Over loading: We create multiple methods with the same name in the class provided these methods has different type of arguments or different number of argument

Example 1:

```
public class A {

	public void test(){// 0
		System.out.println("From test");
	}

	public void test(int i){//1
		System.out.println(i);
	}

	public static void main(String[] args) {
		A a1 = new A();
		a1.test();
		a1.test(100);
	}
}
```

Output:

From test

100

Can we create more than one main method in the same class ?

Example 2:

```
public class A {
```

```
	public static void main(String[] args) {//1
		System.out.println("From built in main method");
		A.main();
	}
	public static void main(){// 0
		System.out.println("From user defined method");
	}
}
```

Output:

From built in main method

From user defined method

Example 3:

```
public class A {
	public static void main(String[] args) {//1
		A a1 = new A();
		a1.emailSender();
		a1.emailSender("avb324");
	}
	public void emailSender(){//0
		System.out.println("Send marketing emailers");
	}
	public void emailSender(String transactionID){//1
		System.out.println("Sending transactional emailer");
	}

}
```

Output:

Send marketing emailers

Sending transactional emailer

Packages:

1. Packages in java are nothing but folders created to store your programs in organized manner

2. Packages resolves naming convention problems in java, that we can create multiple classes with the same

3. When you are using a class present in different package then importing would become mandatory

4. When you are accessing the class present in same package then importing it is not required

5. short for importing class is control + shift + o

Example 1:

```
package p1;

public class A {


}
```

Example 2:

```
package p3.p4.p5;


public class C {


}
```

Example 3:

```
package p1;
```

```
public class A {
	public int i = 10;
}
```

```
package p2;
import p1.A;
public class B {
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
	}

}
```

Outout:

10

Example 4:

```
package p1;

public class A {
	public int i = 10;
}
```

```
package p2;

public class B {
	public static void main(String[] args) {
		p1.A a1 = new p1.A();
		System.out.println(a1.i);
```

```
	}


}
```

Output:

10

Example 5:

```
package p1;


public class A {
	public int i = 10;
}
```

```
package p1;


public class C {
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
	}


}
```

Output:

10

Example 6:

```
package p1;
```

```
public class A {
	public int i = 10;
}
```

```
package p2;

import p1.A;

public class B extends A{
	public static void main(String[] args) {

	}

}
```

Example 7

```
package p1.p2.p3;

public class D {

}
```

```
package p1;

import p1.p2.p3.D;

public class A {

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

```
		D d1 = new D();
	}
}
```

Output:

Example 8:

```
package p1;

public class A {

	public static void main(String[] args) {

	}
}
```

```
package p1;

public class C {
	public static void main(String[] args) {

	}

}
```

```
package p2;
import p1.*;
public class B {
	public static void main(String[] args) {
		A a1 = new A();
```

```
		C c1 = new C();

	}

}
```

Example 9:

```
package p1;

public class A {

	public static void main(String[] args) {

	}
}
package p2;
import p1.A;
import p1.p2.p3.D;
public class B {
	public static void main(String[] args) {
		A a1 = new A();
		D d1 = new D();

	}

}
package p1.p2.p3;
```

```
public class D {


}
```

Access Specifier:

Example 1:

```
package p1;

public class A {

	private int i = 10;
	private void test(){
		System.out.println("From test");
	}
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}
}
```

Output:

10

From test

Example 2:

```
package p1;


public class A {


	private int i = 10;
	private void test(){
		System.out.println("From test");
	}


}
```

```
package p1;


public class B extends A{
	public static void main(String[] args) {
		B b1 = new B();
		System.out.println(b1.i);
		b1.test();
	}


}
```

Output: Error

Example 3:

```
package p1;


public class A {
```

```
	private int i = 10;
	private void test(){
		System.out.println("From test");
	}

}
```

```
package p1;

public class B{
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}

}
```

Output: Error

Example 4:

```
package p1;

public class A {

	private int i = 10;
	private void test(){
		System.out.println("From test");
	}
```

```
}
```

```
package p2;
import p1.A;
public class C extends A{
	public static void main(String[] args) {
		C c1 = new C();
		System.out.println(c1.i);
		c1.test();
	}
}
```

Output: Error

Example 5:

```
package p1;

public class A {

	private int i = 10;
	private void test(){
		System.out.println("From test");
	}

}
```

```
package p2;
```

```
import p1.A;

public class C{
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}
}
```

Output: Error

Examle 6:

```
package p1;


public class A {


	int i = 10;
	void test(){
		System.out.println("From test");
	}
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}


}
```

Example 7:

```
package p1;


public class A {


	int i = 10;

	void test(){

		System.out.println("From test");

	}




}
```

```
package p1;


public class B extends A{

	public static void main(String[] args) {

		B b1 = new B();

		System.out.println(b1.i);

		b1.test();

	}


}
```

Output:

10

From test

Example 8:

```
package p1;


public class A {


	int i = 10;
	void test(){
		System.out.println("From test");
	}



}
```

```
package p1;


public class B{
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}


}
```

Output:

10

From test

Example 9:

```
package p1;
```

```
public class A {

	int i = 10;
	void test(){
		System.out.println("From test");
	}



}
```

```
package p2;
import p1.A;
public class C extends A{
	public static void main(String[] args) {
		C c1 = new C();
		System.out.println(c1.i);
		c1.test();
	}
}
```

Output: Error

Example 10:

```
package p1;

public class A {

	int i = 10;
```

```
	void test(){
		System.out.println("From test");
	}


}
```

```
package p2;
import p1.A;
public class C {
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}
}
```

Output: Error

Note:

a. If you make your class member as private then those members can be accessed only in same class

b. If you make your class member as default then those members can be accessed only in same package

c. If you make your class member as protected then those members can be accessed in same package and different package only through inheritance

d. If you make your class member as public then those members can be accessed every where

Protected Access Specifier:

Example 1:

```
package p1;


public class A {


	protected int i = 10;

	protected void test(){

		System.out.println("From test");

	}

	public static void main(String[] args) {

		A a1 = new A();

		System.out.println(a1.i);

		a1.test();

	}


}
```

Outout:

10

From test

Example 2:

```
package p1;


public class A {


	protected int i = 10;

	protected void test(){

		System.out.println("From test");
```

```
	}

}
```

```
package p1;

public class B extends A{

	public static void main(String[] args) {
		B b1 = new B();
		System.out.println(b1.i);
		b1.test();
	}

}
```

Outout:

10

From test

Example 3:

```
package p1;

public class A {

	protected int i = 10;
	protected void test(){
		System.out.println("From test");
	}
```

```
}
```

```
package p1;


public class B{


	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}


}
```

Output:

10

From test

Example 4:

```
package p1;


public class A {


	protected int i = 10;
	protected void test(){
		System.out.println("From test");
	}
```

```
package p2;

import p1.A;


public class C extends A{
	public static void main(String[] args) {
		C c1 = new C();
		System.out.println(c1.i);
		c1.test();
	}
}
}
```

Output:

10

From test

public access specifier:

Example 1:

```
package p1;


public class A {

	public int i = 10;
	public void test(){
		System.out.println("From test");
	}
```

```
	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}

}
```

Output:

10

From test

Example 2:

```
package p1;

public class A {

	public int i = 10;
	public void test(){
		System.out.println("From test");
	}

}
```

```
package p1;

public class B extends A{
```

```
	public static void main(String[] args) {
		B b1 = new B();
		System.out.println(b1.i);
		b1.test();
	}

}
```

Output:

10

From test

Example 3:

```
package p1;

public class A {

	public int i = 10;
	public void test(){
		System.out.println("From test");
	}

}
```

```
package p1;

public class B{

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

```
        A a1 = new A();
        System.out.println(a1.i);
        a1.test();
    }

}
```

Output:

10

From test

Access Specifier a class supports:

1. public- A public class can be accessed in any packages
2. default- A default class can be accessed only in the same package

private and protected a class would not support

Example 1: for default class

```
package p1;

class A {// This class can be used only in same package

    public int i = 10;
    public void test(){
        System.out.println("From test");
    }
```

```
}

package p1;

public class B{

	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}

}

package p2;

import p1.A;//Error


public class C {
	public static void main(String[] args) {
		A a1 = new A();//Error
		System.out.println(a1.i);
		a1.test();
	}
}
```

Example 2: An example for public class

```
package p1;


public class A {// This class can be used only in same package


	public int i = 10;
	public void test(){
		System.out.println("From test");
	}


}
```

```
package p1;


public class B{


	public static void main(String[] args) {
		A a1 = new A();
		System.out.println(a1.i);
		a1.test();
	}


}
```

```
package p2;


import p1.A;
```

```
public class C {
	public static void main(String[] args) {
		A a1 = new A();//p1
		System.out.println(a1.i);
		a1.test();
	}
}
```

What access specifiers constructors would support ?

a. constructor can be priavte and for such contructors object should be created in same class

b. constructor can be default and for such contructors object should be created any where in same package

c. constructor can be protected and for such contructors object should be created any where in same package which is very similar to default constructors

d. constructor can be public and for such contructors object can be created any where in the program

Example 1:

```
package p1;

public class A {

  private	A(){
		System.out.println("From Constructor A");
	}
```

```
    public static void main(String[] args) {
                A a1 = new A();
        }

}
```

Output:

From Constructor A

Example 2:

```
package p1;

public class A {

    private     A(){
                System.out.println("From Constructor A");
        }


}
```

```
package p2;

import p1.A;

public class B {

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

```
                A a1 = new A();
        }



}
```

Output:

Error

Example 3:

```
package p1;


public class A {
   private      A(){
                System.out.println("From Constructor A");
        }
}
```

```
package p1;


public class C {
        public static void main(String[] args) {
                A a1 = new A();
        }
}
```

Output:

Error

Example 4:

```
package p1;


public class A {
   A(){
                System.out.println("From Constructor A");
        }
   public static void main(String[] args) {
                A a1 = new A();
        }
}
```

Output:

```
From Constructor A
```

Example 5:

```
package p1;


public class A {
   A(){
                System.out.println("From Constructor A");
        }


}
```

```
package p1;


public class C {
```

```
        public static void main(String[] args) {
                A a1 = new A();
        }
}
```

Output:

From Constructor A

Example 6:

```
package p1;

public class A {
   A(){
            System.out.println("From Constructor A");
        }

}
```

```
package p2;

import p1.A;

public class B {

        public static void main(String[] args) {
                A a1 = new A();
            }
```

```
}
```

Output:

Error

Example 7:

```
package p1;


public class A {
	protected A() {
		System.out.println("From Constructor A");
	}

	public static void main(String[] args) {
		A a1 = new A();

	}

}
```

Output:

From Constructor A

Example 8:

```
package p1;


public class A {
	protected A() {
		System.out.println("From Constructor A");
```

```
	}



}
package p1;


public class C {
	public static void main(String[] args) {
		A a1 = new A();
	}
}
```

Output:

From Constructor A

Example 9:

```
package p1;


public class A {
	protected A() {
		System.out.println("From Constructor A");
	}



}
package p2;
```

```
import p1.A;


public class B {


	public static void main(String[] args) {

			A a1 = new A();

		}




}
```

Output:

Error

Example 10:

```
package p1;

public class A {

	public A() {

		System.out.println("From Constructor A");

	}

	public static void main(String[] args) {

		A a1 = new A();

	}

}
```

Output:

From Constructor A

Example 11:

```
package p1;
public class A {
	public A() {
		System.out.println("From Constructor A");
	}

}
package p1;

public class C {
	public static void main(String[] args) {
		A a1 = new A();
	}
}
```

Output:

From Constructor A

Example 12:

```
package p1;
public class A {
	public A() {
		System.out.println("From Constructor A");
	}

}
package p2;
import p1.A;
```

```
public class B extends A{


	public static void main(String[] args) {

		A a1 = new A();

	}

}
```

Output:

From Constructor A

Polymorphism:

Interview Questions

note:

a. During overriding accessspecifiers need not be same

b. During overriding the scope of accessspecifier should not be reduced

Question 1:

```
package p1;

public class A {

	protected void test(){

		System.out.println(100);

	}


}

package p1;

public class B extends A{

	@Override

	void test(){
```

```
		System.out.println(500);
	}
	public static void main(String[] args) {
		B b1 = new B();
		b1.test();
	}

}
```

Output: Error

Question 2:

```
package p1;
public class A {
	void test(){
		System.out.println(100);
	}

}
```

```
package p1;
public class B extends A{
	@Override
	protected void test(){
		System.out.println(500);
	}
	public static void main(String[] args) {
		B b1 = new B();
		b1.test();
```

```
	}
}
```

Output:

500

Question 3:

```
package p1;
public class A {
	public void test(){
		System.out.println(100);
	}

}
```

```
package p1;
public class B extends A{
	@Override
	protected void test(){
		System.out.println(500);
	}
	public static void main(String[] args) {
		B b1 = new B();
		b1.test();
	}
}
```

Output:

Error

Super keyword:

a. It helps us to access members of parent class. super keyword can be used only when inheritance is happening

b. super keyword nnot be used inside static methods.

c. We cannot use super keyword in main method because main method is static

d. using super keyword we can call construtors of parent class. but ensure that to call parent class constructor you are using super keyword in child class constructor

e. super keyword cannot be second statement while calling parent class constructor from child class constructor.

ex: super();

Example 1:

```
package p1;

public class A {
	int i = 10;
	public void test(){
		System.out.println(100);
	}
}
```

```
package p1;

public class B extends A{

	public static void main(String[] args) {
		B b1 = new B();
		b1.x();
	}
```

```
	public void x(){
		System.out.println(super.i);
		super.test();
	}
}
```

Output:

10

100

Example 2:

```
package p1;
public class A {
	int i = 10;
	public void test(){
		System.out.println(100);
	}
}
```

```
package p1;
public class B extends A{

	public static void main(String[] args) {
		B b1 = new B();
		b1.x();
	}
	public static void x(){
		System.out.println(super.i);//Error
```

```
		super.test();//Error
	}
}
```

Output:

Error

Example 3:

```
package p1;
public class A {
	static int i = 10;
	public static void test(){
		System.out.println(100);
	}
}
```

```
package p1;
public class B extends A{

	public static void main(String[] args) {
		B b1 = new B();
		b1.x();
	}
	public void x(){
		System.out.println(super.i);
		super.test();
	}
}
```

Output:

10

100

Example 4:

```
package p1;

public class A {
	A(){
		System.out.println("From Constructor A");
	}
}
```

```
package p1;

public class B extends A{

	B(){
		super();
	}

	public static void main(String[] args) {
		B b1 = new B();

	}

}
```

Output:

From Constructor A

Example 5:

```
package p1;

public class A {
	A(int i){
		System.out.println(i);
	}
}
```

```
package p1;

public class B extends A{

	B(){
		super(500);
	}

	public static void main(String[] args) {
		B b1 = new B();

	}

}
```

Output:

500

Example 6:

```
package p1;

public class A {
```

```
	A(int i){
		System.out.println(i);
	}
}

package p1;
public class B extends A{

	B(){
		System.out.println("From constructor B");
		super(500);
	}

	public static void main(String[] args) {
		B b1 = new B();

	}

}
```

Output:

Error

Example 7:

```
package p1;
public class A {
	A(int i){
		System.out.println(i);
```

```
	}
}
package p1;
public class B extends A{

	B(){
		super(500);
		System.out.println("From constructor B");

	}

	public static void main(String[] args) {
		B b1 = new B();

	}

}
```

Output:

500

From constructor B

Interfaces in java:

Features of interfaces developed in JDK 1.7 version

a. Interfaces can include only abstract methods/incomplete methods in it

b. when a class implements interface then in means you are inheriting incomplete method of interface

into the class and ensure that you complete that method in a class or else you will get an error

c. abstract keyword: Usage of abstract keyword if done on a method then it is to define that the method is incomplete in an interface. But usage of abstract keyword in an interface is optional

d. If a method is made final then overriding of that method is not allowed

Example 1:

```
package interfaces_examples;

public interface A {//Contract MRF Tyres

	public void test();//abstract methods

}
```

```
package interfaces_examples;

public class B implements A{//Dhoni


}
```

Output: Error because test() method is not completed in class

Example 2:

```
package interfaces_examples;

public interface A {

	public void test();

}
```

```
package interfaces_examples;

public class B implements A {

	public void test() {

		System.out.println("From test");

	}

	public static void main(String[] args) {

		B b1 = new B();

		b1.test();
```

```
	}


}
```

Output:

From test

Example 3:

```
package interfaces_examples;

public interface A {
	public abstract void test();
}
```

```
package interfaces_examples;

public class B implements A {
	public void test() {
		System.out.println("From test");
	}
	public static void main(String[] args) {
		B b1 = new B();
		b1.test();
	}
}
```

Ouput:

From test

Example 4:

```
package interfaces_examples;
```

```
public interface A {
	public void test1();
	public void test2();
}
```

```
package interfaces_examples;
public class B implements A {

	public void test1() {
		System.out.println("From test1");
	}
	public void test2() {
		System.out.println("From test2");
	}
	public static void main(String[] args) {
		B b1 = new B();
		b1.test1();
		b1.test2();
	}
}
```

Output:

From test1

From test2

Note:

What is final keyword in java?

a. When a variable is made final then it means that the variable value cannot be changed or re-initialization of that variable is not allowed

b. When a variable is final then initializing that variable is mandatory

c. If a class is made final then inheriting that class is not possible (refer example no. 6)

d. What happens if an array is made final

Example 1:

```
package interfaces_examples;

public class B {

	public static void main(String[] args) {

		final int  i = 10;

		i = 100;

		System.out.println(i);

	}

}
```

Output: Error

Example 2:

```
package interfaces_examples;

public class B {

	public static void main(String[] args) {

		final int  i = 10;

		i = 10;

		System.out.println(i);

	}

}
```

Output: Error

Example 3:

```
package interfaces_examples;

public class B {

	final static int i = 10;

	public static void main(String[] args) {

		B.i = 100;

		System.out.println(B.i);

	}

}
```

Output: Error

Example 4:

```
package interfaces_examples;

public class B {

	final static int i ; //Error becausfinal variable is not initialized

	public static void main(String[] args) {

		System.out.println(B.i);

	}

}
```

Output:

Error

Example 5:

```
package interfaces_examples;

public class B {

	final static int i  = 100;
```

```
	public static void main(String[] args) {
		System.out.println(B.i);
	}
}
```

Output:

100

Example 6:

```
package interfaces_examples;

final public  class A {

}
```

```
package interfaces_examples;

public class B extends A{

}
```

Output:

Error

Example 7:

```
package interfaces_examples;
```

```
public  class A {
	final public void test(){
		System.out.println("From test");
	}
}
```

```
package interfaces_examples;


public class B extends A{


	public void test(){


	}
}
```

Output:

Error

Example 8:

```
public class B{
	public static void main(String[] args) {
	    final int[] intArray = new int[3];
		intArray = new int[4];
	}
}
```

Output:

Error, because the initial size of an array is 3, and because

it is final we cannot resize the array

Interfaces Concept continued:

a. Every variable created in an interface by default is final and static

b. For an interface object cannot be created, but interface reference variable can be created

c. Interfaces supports multiple inheritance in java

d. note:

1. when you are inheriting from class to class we use extends keyword

2. when you are inheriting from interface to interface we use extends keyword

3. when your inheriting from interface to class we use implements keyword

e. If an interface is empty, then such interfaces are called as marker interfaces

f. On a class we can use both extends and implements keyword together, but ensure that extends is used first and then implements

Example 1:

```
public interface A {
	int i;
}
```

Example 2:

```
package appinheritance;

public interface A {
	int i = 10;
}
```

```
package appinheritance;
```

```
public class B {
	public static void main(String[] args) {
		System.out.println(A.i);
	}
}
```

Output:

10

Example 3:

```
package appinheritance;

public interface A {
	int i = 10;
}
```

```
package appinheritance;

public class B {
	public static void main(String[] args) {
		A.i = 100;
		System.out.println(A.i);
	}
}
```

Output:

Error

Example 4:

```
package appinheritance;


public interface A {
	final static int  i = 10;
}


package appinheritance;


public class B {
	public static void main(String[] args) {
		System.out.println(A.i);
	}
}
```

Ouput:

10

Example 5:

```
package appinheritance;


public interface A {
	final static int  i = 10;
}


package appinheritance;


public class B {
```

```
	public static void main(String[] args) {
		A a1 = new A();
	}
}
```

Output:

Error

Example 6:

```
package appinheritance;

public interface A {
	final static int  i = 10;
}
```

```
package appinheritance;

public class B {
	static A a1 ;
	public static void main(String[] args) {
		System.out.println(a1);
	}
}
```

Output:

null

Example 7:

```
package appinheritance;
```

```
public interface A {
	public void test1();
}
```

```
package appinheritance;
public interface B {
	public void test2();
}
```

```
package appinheritance;
public class C implements A,B{

	public void test1() {
		System.out.println("From test1");
	}
	public void test2() {
		System.out.println("From test2");
	}
	public static void main(String[] args) {
		C c1 = new C();
		c1.test1();
		c1.test2();
	}

}
```

Output:

From test1

From test2

Example 8:

```
package appinheritance;

public interface A {

	public void test1();

}
```

```
package appinheritance;

public interface B extends A{//test1() test2()

	public void test2();

}
```

```
package appinheritance;

public class C implements B {


	public static void main(String[] args) {

			C c1 = new C();

			c1.test1();

			c1.test2();

	}


	public void test1() {

		System.out.println("From test1");

	}


	public void test2() {

		System.out.println("From test2");


	}
```

```
}
```

Output:

```
From test1
From test2
```

Example 9:

```
package appinheritance;
public interface A {
	public void test1();
}
package appinheritance;
public interface B {
	public void test2();
}
package appinheritance;
public interface C extends A,B{//test1() test2() test3()
	public void test3();
}
package appinheritance;

public class D implements C{
	public void test1(){
		System.out.println(100);
	}
	public void test2(){
		System.out.println(1000);
```

```
	}
	public void test3(){
		System.out.println(2000);
	}
	public static void main(String[] args) {
		D d1 = new D();
		d1.test1();
		d1.test2();
		d1.test3();
	}
}
```

Output:

100

1000

2000

Example 10:

```
package appinheritance;
public interface A {
	public void test1();
}
```

```
package appinheritance;

public class B {
	public void test2(){
		System.out.println(1000);
	}
```

```
}
```

```
package appinheritance;


public class C  extends B implements A {
	public static void main(String[] args) {
		C c1 = new C();
		c1.test1();
		c1.test2();
	}

	public void test1() {
		System.out.println(2000);
	}
}
```

Output:

100

1000

2000

Purpose of interfaces:

1. It help to design and code the program in a way that it is user friendly (add())

2. It hides implementation details (add())

Example 1:

```
package calculator;
```

```
public interface CalculatorsOperations {
	public void add();
}
//This abstract add method should be implements in 2 ways
//1. add numbers in Ordinary calc
//2. add numbers in Scientific Calc
package calculator;

public class OrdCalc implements CalculatorsOperations{

	public void add() {
		//Will consist login to add bases on OrdCal operation
		System.out.println("Ord Add done");
	}

}
package calculator;

public class ScientificCalc implements CalculatorsOperations{

	public void add() {
		//Shld consist the logic to add based on Sci. Calculations
		System.out.println("Sci Cal done");
	}

}
```

```
package calculator;

public class Operations {

	public static void main(String[] args) {
		OrdCalc o = new OrdCalc();
		o.add();

		ScientificCalc s = new ScientificCalc();
		s.add();

	}

}
```

User:

```
package calculator;

public class Operations {

	public static void main(String[] args) {
		OrdCalc o = new OrdCalc();
		o.add();

		ScientificCalc s = new ScientificCalc();
```

```
        s.add();

    }

}
```

Example 2:

```
public class C {

    public static void main(String[] args) {

        ArrayList a = new ArrayList();
        a.add(10);

        LinkedList l = new LinkedList();
        l.add(100);

    }

}
```

IN JDK 1.8 interfaces ?

1. Functional Interfaces

2. Lamdas expression

3. Misc. concepts added.

Functional Interfaces:

a. A functional interface can consist of only 1 abstract/incomplete method.

b. A functional interface can consist of main method in it

c. A functional interface can consist of any number of complete methods in it. These complete methods are created in a functional interface using "default" keyword

Example 1:

```
@FunctionalInterface

interface A {


public void test();




}



public class B {

public static void main(String[] args){



}


}
```

Example 2:

```
@FunctionalInterface

interface A {


public void test();


public void x();


}




public class B {


public static void main(String[] args){




}


}
```

Output:

Error

Example 3:

```
@FunctionalInterface

interface A {
```

```
public void test();

}

public class B {

public static void main(String[] args){

A a1 = ()->{

System.out.println(100);

};

a1.test();

}

}
```

Output:

100

Example 4:

```
@FunctionalInterface

interface A {
```

```
public void test();
}

public class B {

public static void main(String[] args){

A a1 = ()->System.out.println(100);



a1.test();

}

}
```

Output:

100

Example 5:

```
@FunctionalInterface
interface A {

public void test();
}

public class B {
```

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

A a1 = ()->{

System.out.println(100);

System.out.println(1000);

System.out.println(10000);

};



a1.test();

}

}
```

Example 6:

```
@FunctionalInterface
interface A {

public void test(int i);

}
```

```
public class B {

public static void main(String[] args){

A a1 = (int x)->{

System.out.println(x);

};



a1.test(500);

}

}
```

Output:

500

Example 7:

```
@FunctionalInterface

interface A {

public void test(int i);
```

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

System.out.println("From main");

}

}

public class B{

public static void main(String[] args){

A.main(null);

System.out.println("100");

}

}
```

Output:

From main

100

Example 8:

```
@FunctionalInterface
interface A {

public void test();


default void test1(){

System.out.println(5000);

}


default void test2(){


System.out.println(500);

}

}

public class B{

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

```
A a1 = ()->{

System.out.println(100);

};



a1.test();

a1.test1();

a1.test2();

}

}
```

Ouput

100

5000

500

abstract classes in java ?

a. you can develop both complete and incomplete methods

b. in abstract class to develop incomplete methods abstract keyword in mandatory

c. An abstract class object can never be created

d. An abstract class can consist of main method and we can run abstract class

e. When an inheritance is done from abstract class to abstract class we need not complete the method

f. When an inheritance is done from abstract class to complete class then we need complete the method

g. abstract classes donot support multiple inhritance

note:

1. interfaces are 100% abstract

2. abstract class can be 0% to 100% abstract

Example 1:

```
public  class A {//Error

        public abstract void test();//Error

        public void x(){

        }

}
```

Output: Error

Example 2:

```
public abstract  class A {

        public abstract void test();

        public void x(){

        }

}
```

Output:

Program will compile and run, but will print nothing

Example 3:

```
public abstract  class A {

        public  void test();

}
```

Output:

Error

Example 4:

```
public abstract  class A {

	public static void main(String[] args) {
		A a1 = new A();//Cannot create object
	}

}
```

Output: Error

Example 5:

```
public abstract  class A {
	static int i = 10;
	public static void main(String[] args) {
		System.out.println(A.i);
	}

}
```

Output:

10

Example 6:

```
public abstract  class A {

	public static void main(String[] args) {
		A.test();
	}
	public static void test(){
```

```
                System.out.println(100);
        }
}
```

Output:

100

Example 7:

```
public abstract  class A {

        public abstract void test();
}

public class B extends A{

        public void test() {
                System.out.println(500);
        }
        public static void main(String[] args) {
                B b1 = new B();
                b1.test();
        }

}
```

Output:

500

Example 8:

```
public abstract  class A {


	public abstract void test();


	public void xyz(){
		System.out.println(100);
	}
}
public class B extends A{


	public void test() {
		System.out.println(500);
	}
	public static void main(String[] args) {
		B b1 = new B();
		b1.test();
		b1.xyz();
	}


}
```

Output:

500

100

Example 9:

```
public abstract  class A {

        public abstract void test();

}
public abstract class B extends A{
        public abstract void x();

}
public class C extends B{

        public void x() {
         System.out.println("From x");
        }

        public void test() {
                System.out.println("From test");

        }
        public static void main(String[] args) {
                C c1 = new C();
                c1.x();
                c1.test();
        }
```

```
}
```

Output:

From x

From test

Example 10:

```
public abstract  class A {

        public abstract void test();

}
public abstract class B extends A{
        public abstract void x();

}
public class C extends A,B{//Error


}
```

Output:

Error

Example 11:

```
public interface A {
        public void test1();
```

```
}
public abstract class B implements A{
	public abstract void test2();
}
public class C extends B{

	public static void main(String[] args) {
		C c1= new C();
		c1.test1();
		c1.test2();
	}

	public void test1() {
		System.out.println(100);
	}

	public void test2() {
		System.out.println(500);
	}

}
```

Output:

100

500

Example 12:

```
public interface A {
	public void test1();

}

public abstract class B {
	public abstract void test2();
}
public class C extends B implements A{

	public static void main(String[] args) {
		C c1= new C();
		c1.test1();
		c1.test2();
	}

	public void test1() {
		System.out.println(100);
	}

	public void test2() {
		System.out.println(500);
	}

}
```

Output:

100

500

Example 13:

```
public interface A {
	public void test1();

}

public abstract class B {
	public abstract void test2();
}
public interface D {
	public void test3();
}


public class C extends B implements A,D{

	public static void main(String[] args) {
		C c1= new C();
		c1.test1();
		c1.test2();
		c1.test3();
	}

	public void test1() {
```

```
		System.out.println(100);
	}

	public void test2() {
		System.out.println(500);
	}
	public void test3() {
		System.out.println(5000);
	}
}
```

Output:

100

500

5000

Exception in java and Exception handling

Exceptions are unexpected events that occurs in your program because of bad user input given. Exceptions will hault your program abruptly and hence the software would become unresponsive and no further line of code will execute

Example 1:

```
public class A {

	public static void main(String[] args) {
		int i = 10;
```

```
		int j = 0;


		int k = i / j;


		System.out.println("welcome");
		System.out.println("hello");
		System.out.println("world");
	}
}
```

Exception Handling:

To handle exceptions in java we use try catch block. If any line od code trhrows exception in try block then try block will automatically created an exception object and that objects referemce try block will give it to catch block. Now catch block will suppress the exception and will print the reason for excetion. After the exception is handled furtrher code would execute.

Example 2:

```
public class A {


	public static void main(String[] args) {
		try {
		int i = 10;
		int j = 0;


		int k = i / j;
		System.out.println(100);
		}catch(Exception e) {
			System.out.println(e);
```

```
                }


                System.out.println("welcome");
                System.out.println("hello");
                System.out.println("world");
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
100
welcome
hello
world
```

Example 3:

```
public class A {

        public static void main(String[] args) {
                try {
                int i = 10;
                int j = 0;
                int k = i / j;
                }catch(Exception e) {
                        e.printStackTrace();
                }
```

```
                System.out.println("welcome");
                System.out.println("hello");
                System.out.println("world");
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
        at a.A.main(A.java:9)
welcome
hello
world
```