with your own name; Sometimes you may want your cat to catch a mouse, but other times he can catch a fish, etc.

At high level, overloading is for extending functionalities but keeping the flexibility of code.

Overloading Rules

Compared to overriding, overloading is simpler, thus the rules are more relaxed.

Here are the rules with regard to overloaded methods:

- Overloaded methods must have different argument list.
- Overloaded methods may have different return type (the argument list is different as well).
- Overloaded methods may have different access modifiers.
- Overloaded methods may throw different exceptions.

Program N0. 74

Arithmetic Operations

```
public class MethodOverloadDemo_74
{
	static void arithAdd(int a, int b)
	{
		System.out.println("\n\t This is Method No. 1");
		System.out.println("\n\t Num1 = " + a);
		System.out.println("\n\t Num2 = " + b);
		int sum = a + b;
		System.out.println("\n\t Addition = " + sum);
	}
	static void arithAdd(int a, double b)
	{
		System.out.println("\n\t This is Method No. 2");
		System.out.println("\n\t Num1 = " + a);
		System.out.println("\n\t Num2 = " + b);
		double sum = a + b;
		System.out.println("\n\t Addition = " + sum);
	}
	static double arithAdd(double a, int b)
	{
```

```
            System.out.println("\n\t This is Method No. 3");
            System.out.println("\n\t Num1 = " + a);
            System.out.println("\n\t Num2 = " + b);
            double sum = a+b;
            return sum;
      }
      public static void main(String[] args)
      {
            arithAdd(10,20);
      }
}
```

Program NO. 75

Class BankAccount

```
class BankAccount
{
      private int accId;
      private String custName;
      private double bal;

      public void openAccount()
      {
            System.out.println("\n\t Bank Acc Opened");
      }
      public void openAccount(int id, String name)
      {
            System.out.println("\n\t Bank Account opened with 
Following
                                          Details :");
            accId = id;
            custName = name;
            bal = 0.00;
            System.out.println("\n\t Account Id = " + accId);
            System.out.println("\n\t Cust Name = " + custName);
            System.out.println("\n\t Balance = " + bal);
      }
      public void openAccount(int id, String name, double iniamt)
      {
      System.out.println("\n\t Bank Account opened with Initial 
Amount:");
            accId = id;
            custName = name;
            bal = iniamt;
            System.out.println("\n\t Account Id = " + accId);
            System.out.println("\n\t Cust Name = " + custName);
            System.out.println("\n\t Balance = " + bal);
      }

}
public class BankAccountDemo_75
{
      public static void main(String[] args)
      {
            BankAccount SB1 = new BankAccount();
            SB1.openAccount();

            BankAccount SB2 = new BankAccount();
            SB2.openAccount(101, "Bhartesh");

            BankAccount SB3 = new BankAccount();
            SB3.openAccount(101, "Payal", 1000.00);
      }
}
```

The Constructors

- A class constructor is a special member function of a class that is executed whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.
- A constructor function has exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.
- Java provides a default constructor which takes no arguments and performs no special actions or initializations, when no explicit constructors are provided.

```
Syntax
class <ClassName>
{
   …datamembers/fields;
   public <ClassName>(argument list)
   {
       …statements;
   }
```

Program NO. 76

Class Demo

```
class Demo
{
      int x, y;
      public Demo()           //Constructor Method
      {
            x = 10;
            y = 20;
      }
      void show()
      {
            System.out.println("\n\t x = " + x);
            System.out.println("\n\t y = " + y);
      }
}

public class ConstructorDemo_76
{
      public static void main(String[] args)
      {
            Demo obj1 = new Demo();
            obj1.show();
      }
}
```

Parameterized Constructor

- Constructors may include parameters of various types. When a constructor is parameterized the object declaration statement must pass the initial values to the parameters
- When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition

Program N0. 77

Class Rectangle - Parameterized Constructor

```
import java.util.Scanner;

class Rectangle
{
      public int length, breadth;

      public Rectangle(int l, int b)
      {
            length = l;
            breadth = b;
      }

      public void calcArea()
      {
            int area = length * breadth;
            System.out.println("\n\t Length =  " + length);
            System.out.println("\n\t Breadth = " + breadth);
            System.out.println("\n\t Area = " + area);
      }
}
```

```
public class RectangleDemo
{
	public static void main(String[] args)
	{
		Scanner scan = new Scanner(System.in);
		int l, b;

		System.out.print("\n\t Enter Length :");
		l = scan.nextInt();

		System.out.print("\n\t Enter Breadth :");
		b = scan.nextInt();

		Rectangle rect1 = new Rectangle(l, b);
		rect1.calcArea();
	}
}
```

Overloaded Constructors

- Like methods, constructors can also be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists

Program NO. 78

Class Bank Account

```
class BankAccount
{
	int id;
	String name;
	double bal;

	public BankAccount()
	{
		System.out.println("Bank Acc Opened");
	}

	public BankAccount(int i, String n)
	{
		id = i;
		name = n;
		bal = 0.00;
		System.out.println("\n\t Bank Acc Opened with
		 Following Details :");
		System.out.println("\n\t id = " + id);
		System.out.println("\n\t Name = " + name);
		System.out.println("\n\t Balance = " + bal);
	}
	public BankAccount(int i, String n, double iniamt)
		{
			id = i;
			name = n;
			bal = iniamt;
			System.out.println("\n\t Bank Acc Opened with 
			Following Details :");
			System.out.println("\n\t id = " + id);
			System.out.println("\n\t Name = " + name);
			System.out.println("\n\t Balance =  " + bal);
	}
}

public class BankAccountDemo_78
{
	public static void main(String[] args)
	{
		BankAccount SB1 = new BankAccount(101, "asdfjdsf", 
		   1000.00);
	}
}
```

this() constructor

- "**this**" keyword in Java is a special keyword which can be used to represent current object or instance of any class in Java
- "**this**()" statement is used to call constructor of same class in Java (used to call overloaded constructor)
- It is called Explicit Constructor Invocation. If a class has two overloaded constructor one without argument and another with argument. Then this keyword can be used to call Constructor with argument from Constructor without argument

Program N0. 79

this() statement with Constructor

Class Goods

```
class Goods
{
	int id;
	String name;
	String description;
	public Goods()
	{
		System.out.println("\n\t Goods Created");
	}
	public Goods(int i, String n)
	{
		this();
		id = i;
		name = n;
		System.out.println("\n\t id = " + id);
		System.out.println("\n\t Name = " + name);
	}
	public Goods(int i, String n, String des)
	{
		this(i, n);
		description = des;
		System.out.println("\n\t Description = " +
		  description);
	}

}

public class ThisConstructorDemo
{
	public static void main(String[] args)
	{
		Goods g1 = new Goods(101, "Mouse", "Optical Mouse");
	}
}
```

Note

- this keyword can only be the first statement in Constructor
- A constructor can have either this or super keyword but not both
- this is a final variable in Java and you cannot assign value to this

“this” to refer current class instance variable

- If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity

Program N0. 80

this keyword to refer current class instance variable

```
class Goods
{
	int id;
	String name;
	String description;
	public Goods()
	{
		System.out.println("\n\t Goods Created");
	}
	public Goods(int id, String name)
	{
		this();
		this.id = id;
		this.name = name;
		System.out.println("\n\t id = " + id);
		System.out.println("\n\t Name = " + name);
	}
	public Goods(int id, String n, String des)
	{
		this(id, n);
		description = des;
		System.out.println("\n\t Description = " +
		 description);
	}
}

public class ThisInstanceVar_80
{
	public static void main(String[] args)
	{

	}
}
```

“this” passed as an argument in the method

- The **this** keyword can also be passed as an argument in the method. It is mainly used in the event handling

Program N0. 81

“this” passed as an argument in the method

This keyword with Method

- this keyword can also be used inside methods to call another method from same class

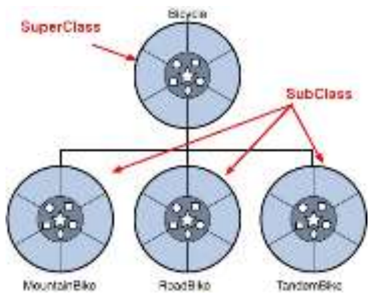
Program N0. 82

“this” keyword with method

```
class Demo
{
    private void method1()
    {
        System.out.println("\n\t Method is Invoked");
    }

    public void method2()
    {
        this.method1();
    }

}

public class ThisMethod_82
{
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.method2();
    }
}
```

Inheritance

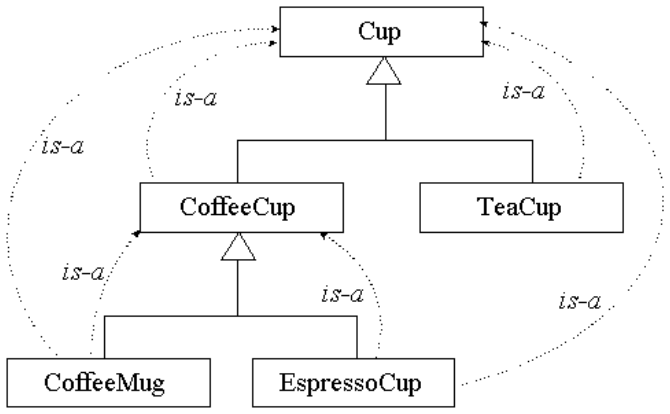
- Inheritance is the ability of a class to inherit from data and behaviors another class to provide the reusability. Inheritance is one of the key features of Object Oriented Programming.
- When a Class extends another class, it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language



- A class that is derived from another class is called a subclass (also a *derived class*, *extended class*, or *child class*).
- The class from which the subclass is derived is called a superclass (also a *base class* or a *parent class*).
- A class can be defined as a "subclass" of another class.
    - The subclass inherits all data attributes of its superclass
    - The subclass inherits all methods of its superclass
    - The subclass inherits all associations of its superclass
- The subclass can:
    - Add new functionality
    - Use inherited functionality
    - Override inherited functionality
- Inheritance is declared using the "extends" keyword
    - If inheritance is not defined, the class extends a class called Object

```
public class Person
{
      private String name;
      private Date dob;
      [...]
}
public class Employee extends Person
{
      private int employeID;
      private int salary;
      private Date startDate;
      [...]
}
```

- Inheritance should create an **IS-A** relationship, meaning the child is a more specific version of the parent
- Inheritance defines **IS-A** relationship between a Super class and its Sub class.



- Syntax

```
class Superclass-Name
{
//methods and fields
}
class Subclass-name extends Superclass-name
{
//methods and fields
}
```

WHY IS INHERITANCE?

As you can see in the above examples, inheritance is for **reusing code**. When you want to extend features of a class, you can write a subclass to inherit all data and behaviors of that superclass. This saves time on writing code.

Another reason for implementing inheritance is for the purpose of **extensibility**. It’s easier to extend a class and add new features than writing a new class from scratch.

And using inheritance promotes the **maintainability** of the code. Imagine you have a superclass and 5 subclasses. When you want to update a common feature of all these classes, you just update the parent class in one place. That’s much easier than updating every single class in case there is no inheritance.

Program N0. 97

Inheritance Demo

```
class SuperDemo
{
      int x, y;
      void getxy(int a, int b)
      {
            x = a;
            y = b;
      }
      void showxy()
      {
            System.out.println("\n\t x = " + x);
            System.out.println("\n\t y = " + y);
      }
}

class DerivedDemo extends SuperDemo
{
      int z;

      void getz(int a)
      {
            z = a;
      }
      void showz()
      {
            System.out.println("\n\t z = " + z);
      }
}

public class InheritanceDemo_113
{
      public static void main(String[] args)
      {
            DerivedDemo d1 = new DerivedDemo();
            d1.getxy(10,20);
            d1.getz(30);

            d1.showxy();
            d1.showz();
      }
}
```

- Private members of the superclass are not inherited by the subclass
- Members that have default accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same package as the superclass
- Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.

- A subclass can extend only one superclass

Program N0. 98

Class Student extends Person

```
class Person
{
	public String name;
	public int age;

	public void getPerson(int a, String n)
	{
		age = a;
		name = n;
	}
	public void showPerson()
	{
		System.out.println("\n\t Age = " + age);
		System.out.println("\n\t Name = " + name);
	}
}

class Student extends Person
{
	public int studid;
	public int marks;

	public void getStudent(int a, String n, int id, int m)
	{
		getPerson(a, n);
		studid = id;
		marks = m;
	}
	public void showStudent()
	{
		System.out.println("\n\t Student id = " + studid);
		showPerson();
		System.out.println("\n\t Marks = " + marks);
	}
}

public class InheritanceDemo_114
{
	public static void main(String[] args)
	{
```

```
                Student stud1 = new Student();
                stud1.getStudent(25, "Amol", 101, 85);
                stud1.showStudent();
        }
}
```

FORMS of Inheritance

- On the basis of class, there can be three types of inheritance
  1. Single
  2. Multilevel
  3. Hierarchical
- Multiple and Hybrid is supported through interface only

Single Inheritance

- The Inheritance hierarchy where one class extends another class is called as Single Inheritance

Program NO. 99

Class Bicycle

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the **superclass**, thus **hiding** it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus **overriding** it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus **hiding** it.

- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword **super**.

Private Members in a Superclass

A subclass does not inherit the `private` members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Multilevel Inheritance

- The Inheritance Hierarchy where one derived class acts as Parent/Super class to another class is called as Multilevel Inheritance

Program N0. 100

Multilevel Inheritance Example

```
class Goods
{
        public String gname;

        public void getGoods(String name)
        {
                gname = name;
        }
}
class Food extends Goods
{
        int cal;
        public void getFood(String name, int c)
        {
                getGoods(name);
                cal = c;
        }
}
class Jam extends Food
{
        String des;
        double price;

        public void getJam(String name, int c, String d, double p)
        {
                getFood(name, c);
                des = d;
                price = p;
        }
```

```
      public void showJam()
      {
            System.out.println("\n\t Name = " + gname);
            System.out.println("\n\t Cal = " + cal);
            System.out.println("\n\t Description = " + des);
            System.out.println("\n\t Price = " + price);
      }
}
public class InheritanceDemo_116
{
      public static void main(String[] args)
      {
            Jam j1 = new Jam();
            j1.getJam("Fruit Jam", 200, "Mix Fruit Jam 200 gm",
25.30);
            j1.showJam();
      }
}
```

Hierarchical Inheritance

- The Inheritance Hierarchy where more than one classes are derived from one Superclass

Program NO. 101

Hierarchical Inheritance – Bank Account Class [Assignment]

Super keyword

- The super is a reference variable that is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.
- Usage of super Keyword
  - super() is used to invoke immediate parent class constructor
  - super is used to refer immediate parent class instance variable
  - super is used to invoke immediate parent class method

Calling Base Class Constructor

- In Java, constructor of base class with no argument gets automatically called in derived class constructor.

Program NO. 102

Calling Base Class Constructor

```
class BaseClass
{
          public BaseClass()
          {
            System.out.println("\n\t This is Base Class Default
                 Constructor");
        }
}
class DerivedClass extends BaseClass
{
        public DerivedClass()
        {
            System.out.println("\n\t This is Derived Class
               Constructor");
        }
}
public class BaseClassConstructor_120
{
        public static void main(String[] args)
        {
                DerivedClass d1 = new DerivedClass();
        }
}
```

- If the superclass/baseclass has parameterized constructor then it is invoked/called using super()
- Use of super() for calling base class constructors must be the first line in derived class constructor

Program N0. 103

Calling Base Class parameterized Constructor

```
class BaseClass
{
        public BaseClass()
        {
            System.out.println("\n\t This is Base Class Default
               Constructor");
        }

        public BaseClass(int a)
        {
            System.out.println("\n\t Base Paramaterized
               Constructor : a = " + a);
        }
}

class DerivedClass extends BaseClass
{
        public DerivedClass(int a)
        {
            super(a);
            System.out.println("\n\t This is Derived Class
              Constructor");
        }
}

public class BaseClassConstructor_121
{
        public static void main(String[] args)
        {
            DerivedClass d = new DerivedClass(10);
        }
}
```

Controlling Access to Members of a Class

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere
- If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package
  - (packages are named groups of related classes)
- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere
- If a class has no modifier (the default, also known as package-private), it is visible only within its own package
- The private modifier specifies that the member can only be accessed in its own class.
- The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

Modifier	Class	Package	Subclass	World
Public	**Y**	**Y**	**Y**	**Y**
protected	**Y**	**Y**	**Y**	**N**
no modifier	**Y**	**Y**	**N**	**N**
Private	**Y**	**N**	**N**	**N**

HAS-A Relationship

- These relationships are mainly based on the usage.
- This determines whether a certain class **HAS-A** certain thing. (This relationship helps to reduce duplication of code as well as bugs)
- If a class has an entity reference (of other class), it is known as Aggregation.
- Aggregation represents HAS-A relationship.

Example

- An Employee object contains some information such as id, name, emailId etc.
- It contains one more object named address, which contains its own information such as city, state, country, zipcode etc.

Program N0.
104

Has-A Relationship Example

```
class Address
{
	public String street,area, city, pincode;


	public Address(String st, String ar, String ct, String pin)
	{
		street = st;
		area = ar;
		city = ct;
		pincode = pin;
	}
	public void showAddress()
	{
		System.out.println("\n\t Address = " + street + ",\n\t " +
area + "," + city + ",\n\t " + pincode);
	}
}

class Supplier
{
	public int suppid;
	public String sname;
	public Address add;

	public Supplier(int sid, String sn, Address ad)
	{
		suppid = sid;
		sname = sn;
		add = ad;
	}
	public void showSupplier()
	{
		System.out.println("\n\t Supplier Info = " + suppid + "\t"
		   + sname);
		add.showAddress();
	}
}

class Customer
{
      public int custid;
      public String cname;
      public Address add;

      public Customer(int cid, String cn, Address ad)
      {
            custid = cid;
            cname = cn;
            add = ad;
      }
      public void showCustomer()
      {
            System.out.println("\n\t Customer Info = " + custid + "\t"
               + cname);
            add.showAddress();
      }
}
public class InheritanceDemo
{
      public static void main(String[] args)
      {
            Address ad = new Address("338/3B, Pragati Colony", "100Ft
road", "Sangli", "416415");
            Supplier supp = new Supplier(101, "Devang",ad);
            supp.showSupplier();

            Customer cust = new Customer(102, "Alfaz",new
Address("100Ft Road","Vishrambag","Sangli","416415"));
            cust.showCustomer();

      }
}
```

Program NO. 105

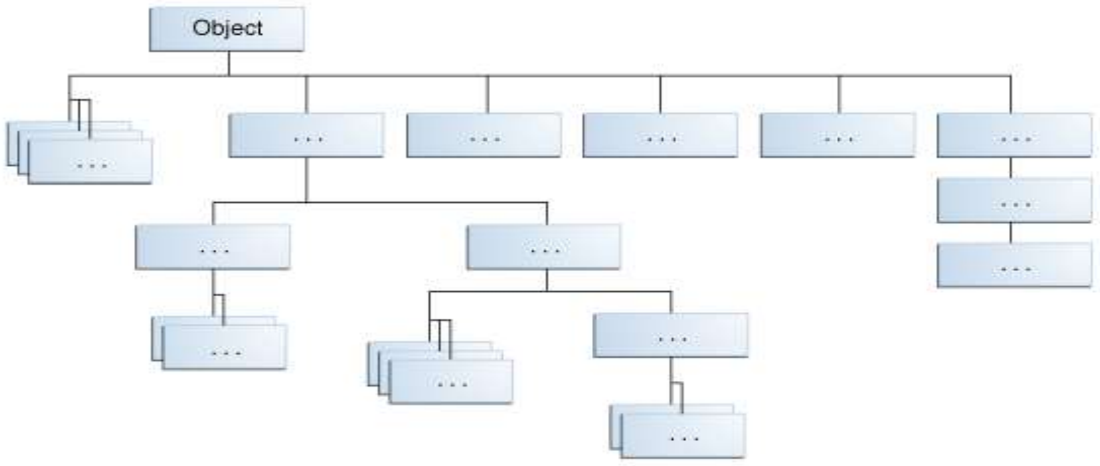
Has-A Relationship Example

```
class Operation
{
         public double square(double n)
         {
            return n*n;
         }
}
class Circle
```

```
{
          Operation op;       //has-a relationship
          double pi = 3.14, rad;

          public Circle(double r)
          {
            rad = r;
          }

          public double calcArea()
          {
            op = new Operation();
            return pi*op.square(rad);
          }
}
public class AggragationDemo_119
{
          public static void main(String[] args)
          {
            Circle cir1 = new Circle(2.50);
            System.out.println("\n\t Area of Circle = " +
               cir1.calcArea());
          }
}
```

The Java Platform Class Hierarchy

The `Object` class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from `Object`, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



- All Classes in the Java Platform are Descendants of Object
- At the top of the hierarchy, `Object` is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
        public MountainBike myBike = new MountainBike();
```

then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` *may be* a `MountainBike`, but it isn't necessarily. Similarly, an `Object` *may be* a `Bicycle` or a `MountainBike`, but it isn't necessarily.

Casting shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
        Object obj = new MountainBike();
```

then `obj` is both an `Object` and a `MountainBike` (until such time as `obj` is assigned another object that is *not* a `MountainBike`). This is called *implicit casting*.

If, on the other hand, we write

```
        MountainBike myBike = obj;
```

we would get a compile-time error because `obj` is not known to the compiler to be a `MountainBike`. However, we can *tell* the compiler that we promise to assign a `MountainBike` to `obj` by *explicit casting:*

```
        MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime checks that `obj` is assigned a `MountainBike` so that the compiler can safely assume that `obj` is a `MountainBike`. If `obj` is not a `MountainBike` at runtime, an exception will be thrown.

Note: You can make a logical test as to the type of a particular object using the `instanceof` operator. This can save you from a runtime error owing to an improper cast. For example:

```
    if (obj instanceof MountainBike) {
        MountainBike myBike = (MountainBike)obj;
    }
```

Here the `instanceof` operator verifies that `obj` refers to a `MountainBike` so that we can make the cast with knowledge that there will be no runtime exception thrown.

What Is Polymorphism?

Polymorphism means '**many forms**'. In OOP, polymorphism means a type can point to different objects at different times. In other words, the actual object to which a reference type refers, can be determined at runtime.

In Java, polymorphism is based on inheritance and overriding.

How is Polymorphism Implemented in Java?

In Java, you can implement polymorphism if you have a super class (or a super interface) with two or more sub classes.

Suppose that we have the following interface and classes:

```
    public interface Animal {
        public void move();
    }

    public class Bird implements Animal {
        public void move() {
            System.out.print("Flying...");
        }
    }

    public class Fish implements Animal {
        public void move() {
            System.out.print("Swimming...");
        }
    }
```

As you can see, we have `Animal` as the super interface, and 3 sub classes: `Dog`, `Bird` and `Fish`.

Because the `Dog` implements `Animal` , or `Dog` is an `Animal`, we can write:

```
    Animal anim = new Dog();
```

Because `Bird` is an `Animal`, it's legal to write:

```
    Animal anim = new Bird();
```

Likewise, it's perfect to write:

```
    Animal anim = new Fish();
```

As you can see, we declare a reference variable called `anim`, which is of type `Animal`. Then we assign this reference variable to 3 different kinds of object: `Dog`, `Bird` and `Fish`.

You see? A reference type can take different objects (many forms). This is the simplest form of polymorphism, got it?

Now we come to a more interesting example to see the power of polymorphism.

Suppose that we have a trainer who teaches animals. We create the `Trainer` class as follows:

```
    public class Trainer {
        public void teach(Animal anim) {
            anim.move();
        }
    }
```

Notice that the `teach()` method accepts any kind of `Animal`. Thus we can pass any objects which are sub types of the `Animal` type. For example:

```
    Trainer trainer = new Trainer();

    Dog dog = new Dog();

    Bird bird = new Bird();

    Fish fish = new Fish();

    trainer.teach(dog);
    trainer.teach(bird);
    trainer.teach(fish);
```

Outputs:

```
Running…
Flying…
Swimming…
```

Here, as you can see, the `teach()` method can accept 'many forms' of `Animal`: `Dog`, `Bird`, `Fish`,… as long as they are sub types of the `Animal` interface.

In the `teach()` method, the `move()` method is invoked on the `Animal` reference. And depending on the actual object type, the appropriate overriding method is called. Thus we see the outputs:

Running… (from the Dog object).
Flying… (from the Dog object).
Running… (from the Dog object).

Why is Polymorphism?

Polymorphism is a robust feature of OOP. It increases the reusability, flexibility and extensibility of code. Take the above example for instance:

- Reusability: the `teach()` method can be re-used for different kinds of objects as long as they are sub types of the `Animal` interface.
- Flexibility: the actual object can be determined at runtime which allows the code run more flexibly.
- Extensibility: when we want to add a new kind of `Animal`, e.g. `Snake`, we just pass an object of `Snake` into the `teach()` method without any modification.

Method Overriding

When you write a class that extends another class (or **implements an interface**), and you re-implement methods of the super class (or super interface), it is called overriding.

Method Overriding is achieved when a subclass overrides non-static methods defined in the superclass, following which the **new method** implementation in the subclass that is executed

The new method definition **must have** the same method signature (i.e., method name and parameters) and return type.

Let's look at a couple of examples.

Example #1:

```
public class Animal {
    public void move() {
        System.out.print("Animal is moving...");
    }
}

public class Dog extends Animal {
    public void move() {
        System.out.print("Dog is running...");
    }
}
```

Here, the Dog class overrides the move() method of the Animal class. In this case, the method in the super class (Animal) is called overridden method. And the method in the sub class (Dog) is called overriding method.

Program N0. 106

Method Overriding

```
class Base
{
	private void show()
	{
		System.out.println("\n\t This is Show() of Base
		   Class");
	}
}

class Child extends Base
{
	public void show()//overriding the superclass show()
	{
		//hide the method of superclass
		System.out.println("\n\t This is Overridden show() of
		     Child Class");
	}
}

public class MethodOverride_122
{
	public static void main(String[] args)
	{
		Child c1 = new Child();
		c1.show();

	}
}
```

Why Overriding?

The benefit of overriding is ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement

Use overriding when you want to re-implement (or re-define) a behavior of the super class. For example: the `Dog` class defines how a dog moves via the `move()` method. A hound runs faster than a dog so you may need to override the `move()` method in the `Hound` class.

Overriding Rules:

Rule #1: Only inherited methods can be overridden.
Inheritable methods are declared with the following access modifiers: public, protected and default (in the same package). That means private and default methods (in different package) cannot be overridden.'

Rule #2: Final and static methods cannot be overridden.

Rule #3: The overriding method must have same argument list as the overridden method.

Rule #4: The overriding method must have same return type (or sub type).
In case of the overriding method's return type is a sub type of the overridden method's return type, it is called co-variant return type.

Rule #5: The overriding method must not have more restrictive access modifier.
For example: if the overridden method is public, you cannot make the overriding method protected, private or default.

Rule #6: The overriding method must not throw new or broader exceptions.
In other words, the overriding method may throw fewer or narrower checked exceptions, or any unchecked exceptions.

Rule #7: Use the super keyword to invoke the overridden method from a sub class.

Rule #8: Constructors cannot be overridden.

Rule #9: Abstract methods must be overridden by the first concrete (non-abstract) sub class.

Rule #10: A static method in a sub class may hide another static one in a super class, and that's called method hiding.

Rule #11: The synchronized modifier has no effect on the rules of overriding.
The synchronized modifier relates to the acquiring and releasing of a monitor object in multi-threaded context, therefore it has totally no effect on the rules of overriding. That means a synchronized method can override a non-synchronized one and vice versa.

Rule #12: The strictfp modifier has no effect on the rules of overriding.

That means the presence or absence of the strictfp modifier has absolutely no effect on the rules of overriding: it's possible that a FP-strict method can override a non-FP-strict one and vice-versa.

Program N0. 107

Method overriding

Car Example

```
class Car
{
	public void move()
	{
		System.out.println("\n\t This car moves normally");
	}
}

class BMW extends Car
{
      public void move()
      {
            System.out.println("\n\t BMW car moves Fast");
      }
}

class Ford extends Car
{
      public void move()
      {
            System.out.println("\n\t Ford Car moves fast and
            Smooth");
      }
}



class xyz extends Car
{
      public void move()
      {
                  System.out.println("\n\t xyz Car moves fast and
                 safe");
      }
}

public class MethodOverride_123
{
      public static void main(String[] args)
      {
            BMW b1 = new BMW();
            b1.move();

            Ford f1 = new Ford();
            f1.move();

            xyz x1 = new xyz();
            x1.move();
      }
}
```

Difference between method Overloading and Method Overriding

Method Overloading	Method Overriding
Method overloading is used to increase the readability of the program	Method overriding is used to provide the specific implementation of the method that is already provided by its super class
Method overloading is performed within a class	Method overriding occurs in two classes that have IS-A relationship.
In case of method overloading parameter must be different	In case of method overriding parameter must be same

Abstract Class and Abstract Methods

An ***abstract class*** is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An ***abstract method*** is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself must be declared `abstract`, as in:

```
public abstract class GraphicObject
{
   // declare fields
   // declare nonabstract methods

    abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared `abstract`.

- Abstract modifier means that the class can be used as a superclass only
- Java Abstract classes are used to declare common characteristics of subclasses.
- It can only be used as a superclass for other classes that extend the abstract class.
- Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree
- Abstraction refers to the ability to make a class abstract in OOP
- An abstract class is one that cannot be instantiated
- All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class
- Abstract class can have any number of constructors, methods, fields

Program N0. 108

Abstract Class Shape

```
abstract class Shape
{
	public double d1, d2;

	public Shape(double a, double b)
	{
		d1 = a;
		d2 = b;
	}

	public void show()
	{
		System.out.println("\n\t d1 = " + d1);
		System.out.println("\n\t d2 = " + d2);
	}
}
```

```
class Rectangle extends Shape
{
      public double area;

      public Rectangle(double l, double b)
      {
            super(l, b);
      }

      public void calcArea()
      {
            area = d1*d2;
            show();
            System.out.println("\n\t Area of Rectangle :" +
              area);
      }
}

public class AbstractClassDemo_124
{
      public static void main(String[] args)
      {
            Rectangle rect1 = new Rectangle(12.20,2.50);
            rect1.calcArea();
      }
}
```

- Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class

Program N0. 109

Abstract Methods

- Methods that are declared **without any body** within an **abstract class** is known as abstract method
- The method body will be defined by its subclass
- Abstract method can never be final and static
- Any class that extends an abstract class must implement all the abstract methods declared by the superclass

```
Syntax
abstract class <classname>
{
        …fields/instance variables
…Constructors/methods
```

```
            public abstract <ReturnType> methodName (args list);
}
```

Program N0. 126

Abstract method Demo

```
abstract class Shape
{
      public double d1, d2;

      public Shape(double a)
      {
            d1 = a;
            d2 = a;
      }
      public Shape(double a, double b)
      {
            d1 = a;
            d2 = b;
      }
      public void show()
      {
            System.out.println("\n\t d1 = " + d1);
            System.out.println("\n\t d2 = " + d2);
      }
      public abstract void calcArea();    //no definition
}

class Rectangle extends Shape
{
      public double area;

      public Rectangle(double l, double b)
      {
            super(l, b);
      }


      public void calcArea()   //overriding superclass
                               //abstract method calcArea()
      {

            area = d1*d2;
            show();
            System.out.println("\n\t Area of Rectangle = " +
            area);
      }
}

class Circle extends Shape
{
      double pi = 3.14;
      double area;

      public Circle(double r)
      {
            super(r);
      }
      public void calcArea()
      {
            area = d1*d2*pi;
            System.out.println("\n\t Area of Circle = " + area);
      }
}

public class AbstractMethodDemo_126
{
      public static void main(String[] args)
      {
            Rectangle rect1 = new Rectangle(2.20, 4.50);
            rect1.calcArea();

            Circle cir1 = new Circle(5.50);
            cir1.calcArea();
      }
}
```

- Abstract classes are not interfaces
- An abstract class can have 0 or more abstract methods
- When an abstract class with abstract method is extended by a non-abstract class, the abstract methods must be overridden by the subclass

Program N0. 127	**Abstract class Demo**

- Abstract methods are usually declared where two ore more subclasses are expected to do a similar thing in different ways through different implementations
- The subclasses (concrete) extend Abstract class and provide different implementations for the abstract methods
- Private methods can not be abstract because they are not visible outside the class
- Static methods can not be abstract. Static methods belong to class, we access static methods using class name instead of object, overriding is only implemented using objects at runtime, hence static method can not be abstract

Interface

- An interface describes aspects of a class other than those that it inherits from its parent.
- **An interface is a set of requirements that the class must implement.**
- Interface is a pure abstract class
- Interface is syntactically similar to classes, but cannot be instantiated.
- The methods in the interface are declared without any body (abstract)
- Interface is used to achieve complete abstraction in java.
- Interface must contain abstract methods only

```
Syntax
interface InterfaceName
{
   constant definitions
   method declarations (without implementations)
}
```

- A class implements an interface by doing this:

```
      class SomeClass extends SomeParent implements
   interfaceName
      {
            . . .
            . . .
      }
```

- A class always extends just one parent but may implement

  several interfaces.

- **Abstract methods** : Methods that are declared, with no implementation
- **Abstract class** : A class with abstract methods, not meant to be instantiated
- **Interface** : A named collection of method definitions (without implementations)

Program N0. 128

Interface Demo

```
interface DemoInterface
{
        public abstract void display();// method without body
}

class Demo1 implements DemoInterface
{
        public void display()
        {
                System.out.println("\n\t This is Demo1 display()");
        }
}

class Demo2 implements DemoInterface
{
        public void display()
        {
                System.out.println("\n\t This is Demo2 display()");
```

```
	}

}
public class InterfaceDemo_128
{
	public static void main(String[] args)
	{
		Demo1 d1 = new Demo1();
		d1.display();

		Demo2 d2 = new Demo2();
		d2.display();
	}
}
```

- All interface methods must be overridden with public only as overridden method cannot have a weaker access specifier
- Methods inside interface must not be static, final
- All variables declared inside interface are implicitly public static final variables
- All methods declared inside java interfaces are implicitly public and abstract, even if you don't use public or abstract keyword
- Interface can extend one or more other interface
- As with abstract classes, with interfaces also, objects cannot be created but reference variables can be created.
- Interface reference variable can be assigned with concrete subclass objects. Once assigned, the interface reference variable works like an object.

Difference between an interface and an abstract class?

- An interface cannot implement any methods, whereas an abstract class can
- A class can implement many interfaces but can have only one superclass (abstract or not)
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

Abstract class:

```
	class Apple extends Food { … }
```

Interface:

```
	public class Person implements Student, Athlete, Chef { … }
```

Abstract class	Interface
Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classe	Interface is a Java Object containing method declaration but no implementation. The classes which

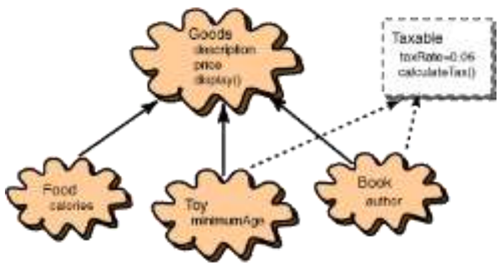
s.	implement the Interfaces must provide the method definition for all the methods.
Abstract class is a Class prefix with an abstract keyword followed by Class definition.	Interface is a pure abstract class which starts with interface keyword.
Abstract class can also contain concrete methods.	Whereas, Interface contains all abstract methods and final variable declarations.
Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes.	Interfaces are useful in a situation that all properties should be implemented.

Example

- Let us create a database program for a store. The store sells:
- Goods, each of which has the attributes:
  - description
  - price
- The types of goods are:
  - Food — with an attribute calories. **Food objects are not taxable**
  - Toy — with an attribute minimum age. **Toy objects are taxable**
  - Book — with an attribute author. **Book objects are taxable**
- There are many things that are taxable that are not goods, such as services or entertainment. Also, not all goods are taxable.
- So we want to have the concept **taxable** as a separate concept, not part of the concept of Goods. Here is what the concept Taxable looks like:
- A Taxable item,
  - has a taxRate of 5 percent,
  - has a calculateTax() method.
- When implemented in Java, these concepts will appear as classes and an interface.

.

Concept	Parent Class, Child Class, or Interface?
Goods	Parent Class
Food	Child Class
Toy	Child Class
Book	Child Class
Taxable	Interface



Food, Toy, and Book extend their parent, Goods

Toy and Book implement the interface Taxable

Program N0. **Interface Example**

```
class Goods
{
        public String des;
        public double price;

        public Goods(String d, double p)
        {
                des = d;
                price = p;
        }
        public void showGoods()
        {
                System.out.println("\n\t Description = " + des);
                System.out.println("\n\t Price = " + price);
        }
}

interface Taxable
{
        public static final double taxRate = 0.05;
        public abstract double calcTax();
}

class Food extends Goods
{
        public int cal;
        public Food(String d, double p, int c)
        {
                super(d, p);
                cal = c;
        }
        public void showFood()
        {
                showGoods();
                System.out.println("\n\t Calories = " + cal);
        }
}


class Toy extends Goods implements Taxable
{
        public int age;
        public Toy(String d, double p, int a)
        {
                super(d, p);
                age = a;
        }

        public double calcTax()
        {
                return price*taxRate;
        }

        public void showToy()
        {
                showGoods();
                System.out.println("\n\t Age = " + age + " years");
                System.out.println("\n\t Tax = " + calcTax());
        }
}


public class InterfaceDemo_129
{
        public static void main(String[] args)
        {
                Food jam = new Food("Mix Fruit Jam", 150.00, 200);
                jam.showFood();

                Toy car1 = new Toy("Police Car", 350.00, 5);
                car1.showToy();

                Taxable toy1 = new Toy("Car", 700.00, 5);
                Toy t = (Toy)toy1;
                t.showToy();
        }
}
```

Interface extends interface

```
interface Singer
{
    void sing();
    void warmUpVoice();
}

interface Dancer
{
    void dance();
    void stretchLegs();
}
```

```
interface Talented extends Singer, Dancer
{
          // can sing and dance. Wowwee.
}
```

Where can interfaces are used?

- You can pass an interface as a parameter or assign a class to an interface variable, just like you would to an abstract class.

```
Example:
Food myLunch = new Sandwich();
Food mySnack = new Apple();
Student amit = new Person();
//assuming that Person implements Student
```

- If Person has methods eat(Food f) and teach(Student s), the following is possible:

```
Person sumit= new Person();
sumit.teach(amit);
sumit.eat(myLunch);
```