

Module III

Inheritance basics, method overriding, abstract classes, interface. Defining and importing packages. Exception handling fundamentals, multiple catch and nested try statements.

Inheritance

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

The class which inherits the properties of other class is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

```
class Animal
{
    void eat()
    {
        System.out.println("eating...");
    }
}

class Dog extends Animal
{
    void bark()
    {
        System.out.println("barking...");
    }
}
```

```

class Inheritance
{
    public static void main(String args[])
    {
        Dog d=new Dog();
        d.bark();
        d.eat();
    }
}

```

Example

```

class A
{
    int i,j;
    void display()
    {
        System.out.print(i+" "+j);
    }
}

class B extends A
{
    int k;
    void show()
    {
        System.out.print(" "+k);
    }
    void sum()
    {
        System.out.print("\nSUM=");
        System.out.print(i+j+k);
    }
}

class Inheritance
{

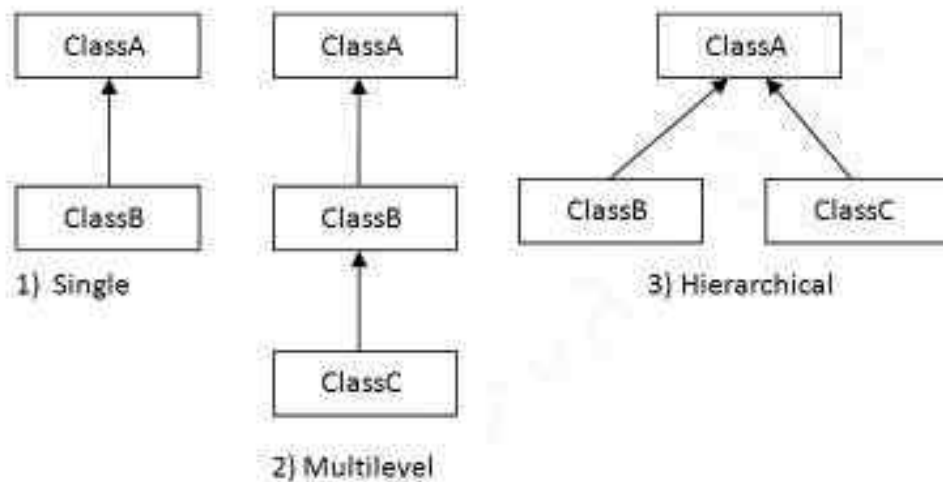
```

```

public static void main(String args[])
{
    A a1=new A();
    B b1=new B();
    a1.i=10;
    a1.j=20;
    System.out.println("\n using the object of the super class");
    a1.display();
    b1.i=100;
    b1.j=200;
    b1.k=300;
    System.out.println("\n using the object of the sub class");
    b1.display();
    b1.show();
    b1.sum(); } }

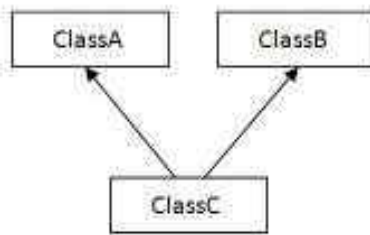
```

Types of Inheritance

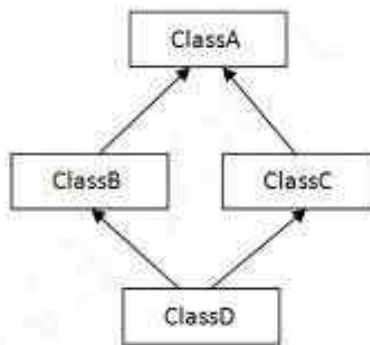


Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance.



4) Multiple



5) Hybrid

Single Inheritance Example

```

class Animal
{
void eat()
{System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance
{
public static void main(String args[])
{
Dog d=new Dog();
d.bark();
d.eat();
}}
  
```

Output:

barking...
eating...

Multilevel Inheritance Example

```

class Animal
{
  
```

```
void eat(){System.out.println("eating...");}
}
class Dog extends Animal
{
void bark(){System.out.println("barking...");}
}

class BabyDog extends Dog
{
void weep(){System.out.println("weeping...");}
}
class TestInheritance
{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

weeping...
barking...
eating...

Hierarchical Inheritance Example

```
class Animal
{
void eat(){System.out.println("eating...");}
}
}
class Dog extends Animal
{
```

```

void bark(){System.out.println("barking...");}
}
class Cat extends Animal
{
void meow(){System.out.println("meowing...");}
}
class TestInheritance
{
public static void main(String args[])
{
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}

```

Output:

```

meowing...
eating...

```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

super keyword in java

The super keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

super can be used to refer immediate parent class instance variable.

super can be used to invoke immediate parent class method.

super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal
{
    String color="white";
}
class Dog extends Animal
{
    String color="black";
    void printColor()
    {
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}
class TestSuper
{
    public static void main(String args[])
    {
        Dog d=new Dog();
        d.printColor();
    }
}
```

Output:

```
black
white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal
{
void eat(){System.out.println("eating...");
}
}
class Dog extends Animal
{
void eat(){System.out.println("eating bread...");
}
void bark()
{System.out.println("barking...");
}
void work()
{
super.eat();
bark();
}
}
class TestSuper
{
public static void main(String args[])
{
Dog d=new Dog();
d.work();
}}
```

Output:

```
eating...
barking...
```


In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal
{
    Animal()
    {System.out.println("animal is created");
    }
}

class Dog extends Animal
{
    Dog()
    {
        super();
        System.out.println("dog is created");
    }
}

class TestSuper
{
    public static void main(String args[])
    {
        Dog d=new Dog();
    }
}
```

Output:

animal is created

dog is created

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person
```

```

{
int id;
String name;
Person(int i,String n)
{
id=i;
name=n;
}
}
class Emp extends Person
{
float salary;
Emp(int id,String name,float salary)
{
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);
}
}
class TestSuper5
{
public static void main(String[] args)
{
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}

```

Output:

1 ankit 45000

Method Overriding

- when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override the method in the superclass*.

- *When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.*

Example

```

class A
{
    void callme()
    {
        System.out.println("This is a method in class A");
    }
}

class B extends A
{
    void callme()
    {
        System.out.println("This is a method in class B");
    }
}

class C extends A
{
    void callme()
    {
        System.out.println("This is a method in class C");
    }
}

class Polymorphism
{
    public static void main(String args[])
    {
        C c1=new C();
        c1.callme();
    }
}

```

Dynamic Method dispatch

- A call to overridden method is resolved at run time.
- Run time polymorphism

Working

- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon *the type of the object* being referred to at the time the call occurs.

Example

```
class A
{
    void callme()
    {
        System.out.println("This is a method in class A");
    }
}

class B extends A
{
    void callme()
    {
        System.out.println("This is a method in class B");
    }
}

class C extends A
{
    void callme()
    {
        System.out.println("This is a method in class C");
    }
}

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

```

    B b1=new B();
    C c1=new C();
    A temp;
    temp=a1;
    temp.callme();
    temp=b1;
    temp.callme();
    temp=c1;
    temp.callme();
}
}

```

Abstract classes

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

There are two ways to achieve abstraction in java

Abstract class

Interface

Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

```

abstract class A
{
}

```

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

abstract void printStatus();//no body and abstract

Example program

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.

A factory method is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

abstract class Shape

```
{
abstract void draw();
}

        class Rectangle extends Shape
        {
            void draw()
            {
                System.out.println("drawing rectangle");
            }
        }

        class Circle extends Shape
        {
            void draw()
            {System.out.println("drawing circle");}
        }

class TestAbstraction
{
    public static void main(String args[])
    {
        Shape s=new Circle();
        s.draw();
    } }
```

Output

drawing circle

An abstract class can have **data member, abstract method, method body, constructor and even main() method.**

```
abstract class Bike
{
    Bike()
    {System.out.println("bike is created");}
```

abstract void run();

```
    void changeGear()
    {System.out.println("gear changed");}
}
```

```
class Honda extends Bike
{
    void run()
    {System.out.println("running safely..");}
}

class TestAbstraction
{
    public static void main(String args[])
    {
        Bike obj = new Bike();
        obj.run();
        obj.changeGear();
    }
}
```

Output

```
bike is created
running safely
gear changed
```

final keyword in Java

Three uses : -

- Equivalent of constant
- To prevent method overriding
- To prevent inheritance

To prevent method overriding

Methods declared as final **cannot be overridden**

Example

```
class A
{
final void callme()
{
    System.out.println("This is a method in class A");
}
}
class B extends A
{
void callme() // Error!
{
    System.out.println("This is a method in class B");
}
}
```

To prevent inheritance of a class

```
final class A
{
    .....
}
class B extends A // illegal
{
    .....
```



```
}
```

A class with final keyword will consider all methods declared as final. Final and abstract cannot be used simultaneously with a class

Packages

- *Packages* are containers for classes
- *Packages* manage namespaces
- Namespaces allow to group entities like classes, objects and functions under a name
- Classes inside a package cannot be accessed by code outside that package

package pkg;

- First statement in java source code
- If no package is specified, the class name is put into default package with out any name
- Class files must be stored in the same directory of the package

Hierarchy of packages

```
package pkg1.pkg2.pkg3.....;
```

Example

package MyPack;

```
class Balance
```

```
{
```

```
String name;
```

```
int bal;
```

```
Balance(String n, int b)
```

```
{
```

```
name=n;
```

```
bal=b;
```

```
}
```

```
void show()
```

```
{
```

```
System.out.println("-->");
```

```
System.out.println(name+": $" + bal);
```

```

}
}

class PackageTest
{
    public static void main(String args[])
    {
        Balance b[]=new Balance[3];
        b[0]=new Balance("ABC",123);
        b[1]=new Balance("EFG",157);
        b[2]=new Balance("IJK",10);
        for(int i=0;i<3;i++)
            b[i].show();
    }
}

```

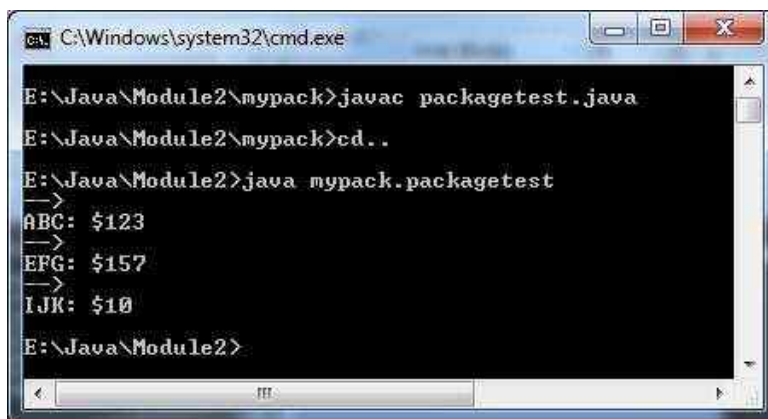
Save the file in a directory MyPack

Compile the file `javac PackageTest.java`

Go back to the previous directory

Run the program using

`java MyPack.PackageTest`



```

C:\Windows\system32\cmd.exe

E:\Java\Module2\mypack>javac packagetest.java
E:\Java\Module2\mypack>cd..
E:\Java\Module2>java mypack.packagetest
->
ABC: $123
->
EFG: $157
->
IJK: $10
E:\Java\Module2>

```

Importing packages

- `import pkg1.pkg2.classname;`
`import pkg1.*;`
- All standard classes in java are under the package java.
- If a programmer need these classes he can import those

Eg:- `import java.io.*;`

Four access controls

- **Private** – only within class
- **Default** – it is visible to subclasses as well as to other classes in the *same package*.
- **Protected** – If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.
- **Public** – can be accessed from anywhere

Interfaces

Syntactically similar to classes but they lack instance variables and their methods are declared without any body

- all method inside interface must be abstract

- *Using interface, you can specify, What a class must do, but not how it does it.*

One class can implement any number of interfaces

access interface-name

```
{
    return-type method1(parameter list);
    return-type method2(parameter list);
    type final-variable1=value;
    type final-variable2=value;
    //...
    return-type methodN(parameter list);
    type final-variableN=value;
}
```

access – public or default

- Access is public or not used(default)
- Name is a valid identifier
- Methods have no bodies.
- Variables are implicitly **final** and **static**, and must be initialized.
- All methods and variables are implicitly public if interface is public

public interface Callback

```
{
    void callback(int param);
}
```

```
}
```

Implementing interface

```
access class classname [implements interface1,[interface2],....]
```

```
{
    //class-body
}
```

Note:-

- More than one interfaces can be used.
- Type signature of method in class must match with the method declaration in interface

Example

interface Callback

```
{
    void callback(int param);
}
```

class Client implements Callback

```
{
    public void callback(int param) // interface method must be public
    {
        System.out.println("Callback called with "+param);
    }
    void nonIfaceMeth()
    {
        System.out.println("Classes that implement interfaces may also define other members");
    }
}
```

```
class TestIface
```

```
{
    public static void main(String args[])
    {
        Callback c = new Client();
        c.callback(42);
    }
}
```

Variables in Interfaces

```
interface SharedConstants
{
    int final NO = 0;
    int final YES = 1;
}
```

Interfaces can be extended

- One interface can inherit other

```
interface a
{
    void meth1();
    void meth2();
}

// b now include meth1(),meth2() – it adds meth3()
    interface b extends a
    {
        void meth3();
    }
    class Example implements b
    {
        public void meth1()
        {
            System.out.println("method1");
        }
        public void meth2()
        {
            System.out.println("method2");
        }
        public void meth3()
        {
            System.out.println("method3");
        }
    }
```

```

        }
    }
class Inheritance
{
    public static void main(String args[])
    {
        Example e=new Example();
        e.meth1();
        e.meth2();
        e.meth3();
    }
}

```

Exception Handling

- An *exception* is an abnormal condition that arises in a code sequence at **run time**.
- Exception is an object that describes exceptional condition that has occurred in a piece of code
- Exception object is created and thrown from the method that generate error
- It is caught and handled automatically or manually

Five key words

- try
- catch
- throw
- throws
- finally

Uncaught Exceptions

- When the Java run-time system detects the attempt to divide by zero
 1. It constructs a new exception object and then *throws* this exception.
 2. This causes the execution of **Exc0** to stop

The exception is caught by the default handler provided by Java run-time

```

class Exc0 {
    public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
    }
}

```

```
}
```

Output

```
java.lang.ArithmeticException: / by zero
```

```
at Exc0.main(Exc0.java:4)
```

General form of exception handling block

```
try
{
    block of code to monitor for errors
}
catch(Exceptiontype1 objname)
{
    Handler for exception type1
}
catch(Exceptiontype2 objname)
{
    Handler for exception type2
}
.....
finally
{
    // block of code to be executed before try block ends
}
```

Exception Types

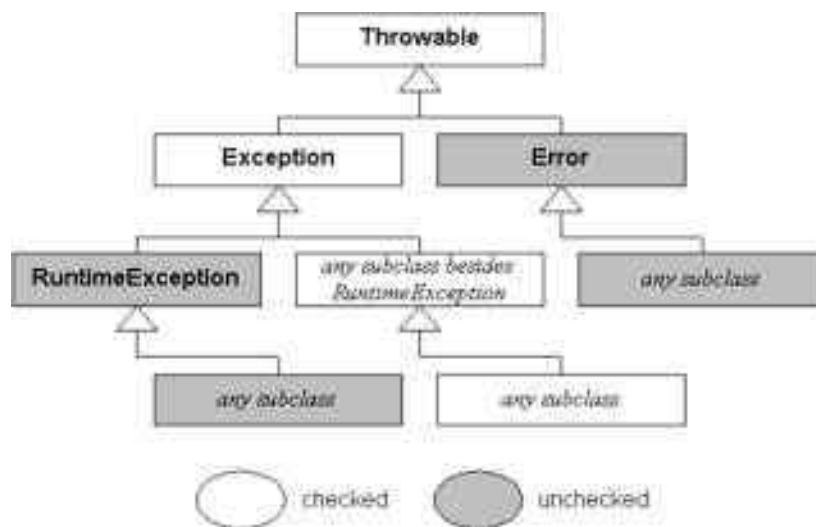
All exceptions are subclasses of the built in class **Throwable**

1. Exception

- This class is used for exceptional conditions that user programs should catch.
- There is an important subclass of **Exception**, called **RuntimeException**.(division by zero, invalid array indexing etc)

2. Error

- Defines exceptions that are not caught by normal circumstances.
- Those errors are managed by Java run time. Eg : stack overflow



unchecked exceptions - compiler does not check to see if a method handles or throws these exceptions.

checked exceptions - The method can generate one of these exceptions and does not handle it itself.(use *throws*)

Using try catch

```

class Try1
{
    public static void main(String args[])
    {
        int a,d;

        try
        {
            d=0;
            a=42/d;
            System.out.println("a=" +a);
        }
        catch(ArithmeticException e )
        {
            System.out.println(e);
        }

        System.out.println("This is after try catch");
    }
}
  
```



```
}
```



```
C:\Windows\system32\cmd.exe
E:\Java\Module2\Exception>java try1
java.lang.ArithmeticException: / by zero
This is after try catch
E:\Java\Module2\Exception>
```

Multiple Catch

- When more than one exception could be raised from a single piece of code
- Specify more than one catch block
- Each catch statement will be inspected in order
 - The first matching catch is selected and rest is bypassed

```
class Try1
```

```
{
    public static void main(String args[])
    {
        try
        {
            int a=args.length;
            System.out.println("a=" +a);
            int b=42/a;
            int c[]={1};
            c[42]=99;
        }
        catch(ArithmeticException ae )
        {
            System.out.println(ae);
        }
        catch(ArrayIndexOutOfBoundsException arr )
        {
            System.out.println(arr);
        }
    }
}
```

```

        System.out.println("This is after try catch");
    }
}

```



```

C:\Windows\system32\cmd.exe

E:\Java\Module2\Exception>java try1
a=0
java.lang.ArithmeticException: / by zero
This is after try catch
E:\Java\Module2\Exception>

```



```

C:\Windows\system32\cmd.exe

E:\Java\Module2\Exception>java try1 testarguments
a=1
java.lang.ArrayIndexOutOfBoundsException: 42
This is after try catch
E:\Java\Module2\Exception>

```

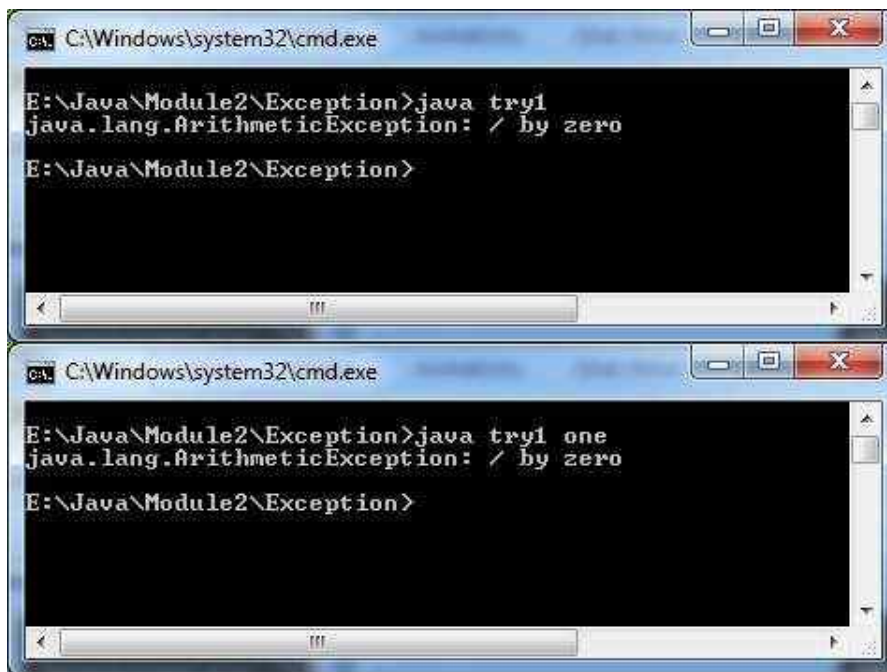
Nested try statements

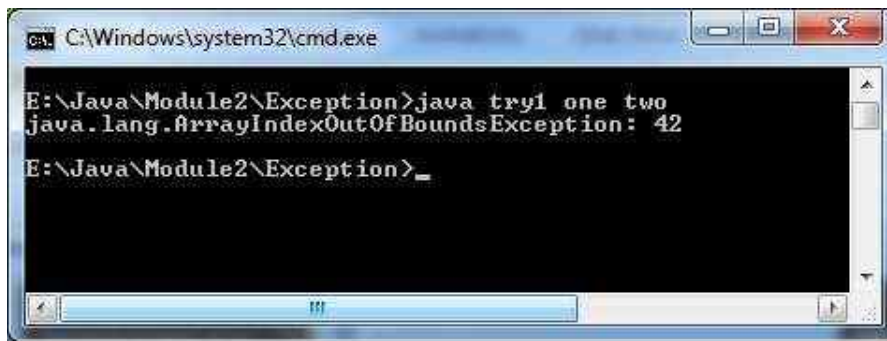
```

class Try1
{
    public static void main(String args[])
    {
        try
        {
            int a=args.length;
            int b=42/a;
            try
            {
                if(a==1)
                    a=a/(a-a);
                if(a==2)
                {
                    int c[]={1};

```

```
        c[42]=99;
    }
}
catch(ArrayIndexOutOfBoundsException arr)
{
    System.out.println(arr);
}
}
catch(ArithmeticException ae )
{
    System.out.println(ae);
}
}
```





```

C:\Windows\system32\cmd.exe
E:\Java\Module2\Exception>java try1 one two
java.lang.ArrayIndexOutOfBoundsException: 42
E:\Java\Module2\Exception>_

```

Throw

You have only been catching exceptions that are thrown by the Java run time system. However, it is possible for our pgm to throw an exception explicitly.

Syntax: throw ThrowableInstance;

Working

- Flow of execution stops at throw statement.
- Next, search for a immediate matching try catch statement.
- If no match found default handler will work

There are two ways to obtain a throwable object.

- Using a parameter into a **catch clause**
- Creating one with **new operator**

Example

```

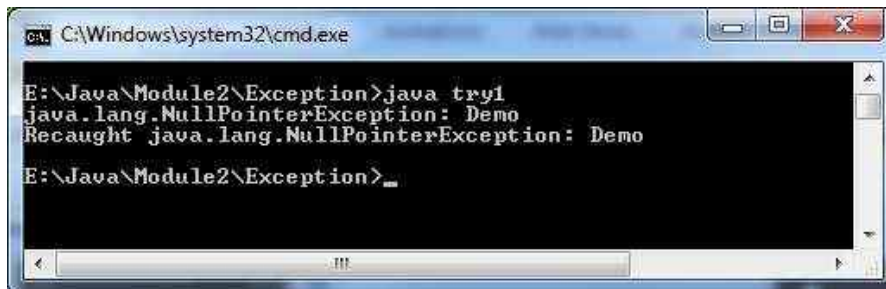
class try1
{
    static void throwdemo()
    {
        try
        {
            throw new NullPointerException("Demo");
        }
        catch(NullPointerException n)
        {
            System.out.println(n);
            throw n;
        }
    }
}

```

```

    }
}
public static void main(String args[])
{
    try
    {
        throwdemo();
    }
    catch(NullPointerException ne)
    {
        System.out.println("Recaught "+ne);
    }
}
}

```



throws

- If a method is capable of generating an exception that it cannot handle
- Include a throws clause in the method's declaration
- Can notify the caller about possible exceptions

- Syntax:-

type method-name(parameter list) throws exception list

{

// body of the method

}

- Exception list is a comma separated list

Note:-

- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

```
class ThrowsDemo
{
    static void throwOne() throws IllegalAccessException
    {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }

    public static void main(String args[]) {
        try
        {
            throwOne();
        }
        catch (IllegalAccessException e)
        {
            System.out.println("Caught " + e);
        }
    }
}
```

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws

	IOException,SQLException.
--	---------------------------

Java throw example

```
void m(){
throw new ArithmeticException("sorry");
}
```

Java throws example

```
void m()throws ArithmeticException{
//method code
}
```

Java throw and throws example

```
void m()throws ArithmeticException{
throw new ArithmeticException("sorry");
}
```

finally

Finally is a block of statements that will be executed immediately after try catch block

Example

```
class try1
{
static void a()
{
try
{
System.out.println("Inside A");
}
finally
{
System.out.println("Inside A's Finally");
}
}
static void b()
{
```

```
try
{
    System.out.println("Inside B");
}
finally
{
    System.out.println("Inside B's Finally");
    return;
}
}
public static void main(String args[])
{
    a();
    b();

}
}
```



```
C:\Windows\system32\cmd.exe
E:\Java\Module2\Exception>java try1
Inside A
Inside A's Finally
Inside B
Inside B's Finally
E:\Java\Module2\Exception>_
```

Module IV