# Abstract class

Providing necessary information and hiding unnecessary information is called abstraction.

Abstract class and interfaces we need to have abstraction.

We cannot create an object for an abstract class.

We need to inherit an abstract class and access abstract methods.

An abstract method cannot be final.

An abstract method cannot be static, static methods cannot be overridden.

An abstract method cannot be private.

An abstract class cannot be final.

The final class can be instantiated, we can create objects.

An abstract class cannot be instantiated.

The final class cannot be inherited.

An abstract class can have a constructor and those constructors are called whenever objects are created to its subclasses.

Refer: Code snippet – AbstractClassAndInheritence

|  |
| --- |
| **public** **class** AbstractClassAndInheritence {  **public** **static** **void** main(String[] args)  {  A obj = **new** B();  obj.print();  obj.show();    B obj1 = **new** B();  obj1.print();  obj1.show();  }  }  **abstract** **class** A  {  **public** **abstract** **void** print();    **public** **void** show()  {  System.***out***.println("A Class - Show method");  }  }  **class** B **extends** A  {  @Override  **public** **void** show()  {  System.***out***.println("B Class - Show method");  }    **public** **void** print()  {  System.***out***.println("Print Method");  }  } |

# Interface

It is a collection of public static final variables and public abstract methods.

In the interface, all variables are implicitly public static final and all methods are implicitly public abstract.

The interface itself is implicitly abstract.

Graphical user interface, application, Word

Description automatically generated

Class A{} => Concreate class

Abstract class B{} => Abstract class

Interface c{} => abstract interface

Abstract interface D{} => abstract interface

The interface cannot be instantiated and cannot create an object.

The interface can be inherited into a class by using extends keyword.

Whenever an interface is inherited all abstract methods of an interface must be overridden in a sub-class or sub-class must be declared with an abstract keyword otherwise compile-time error.

Refer code snippet-interfaceExample1

|  |
| --- |
| **package** abstractclassesandinterfaces;  **public** **class** interfaceExample1  {  **public** **static** **void** main(String[] args)  {  xyz obj = **new** xyz();  {  obj.show();  obj.print();  }  }  }  **interface** abc  {  **int** ***i*** = 10;  **public** **void** show();  **public** **void** print();  }  **class** xyz **implements** abc  {  **public** **void** show()  {  System.***out***.println(abc.***i***);  }    **public** **void** print()  {  System.***out***.println(abc.***i***);  }  } |

Java compilers generate .class files for interfaces also.

1. Class A{}

Class B extends A{}

1. Interface A{}

Class B implements A{}

1. Interface A{}

Interface B extends A{}

1. Class cannot be inherited into the interface.

Interfaces are introduced to achieve multiple inheritances.

Class A{}

Class B{}

Interface C{}

Interface D{}

Class Demo extends A,B =>Error

Class Demo implements C,D => Valid

Class Demo extends A implements C => Valid

Class Demo extends A implements C,D => Valid

Interface Demo extends C,D => Valid

If we have all concrete methods then use concrete class, because it helps to create objects.

If we have all abstract methods then use interface, because this helps in multiple inheritances.

If we have both abstract methods and concrete methods, then use an abstract class.

## Default methods in interface

* This feature allows to write concreate instance methods in interface by prefixing default keyword.

Refer code snippet - DefaultMethodInInterface

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  **interface** Test  {  **default** **public** **void** show()  {  System.***out***.println("Show");  }  }  **public** **class** DefaultMethodInInterface **implements** Test  {  **public** **static** **void** main(String[] args)  {  DefaultMethodInInterface obj = **new** DefaultMethodInInterface();  obj.show();  //or  Test t = **new** DefaultMethodInInterface();  t.show();  }  }  //output  Show  Show |

## Static Methods in an interface

* This feature allows to write class (static ) methods in interface.

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  **interface** Test1  {  **public** **static** **void** display()  {  System.***out***.println("display");  }  }  **public** **class** StaticMethodsInInterface  {  **public** **static** **void** main(String[] args)  {  Test1.*display*();  }  }  //OUTPUT  display |

## @FunctionalInterface Annotation

* This function ensures that the interface cannot have more than one abstract method.

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  @FunctionalInterface  **interface** Test123  {  **void** display(); //=> public abstract method, only one is allowed.  **default** **public** **void** print()  {  System.***out***.println("Print");  }  }  @FunctionalInterface  **interface** show **extends** Test123  {  **default** **void** show1()  {  System.***out***.println("show1");  }    //Still Functional Interface because only one public abstract method.  }  **public** **class** FuntionalInterfaceAnnotationExample **implements** show  {  **public** **static** **void** main(String[] args)  {  FuntionalInterfaceAnnotationExample obj = **new** FuntionalInterfaceAnnotationExample();    obj.print();  obj.show1();  obj.display();  }  **public** **void** display()  {  System.***out***.println("display");  }  }  //OUTPUT  Print  show1  display |

## More ways to implement Interface

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  @FunctionalInterface  **interface** Test1234  {  **public** **void** print();  }  **public** **class** MoreWaysToImplementInterface  {  **public** **static** **void** main(String[] args)  {  Test1234 t = **new** Test1234()  {  **public** **void** print()  {  System.***out***.println("Print");  }  };  t.print();  }  }  //OUTPUT  Print |

## Functional Interfaces

* An interface that contains only one abstract method is called a functional interface.
* Functional interfaces are required in functional programming.
* Functional programming means, that a function can be passed as a parameter to another function and the function can return another function.

Predefined Functional Interfaces

Interfaces Abstract Methods

========= ===============

Function<T,R> R apply(T t)

Predicate<T> Boolean test(T t)

Consumer<T> void accept(T t)

Supplier<T> T get()

* The above interfaces are part of java.util.function package

### Function

@FunctionalInterface

**interface** Test2<T,R>

{

       R cube(T x);

}

**public** **class** Demo1 {

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

       {    Test2<Integer,Integer> t = x -> x\*x\*x;

**int** y = t.cube(10);

             System.***out***.println(y);

       }

}

|  |
| --- |
| **//Program to multiply 3 numbers using Predefined Functional - Function Interface**  **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Function;  **public** **class** PredefinedInterface\_FuctionExample1  {  **public** **static** **void** main(String[] args)  {  Function<Integer,Integer> f = x -> x\*x\*x;  **int** res = f.apply(10);  System.***out***.println(res);  }  }  //OUTPUT  1000 |

|  |
| --- |
| **//Program to get length of a string using Predefined Functional - Function Interface**  **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Function;  **public** **class** PredefinedInterface\_FuctionExample2 {  **public** **static** **void** main(String[] args)  {  Function<String,Integer> f = x -> x.length();  **int** len = f.apply("Welcome");  System.***out***.println(len);  }  }  //OUTPUT  7 |

|  |
| --- |
| **//Program to display string uppercase**  **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Function;  **public** **class** PredefinedInterface\_FuctionExample3 {  **public** **static** **void** main(String[] args) {  Function<String,String> f = x -> x.toUpperCase();  String str = f.apply("Welcome");  System.***out***.println(str);  }  }  //OUTPUT  WELCOME |

### Predicate

Programs to demonstrate predicate interface

|  |
| --- |
| **//Program to check whether given number is positive or negative number**  **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Predicate;  **public** **class** PredefinedInterface\_PredicateExample1 {  **public** **static** **void** main(String[] args) {  Predicate<Integer> p = x -> x>0;  **boolean** res = p.test(100);  **if** (res)  {  System.***out***.println("Positive Number");  }  **else**  {  System.***out***.println("Zero or Negative Number");  }  }  //OUTPUT  Positive Number |
|  |

|  |
| --- |
| **//Program to check Even or odd numbers**  **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Predicate;  **public** **class** PredefinedInterface\_PredicateExample2 {  **public** **static** **void** main(String[] args) {  Predicate<Integer> p = x -> x%2==0;  **boolean** res = p.test(100);  **if** (res)  {  System.***out***.println("Even Number");  }  **else**  {  System.***out***.println("Odd Number");  }  }  }  //OUTPUT  Even Number |

### Consumer

Program to demonstrate Consumer interface

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Consumer;  **public** **class** PredefinedInterface\_ConsumerExample1 {  **public** **static** **void** main(String[] args) {  Consumer<Integer> c = x -> System.***out***.println(x);  c.accept(10);  }  }  //OUTPUT  10 |

### Supplier

Program to demonstrate Supplier interface

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  **import** java.util.function.Supplier;  **public** **class** PredefinedInterface\_SupplierExample1 {  **public** **static** **void** main(String[] args) {  Supplier<Integer> t = () -> 39;  **int** x = t.get();  System.***out***.println(x);  }  }  //OUTPUT  39 |

# Encapsulation

The binding of methods and variables into a class is called encapsulation.

Refer code snippet - EncapsulationExample1

|  |
| --- |
| **package** AbstractClassesAndInterfacesAndEncapsulation;  **public** **class** EncapsulationExample1  {  **public** **static** **void** main(String[] args)  {  Employee e1 = **new** Employee();  e1.setEmpno(101);  e1.setName("Gowtham");  e1.setSalary(10100);    System.***out***.println(e1.getEmpno()+" - "+e1.getName()+" - "+e1.getSalary());  }  }  **class** Employee  {  **private** **int** empno;  **private** String name;  **private** **float** salary;  **public** **int** getEmpno() {  **return** empno;  }  **public** **void** setEmpno(**int** empno) {  **this**.empno = empno;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **float** getSalary() {  **return** salary;  }  **public** **void** setSalary(**float** salary) {  **this**.salary = salary;  }  } |