**Interfaces**

Interfaces is a powerful concept of Java which helps in achieving multiple inheritance to certain extent.

Interface is similar to a abstract class except that all the methods in it are abstract. We can not include the method body for any of the methods in interface i.e. we can not define concrete methods in interface.

The syntax for creating an *interface* is

***interface interface-name  
{  
    return-type method-name-1(parameters);  
  
    return-type method-name-2(parameters);  
}***

Here interface is a keyword, *interface-name* is the name of the interface. The methods can be declared as shown. Please note that there is no method body enclosed in flower brackets { }.

class CricketPlayersUsingInterfaces  
{  
    public static void main(String s[])  
    {  
      
        StrongBatsmen sachin = new StrongBatsmen("Sachin", 250, 11324, 100, 125);  
        StrongWicketKeeper dhoni = new StrongWicketKeeper("Dhoni", 153, 6021, 120, 67);  
        StrongBatsmen shewag = new StrongBatsmen("Shewag", 110, 4341, 22, 40);  
        AllRounderBatsmen yuvraj = new AllRounderBatsmen("Yuvraj", 105, 6533, 15, 46);  
        StrongBatsmen kohli = new StrongBatsmen("Kohli", 75, 4003, 25, 60);  
        AllRounderBatsmen raina = new AllRounderBatsmen("Raina", 34, 2600, 12, 19);  
        AllRounderBatsmen rohit = new AllRounderBatsmen("Rohit", 25, 1500, 5, 9);  
        StrongBowler harbhajan = new StrongBowler("Harbhajan", 189, 1500, 320, 4);  
        StrongBowler zaheer = new StrongBowler("Zaheer", 150, 900, 220, 4);  
        StrongBowler umesh = new StrongBowler("Umesh", 25, 150, 105, 2);  
        AllRounderBowler ashwin = new AllRounderBowler("Aswin", 15, 200, 60, 2);  
      
        bowl(1, zaheer);  
        bowl(2, umesh);  
        bowl(3, ashwin);  
        bowl(4, yuvraj);  
        bowl(5, raina);  
    }  
      
    public static void bowl(int overNumber, IBowler iBowler)  
    {  
        System.out.println("Bowling over " + overNumber);  
        System.out.println("--------------------------");  
        iBowler.bowlYorkers();  
        iBowler.takeWickets();  
    }  
}  
  
abstract class Player  
{  
    String name;  
    int matchesPlayed;  
    int runsScored;  
  
    Player(String name, int matchesPlayed, int runsScored)  
    {  
        this.name = name;  
        this.matchesPlayed = matchesPlayed;  
        this.runsScored = runsScored;  
    }  
      
      
    public void bat()  
    {  
    }  
  
    public void makeSomeRuns()  
    {  
    }  
  
    void print()  
    {  
        System.out.print(name  + " played " + matchesPlayed + " matches and scored " + runsScored + " runs.");  
    }  
}  
  
class StrongBatsmen extends Player implements IBatsmen  
{  
    int numberOfCenturies;  
    int numberOfHalfCenturies;  
  
    StrongBatsmen(String name, int matchesPlayed, int runsScored, int numberOfCenturies, int numberOfHalfCenturies)  
    {  
        super(name, matchesPlayed, runsScored);  
        this.numberOfCenturies = numberOfCenturies;  
        this.numberOfHalfCenturies = numberOfHalfCenturies;  
    }  
  
    public void openInnings()  
    {  
    }  
  
    public void makeCentury()  
    {  
    }  
      
    public void makeHalfCentury()  
    {  
    }  
  
    void print()  
    {      
        super.print();  
        System.out.print(" He is a strong batsmen and made " + numberOfCenturies +  " centuries and " + numberOfHalfCenturies + " half centuries.");  
    }  
}  
  
  
class StrongBowler extends Player implements IBowler  
{  
    int numberOfWickets;  
    int numberOf5WicketInnings;  
      
    StrongBowler(String name, int matchesPlayed, int runsScored, int numberOfWickets, int numberOf5WicketInnings)  
    {  
        super(name, matchesPlayed, runsScored);  
        this.numberOfWickets= numberOfWickets;  
        this.numberOf5WicketInnings = numberOf5WicketInnings;  
    }  
  
    public void openInnings()  
    {  
    }  
  
    public void bowlYorkers()  
    {  
        System.out.println("Strong Bowler " + name + " is bowling yorkers.");  
    }  
      
    public void takeWickets()  
    {  
        System.out.println("Strong Bowler " + name + " is taking wickets.");  
    }  
  
    void print()  
    {      
        super.print();  
        System.out.print(" He is also a good bowler and has taken " + numberOfWickets + " wickets. He has " + numberOf5WicketInnings + " 5WI(5-Wicket Innings) in his account.");  
    }  
  
}  
  
class StrongWicketKeeper extends Player implements IKeeper  
{  
    int numberOfCatches;  
    int numberOfStumpings;  
      
    StrongWicketKeeper(String name, int matchesPlayed, int runsScored, int numberOfCatches, int numberOfStumpings)  
    {  
        super(name, matchesPlayed, runsScored);  
        this.numberOfCatches = numberOfCatches;  
        this.numberOfStumpings = numberOfStumpings;  
    }  
  
    public void keepWickets()  
    {  
    }  
      
    public void stumpBatsmen()  
    {  
    }  
      
    public void makeAppeals()  
    {  
    }  
  
    void print()  
    {      
        super.print();  
        System.out.print(" He also keeps the wickets and has " + numberOfCatches + " catches and " + numberOfStumpings + " stumpings in his account.");  
    }  
}  
  
class AllRounderBatsmen extends StrongBatsmen implements IBowler  
{  
  
    AllRounderBatsmen(String name, int matchesPlayed, int runsScored, int numberOfCenturies, int numberOfHalfCenturies)  
    {  
        super(name, matchesPlayed, runsScored, numberOfCenturies, numberOfHalfCenturies);  
    }  
  
    public void openInnings()  
    {  
    }  
  
    public void bowlYorkers()  
    {  
        System.out.println("All Rounder Batsmen " + name + " is bowling yorkers.");  
    }  
      
    public void takeWickets()  
    {  
        System.out.println("All Rounder Batsmen " + name + " is taking wickets.");  
    }  
}  
  
class AllRounderBowler extends StrongBowler implements IBatsmen  
{  
  
    AllRounderBowler(String name, int matchesPlayed, int runsScored, int numberOfWickets, int numberOf5WicketInnings)  
    {  
        super(name, matchesPlayed, runsScored, numberOfWickets, numberOf5WicketInnings);  
    }  
  
    public void openInnings()  
    {  
    }  
  
    public void makeCentury()  
    {  
    }  
      
    public void makeHalfCentury()  
    {  
    }  
}  
  
  
  
interface IBatsmen  
{  
    void bat();  
  
    void makeSomeRuns();  
  
    void openInnings();  
  
    void makeCentury();  
  
    void makeHalfCentury();  
}  
  
interface IBowler  
{  
    void openInnings();  
  
    void bowlYorkers();  
  
    void takeWickets();  
}  
  
interface IKeeper  
{  
    void keepWickets();  
  
    void stumpBatsmen();  
  
    void makeAppeals();  
}  
  
  
interface ICaptain  
{  
    void selectTeam();  
      
    void setField();  
}

**Differences between Abstract and Interfaces**

|  |  |
| --- | --- |
| **Interfaces** | **Abstract Classes** |
| Interfaces can contain only abstract methods. We can not define any concrete methods in an interface. | Abstract classes can contain both abstract methods and concrete methods. |
| Interfaces can not have any member variables. But they can have static variables. | Abstract classes can contain member variables. |
| All methods in an interface are by default public. We can not change the access specifier to private or protected. | Abstract methods have to be either protected or public. They can not be private. |
| A class can implement multiple interfaces using implements keyword. | A class can extend only one abstract class. |
| An interface can extend multiple interfaces using the extends keyword. | An abstract class can only extend one other class. |

**Similarities:**

* Similar to abstract classes, we can not create objects of interfaces, but they can be used as references. Any object of the class implementing that interface can be assigned to that reference. See LINE A and LINE D below.
* Every class implementing an interface has 'IS-A' relation with interface, so where ever we can use the interface, we can also use the class which implements it.
* Similar to abstract classes, the abstract methods can be called using the interface references. See LINE B below.
* As with classes extending abstract class, any class implementing an interface should implement all the methods of the interface. If it is not implementing all the methods, then that class should be marked as abstract
* interface IA  
  {  
      void printA();  
  }  
    
  class A implements IA  
  {  
      public void printA()  
      {  
          System.out.println("A's implementation of printA method.");  
      }  
  }  
    
  ...  
    
  IA ia = new A(); // Valid - LINE A  
  ia.printA();  // Valid - LINE B  
    
  A a1 = new A();  
  ia = a1; // Valid - LINE D  
    
  IA ia2 = new IA(); // INVALID  
  ...

**Example of a Banking System using Interfaces**

class TestAccountInterface  
{  
    public static void main(String s[])  
    {  
        IAccount account = new HDFCAccount();  
      
        System.out.println("Transacting using HDFC Account");  
        transactOnAccount(account);  
        System.out.println();  
      
        account = new StateBankAccount();  
      
        System.out.println("Transacting using State Bank Account");  
        transactOnAccount(account);  
    }  
      
    public static void transactOnAccount(IAccount account)  
    {  
        System.out.println("------------------------------");  
        account.deposit(10000.0);  
        printBalance("depositing 10,000.0", account);  
        account.withdraw(2500.0);  
        printBalance("withdrawing 2,500.0", account);  
        account.withdraw(4100.0);  
        printBalance("withdrawing 4,100.0", account);  
        account.deposit(5000.0);  
        printBalance("depositing 5,000.0", account);  
        System.out.println("------------------------------");  
    }  
      
    public static void printBalance(String message, IAccount account)  
    {  
        System.out.println("The balance after " + message + " is " + account.getBalance() +".");  
    }  
  
}  
  
interface IAccount  
{  
    double getBalance();  
  
    void deposit(double amount);  
  
    void withdraw(double amount);  
}  
  
class HDFCAccount implements IAccount  
{  
    double deposits;  
    double withdrawals;  
  
  
    public double getBalance()  
    {  
        return deposits - withdrawals;  
    }  
  
    public void deposit(double amount)  
    {  
        deposits += amount;  
    }  
  
    public void withdraw(double amount)  
    {  
        withdrawals += amount;  
    }  
}  
  
class StateBankAccount implements IAccount  
{  
    double balance;  
  
    public double getBalance()  
    {  
        return balance;  
    }  
  
    public void deposit(double amount)  
    {  
        balance += amount;  
    }  
  
    public void withdraw(double amount)  
    {  
        balance -= amount;  
    }  
}

**Interfaces – Creating Them with Examples**

* A simple interface. When class A implements IA, then it has to implement all the methods in the interface IA.

interface IA  
{  
    void method1();  
      
    int method2(int x, int y);  
}  
  
class A implements IA  
{  
    public void method1()  
    {  
        // Some code here  
    }  
  
    public int method2(int x, int y)  
    {  
        // Some code here  
    }  
}

* An interface extending other interface. Here interface IB extends interface IA and when B implements interface IB, it has to implement the methods from interface IA as well as interface IB.

interface IA  
{  
    void method1();  
      
    int method2(int x, int y);  
}  
  
interface IB extends IA  
{  
    double method3();  
}  
  
class B implements IB  
{  
    public void method1()  
    {  
        // Some code here  
    }  
  
    public int method2(int x, int y)  
    {  
        // Some code here  
    }  
  
    public double method3()  
    {  
        // Some code here  
    }  
}

* An interface extending multiple interfaces. Here interface ID extends from IA, IB and IC. If we define any class implementing ID, then the methods methodA, methodB, methodC and methodD has to be implemented.

interface IA  
{  
    void methodA();  
}  
  
interface IB  
{  
    void methodB();  
}  
  
interface IC  
{  
    void methodC();  
}  
  
interface ID extends IA, IB, IC  
{  
    void methodD();  
}

* Marker Interface

interface IA  
{  
}  
  
interface IB  
{  
    void methodB();  
}  
  
interface IC extends IA, IB  
{  
}

* An interface can have constant (public static final) variables. In fact, it can have only public static final variables. Either we put these modifiers before or not, they will be considered as public static final. For e.g., PI, PI\_2 and PI\_3 are such variables. Even though we do not put the modifiers public static final before, they will behave as if they are public static final.

interface IA  
{  
    public static final double PI = 3.14;  
    double PI\_2 = 3.14;  
    static double PI\_3 = 3.14;  
  
    void method1();  
}

**Packages and Access Control**

package com.naturalresources;  
  
class Light  
{  
    double frequency;  
    double wavelength;  
}

package com.electrical.home;  
  
class Light  
{  
    double wattage;  
    double price;  
    String color;  
}

com.naturalresouces.Light naturalLight = new com.naturalresouces.Light();  
System.out.println("Frequency of natural Light : " + naturalLight.frequency);  
  
// Creating the electrical Light  
com.electrical.home.Light electricLight = new com.electrical.home.Light();  
System.out.println("Wattage of Electric Light : " + electricLight.wattage);

**Access Control**

Access Control is a way of preventing the misuse of data belonging to a class. The following program shows how the data can be corrupted or can become inconsistent.

class TestStudentMarks  
{  
    public static void main(String arg[])  
    {  
        Student uday = new Student("Uday");  
          
        uday.subject1 = 87;  
        uday.subject2 = 65;  
        uday.subject3 = 93;  
          
        uday.total\_marks = 235;  
          
        System.out.println("Subject 1 = " + uday.subject1);  
        System.out.println("Subject 2 = " + uday.subject2);  
        System.out.println("Subject 3 = " + uday.subject3);  
        System.out.println("Total = " + uday.total\_marks);      
    }  
}  
  
class Student  
{  
    String name;  
  
    Student(String name)  
    {  
        this.name = name;  
    }  
  
    int subject1;  
    int subject2;  
    int subject3;  
    int total\_marks;  
}

**There is a problem with above approach**

class TestStudentMarksImproved  
{  
    public static void main(String arg[])  
    {  
        Student uday = new Student("uday ");  
          
        uday.setMarks(87, 65, 93);  
          
        System.out.println("Subject 1 = " + uday.getSubject1Marks());  
        System.out.println("Subject 2 = " + uday.getSubject2Marks());  
        System.out.println("Subject 3 = " + uday.getSubject3Marks());  
        System.out.println("Total = " + uday.getTotalMarks());  
          
          
        // LINE X - THROWS COMPILATION ERROR  
        // madhavan.subject1 = 87;   
        // LINE Y - THROWS COMPILATION ERROR  
        // System.out.println("Subject 1 = " + uday.subject1);  
          
      
    }  
}  
  
class Student  
{  
    private String name;  
    private int subject1;  
    private int subject2;  
    private int subject3;  
    private int total\_marks;  
  
    Student(String name)  
    {  
        this.name = name;  
    }  
  
    void setMarks(int subject1, int subject2, int subject3)  
    {  
        this.subject1 = subject1;  
        this.subject2 = subject2;  
        this.subject3 = subject3;  
  
        this.total\_marks = subject1 + subject2 + subject3; // LINE A  
    }  
  
    int getSubject1Marks()  
    {  
        return subject1; // LINE B  
    }  
  
    int getSubject2Marks()  
    {  
        return subject2;  
    }  
  
    int getSubject3Marks()  
    {  
        return subject3;  
    }  
  
    int getTotalMarks()  
    {  
        return total\_marks;  
    }  
}

**Access Modifiers in Java**

**Private:**The most strict level of access control allowed in Java is private. Any member variable or method marked as private will be only accessible in that class. But we might not need such a strong restriction all the time.

**Note**: that while private, protected and public are keywords, the *default* access level is N OT a keyword. If we do not place either private, protected or public before a member variable or method, then it is automatically considered as *default* access level

**Default**:  
If a variable or method is marked as default by not placing any access specifier, then that variable or method can be only accessed in that class, its same package sub-classes or in any other class belonging to the same package. But that variable or method can not be accessed in a sub-class or non-subclass belonging to a different package.

**Protected**   
 If a variable or method is marked as protected, then that variable or method can be only accessed in that class, its sub-classes (same-package or different package) or in any other class belonging to the same package. That variable or method is only not accessible in a non-subclass belonging to a different package.

**Public**

If a variable or method is marked as public, then that variable or method is accessible everywhere. There is no restriction on its access.

**The following table summarizes the access control.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Private** | **No Modifier** | **Protected** | **Public** |
| Same class | Yes | Yes | Yes | Yes |
| Same package sub-class | No | Yes | Yes | Yes |
| Same package non sub-class | No | Yes | Yes | Yes |
| Different package sub-class | No | No | Yes | Yes |
| Different package non sub-class | No | No | No | Yes |

**final, static and others**

**final:**

The final keyword helps in making a variable unmodifiable after the initialization. If a variable is defined with final keyword, then it can not be modified after the initialization. This kind of check prevents the developers from accidentally modifying a variable, a method or a class.

* **Local variable:** If a variable is declared with the keyword final, then the value can not be changed after initialization. Here the variable pi is declared and initialized to 3.14, if we want to change it 3.45, then it causes a compilation error.

final double pi = 3.14;  
pi = 3.45; // Causes a compilation error

* **Member variable:** If a member variable of a class is marked as final, then it has to be initialized while it is declared or initialized in the constructor. Otherwise it causes a compilation error. Here the initializations in class A and B are correct, where as for class C, it is a problem since the variable j is neither initialized when declared nor initialized in the constructor.

class A  
{  
    final int i = 10;  
}  
  
class B  
{  
    final int k;  
      
    B(int k)  
    {  
        this.k = k;  
    }  
}  
  
class C  
{  
    final int j; // Causes a compilation error  
  
    C()  
    {  
    }  
}

* **Class or static variable:** If a static variable of a class is marked as final, then it has to be initialized while it is declared or initialized in the static block. Otherwise it causes a compilation error. Here the initializations in class M and N are correct, where as for class O, it is a problem since the variable j is neither initialized when declared nor initialized in the static block.

class M  
{  
    static final int i = 10;  
}  
  
class N  
{  
    static final int k;  
      
    static  
    {  
        k = 4 \* 20;  
    }  
}  
  
class O  
{  
    static final int j; // Causes a compilation error  
}

* **Method**: If a method of a class is marked as final, then it can not be overridden in any of its sub-classes. It causes a compilation error, if we try to override a final method.

class X  
{  
    final void print()  
    {  
        System.out.println("This method can not be overridden.");  
    }  
}  
  
class Y extends X  
{  
    void print() // Causes a compilation error, can not override a final method  
    {  
    }      
}

* **Class**: If a class itself is marked as final, then it can not be inherited (or extended). It causes a compilation error, if we try to extend a final class. Here class P is marked as final, hence it can not be inherited.

final class P  
{  
}  
  
class Q extends P  
{  
    // Causes a compilation error, can not inherit a final class  
}

**static:**

static keyword helps in declaring class variables. static variables can be used to define global variables.

As we know, with member variables of a class, if multiple objects are created, then every object will have its variable and changing variable of one object does not impact the variable of the other object.

class A  
{  
    int i;  
}  
...  
A a1 = new A();  
A a2 = new A();  
a1.i = 30;  
a2.i = 50;

It can be used to track number of objects created or some other usecase

class CountStudents  
{  
    public static void main(String s[])  
    {  
        Student st1 = new Student("Manohar", 34, 'A');  
        Student st2 = new Student("Uday", 78, 'B');  
        System.out.println("Number of students after st1, st2 : " + Student.count);  
        Student st3 = new Student("Kartik", 65, 'A');  
        System.out.println("Number of students after st3: " + Student.count);  
        Student st4 = createStudent();  
        System.out.println("Number of students after st4: " + Student.count);  
        System.out.println("Print count using objects : " + st1.count + " " + st2.count + " " + st3.count + " " + st4.count);  
    }  
      
    public static Student createStudent()  
    {  
        return new Student("New Student", 59, 'C');  
    }  
}  
  
class Student  
{  
    // Static variable  
    static int count = 0; // LINE A  
  
    // Member variables  
    String name;  
    int marks;  
    char section;  
  
    Student(String name, int marks, char section)  
    {  
        this.name = name;  
        this.marks = marks;  
        this.section = section;  
        // Increment the static variable  
        count++; // LINE B  
    }  
}

**Marking static variables as final**

The static variables of any class can be marked as final, so that they become constant or unmodifiable. Declaring the constants as public static final is a very common practice.

public static final double PI = 3.14;  
public static final int NUMBER\_OF\_BALLS\_IN\_A\_OVER = 6;

**Other uses - static methods**

**Static blocks:**

We can use static blocks inside a class to initialize static variables or do any one time activities, we want to perform before the class is used.

class Temperature  
{  
    public static final double FEET\_TO\_METER\_CONVERSION = 0.3048;  
    public static final double METER\_TO\_FEET\_CONVERSION;  
      
    static  
    {  
        METER\_TO\_FEET\_CONVERSION = 1 / FEET\_TO\_METER\_CONVERSION;  
        // Do other one time activities, like loading data from file, creating database connection pool etc.  
    }  
  
    static  
    {  
        // Do some more activities  
    }  
}

As shown above, we can have as many static blocks as needed and they will be called in the same order as they appear in the file. Also note that they will be called only once, no matter how many objects are created or how many times the class is accessed.

**Static methods:**

In any class, we can also define static methods. Similar to static variables, we can access static methods, using the class name and do not need any object for calling them.

When we create static methods as shown below, we need not create unnecessary new objects, every time we want to make a conversion.

class ConversionUtils  
{  
    public static double convertToFeet(double meters)  
    {  
        ...  
    }  
  
    public static double convertToMeters(double feet)  
    {  
        ...  
    }  
  
    public static double convertToFahrenheit(double celsius)  
    {  
        ...  
    }  
  
    public static double convertToCelsius(double fahrenheit)  
    {  
        ...  
    }  
}  
....  
double meters = ConversionUtils.convertToMeters(300.25);  
double fahrenheit = ConversionUtils.convertToFahrenheit(37.67);

**static and non-static methods**

we can have static methods and variables in a class. We can also have non-static methods and variables. These are nothing but the normal member variables and methods.

class A  
{  
    static int i\_static = 0;  
    int j\_non\_static;  
  
    public static void printStatic()  
    {  
        System.out.println("i\_static : " + i\_static);  
  
        // The below two statements causes compilation errors  
        // System.out.println("j\_non\_static : " + j\_non\_static);  
        // printNonStatic();  
    }  
  
    public void printNonStatic()  
    {  
        // All the three statements work fine.  
  
        System.out.println("i\_static : " + i\_static);  
        System.out.println("j\_non\_static : " + j\_non\_static);  
        printStatic();  
    }  
}

**The access rules are :**

* We can access a static variable from a static method. e.g., i\_static is accessed from printStatic() method.
* We CAN NOT access a non-static variable or non-static method from a static method. e.g., j\_non\_static and printNonStatic() can not be accessed from printStatic() method.
* We can access a non-static variable from a non-static method. e.g., j\_non\_static can be accessed from printNonStatic() method.
* We can also access a static variable or static method from a non-static method. e.g., i\_static and printStatic() can be accessed from printNonStatic() method.

**Understanding A simple Java Program**

class PrintHelloWorld  
{  
    public static void main(String args[])  
    {  
        System.out.println("Hello World");  
    }  
}

* **Why class:** In Java, every line of code/statement should be included as part of a class, that is the reason, we have defined a class with the name of the program PrintHelloWorld.
* **Why main:** When we run the program using java PrintHelloWorld it will look for the method main in the class PrintHelloWorld. The method main is the starting point of execution of the program.
* **Why static:** If we want to call a method, we should have an object, since we can not create an object without going into main method, it will become a problem. If we make the main method as static then we can call it with out creating any object.
* **Why public:** Since the main method should be accessible from outside of the program and probably from different packages, you need to make it public. If you make it private or protected or default, then the method can not be accessed from a different package.
* **Why void:** Even if return any parameter from the main method, the program does not know what to do with it. If you want to return a process exit code, we should use the method System.exit and the not return the value from main method.
* **Why String[]:** One way to pass parameters to java program, is to use the command line arguments. When we run the program with command line arguments, the arguments are converted into a String array and passed as a parameter to the main method.