Unit 3
Package, Multithreading, Exception Handling

Packages

- A package as the name suggests is a pack(group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces.
- A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:
  - ✔ Built-in Packages (packages from the Java API)
  - ✔ User-defined Packages (create your own packages)

Packages

- Packages-**Built-in Packages (packages from the Java API) :**

✔ The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

✔ The library contains components for managing input, database programming, and much much more.

✔ The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

✔ To use a class or a package from the library, you need to use the **import** keyword:

Packages

Syntax

import *package.name.Class*; // Import a single class
import *package.name*.*; // Import the whole package

Example:

- **Import a Class**
  - √ If you find a class you want to use, for example, the Scanner class, **which is used to get user input**, write the following code:

import java.util.Scanner;

  - ✔ In the example above, java.util is a package, while Scanner is a class of the java.util package.

Packages

- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read a complete line:

Example

Using the Scanner class to get user input:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Packages

- **Import a Package**
  - ✔ There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other utility classes.
  - ✔ To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the classes in the java.util package:

Syntax:

```
import java.util.*;
```

Example:

```
import java.util.*; // import the java.util package

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    String userName;

    // Enter username and press Enter
    System.out.println("Enter username");
    userName = myObj.nextLine();

    System.out.println("Username is: " + userName);
  }
}
```

Packages-User-defined Packages

- To create your own package, you need to understand that Java use a file system directory to store them. Just like folders on your computer:

- To create a package, use the package keyword

Syntax:

package package_name;

Packages-User-defined Packages

Example:

MyPackageClass.java // file name

```
package mypack;
class MyPackageClass
{
public static void main(String[] args)
Sop("This is my package!");
}
}
```

To compile the java file:

C:\Users*Your Name*>javac MyPackageClass.java

To compile the package:

C:\Users*Your Name*>javac -d . MyPackageClass.java

✔This forces the compiler to create the "mypack" package.

✔The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

✔**Note:** The package name should be written in lower case to avoid conflict with class names.

Packages-User-defined Packages

- When we compiled the package in the example above, a new folder was created, called "mypack".
- To run the **MyPackageClass.java** file, write the following:

C:\Users*Your Name*>java
mypack.MyPackageClass

- The output will be:

This is my package!

Advantages of using a package in Java

- These are the reasons why you should use packages in Java:

✔ **Reusability**: While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.

✔ **Better Organization**: Again, in large java projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.

✔ **Name Conflicts**: We can define two classes with the same name in different packages so to avoid name collision, we can use packages

How to access package from another package?

- There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

How to access package from another package?

1) Using packagename

- ✔ If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- ✔ The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg()
{
System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[])
{
   A obj = new A();
   obj.msg();
  }
}
```

How to access package from another package?

2) Using packagename.classname

✔ If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg()
{
System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[])
{
   A obj = new A();
   obj.msg();
  }
}
```

How to access package from another package?

3) Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A
{
public void msg()
{System.out.println("Hello");}
}
```

Output: Hello

```
//save by B.java
package mypack;
class B
{
  public static void main(String args[])
{
 pack.A obj = new pack.A();//using fully q
ualified name
   obj.msg();
  }
}
```

Access Modifier

- There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
  - ✔ Private
  - ✔ Default
  - ✔ Protected
  - ✔ public
- There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

Access Modifier

1) private access modifier

- The private access modifier is accessible only within class.
- **Example of private access modifier**
  - ✔ In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Access Modifier

Role of Private Constructor

- If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A
{
        private A()
        {}//private constructor
        void msg()
        {
        System.out.println("Hello java");}
        }
        public class Simple
        {
        public static void main(String args[]){
        A obj=new A();//Compile Time Error
         }
 }
```

Access Modifier

2) default access modifier

* If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.Example of default access modifier
* In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A
{
  void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[])
{
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

Access Modifer

3) protected access modifier

* The **protected access modifier** is accessible within package and outside the package but through inheritance only.
* The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
* Example of protected access modifier
* In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A
{
protected void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B extends A{
  public static void main(String args[])
{
   B obj = new B();
   obj.msg();
  }
}
```

Access Modifier

4) public access modifier

* The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A
{
public void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[])
{
   A obj = new A();
   obj.msg();
  }
}
```

Output:Hello

Understanding all java access modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- In this page, we will learn about Java exceptions, its type and the difference between checked and unchecked exceptions.
- **What is Exception in Java**
- **Dictionary Meaning:** Exception is an abnormal condition.
- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- **What is Exception Handling**
- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Exception Handling in Java

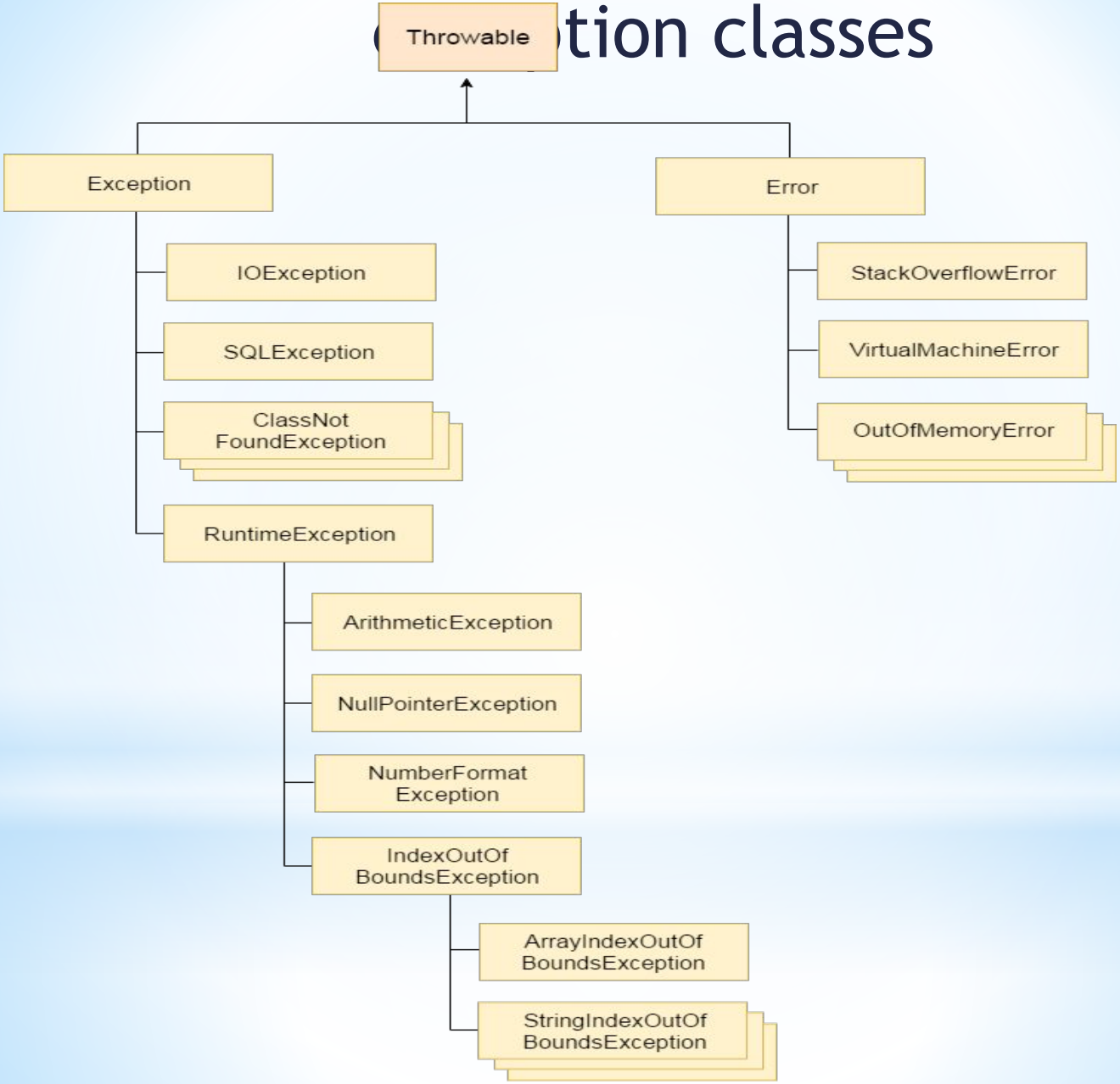
Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of java exception classes



Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:
  - ✔ Checked Exception
  - ✔ Unchecked Exception
  - ✔ Error
- Difference between Checked and Unchecked Exceptions

1) Checked Exception

- ✔ The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

- ✔ The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

- ✔ Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError,

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

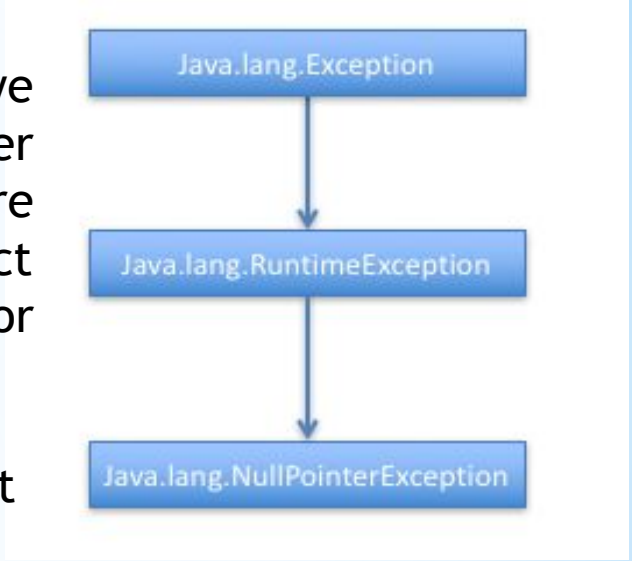
```
public class JavaExceptionExample
{
  public static void main(String args[])
{
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e)
{System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...")
}
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero rest of the code...

NullPointer exception

NullPointer Exception is a runtime exception, so we don't need to catch it in program. NullPointer Exception is raised in an application when we are trying to do some operation on **null** where an object is required. Some of the common reasons for NullPointer Exception in java programs are;



1. Invoking a method on an object instance but at runtime the object is null.
2. Accessing variables of an object instance that is null at runtime.
3. Throwing null in the program
4. Accessing index or modifying value of an index of an array that is null
5. Checking length of an array that is null at runtime.

NullPointer Exception Example

1. **NullPointerException when calling instance method**

```
public class Temp
{
	public static void main(String[] args)
	{
		Temp t = initT();
		t.foo("Hi");
	}
	private static Temp initT()
	{
		return null;
	}
	public void foo(String s)
	{
System.out.println(s.toLowerCase());
	 }
 }
```

OUTPUT :

Exception in thread "main" java.lang.NullPointerException at Temp.main(Temp.java:7)

2. **Java NullPointer Exception while accessing/modifying field of null object**

```
public class Temp
 {
       public int x = 10;
       public static void main(String[] args)
      {
            Temp t = initT(); int i = t.x;
      }
      private static Temp initT()
      {
            return null;
      }
 }
```

OUTPUT :

```
Exception in thread "main" java.lang.NullPointerException at
Temp.main(Temp.java:9)
```

3. Java NullPointerException when null is passed in method argument

```
public class Temp
{
	public static void main(String[] args)
	{
		foo(null);
	}
	 public static void foo(String s)
	{
		System.out.println(s.toLowerCase());
	}
 }
```

4. **java.lang.NullPointerException when null is thrown**

```
public class Temp
{
      public static void main(String[] args)
      {
       throw null;
       }
 }
```

OUTPUT :

```
Exception in thread "main" java.lang.NullPointerException at
Temp.main(Temp.java:5)
```

5. **java.lang.NullPointerException when getting length of null array**

```
public class Temp
{
	public static void main(String[] args)
	{
		int[] data = null; int len = data.length;
	}
}
```

OUTPUT :

Exception in thread "main" java.lang.NullPointerException at Temp.main(Temp.java:7)

6. NullPointerException when accessing index value of null array

```
public class Temp {

	public static void main(String[] args) {

		int[] data = null;

		int len = data[2];
	}

}
```

OUTPUT :

```
Exception in thread "main"
java.lang.NullPointerException
	at Temp.main(Temp.java:7)
```

7. java.lang.NullPointerException when synchronized on null object

```
public class Temp
{

	public static String mutex = null;

	public static void main(String[] args)
	{

		synchronized(mutex)
		{
			System.out.println("synchronized block");
		}

	}

}
```

NumberFormatException

A Java NumberFormatException usually occurs when you try to do something like convert a String to a numeric value, like an int, float, double, long, etc

NumberFormatException example

```
package com.devdaily.javasamples;
public class ConvertStringToNumber
{
        public static void main(String[] args)
        {
                try
                {
                // intentional error String s = "FOOBAR";
                int i = Integer.parseInt(s);

                // this line of code will never be reached
                System.out.println("int value = " + i);
                }

                catch(NumberFormatException nfe)
                {
                        nfe.printStackTrace();
                }
        }
}
```

Array IndexOutOfBound Exception

- ArrayIndexOutOfBoundsException is thrown to indicate that we are trying to access array element with an illegal index.
- This exception is thrown when the index is either negative or greater than or equal to the size of the array.

ArrayIndexOutOfBounds Exception Example

```
package com.journaldev.exceptions;
import java.util.Scanner;
public class ArrayIndexOutOfBoundsExceptionExample
{
	public static void main(String[] args)
	{
	Scanner sc = new Scanner(System.in);
	System.out.println("Enter size of int array:");
	int size = sc.nextInt();
	int[] intArray = new int[size];
	for (int i = 0; i < size; i++)
	{
	System.out.println("Please enter int value at index " + i + ":");
	intArray[i] = sc.nextInt();
	}
	System.out.println("Enter array index to get the value:");
	int index = sc.nextInt();
	sc.close();
	System.out.println("Value at " + index + " = " + intArray[index]);
	}
}
```

OUTPUT :

```
Enter size of int array:
3
Please enter int value at index 0:
1
Please enter int value at index 1:
2
Please enter int value at index 2:
3
Enter array index to get the value:
4
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
      at
com.journaldev.exceptions.ArrayIndexOutOfBoundsExceptionExampl
e.main(ArrayIndexOutOfBoundsExceptionExample.java:23)
```

Java try-catch block

- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}
catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}
finally{}
```

Java catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

- The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

```
public class TryCatchExample1
{
    public static void main(String[] args)
     {
       int data=50/0; //may throw exception
       System.out.println("rest of the code");
      }

}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Solution by exception handling

```
public class TryCatchExample2
{
    public static void main(String[] args)
    {
        try
        {
            int data=50/0; //may throw exception
        }
            //handling the exception
            catch(ArithmeticException e)
            {
                System.out.println(e);
            }
        System.out.println("rest of the code");
    }
}
```

Output:

java.lang.ArithmeticException: / by zero rest of the code

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

```
public class TryCatchExample3
{
    public static void main(String[] args)
    {
        try
        {
            int data=50/0; //may throw exception
                // if exception occurs,
            //the remaining statement will not exceute

            System.out.println("rest of the code");
        }
         // handling the exception
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}
```

Output:

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

catch multiple exceptions

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
- **Points to remember**
- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example 1

```
public class MultipleCatchBlock1
{
   public static void main(String[] args)
   {
        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
         catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
         catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
         catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
          System.out.println("rest of the code");
    }
}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

```
public class MultipleCatchBlock2 {
   public static void main(String[] args) {
        try{
            int a[]=new int[5];

            System.out.println(a[10]);
           }
           catch(ArithmeticException e)
              {
               System.out.println("Arithmetic Exception occurs");
              }
           catch(ArrayIndexOutOfBoundsException e)
              {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
              }
           catch(Exception e)
              {
               System.out.println("Parent Exception occurs");
              }
           System.out.println("rest of the code");
   }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

Example 3

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is invoked.

```
public class MultipleCatchBlock3
{
public static void main(String[] args) {

   try{
         int a[]=new int[5];
         a[5]=30/0;
         System.out.println(a[10]);
         }
   catch(ArithmeticException e)
   {
    System.out.println("Arithmetic Excep
tion occurs");
    }
catch(ArrayIndexOutOfBoundsException
e){
 System.out.println("ArrayIndexOutOfBo
unds Exception occurs");
  }
 catch(Exception e)
    {
     System.out.println("Parent Exception o
ccurs");
     }
   System.out.println("rest of the code");

    }
}
```

Output:

```
Arithmetic Exception occurs rest of the
code
```

Java Nested try block

- The try block within a try block is known as nested try block in java.

Why use nested try block

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
try
{
        statement 1;
       statement 2;
       try
       {
         statement 1;
               statement 2;
       }
       catch(Exception e)
       {
       }
}
catch(Exception e)
{
}
```

Java nested try example

```
class Excep6{
 public static void main(String args[]){
  try{
      try{
           System.out.println("going to divide");
           int b =39/0;
            }
       catch(ArithmeticException e){System.out.println(e);}

      try{
            int a[]=new int[5];
            a[5]=4;
             }
       catch(ArrayIndexOutOfBoundsException e)
            {  System.out.println(e);}
   System.out.println("other statement);
  }
catch(Exception e)
{
System.out.println("handeled");}
  System.out.println("normal flow..");
 } }
```

Java throw exception

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.
- The syntax of java throw keyword is given below.

```
throw exception;
```

Example of throw IOException

```
throw new IOException("sorry device error);
```

java throw keyword example

- In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the Arithmetic Exception otherwise print a message welcome to vote.

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java throws keyword

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of java throws

```
return_type method_name() throws exception_class_name{
//method code
}
```

- **Which exception should be declared**

checked exception only, because:

- **unchecked Exception:** under your control so correct your code.

error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

- **Advantage of Java throws keyword**

I. Now Checked Exception can be propagated (forwarded in call stack).

II. It provides information to the caller of the method about the exception.

Java throws example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p()
{
   try{
    n();
   }
catch(Exception e)
{System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }  }
```

Output:

```
exception handled
normal flow...
```

Throws Example

- There are two cases:

Case1: You caught the exception i.e. handle the exception using try/catch.

Case2: You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
void method()throws IOException{
 throw new IOException("device error");
 }
}
public class Testthrows2
{
public static void main(String args[]){
try{
            M m=new M();
            m.method();
      }
catch(Exception e)
{
System.out.println("exception handled");}
  System.out.println("normal flow...");
  }
}
```

Output:exception handled
normal flow...

Case2: You declare the exception

A)In case you declare the exception, if exception does not occur, the code will be executed fine.

B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation per
formed");
 }
}
class Testthrows3{
   public static void main(String args[])thr
ows IOException
{//declare exception
     M m=new M();
     m.method();
    System.out.println("normal flow...");
  } }
```

Output: device operation performed normal flow...

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])thro
ws IOException
{  //declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

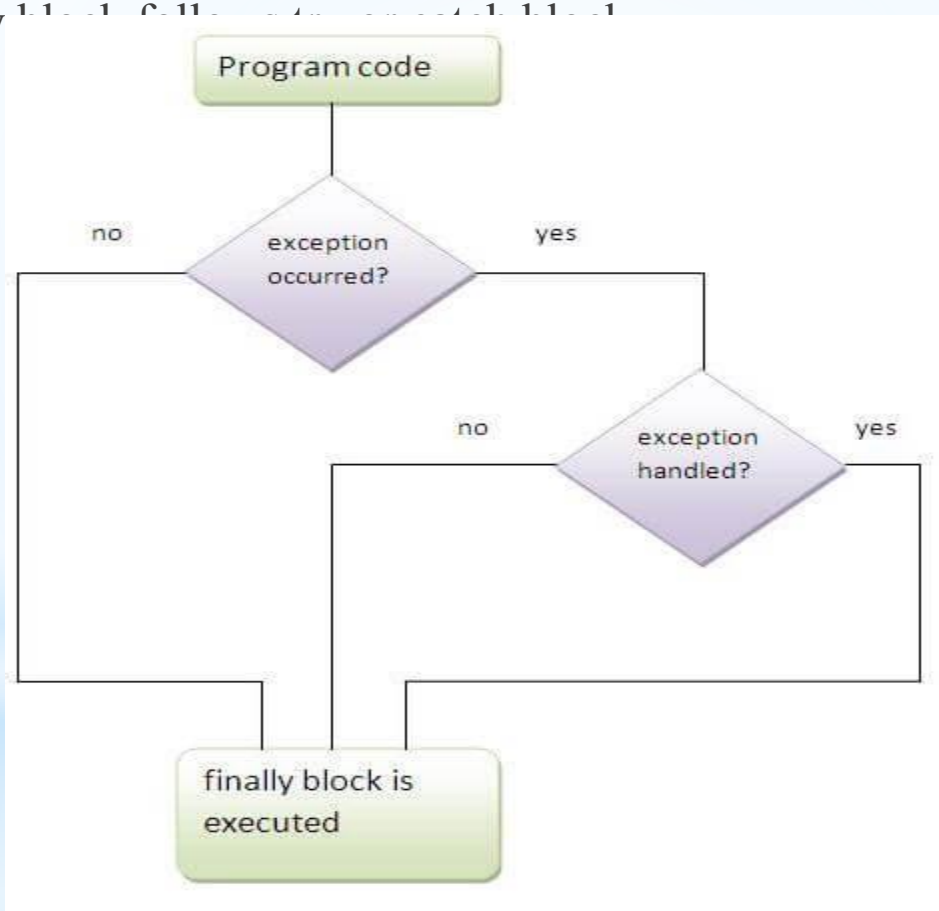
Output:Runtime Exception

Difference between throw and throws in Java

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 finally block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally



Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Case 1: java finally example where **exception doesn't occur**

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e)
{System.out.println(e);}
  finally
{System.out.println("finally block is always ex
ecuted");}
  System.out.println("rest of the code...");
  }
}
```

Output:5

finally block is always executed

rest of the code...

Case 2: java finally example where exception occurs and not handled.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e)
{System.out.println(e);}
  finally
{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:

finally block is always executed Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3:java finally example where exception occurs and handled.

```
public class TestFinallyBlock2{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println(e);}
  finally{System.out.println("finally block is always execu
ted");}
  System.out.println("rest of the code...");
  }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero finally block is always executed
rest of the code...

Multithreading

- **Multithreading in java** is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.
- **Advantages of Java Multithreading**

I. 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

II. 2) You **can perform many operations together, so it saves time**.

III. 3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:
  1) Process-based Multitasking (Multiprocessing)
  2) Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

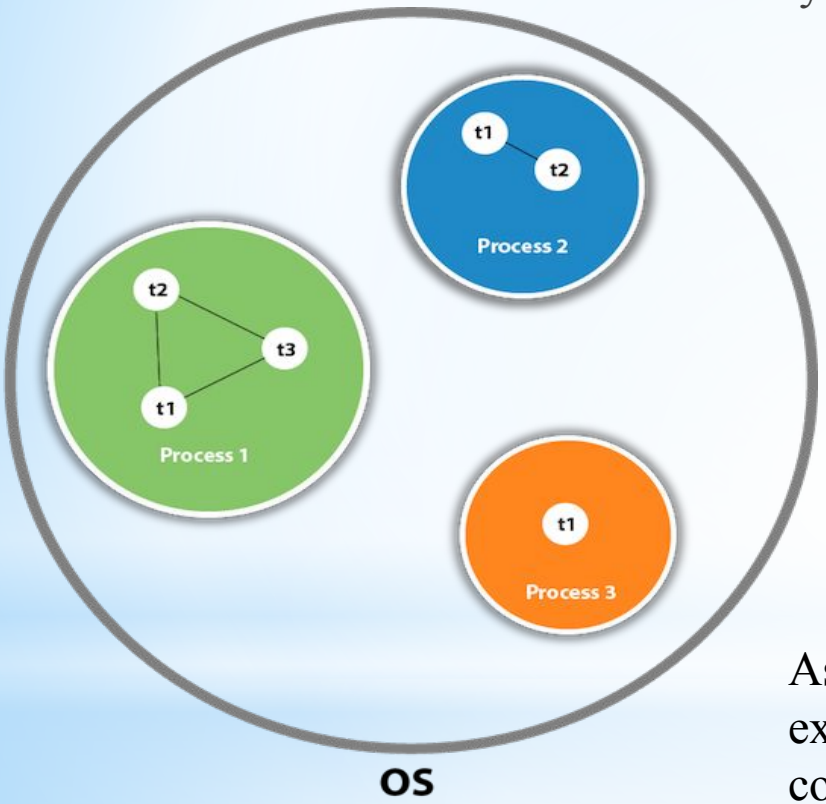
- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

What is Thread in java

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

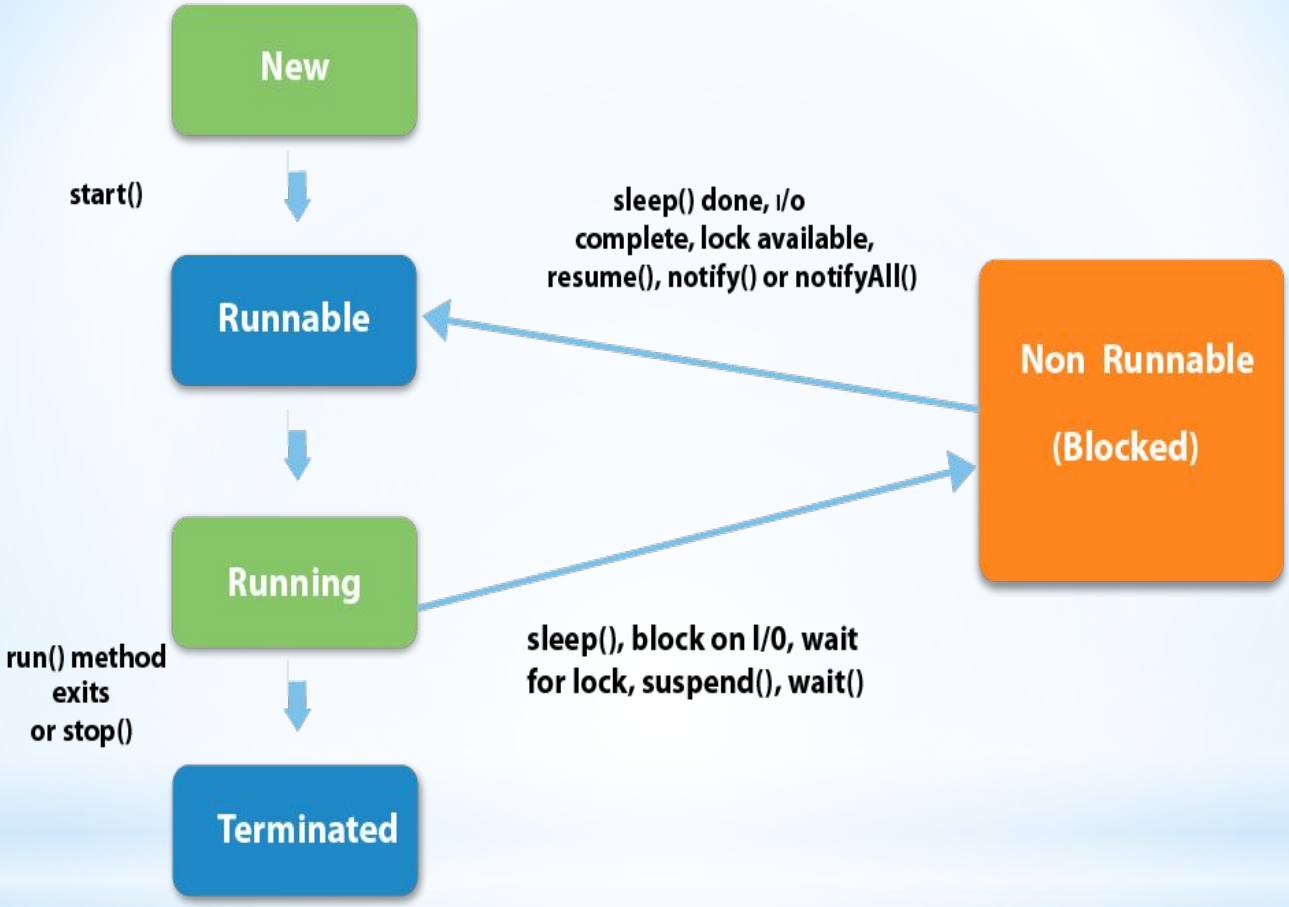
Java Thread class

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.

Life cycle of a Thread (Thread States)

- A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

I. New
II. Runnable
III. Running
IV. Non-Runnable (Blocked)
V. Terminated


New
start()
Runnable
Running
run() method
exits
or stop()
Terminated
sleep() done, i/o
complete, lock available,
resume(), notify() or notifyAll()
Non Runnable
(Blocked)
sleep(), block on I/0, wait
for lock, suspend(), wait()

1) New

- The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

- The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

- The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

- A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

- **Thread class:**

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

I. Thread()
II. Thread(String name)
III. Thread(Runnable r)
IV. Thread(Runnable r,String name)

How to create thread

- **Runnable interface:**

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

- **Starting a thread:**

start() method of Thread class is used to start a newly created thread. It performs following tasks:A new thread starts(with new callstack).
The thread moves from New state to the Runnable state.
When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();
t1.start();
 }
}
```

Output: thread is running...

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output: thread is running...

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Example : priority of a Thread

```
class TestMultiPriority1 extends Thread
{
 public void run()
{
System.out.println("running thread name is:"+Thread.currentThread().getN
ame());
System.out.println("running thread priority is:"+Thread.currentThread().ge
tPriority());
}
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
```

Output: running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

The join() method

- The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

```
public void join()throws Interrupted Exception
public void join(long milliseconds)throws Interrupted Exception
```

Example of join() method

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  } }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);
}
 t2.start();
 t3.start();
 }
}
```

Output:

```
1
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```