Object-oriented programming allows you to define new classes from existing classes. This is called inheritance. Inheritance is an important and powerful feature for reusing software.

Diagram

Description automatically generatedInheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

In this example, there is a superclass geometricObject which has attributes of filled of Boolean type and color of String type. It has methods and constructors in it which are used to perform some functions or invoked to create an object respectively. This superclass has its own subclasses Circle and Rectangle. Circle has attributes of radius of type double. This subclass Circle has methods of getRadius, setRadius and getArea which do the assigned function. Similarly, there is another subclass Rectangle which has attributes of width and length which have methods of getWidth, getHeight, setHeight, setWidth & getArea.

According to inheritance, as Class Circle is a subclass of geometricObject, it will inherit all the properties of its superclass (geometricObject) and it can also have its own properties which we have seen earlier. Now, Circle cannot inherit Rectangle as its superclass as Rectangle is already a subclass of its own “same” superclass which is geometricObject.

For a class to inherit properties of its superclass, we use the ‘extend’ keyword like following

class *subclass* extends *superclass*

Inheritance can be of three types:

* Simple Inheritance
* Multi-Level Inheritance
* Hierarchical Inheritance

Simple Inheritance can be defined simply as a parent base class having only one daughter class. Multi-Level Inheritance can be defined as a super base class having a daughter base class which again has a daughter class. Hierarchical Inheritance can be defined as a Base class having two or more daughter classes, more like the example described above.

Java does not support Multiple Inheritance, i.e., a single daughter class having two or more base parent classes.

Following points must be noted for Inheritance:

* Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass.
* Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public accessors/mutators if defined in the superclass.
* Not all is-a relationships should be modelled using inheritance. For example, a square is a rectangle, but you should not extend a Square class from a Rectangle class, because the width and height properties are not appropriate for a square. Instead, you should define a Square class to extend the GeometricObject class and define the side property for the side of a square.
* Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of reusing methods. For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.

The keyword super refers to the superclass and can be used to invoke the super class’s methods and constructors. The ‘this’ Reference, introduced the use of the keyword this to reference the calling object. The keyword super refers to the superclass of the class in which super appears. It can be used in two ways:

* To call a superclass constructor.
* To call a superclass method.

A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited by a subclass. They can only be invoked from the constructors of the subclasses using the keyword super.

The syntax to call a superclass’s constructor is super(), or super(parameters)

The statement super() invokes the no-arg constructor of its superclass, and the statement super(arguments) invokes the superclass constructor that matches the arguments. The statement super() or super(arguments) must be the first statement of the subclass’s constructor. This is the only way to explicitly invoke a superclass constructor.

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts super() as the first statement in the constructor.

In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is called constructor chaining.

The keyword super can also be used to reference a method other than the constructor in the superclass. The syntax is: super.method()

To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

An instance method can be overridden only if it is accessible. Thus, a private method cannot be overridden because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax SuperClassName.staticMethodName.

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.

Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

Dynamic Method Dispatching

* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
* Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
* A superclass reference variable can refer to a subclass object.
* When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.

*Source Code 1: simpleGeometricObject.java*

public class simpleGeometricObject {

    private String color = "White";

    private boolean filled;

    private java.util.Date dateCreated;

    public simpleGeometricObject(){

        dateCreated = new java.util.Date();

    }

    public String getColor(){

        return color;

    }

    public void setColor(String color){

        this.color = color;

    }

    public boolean isFilled(){

        return filled;

    }

    public void setFilled(boolean filled){

        this.filled = filled;

    }

    public java.util.Date getDateCreated(){

        return dateCreated;

    }

    void disp(){

        System.out.println("Object created on: " + dateCreated);

        System.out.println("Color of the Object: " + color);

        System.out.println("Status\_Filled: " + filled);

    }

}

*Source Code 2: circle.java*

public class circle extends simpleGeometricObject{

    private double radius;

    public circle(){}

    public circle(double radius){

        this.radius = radius;

    }

    public circle(double radius, String color, boolean filled){

        this.radius = radius;

        setColor(color);

        setFilled(filled);

    }

    public double getRadius(){

        return radius;

    }

    public void setRadius(double radius){

        this.radius = radius;

    }

    public double getArea(){

        return radius\*radius\*Math.PI;

    }

    public double getPerimeter(){

        return 2\*radius\*Math.PI;

    }

    public double getDiameter(){

        return 2\*radius;

    }

    void disp(){

        super.disp();

        System.out.println("Radius of circle: " + radius);

        System.out.println("Diameter of circle: " + getDiameter());

        System.out.println("Perimeter of circle: " + getPerimeter());

        System.out.println("Area of circle: " + getArea());

    }

}

*Source Code 3: testGeometricObjects.java*

import java.util.Scanner;

public class testGeometricObjects {

    public static void main(String [] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter radius of circle 1: ");

        double radius1 = sc.nextDouble();

        System.out.print("Enter color of circle 1: ");

        String color1 = sc.next();

        System.out.print("Is the circle filled? ");

        boolean filled1 = sc.nextBoolean();

        System.out.print("Enter radius of circle 2: ");

        double radius2 = sc.nextDouble();

        System.out.print("Enter color of circle 2: ");

        String color2 = sc.next();

        System.out.print("Is the circle filled? ");

        boolean filled2 = sc.nextBoolean();

        circle c1 = new circle(radius1, color1, filled1); //Parametrized Constructor to invoke object c1

        circle c2 = new circle(); //No-arg constructor to invoke an empty object

        //setting parameters of object c2

        c2.setColor(color2);

        c2.setFilled(filled2);

        c2.setRadius(radius2);

        c1.disp(); //Calls display function for c1

        c2.disp(); //Calls display function for c2

        sc.close();

    }

}

*Output:*

Enter radius of circle 1: 10

Enter color of circle 1: blue

Is the circle filled? True

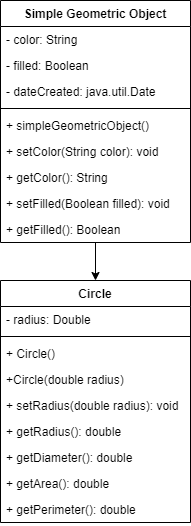
Enter radius of circle 2: 20

Enter color of circle 2: black

Is the circle filled? False

Object created on: Fri Sep 16 17:29:00 IST 2022

Color of the Object: blue

Status\_Filled: true

Radius of circle: 10.0

Diameter of circle: 20.0

Perimeter of circle: 62.83185307179586

Area of circle: 314.1592653589793

Object created on: Fri Sep 16 17:29:00 IST 2022

Color of the Object: black

Status\_Filled: false

Radius of circle: 20.0

Diameter of circle: 40.0

Perimeter of circle: 125.66370614359172

Area of circle: 1256.6370614359173

*Description of the Code:* The code that has been written here is the simplest example of Simple Inheritance where there is a parent base class and a daughter class. The Parent class is GeometricObject, and the daughter class is Circle. GeometricObject has methods which are inherited by Circle which itself has its own methods and variable. There is an overriding method disp() which is solved by using the super keyword whose main function is to call the overriding method of the next superclass which is here GeometricObject.

The following set of codes demonstrate the combined usage of all the three types of inheritance as discussed in previous points.

*Code 1: student.java*

public class student {

    private int age;

    private String name;

    public student(){}

    public student(int age, String name){

        this.age = age;

        this.name = name;

    }

    public String getName(){

        return name;

    }

    public int getAge(){

        return age;

    }

    public void setAge(int age){

        this.age = age;

    }

    public void setName(String name){

        this.name = name;

    }

    void disp(){

        System.out.println("Name of Student: " + name);

        System.out.println("Age of Student: " + age);

    }

}

*Code 2: engineeringStudent.java*

public class engineeringStudent extends student{

    private String rollnumber;

    private double cgpa;

    public engineeringStudent(){}

    public engineeringStudent(String rollnumber, double cgpa){

        this.rollnumber = rollnumber;

        this.cgpa = cgpa;

    }

    public String getrollNumber(){

        return rollnumber;

    }

    public double getCGPA(){

        return cgpa;

    }

    public void setCGPA(double cgpa){

        this.cgpa = cgpa;

    }

    public void setRollNumber(String rollnumber){

        this.rollnumber = rollnumber;

    }

    void disp(){

        super.disp();

        System.out.println("Roll Number: " + rollnumber);

        System.out.println("CGPA: " + cgpa);

    }

}

*Code 3: highschoolstudent.java*

public class highschoolstudent extends student{

    private double physmarks;

    private double mathmarks;

    private double chemmarks;

    highschoolstudent(){}

    highschoolstudent(double physmarks, double mathmarks, double chemmarks){

        this.chemmarks = chemmarks;

        this.mathmarks = mathmarks;

        this.physmarks = physmarks;

    }

    public double getphysmarks(){

        return physmarks;

    }

    public double getmathmarks(){

        return mathmarks;

    }

    public double getchemmarks(){

        return chemmarks;

    }

    public void setphysmarks(double physmarks){

        this.physmarks = physmarks;

    }

    public void setchemmarks(double chemmarks){

        this.chemmarks = chemmarks;

    }

    public void setmathmarks(double mathmarks){

        this.mathmarks = mathmarks;

    }

    void disp(){

        super.disp();

        System.out.println("Physics Marks: " + physmarks);

        System.out.println("Chemistry Marks: " + chemmarks);

        System.out.println("Mathematics Marks: " + mathmarks);

    }

}

*Code 4: cseStudent.java*

public class cseStudent extends engineeringStudent{

    private String language;

    private String projectname;

    cseStudent(){}

    cseStudent(String language, String projectname){

        this.language = language;

        this.projectname = projectname;

    }

    public String getLanguage(){

        return language;

    }

    public String getprojectname(){

        return projectname;

    }

    public void setlanguage(String language){

        this.language = language;

    }

    public void setproject(String projectname){

        this.projectname = projectname;

    }

    void disp(){

        super.disp();

        System.out.println("Language: " + language);

        System.out.println("Project Name: " + projectname);

    }

}

*Code 5: MultiInheritance.java*

public class MultiInheritance {

    public static void main(String [] args) {

        highschoolstudent s1 = new highschoolstudent(); //High School Student

        s1.setAge(18);

        s1.setName("ABC");

        s1.setchemmarks(69);

        s1.setmathmarks(89);

        s1.setphysmarks(78);

        s1.disp();

        cseStudent s2 = new cseStudent(); //CSE Student

        s2.setAge(19);

        s2.setName("XYZ");

        s2.setCGPA(9.89);

        s2.setRollNumber("21BCP050");

        s2.setlanguage("Java");

        s2.setproject("Project 1");

        s2.disp();

    }

}

*Output:*

Name of Student: ABC

Age of Student: 18

Physics Marks: 78.0

Chemistry Marks: 69.0

Mathematics Marks: 89.0

Name of Student: XYZ

Age of Student: 19

Roll Number: 21BCP050

CGPA: 9.89

Language: Java

Project Name: Project 1

Table

Description automatically generated*Explanation of the code:* The code written here demonstrates all of simple, multi-level and hierarchical inheritance. The super base class is Student here which has methods getName, getAge, setName, setAge and disp. This super base class has two daughter base classes Engineering Student and High School Student. Engineering Student has methods getrollnumber, getcgpa, setrollnumber, setcgpa, disp while High School student has methods of getphysmarks, getmathmarks, getchemmarks, setphysmarks, setmathmarks, setchemmarks, disp. The daughter base class Engineering student has its own derived class of CSE student which has methods getprojectname, getlanguage, setprojectname, setlanguage, disp. Note that all of the classes have an overridden method called disp. This is solved by using the super keyword. Now, the super.disp() when executed in CSE student, calls the disp method of the immediate base class, which here is Engineering student, and the super.disp() method in Engineering student calls the disp method of student class. There can be made no direct calls of an overridden method from a derived daughter class to a super base class by skipping any class in between.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

In the inheritance hierarchy, classes become more specific and concrete with each new subclass. If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure that a superclass contains common features of its subclasses. Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an abstract class.

An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in concrete subclasses.

In the inheritance hierarchy, classes become more specific and concrete with each new subclass. If you move from a subclass back up to a superclass, the classes become more general and less specific. Class design should ensure that a superclass contains common features of its subclasses. Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an abstract class.

Some points to be noted about abstract class:

* An abstract method cannot be contained in a nonabstract class. If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined as abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented. Also note that abstract methods are non-static. **Abstract method in abstract class.**
* An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class. **Object cannot be created from abstract class.**
* A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn’t contain any abstract methods. In this case, you cannot create instances of the class using the new operator. This class is used as a base class for defining subclasses. **Abstract class without abstract method.**
* A subclass can override a method from its superclass to define it as abstract. This is very unusual, but it is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined as abstract. **Concrete method can be overridden to be abstract.**
* A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract. **Superclass of abstract class may be concrete**
* You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. **Abstract class as a type.**

Following series of code demonstrates the usage of abstract classes and methods

*Code: geometricObjects.java*

*abstract* *class* geometricObjects {

*private* *String* color;

*private* *boolean* filled;

*protected* geometricObjects() {}

*protected* geometricObjects(*String* color, *boolean* filled) {

*this*.color = color;

*this*.filled = filled;

    }

*protected* geometricObjects(*boolean* filled) {

*this*.filled = filled;

    }

*protected* geometricObjects(*String* color) {

*this*.color = color;

    }

*abstract* *double* getArea();

*abstract* *double* getPerimeter();

*protected* *String* getColor() {

*return* color;

    }

*protected* *void* setColor(*String* color) {

*this*.color = color;

    }

*protected* *boolean* isFilled() {

*return* filled;

    }

*protected* *void* setFilled(*boolean* filled) {

*this*.filled = filled;

    }

*protected* *void* disp(){

        System.out.println("Color: "+color);

        System.out.println("Filled: "+filled);

    }

}

*Code: rectangle.java*

*public* *class* rectangle *extends* geometricObjects{

*private* *double* width;

*private* *double* height;

*public* rectangle() {}

*public* rectangle(*double* width, *double* height) {

*this*.width = width;

*this*.height = height;

    }

*public* rectangle(*double* width, *double* height, *String* color, *boolean* filled) {

*super*(color, filled);

*this*.width = width;

*this*.height = height;

    }

*public* *double* getWidth() {

*return* width;

    }

*public* *void* setWidth(*double* width) {

*this*.width = width;

    }

*public* *double* getHeight() {

*return* height;

    }

*public* *void* setHeight(*double* height) {

*this*.height = height;

    }

*//Defining the abstract methods*

*public* *double* getArea() {

*return* width\*height;

    }

*public* *double* getPerimeter() {

*return* 2\*(width+height);

    }

*public* *void* disp(){

*super*.disp();

        System.out.println("Width: "+width);

        System.out.println("Height: "+height);

        System.out.println("Area: "+getArea());

        System.out.println("Perimeter: "+getPerimeter());

    }

}

*Code: circle.java*

*public* *class* circle *extends* geometricObjects{

*private* *double* radius;

*public* circle() {}

*public* circle(*double* radius) {

*this*.radius = radius;

    }

*public* circle(*double* radius, *String* color, *boolean* filled) {

*super*(color, filled);

*this*.radius = radius;

    }

*public* *double* getRadius() {

*return* radius;

    }

*public* *void* setRadius(*double* radius) {

*this*.radius = radius;

    }

*//Defining the abstract methods*

*public* *double* getArea() {

*return* radius\*radius\*Math.PI;

    }

*public* *double* getPerimeter() {

*return* 2\*radius\*Math.PI;

    }

*public* *void* disp(){

*super*.disp();

        System.out.println("Radius: "+radius);

        System.out.println("Area: "+getArea());

        System.out.println("Perimeter: "+getPerimeter());

    }

}

*Code: triangle.java*

*public* *class* triangle *extends* geometricObjects{

*private* *double* side1;

*private* *double* side2;

*private* *double* side3;

*public* triangle() {}

*public* triangle(*double* side1, *double* side2, *double* side3) {

*this*.side1 = side1;

*this*.side2 = side2;

*this*.side3 = side3;

    }

*public* triangle(*double* side1, *double* side2, *double* side3, *String* color, *boolean* filled) {

*super*(color, filled);

*this*.side1 = side1;

*this*.side2 = side2;

*this*.side3 = side3;

    }

*public* *double* getSide1() {

*return* side1;

    }

*public* *void* setSide1(*double* side1) {

*this*.side1 = side1;

    }

*public* *double* getSide2() {

*return* side2;

    }

*public* *void* setSide2(*double* side2) {

*this*.side2 = side2;

    }

*public* *double* getSide3() {

*return* side3;

    }

*public* *void* setSide3(*double* side3) {

*this*.side3 = side3;

    }

*//Defining the abstract methods*

*public* *double* getArea() {

*double* s = (side1+side2+side3)/2;

*return* Math.sqrt(s\*(s-side1)\*(s-side2)\*(s-side3));

    }

*public* *double* getPerimeter() {

*return* side1+side2+side3;

    }

*public* *void* disp(){

*super*.disp();

        System.out.println("Side1: "+side1);

        System.out.println("Side2: "+side2);

        System.out.println("Side3: "+side3);

        System.out.println("Area: "+getArea());

        System.out.println("Perimeter: "+getPerimeter());

    }

}

*Code: testObjects.java*

*import* *java.util.Scanner*;

*public* *class* testObjects {

*public* *static* *void* main(*String* [] args) {

*Scanner* input = *new* Scanner(System.in);

        System.out.println("CIRCLE");

        System.out.println("Enter the radius of the circle: ");

*double* radius = input.nextDouble();

        System.out.println("Enter the color of the circle: ");

*String* color = input.next();

        System.out.println("Is the circle filled? (true/false): ");

*boolean* filled = input.nextBoolean();

        circle c = *new* circle(radius, color, filled);

        System.out.println();

        System.out.println("RECTANGLE");

        System.out.println("Enter the width of the rectangle: ");

*double* width = input.nextDouble();

        System.out.println("Enter the height of the rectangle: ");

*double* height = input.nextDouble();

        System.out.println("Enter the color of the rectangle: ");

        color = input.next();

        System.out.println("Is the rectangle filled? (true/false): ");

        filled = input.nextBoolean();

        rectangle r = *new* rectangle(width, height, color, filled);

        System.out.println();

        System.out.println("TRIANGLE");

        System.out.println("Enter the side 1 of triangle: ");

*double* side1 = input.nextDouble();

        System.out.println("Enter the side 2 of triangle: ");

*double* side2 = input.nextDouble();

        System.out.println("Enter the side 3 of triangle: ");

*double* side3 = input.nextDouble();

        System.out.println("Enter the color of the triangle: ");

        color = input.next();

        System.out.println("Is the triangle filled? (true/false): ");

        filled = input.nextBoolean();

        triangle t = *new* triangle(side1, side2, side3, color, filled);

        System.out.println();

        System.out.println("Circle: ");

        c.disp();

        System.out.println();

        System.out.println("Rectangle: ");

        r.disp();

        System.out.println();

        System.out.println("Triangle: ");

        t.disp();

    }

}

*Output:*

CIRCLE

Enter the radius of the circle:

10

Enter the color of the circle:

blue

Is the circle filled? (true/false):

true

RECTANGLE

Enter the width of the rectangle:

69

Enter the height of the rectangle:

96

Enter the color of the rectangle:

red

Is the rectangle filled? (true/false):

false

TRIANGLE

Enter the side 1 of triangle:

30

Enter the side 2 of triangle:

40

Enter the side 3 of triangle:

50

Enter the color of the triangle:

brown

Is the triangle filled? (true/false):

True

Circle:

Color: blue

Filled: true

Radius: 10.0

Area: 314.1592653589793

Perimeter: 62.83185307179586

Rectangle:

Color: red

Filled: false

Width: 69.0

Height: 96.0

Area: 6624.0

Perimeter: 330.0

Triangle:

Color: brown

Filled: true

Side1: 30.0

Side2: 40.0

Side3: 50.0

Area: 600.0

Perimeter: 120.0

*Code Explanation:* The code written here has a base class GeometricObject which is abstract in nature, i.e., no object can be instantiated from this class. It also has two abstract methods defined of getArea and getPerimeter, i.e., while defining methods of subclass, these two methods need to be defined separately in each subclass. This is also seen in all three subclasses which are circle, rectangle, triangle. There is one overriding method disp which is executed by using the super keyword. Note that super keyword is also used to specify the data members of the abstract class which are filled and colour.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

An interface is a class-like construct that contains only constants and abstract methods.

An Interface in Java programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a class. A Java interface contains 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 the 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.

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

Why do we use an Interface?

* It is used to achieve total abstraction.
* Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
* It is also used to achieve loose coupling.
* Interfaces are used to implement abstraction.

Interfaces are used although abstract classes exist because abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

The relationship between the class and the interface is known as interface inheritance. Since interface inheritance and class inheritance are essentially the same, we will simply refer to both as inheritance.

access\_modifier *interface* interface\_name{

*// declare constant fields*

*// declare methods that abstract by default.*

}

Example for Interface

*interface* interface1{

*void* function1();

*int* a = 5;

}

*interface* interface2{

*void* function2();

*int* b = 6;

}

*class* class1 *implements* interface1, interface2{

*public* *void* function1(){

        System.out.println("Defined a function1 and called it");

        System.out.println("Value of a (integer assigned in interface1) is: " + a);

    }

*public* *void* function2(){

        System.out.println("Defined a function2 and called it");

        System.out.println("Value of b (integer assigned in interface2) is: " + b);

        System.out.println(b);

    }

}

*class* interfaces{

*public* *static* *void* main(*String*[] args){

        class1 obj = *new* class1();

        obj.function1();

        obj.function2();

    }

}

It should be noted that each method which has been defined in interface needs to be defined again in a class which is implementing the interface1 and interface2. The variables that are defined in the interface are not changeable, i.e., the variable is declared static and final. Constructors cannot be implemented in any interface.

Diagram

Description automatically generated