**INTERFACE**

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

* An interface can contain any number of methods.
* An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
* The byte code of an interface appears in a **.class** file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

/\* File name : NameOfInterface.java \*/

import java.lang.\*;

// Any number of import statements

public interface NameOfInterface {

 // Any number of final, static fields

 // Any number of abstract method declarations\

}

Interfaces have the following properties −

* An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.

Example

/\* File name : Animal.java \*/

interface Animal {

 public void eat();

 public void travel();

}

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

/\* File name : MammalInt.java \*/

public class MammalInt implements Animal {

 public void eat() {

 System.out.println("Mammal eats");

 }

 public void travel() {

 System.out.println("Mammal travels");

 }

 public int noOfLegs() {

 return 0;

 }

 public static void main(String args[]) {

 MammalInt m = new MammalInt();

 m.eat();

 m.travel();

 }

}

This will produce the following result −

Output

Mammal eats

Mammal travels

When overriding methods defined in interfaces, there are several rules to be followed −

* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
* The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
* An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

* A class can implement more than one interface at a time.
* A class can extend only one class, but implement many interfaces.
* An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

// Filename: Sports.java

public interface Sports {

 public void setHomeTeam(String name);

 public void setVisitingTeam(String name);

}

// Filename: Football.java

public interface Football extends Sports {

 public void homeTeamScored(int points);

 public void visitingTeamScored(int points);

 public void endOfQuarter(int quarter);

}

// Filename: Hockey.java

public interface Hockey extends Sports {

 public void homeGoalScored();

 public void visitingGoalScored();

 public void endOfPeriod(int period);

 public void overtimePeriod(int ot);

}

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

public interface Hockey extends Sports, Event

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

Example

package java.util;

public interface EventListener

{}

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

**Creates a common parent** − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

**Adds a data type to a class** − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

**Abstraction**

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

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

In Java, abstraction is achieved using Abstract classes and interfaces.

## Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

* Abstract classes may or may not contain *abstract methods*, i.e., methods without body ( public void get(); )
* But, if a class has at least one abstract method, then the class **must** be declared abstract.
* If a class is declared abstract, it cannot be instantiated.
* To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
* If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

### Example

.

/\* File name : Employee.java \*/

public abstract class Employee {

 private String name;

 private String address;

 private int number;

 public Employee(String name, String address, int number) {

 System.out.println("Constructing an Employee");

 this.name = name;

 this.address = address;

 this.number = number;

 }

 public double computePay() {

 System.out.println("Inside Employee computePay");

 return 0.0;

 }

 public void mailCheck() {

 System.out.println("Mailing a check to " + this.name + " " + this.address);

 }

 public String toString() {

 return name + " " + address + " " + number;

 }

 public String getName() {

 return name;

 }

 public String getAddress() {

 return address;

 }

 public void setAddress(String newAddress) {

 address = newAddress;

 }

 public int getNumber() {

 return number;

 }

}

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

Now you can try to instantiate the Employee class in the following way −

/\* File name : AbstractDemo.java \*/

public class AbstractDemo {

 public static void main(String [] args) {

 /\* Following is not allowed and would raise error \*/

 Employee e = new Employee("George W.", "Houston, TX", 43);

 System.out.println("\n Call mailCheck using Employee reference--");

 e.mailCheck();

 }

}

When you compile the above class, it gives you the following error −

Employee.java:46: Employee is abstract; cannot be instantiated

 Employee e = new Employee("George W.", "Houston, TX", 43);

 ^

1 error

## Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way −

### Example

/\* File name : Salary.java \*/

public class Salary extends Employee {

 private double salary; // Annual salary

 public Salary(String name, String address, int number, double salary) {

 super(name, address, number);

 setSalary(salary);

 }

 public void mailCheck() {

 System.out.println("Within mailCheck of Salary class ");

 System.out.println("Mailing check to " + getName() + " with salary " + salary);

 }

 public double getSalary() {

 return salary;

 }

 public void setSalary(double newSalary) {

 if(newSalary >= 0.0) {

 salary = newSalary;

 }

 }

 public double computePay() {

 System.out.println("Computing salary pay for " + getName());

 return salary/52;

 }

}

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

/\* File name : AbstractDemo.java \*/

public class AbstractDemo {

 public static void main(String [] args) {

 Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);

 Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

 System.out.println("Call mailCheck using Salary reference --");

 s.mailCheck();

 System.out.println("\n Call mailCheck using Employee reference--");

 e.mailCheck();

 }

}

This produces the following result −

### Output

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Mohtashim with salary 3600.0

 Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

## Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

* **abstract** keyword is used to declare the method as abstract.
* You have to place the **abstract** keyword before the method name in the method declaration.
* An abstract method contains a method signature, but no method body.
* Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Following is an example of the abstract method.

### Example

public abstract class Employee {

 private String name;

 private String address;

 private int number;

 public abstract double computePay();

 // Remainder of class definition

}

Declaring a method as abstract has two consequences −

* The class containing it must be declared as abstract.
* Any class inheriting the current class must either override the abstract method or declare itself as abstract.

**Note** − Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below −

/\* File name : Salary.java \*/

public class Salary extends Employee {

 private double salary; // Annual salary

 public double computePay() {

 System.out.println("Computing salary pay for " + getName());

 return salary/52;

 }

 // Remainder of class definition

}

**Polymorphism**

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example

Let us look at an example.

public interface Vegetarian{}

public class Animal{}

public class Deer extends Animal implements Vegetarian{}

Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples −

* A Deer IS-A Animal
* A Deer IS-A Vegetarian
* A Deer IS-A Deer
* A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal −

Example

Deer d = new Deer();

Animal a = d;

Vegetarian v = d;

Object o = d;

All the reference variables d, a, v, o refer to the same Deer object in the heap.

Virtual Methods

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

A child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

Example

/\* File name : Employee.java \*/

public class Employee {

 private String name;

 private String address;

 private int number;

 public Employee(String name, String address, int number) {

 System.out.println("Constructing an Employee");

 this.name = name;

 this.address = address;

 this.number = number;

 }

 public void mailCheck() {

 System.out.println("Mailing a check to " + this.name + " " + this.address);

 }

 public String toString() {

 return name + " " + address + " " + number;

 }

 public String getName() {

 return name;

 }

 public String getAddress() {

 return address;

 }

 public void setAddress(String newAddress) {

 address = newAddress;

 }

 public int getNumber() {

 return number;

 }

}

Now suppose we extend Employee class as follows −

/\* File name : Salary.java \*/

public class Salary extends Employee {

 private double salary; // Annual salary

 public Salary(String name, String address, int number, double salary) {

 super(name, address, number);

 setSalary(salary);

 }

 public void mailCheck() {

 System.out.println("Within mailCheck of Salary class ");

 System.out.println("Mailing check to " + getName()

 + " with salary " + salary);

 }

 public double getSalary() {

 return salary;

 }

 public void setSalary(double newSalary) {

 if(newSalary >= 0.0) {

 salary = newSalary;

 }

 }

 public double computePay() {

 System.out.println("Computing salary pay for " + getName());

 return salary/52;

 }

}

/\* File name : VirtualDemo.java \*/

public class VirtualDemo {

 public static void main(String [] args) {

 Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);

 Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

 System.out.println("Call mailCheck using Salary reference --");

 s.mailCheck();

 System.out.println("\n Call mailCheck using Employee reference--");

 e.mailCheck();

 }

}

This will produce the following result −

Output

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

Here, we instantiate two Salary objects. One using a Salary reference **s**, and the other using an Employee reference **e**.

While invoking *s.mailCheck()*, the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

mailCheck() on **e** is quite different because **e** is an Employee reference. When the compiler sees *e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and these methods are referred to as virtual methods. An overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

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.

In object-oriented terms, overriding means to override the functionality of an existing method.

Example

Let us look at an example.

[Live Demo](http://tpcg.io/R135iD)

class Animal {

 public void move() {

 System.out.println("Animals can move");

 }

}

class Dog extends Animal {

 public void move() {

 System.out.println("Dogs can walk and run");

 }

}

public class TestDog {

 public static void main(String args[]) {

 Animal a = new Animal(); // Animal reference and object

 Animal b = new Dog(); // Animal reference but Dog object

 a.move(); // runs the method in Animal class

 b.move(); // runs the method in Dog class

 }

}

This will produce the following result −

Output

Animals can move

Dogs can walk and run

In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Consider the following example −

Example

[Live Demo](http://tpcg.io/VHj8iU)

class Animal {

 public void move() {

 System.out.println("Animals can move");

 }

}

class Dog extends Animal {

 public void move() {

 System.out.println("Dogs can walk and run");

 }

 public void bark() {

 System.out.println("Dogs can bark");

 }

}

public class TestDog {

 public static void main(String args[]) {

 Animal a = new Animal(); // Animal reference and object

 Animal b = new Dog(); // Animal reference but Dog object

 a.move(); // runs the method in Animal class

 b.move(); // runs the method in Dog class

 b.bark();

 }

}

This will produce the following result −

Output

TestDog.java:26: error: cannot find symbol

 b.bark();

 ^

 symbol: method bark()

 location: variable b of type Animal

1 error

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for Method Overriding

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.

Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

Example

[Live Demo](http://tpcg.io/r51jFh)

class Animal {

 public void move() {

 System.out.println("Animals can move");

 }

}

class Dog extends Animal {

 public void move() {

 super.move(); // invokes the super class method

 System.out.println("Dogs can walk and run");

 }

}

public class TestDog {

 public static void main(String args[]) {

 Animal b = new Dog(); // Animal reference but Dog object

 b.move(); // runs the method in Dog class

 }

}

This will produce the following result −

Output

Animals can move

Dogs can walk and run

Java Inner Classes

In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

To access the inner class, create an object of the outer class, and then create an object of the inner class:

Example

class OuterClass {

 int x = 10;

 class InnerClass {

 int y = 5;

 }

}

public class MyMainClass {

 public static void main(String[] args) {

 OuterClass myOuter = new OuterClass();

 OuterClass.InnerClass myInner = myOuter.new InnerClass();

 System.out.println(myInner.y + myOuter.x);

 }

}

// Outputs 15 (5 + 10)

Private Inner Class

Unlike a "regular" class, an inner class can be private or protected. If you don't want outside objects to access the inner class, declare the class as private:

Example

class OuterClass {

 int x = 10;

 private class InnerClass {

 int y = 5;

 }

}

public class MyMainClass {

 public static void main(String[] args) {

 OuterClass myOuter = new OuterClass();

 OuterClass.InnerClass myInner = myOuter.new InnerClass();

 System.out.println(myInner.y + myOuter.x);

 }

}

If you try to access a private inner class from an outside class (MyMainClass), an error occurs:

MyMainClass.java:12: error: OuterClass.InnerClass has private access in OuterClass

 OuterClass.InnerClass myInner = myOuter.new InnerClass();

 ^

Static Inner Class

An inner class can also be static, which means that you can access it without creating an object of the outer class:

Example

class OuterClass {

 int x = 10;

 static class InnerClass {

 int y = 5;

 }

}

public class MyMainClass {

 public static void main(String[] args) {

 OuterClass.InnerClass myInner = new OuterClass.InnerClass();

 System.out.println(myInner.y);

 }

}

// Outputs 5

Note: just like static attributes and methods, a static inner class does not have access to members of the outer class.

Access Outer Class From Inner Class

One advantage of inner classes, is that they can access attributes and methods of the outer class:

Example

class OuterClass {

 int x = 10;

 class InnerClass {

 public int myInnerMethod() {

 return x;

 }

 }

}

public class MyMainClass {

 public static void main(String args[]) {

 OuterClass myOuter = new OuterClass();

 OuterClass.InnerClass myInner = myOuter.new InnerClass();

 System.out.println(myInner.myInnerMethod());

 }

}

// Outputs 10