|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# OOP: Abstraction

## declaration

### abstract class

Abstract class is a class marked as "abstract". Abstract class can't be instantiated as object but implemented by a concrete sub-class.

Abstract method is a method marked as "abstract" defined in abstract class. Abstract method can't be implemented as it is declared, but should be implemented by the methods define d in the concrete sub-class.

Abstract class and methods define design principles, which further class development should be consistent.

The abstract class "Insect" is implemented by sub-class "Beetle", which implements all abstract methods.

|  |
| --- |
| standard approach: |
| abstract class Insect {  abstract int getLegs();  }  class Beetle extends Insect{  int getLegs(){  System.out.println("number of legs");  return 10;  }  } |

Therefore, we can summarize some features regarding of abstract class:

* class and methods must be marked as "abstract". abstract methods can't be defined in non-abstract class.
* concrete class must implements all abstract methods if the concrete class extends abstract class.
* Only declare abstract methods in abstract class, but no method body is defined in abstract methods.
* concrete methods should be consistent or compatible with the signatures of abstract methods: method name, return type and arguments should be consistent. access identifier should not broaden parent access.
* Abstract class can extends other abstract class or interface. Concrete class can only extends one abstract class rather than multiple ones.

### Interface

Interface is template of a class. Interface is a collection of abstract methods and static constants.

**An interface is implicitly declared as public and abstract.**

**Interface can't be marked as private/final** because interface should be implemented by other classes or extended by other interfaces.

Define an interface

|  |
| --- |
| An interface is implicitly public and abstract. The blow approaches are equal. |
| interface Test{} |
| public interface Test{} |
| public abstract interface Test{} |
| abstract interface Test{} |

|  |
| --- |
|  |
|  |

### compare

|  |  |  |
| --- | --- | --- |
|  | **interface** | **abstract class** |
| **define** | interface <name>{  …  } | abstract class <name>{  …  } |
| **variables** | static constants, which are public, static, final variables.  must be initialized at the time of declaration | The same as virtual(non-abstract) class |
| **initialization** | No constructor.  No instance or static block |
| **access identifier** | default is public  can't be private/protected |
| **final methods** | No final method |
| **default methods** | default is abstract methods  instance methods must be marked as "default" | default is instance methods  abstract methods must be marked as "abstract" |
| **abstract methods** | public  can’t be private/protected/final  don't include method body. | public/protected  can't be private/final  don't include method body. |
| **inheritance** | interface don't inherit from java.lang.Object. | In java, all class including abstract class extends java.lang.Object. |
| **inheritance (implements)** | concrete implements interface. | abstract implements interface |
| **inheritance (extends)** | interface extends interface.  abstract extends interface.  interface can't extends class | concrete extends abstract  abstract extends abstract |
| **multiple-inheritance** | concrete implements any number of interfaces  Interfaces support multiple inheritance | concrete extends only one abstract class  abstract classes only support single inheritance. Multiple inheritance is not allowed |
| **concrete** | can’t be instantiated.  concrete class must implement all abstract methods | |
| **static method** | must be marked as static. methods body must be defined | |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## variables

### variables

Pattern of variables defined in abstract class is same as that defined in non-abstract class.

|  |
| --- |
| output: 2,10,10,10 |
| public class AbstractVariable {  public static void main(String[] args) {  duck c = new duck();  c.print();  }  }  abstract class bird{  static int wings=2;  }  class duck extends bird{  duck(){  System.out.println(wings);  wings=10;  }  void print(){  System.out.println(wings);  System.out.println(this.wings);  System.out.println(super.wings);  }  } |

|  |
| --- |
| output: 0,10,10,2. Variables in abstract class is hidden. |
| public class AbstractVariable {  public static void main(String[] args) {  duck c = new duck();  c.print();  }    }  abstract class bird{  static int wings=2;  }  class duck extends bird{  int wings;  duck(){  System.out.println(wings);  wings=10;  }  void print(){  System.out.println(wings);  System.out.println(this.wings);  System.out.println(super.wings);  }  } |

### define interface variables

Interface variables define in interface are **constants**. They must be **public, static and final** even if they are not explicitly declared.

They can't be marked as private/protected because they public.

They can't be assigned new value because they are final.

Interface variable can't be overridden but hidden by variables defined in concrete class because they are static.

They must be initialized at the time of declaration.

|  |
| --- |
| all variables are suggested to be marked as "public static final". The approach below are equal |
| interface Test{  public static final int age = 0;  } |
| interface Test{  int age = 0;  } |
| interface Test{  public int age = 0;  } |
| interface Test{  abstract int age = 0;  } |
| interface Test{  final int age = 0;  } |
|  |

### reference interface static

Like other class variables, interface variables can't be overridden but hidden.

|  |
| --- |
|  |
| interface MyVariable{  int weight=1;  }  public class InterfaceVariable implements MyVariable{  int weight = 10;  public static void main(String[] args){  InterfaceVariable a = new InterfaceVariable();  //hide interface variable  System.out.println(**a.weight**);  // implement interface variable  System.out.println(**MyVariable.weight**);  }  } |
| output:  10  0 |
| public class InterfaceVariable2 implements Animal{  static int age;  static{  age=10;  }  public static void main(String[] args) {  System.out.println(**age**);  System.out.println(**Tiger.age**);  System.out.println(**Animal.age**);  }  }  interface Tiger extends Animal{  static int age=1;  }  interface Animal{  static int age=0;  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## methods

### methods

Interface and interface methods must be public in default. They can't be marked as private or protected.

Interface and interface methods can't be marked as final.

Interface variables are constants which is public, final, and static.

### abstract methods

Method body is not allowed for interface abstract methods.

Interface abstract methods are **implicitly public and abstract**. Interface methods can't be marked as final, strictfp, or native. Interface methods can't be marked as private. Concrete methods must be marked as public.

concrete class must concrete all interface abstract methods. All implements methods must be the same signatures with abstract methods. Note: those methods must be marked as "public".

|  |
| --- |
| interface methods are implicitly public and abstract. The blow approaches are equal. |
| interface Test{  void getName();  } |
| interface Test{  public void getName();  } |
| interface Test{  public abstract void getName();  } |
| interface Test{  abstract void getName();  } |

|  |
| --- |
| concrete abstract method in virtual class, and implement instance method |
| public class InterfaceAbstract {  public static void main(String[] args){  **new Tiger().getName();**  }  }  class Tiger implements Animal{  String name;  Tiger(){  this.name = "tiger";  }  **public String getName(){**  System.out.println(this.name);  return this.name;  }  }  interface Animal{  **String getName();**  } |

|  |
| --- |
| output: tiger  concrete abstract method in abstract class. implement this method use casting |
| public class InterfaceAbstract2 extends Tiger{  public static void main(String[] args){  **Tiger t = new InterfaceAbstract2();**  t.getName();  }  }  abstract class Tiger implements Animal{  String name = "tiger";  **public String getName(){**  System.out.println(this.name);  return this.name;  }  }  interface Animal{  String getName();  } |

|  |
| --- |
| output: other  child method override instance method defined in abstract class. |
| public class InterfaceAbstract2 extends Tiger{  public String getName(){  System.out.println("other");  return "other";  }  public static void main(String[] args){  **Tiger t = new InterfaceAbstract2();**  t.getName();  }  }  abstract class Tiger implements Animal{  String name = "tiger";  public String getName(){  System.out.println(this.name);  return this.name;  }  }  interface Animal{  String getName();  } |

|  |
| --- |
| output: tiger  concrete abstract method in interface, and implement interface using casting. |
| public class InterfaceAbstract3 implements Tiger{  public static void main(String[] args){  **Tiger t = new InterfaceAbstract3();**  t.getName();  }  }  interface Tiger extends Animal{  String name = "tiger";  default String getName(){  System.out.println(this.name);  return this.name;  }  }  interface Animal{  String getName();  } |

## virtual methods

### interface default methods

Default methods are the same as instance methods in virtual class but can't be invoked by instantiation of interface. Concrete class inherit default methods, which could be invoked by object instantiation. Abstract class inherit default methods, which could be invoked by casting.

default methods must be marked as "default".

default methods must define method body.

default methods are public access by default. private/protected is not allowed.

default methods can't be marked as "final" or "abstract".

Default interface methods could improve backward compatibility. Interface developers can add a new method to an interface without requiring developers using the interface to recompile their code.

Here are the definitions of default methods.

|  |
| --- |
| They are equal |
| interface Test{  default void getName(){ }  } |
| interface Test{  public default void getName(){ }  } |
| interface Test{  default void getName(){ ; }  } |
| interface Test{  default void getName(){ return; }  } |

In concrete class, default method is invoked as instance method

|  |
| --- |
| output: hop |
| public class InterfaceDefault implements Frog {  public static void main(String[] args){  new InterfaceDefault().go();  }  void go(){  **System.out.println(hop());**  }  }  interface Frog{  **default String hop(){**  return "hop";  }  } |

Methods of concrete class may override default methods. It is ok to invoke default methods with the pattern "<interface>.super.<default method>"

|  |
| --- |
| output:  scrawl  hop |
| public class InterfaceDefaultInherit implements Frog {  public static void main(String[] args){  new InterfaceDefaultInherit().go();  }  void go(){  **System.out.println(hop()); //override default method**  **System.out.println(Frog.super.hop()); // invoke interface default**  }  public String hop(){  return "scrawl";  }  }  interface Frog{  default String hop(){  return "hop";  }  } |

Invoke default methods using casting if abstract class implements interface. default methods could be overridden by abstract class or concrete class, in which the methods should be explicitly marked as "public".

|  |
| --- |
| output: animal |
| public class InterfaceDefault extends Test{  public static void main(String[] args){  **Test t = new InterfaceDefault();**  System.out.println(t.getName());  }  }  abstract class Test implements Animal{ }  interface Animal{  default String getName(){  return "animal";  }  } |
| output: tiger  method in concrete class or abstract class could override default methods.  Note: the methods should be marked as public |
| public class InterfaceDefault extends Test{  **public String getName(){**  return "tiger";  }  public static void main(String[] args){  Test t = new InterfaceDefault();  System.out.println(t.getName());  }  }  abstract class Test implements Animal{  **public String getName(){**  return "test";  }  }  interface Animal{  default String getName(){  return "animal";  }  } |
|  |
|  |

### instance methods in abstract class

|  |
| --- |
| It is ok to define non-abstract methods in abstract class. |
| public class AbstractConcrete{  public static void main(String[] args) {  duck c = new duck();  c.print();  }  }  abstract class bird{  void print(){ System.out.println("bird"); }  }  class duck extends bird{} |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## implement abstract methods

concrete class -> abstract class

concrete class -> abstract class -> abstract class

concrete class -> abstract class -> interface

concrete class -> abstract class, interface

abstract class -> non-abstract class

### virtual extends abstract

|  |
| --- |
| concrete class must implement all abstract methods. |
| **class Beetle extends Insect{**  int getLegs(){  System.out.println("number of legs");  return 10;  }  int getSections(){  System.out.println("number of sections");  return 10;  }  }  **abstract class Insect {**  abstract int getLegs();  abstract int getSections();  } |

define constructor in abstract class

|  |
| --- |
| output: 13 |
| public class Test extends Insect{  public Test(String color) {  System.out.println("3");  }  public static void main(String[] args) {  new Test("bee");  }  }  abstract class Insect {  public Insect() { System.out.println("1"); }  public Insect(String color) { System.out.println("2"); }  } |

### virtual implements interface

concrete class extends interface

|  |
| --- |
|  |
| **public class InterfaceTest implements MyInterface{**  public void print(){  System.out.println("concrete1");  }  public static void main(String[] args){  InterfaceTest c = new InterfaceTest();  c.print();  }  }  **interface MyInterface{**  void print();  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### abstract implements interface

Abstract class implements interface:

* Abstract class implements interface abstract methods, which are implemented by concrete class.
* abstract class implements interface abstract methods, which are implemented by concrete class.

|  |
| --- |
| output:  concrete: print  concrete: test |
| **public class InterfaceExtends extends MyAbstract{**  public void print(){  System.out.println("concrete: print");  }  public void test(){  System.out.println("concrete: test");  }  public static void main(String[] args){  InterfaceExtends c = new InterfaceExtends();  c.print();  c.test();  }  }  **abstract class MyAbstract implements Test{**  abstract void print();  }  **interface Test{**  void test();  } |

abstract class implements interface abstract methods.

|  |
| --- |
| output:  concrete: print  concrete: test |
| public class InterfaceExtends extends MyAbstract{  public void print(){  System.out.println("concrete: print");  }  **public void test(){**  **System.out.println("concrete: test");**  **}**  public static void main(String[] args){  InterfaceExtends c = new InterfaceExtends();  c.print();  c.test();  }  }  abstract class MyAbstract implements Test{  abstract void print();  **public void test(){**  **System.out.println("abstract: test");**  **}**  }  interface Test{  **void test();**  } |
| output:  concrete: print  abstract: test |
| public class InterfaceExtends extends MyAbstract{  public void print(){  System.out.println("concrete: print");  }  public static void main(String[] args){  InterfaceExtends c = new InterfaceExtends();  c.print();  c.test();  }  }  abstract class MyAbstract implements Test{  abstract void print();  **public void test(){**  **System.out.println("abstract: test");**  **}**  }  interface Test{  **void test();**  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### interface extends interface

interface extends other interface

|  |
| --- |
|  |
| **public class InterfaceInherit implements MyInterface{**  public void print(){  System.out.println("concrete: print");  }  public void test(){  System.out.println("concrete: test");  }  public static void main(String[] args){  InterfaceInherit c = new InterfaceInherit();  c.print();  c.test();  }  }  **interface MyInterface extends Test{**  void print();  }  **interface Test{**  void test();  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### abstract extends abstract

|  |
| --- |
| abstract class extends another abstract class, and concrete class must implement all abstract methods define in both of them. |
| abstract class Life{  abstract int getSections();  }  **abstract class Insect extends Life{**  abstract int getLegs();  }  **class Beetle extends Insect{**  int getLegs(){  System.out.println("number of legs");  return 10;  }  public int getSections(){  System.out.println("number of sections");  return 10;  }  } |

|  |
| --- |
| concrete class extends an abstract class and implements an interface. |
| interface Life{  abstract int getSections();  }  abstract class Insect{  abstract int getLegs();  }  **class Beetle extends Insect implements Life{**  int getLegs(){  System.out.println("number of legs");  return 10;  }  public int getSections(){  System.out.println("number of sections");  return 10;  }  } |

### abstract extends non-abstract

|  |
| --- |
| concrete class -> abstract class -> non-abstract class |
| public class AbstractExtends extends Tiger{  int age = 10;  int getAge(){  System.out.println(this.age);  return this.age;  }  public static void main(String[] args){  AbstractExtends c = new AbstractExtends();  c.getAge();  System.out.println(c.getName());  }  }  class Animal{  String getName(){  return "animal";  }  }  abstract class Tiger extends Animal{  abstract int getAge();  } |

## access identifiers

The rule of access compatibility: the order is public > protected > default(package private) > private. Access of concrete methods should be broader than that of abstract methods. The same as the rule of access that works in virtual class,

If abstract method is default, the concrete methods could be default/protected/public

If abstract method is protected, the concrete methods could be protected/public

If abstract method is public, the concrete methods should be public

Abstract methods defined in abstract class can't be private but default/protected/public is ok.

Abstract methods defined in interface is public in default. That can't be private/protected.

**Therefore, private identifier should not appear in abstract methods and concrete methods.**

### virtual class concrete abstract

|  |
| --- |
|  |
| class Beetle extends Insect{  **protected int getLegs(){**  System.out.println("number of legs");  return 10;  }  }  abstract class Insect {  **abstract int getLegs();**  } |
| class Beetle extends Insect{  **protected int getLegs(){**  System.out.println("number of legs");  return 10;  }  }  abstract class Insect {  **protected abstract int getLegs();**  } |
| class Beetle extends Insect{  **public int getLegs(){**  System.out.println("number of legs");  return 10;  }  }  abstract class Insect {  **public abstract int getLegs();**  } |

**class concrete interface abstract methods**

The concrete method must be public because all inerface abstract methods are public in default.

|  |
| --- |
| the concrete method must be public |
| class Beetle implements Insect{  **public int getLegs(){**  System.out.println("number of legs");  return 10;  }  }  interface Insect {  int getLegs();  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### abstract class concrete abstract

|  |
| --- |
|  |
| public class ClassTest extends Beetle{  public static void main(String[] args){  Beetle t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle extends Insect{  **int getLegs(){ //could be default/protected/public**  System.out.println("Legs of beetle");  return 10;  }  }  abstract class Insect {  **abstract int getLegs();**  } |
| public class ClassTest extends Beetle{  public static void main(String[] args){  Beetle t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle extends Insect{  **protected int getLegs(){ //could be protected/public**  System.out.println("Legs of beetle");  return 10;  }  }  abstract class Insect {  **protected abstract int getLegs();**  } |
| public class ClassTest extends Beetle{  public static void main(String[] args){  Beetle t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle extends Insect{  **public int getLegs(){ // must be public**  System.out.println("Legs of beetle");  return 10;  }  }  abstract class Insect {  **public abstract int getLegs();**  } |

|  |
| --- |
| concrete method must be public if it is interface abstract method |
| public class ClassTest extends Beetle{  public static void main(String[] args){  Beetle t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle implements Insect{  **public int getLegs(){**  System.out.println("Legs of beetle");  return 10;  }  }  interface Insect {  int getLegs();  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### override

The access of overridden method > concrete method > abstract method

|  |
| --- |
|  |
| public class ClassTest extends Beetle{  **int getLegs(){**  System.out.println("Legs of Test");  return 10;  }  public static void main(String[] args){  ClassTest t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle extends Insect{  **int getLegs(){**  System.out.println("Legs of beetle");  return 10;  }  }  abstract class Insect {  **abstract int getLegs();**  } |
| public class ClassTest extends Beetle{  **public int getLegs(){ //should be protected/public**  System.out.println("Legs of Test");  return 10;  }  public static void main(String[] args){  ClassTest t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle extends Insect{  **protected int getLegs(){**  System.out.println("Legs of beetle");  return 10;  }  }  abstract class Insect {  **abstract int getLegs();**  } |

**For interface, either concrete method or overridden method should be public.**

|  |
| --- |
|  |
| public class ClassTest extends Beetle{  public int getLegs(){  System.out.println("Legs of Test");  return 10;  }  public static void main(String[] args){  ClassTest t = new ClassTest();  t.getLegs();  }  }  abstract class Beetle implements Insect{  public int getLegs(){  System.out.println("Legs of beetle");  return 10;  }  }  interface Insect {  int getLegs();  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## lambda expression

### Functional interface

lambda expression is object in Java8 (In python, lambda expression is lambda function). lambda expression is concrete of an **functional interface**. Functional interface consists of a single abstract method and no other methods.

|  |
| --- |
| Interface Runnable define single one abstract method run(). |
| @FunctionalInterface  public interface Runnable {  public abstract void run();  } |

There are a few examples of the functional interfaces newly added in Java 8:

|  |
| --- |
|  |
| public interface Predicate {  boolean test(T t);  }  public interface Function<T, R> {  R apply(T t);  }  public interface BinaryOperator {  T apply(T left, T right);  }  public interface Consumer {  void accept(T t);  }  public interface Supplier {  T get();  } |

|  |
| --- |
|  |
|  |

### lambda expression

parameters -> expression

|  |
| --- |
| ()-> true;  ()-> {return true;} |
| x -> x\*x;  (x) -> x\*x;  (x) -> {return x\*x;}  (int x) -> x\*x;  (int x) -> {return x\*x;} |
| (x, y) -> x\*y;  (x, y) -> { return x\*y;}  (int x, int y) -> x\*y;  (int x, int y) -> { return x\*y;} |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### anonymous class

Define a functional interface, single one abstract method call() is defined. The below is a common approach: create a concrete class Lambda1 that implements this abstract method. This concrete class could be normal virtual class with name or anonymous class.

|  |
| --- |
| concrete call() in class Lambda1 or anonymous class |
| public class Lambda1 implements LambdaFunc{  public void call(){  System.out.println("Hello world");  }  public static void main(String[] args){  **LambdaFunc t = new Lambda1();**  **t.call();**  }  }  interface LambdaFunc{  void call();  } |
| public class Lambda2{  public static void main(String[] args){  **LambdaFunc t = new LambdaFunc() {**  **public void call(){**  **System.out.println("Hello world");**  **}**  **};**  t.call();  }  }  interface LambdaFunc{  void call();  } |

Here is more concise approach for lambda expression: apply lambda expression as anonymous class which concrete this interface method.

Note: the literal at the beginning is interface name, followed by lambda expression, which concrete interface method.

|  |
| --- |
| output: Hello world |
| public class Lambda3 {  public static void main(String[] args){  **LambdaFunc t = () -> System.out.println("Hello world");**  t.call();  }  }  interface LambdaFunc{  void call();  } |

|  |
| --- |
| output: 1 |
| public class LambdaTest {  public static void main(String[] args){  **Test t = (a) -> a++;**  int x= 1;  System.out.println(t.go(x));  }  }  interface Test{  int go(int x);  } |

|  |
| --- |
| output: 25 |
| public class LambdaTest {  public static void main(String[] args){  Test t = (int a) -> {  a +=4;  return a\*a;  };  int x= 1;  System.out.println(t.go(x));  }  }  interface Test{  int go(int x);  } |

|  |
| --- |
|  |
|  |

### pass arguments

// lambda function introducted in Java 8

// functional programming: pass function as argument into method

// That method is called reusable and abstract process.

1. declare functional interface. The interface defines a single method.

2. declare a class. In this class, declare a method and apply the interface as one of arguments. In method body, invoke the interface method.

3. invoke class method by passing various lambda functions

|  |
| --- |
| output:  1.0  18.0  28.0 |
| import java.util.ArrayList;  import java.util.List;  public class LambdaDefine2{  public static void execute(List<Integer> list, Computer c){  double result = 1D;  if(list != null){  for (Integer i : list) {  if(i != 0)  result = c.compute(result, i);  }  }  System.out.println(result);  }  public static void main(String[] args){  List<Integer> nums = new ArrayList<Integer>();  execute(nums, null);  nums.add(2);nums.add(14);nums.add(1);  // accumulative add  execute(nums, (a,b)->{ return (b!=0) ? (a+b) : a;});  // accumulative multiply  execute(nums, (a,b)->{ return a \* b;} );  }  }  interface Computer{  public double compute(double a, int b);  } |

|  |
| --- |
| output:  pass subtraction fucnion:5  pass sum fucnion:15  pass multiply fucnion:50  pass division fucnion:2 |
| public class LambdaDefine{  int operate(int a, int b, MathOperation m){  // use the funcion defined in the interface  return m.operation(a, b);  }  public static void main(String[] args){  LambdaDefine c1 = new LambdaDefine();  int res;  res = c1.operate(10,5, (a,b) -> a-b);  System.out.println("pass subtraction fucnion:" + res);  res = c1.operate(10,5, (a,b) -> { return a+b; });  System.out.println("pass sum fucnion:" + res);  res = c1.operate(10,5, (int a, int b) -> a\*b);  System.out.println("pass multiply fucnion:" + res);  MathOperation division = (int a, int b) -> {  if (b != 0) return a/b;  return 0;  };  res = c1.operate(10,5, division);  System.out.println("pass division fucnion:" + res);  }  }  interface MathOperation {  int operation(int a, int b);  } |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |