Multi Threading in Java

Introduction

- **Process**
  - it is an instance of a computer program that is executed sequentially.
- **Thread**
  - Thread is light weight sub process.
  - It is separate path of execution
  - It shares the memory area of process.

- **Thread based Multitasking**
  - it is a feature that allows a single program to perform more than one task simultaneously.
  - Executing several task simultaneously where each task is separate and an independent part of same program.
  - These each independent part of the program is called Thread.
  - Example:
    - Using a browser we can navigate through the webpage and at the same time download a file.

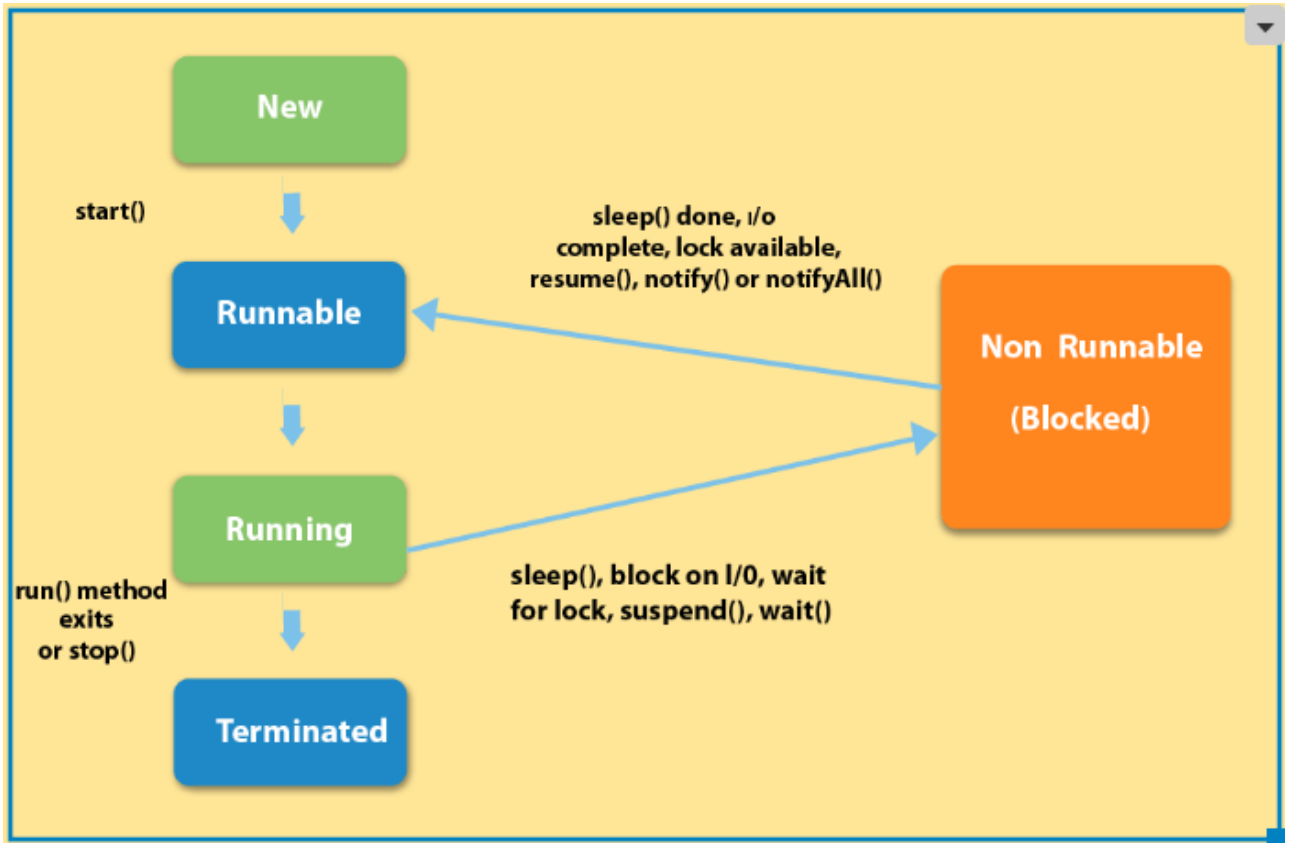
Note:

- There can be multiple processes inside the OS and one process can have multiple threads.
- The concept of threading in java is used to increase the execution speed of program.
- Multimedia graphics-animations, Server developments are areas where thread based multi-tasking can be used.

Multi Threaded Programming

- Multi Threaded programs contains two or more threads that can run concurrently, so that single program can perform two or more tasks simultaneously.
- In Multi threaded programming several threads of execution may be associated with single process.
- So, Process that has single thread is referred as Single-Threaded process.
- While Process that has multiple threads is referred as Multi threaded process.

Thread Life cycle



- A Thread in Java can be in one of the following states during the execution of program.
  - New
  - Runnable
  - Running
  - Blocked
  - Terminated.
- **New:**
  - When a new thread is created, and thread has not yet started to run then it is in **new state.**
  - The Thread is in new state if you create an instance of Thread class but has not invoke the start() method.

- **Runnable:**
  - A thread that is ready to run is moved to runnable state.
  - A thread in this state might be ready to run at any instance of time.
  - It is responsibility of thread scheduler to give the thread time to run.
  - After invocation of start() method on new thread, the thread becomes runnable.

- **Running:**
  - A thread is in running state if thread scheduler has selected it for execution.
  - A thread scheduler picks up one of the thread from the runnable thread pool to start thread execution.
  - A thread can change its state from running to runnable or Blocked depending upon time slicing ,completion of run() method or waiting for some resources.

- **Non-Runnable/Blocked**
  - When a thread is temporarily inactive, then it is in blocked/waiting state.
  - This is the state when thread is alive but not eligible to run.
  - When thread is waiting for some I/O event to complete or waiting for other thread to complete its task then it lies in this state.
  - A thread in this state cannot continue its execution until it is moved to runnable state.

- **Terminated**
  - A thread enters in terminated state when it completes it execution.
  - Thread is in terminated state when its run() method exits.

Main Thread and Garbage Collection Thread

- Any Simple java program is always running in a multi-threaded environment due to
  - Main thread
  - Garbage collection thread
- Even if you do not create any thread in your program , by default a thread called "Main thread" is still created by JVM to start execution of application.
- You can control main thread by obtaining reference of it by calling "currentThread()" Method.

- Main thread is one from which other child thread can be created.
- Main thread must be the last thread to finish execution because when it stops the program is terminated.
- Garbage collection thread is always running in background to clean up discarded objects and reclaim the memory.

Example :

```
class MainThreadDemo
{
    public static void main(String[] args)
    {
        Thread t  = Thread.currentThread();
        System.out.println(t.getName());
        System.out.println("Changing name of Thread");
        t.setName("My Main Thread");
        System.out.println(t.getName());
    }
}
```

Implementing Thread in Java

- In Java Thread can be implemented/Created by using two mechanism
  - Extending the **Thread** class
  - Implementing the **Runnable** Interface.

Steps for Creating Thread By Extending Thread class

- Create a subclass which extends the java.lang.Thread class
- Override the run() method in that subclass from the Thread class to define the code executed by the Thread.

```
class Mythread extends Thread
{
    public void run()
    {

        //Job of Mythread
        for(int i = 0 ;i<10;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
```

- Create an instance of subclass.
- Invoke the start() method on the instance of the class to make thread eligible for running.

```
class ThreadDemo1
{
    public static void main(String[] args)
    {
        Mythread mth = new Mythread();
        mth.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
}
```

Creating Thread By implementing Runnable interface

- The easiest way to create thread is to create a class that implements the Runnable interface.
- You can create a thread on any object that implements Runnable.
- To implement Runnable , a class only need to implements a single method call **run()**
  - public void run()
- Once you create a class that implements Runnable , you will instantiate an object of type **Thread** using that class .

- **Thread** defines several constructors. The one that we will use is shown here:
  - Thread(Runnable *threadOb*, String *threadName*)
  - In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface.
  - The name of the new thread is specified by *threadName*

- After the new thread is created, it will not start running until you call its **start( )** method,
- The **start( )** method is shown here:
  - void start( )

Steps:

- Create a class that implements **Runnable interface**
- Provide implementation of **run()** method of Runnable interface

```
class MyThread1 implements Runnable
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
```

- Create an instance of the **Thread** class and pass your class objects to its constructor as a parameters.
- Call Thread class **start()** method.

```
class ThreadDemo2
{
    public static void main(String[] args)
    {
        MyThread1 t1 = new MyThread1();
        Thread t = new Thread(t1,"Custom Thread");
        t.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("Main Thread");
        }
    }
}
```

Thread Priority in Java

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- A thread's priority is used by thread scheduler to select thread from thread pool and allocate processor to it.
- Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.
- Accepted value of priority for a thread is in range of 1 to 10.

- There are 3 static variables defined in Thread class for priority.
  - **public static int MIN_PRIORITY:** This is minimum priority that a thread can have. Value for this is 1.
  - **public static int NORM_PRIORITY:** This is default priority of a thread if do not explicitly define it. Value for this is 5.
  - **public static int MAX_PRIORITY:** This is maximum priority of a thread. Value for this is 10.

- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.
  - This is its general form:
    - **final void setPriority(int *level*)**
    - The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**.
- You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**,
  - **final int getPriority( )**

```java
class PriorityThread extends Thread
{
    public void run()
    {
            System.out.println(Thread.currentThread().getName() +" is executing with priority" + Thread.currentThread().getPriority());

    }
}
class ThreadPriorityDemo
{
    public static void main(String[] args)
    {
        PriorityThread p1 = new PriorityThread();
        PriorityThread p2 = new PriorityThread();
        PriorityThread p3 = new PriorityThread();
        p1.setName("Thread1");
        p2.setName("Thread2");
        p3.setName("Thread3");
        p1.setPriority(Thread.NORM_PRIORITY-2);
        p2.setPriority(Thread.MIN_PRIORITY + 3);
        p3.setPriority(Thread.MAX_PRIORITY -1);
        p1.start();
        p2.start();
        p3.start();
        System.out.println(Thread.currentThread().getName() +" is executing with priority" + Thread.currentThread().getPriority());

    }

}
```

Note :

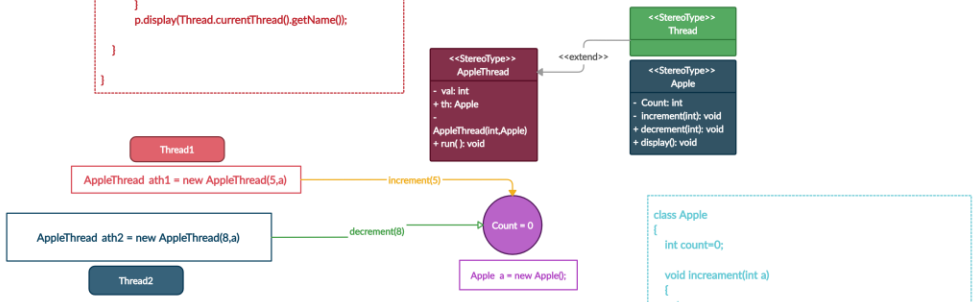
- In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.

Thread Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor or lock. (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.
- You can synchronize your code in either of two ways.
  - Using Synchronized Methods
  - The synchronized Statement

```
class AppleThread extends Thread
{
   int val;
   play p;
   AppleThread(nt  val,Apple p)
   {
      this.p=p;
      this.val=val;
   }

   public void run()
   {

    if(Thread.currentThread().getName().equals("Thread1"))
        {
           p.increament(val);
        }
        else
        {
           p.decreament(val);
        }
        p.display(Thread.currentThread().getName());

   }

}
```




```
class Apple
{
   int count=0;

   void increament(int a)
   {
      try
      {
         Thread.sleep(5000);
      }
      catch(InterruptedException e)
      {
         System.out.println(e);
      }
      count=count+a;
   }

   void decreament(int a)
   {
      try
      {
         Thread.sleep(5000);
      }
      catch(InterruptedException e)
      {
         System.out.println(e);
      }
      count = count-a;
   }
   void display(String name)
   {
      System.out.println(name+"has updated value of
count="+count);
   }

}
```

```
class Apple
{
    int count=0;

    void increament(int a)
    {
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }
        count=count+a;
    }

    void decreament(int a)
    {
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }
        count = count-a;
    }
    void display(String name)
    {
        System.out.println(name+"has updated value of count="+count);
    }

}
```

```
class Applethread extends Thread
{
    int val;
    Apple p;
    Applethread(play p,int val)
    {
        this.p=p;
        this.val=val;
    }
    public void run()
    {

            if(Thread.currentThread().getName().equals("Thread1"))
            {
                p.increament(val);
            }
            else
            {
                p.decreament(val);
            }
            p.display(Thread.currentThread().getName());
    }
}
```

```
class SynchronizedDemo1
{
    public static void main(String[] args)
    {
        Apple p = new Apple();
        Applethread p1 = new Applethread(p,5);
        Applethread p2 = new Applethread(p,2);
        p1.setName("Thread1");
        p2.setName("Thread2");
        p2.start();
        p1.start();
    }
}
```

Using Synchronized Method

- in Java, because all objects have their own implicit monitor/lock associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance must wait.
- To exit the monitor and provide control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

- **Syntax:**
  - **synchronized** return_type method_name(parameterlist)
  - {
  - //Body of the method
  - }

The Synchronized Block

- A block which contain synchronized keyword is called synchronized Block.
- Synchronized block can be used to perform synchronization on any specific resource/code of the method.

- **General syntax**
  - synchronized (object reference expression) {
  - //code block
  - }

```java
class Caller
{
    public void display(String msg)
    {
        System.out.print("["+msg);
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException e)
        {
            System.out.println("caught"+ e);
        }

        System.out.println("]");
    }
}
class CallerThread extends Thread
{
    Caller c;
    String s;

    CallerThread(Caller c,String s)
    {
        this.c = c;
        this.s = s;
    }

    public void run()
    {
        c.display(s);
    }
}
```

```java
class SynchronizationDemo2
{
    public static void main(String[] args)
    {
        Caller cr = new Caller();
        CallerThread cth1 = new CallerThread(cr, "Hello");
        CallerThread cth2 = new CallerThread(cr, "Morning");
        cth1.start();
        cth2.start();
    }
}
```

Exception Handling

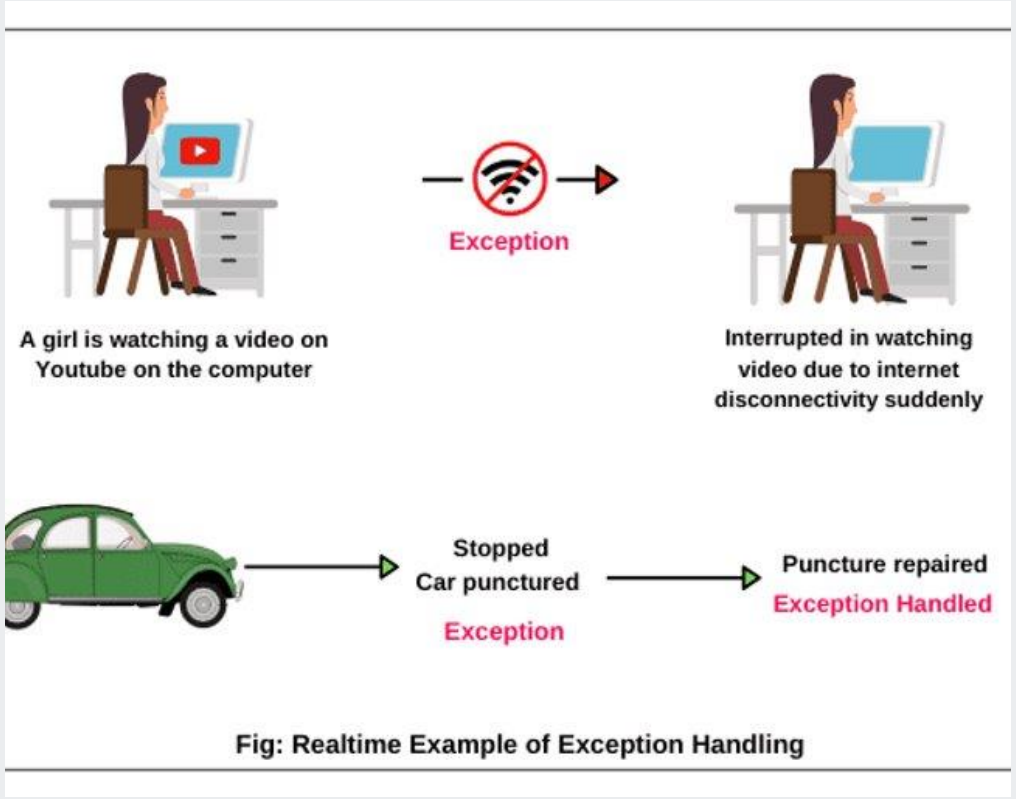


Fig: Realtime Example of Exception Handling

Introduction

An Exception is an unwanted or unexpected event ,which occur during execution of program and disturb the normal flow of the program execution.

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- This exception object contains detail information about an exception, which can be retrieved and processed.

Reason For Exceptions


```
1  public class JavaFehlerBehandlung {
2
3      public static void main(String[] args) {
4          int[] zahlen = new int[1];//neues Integer-Array mit einm Fach
5          zahlen[0] = 1; //Belegung des ersten Faches mit der Zahl 1
6
7          //For-Schleife durchläuft die ersten 10-Fächer des Arrays
8          for (int i = 0; i <= 10; i++) {
9              System.out.println("Fach " + i + ": " + zahlen[i]);
10         }
11     }
12 }
```

JAVA BLOG

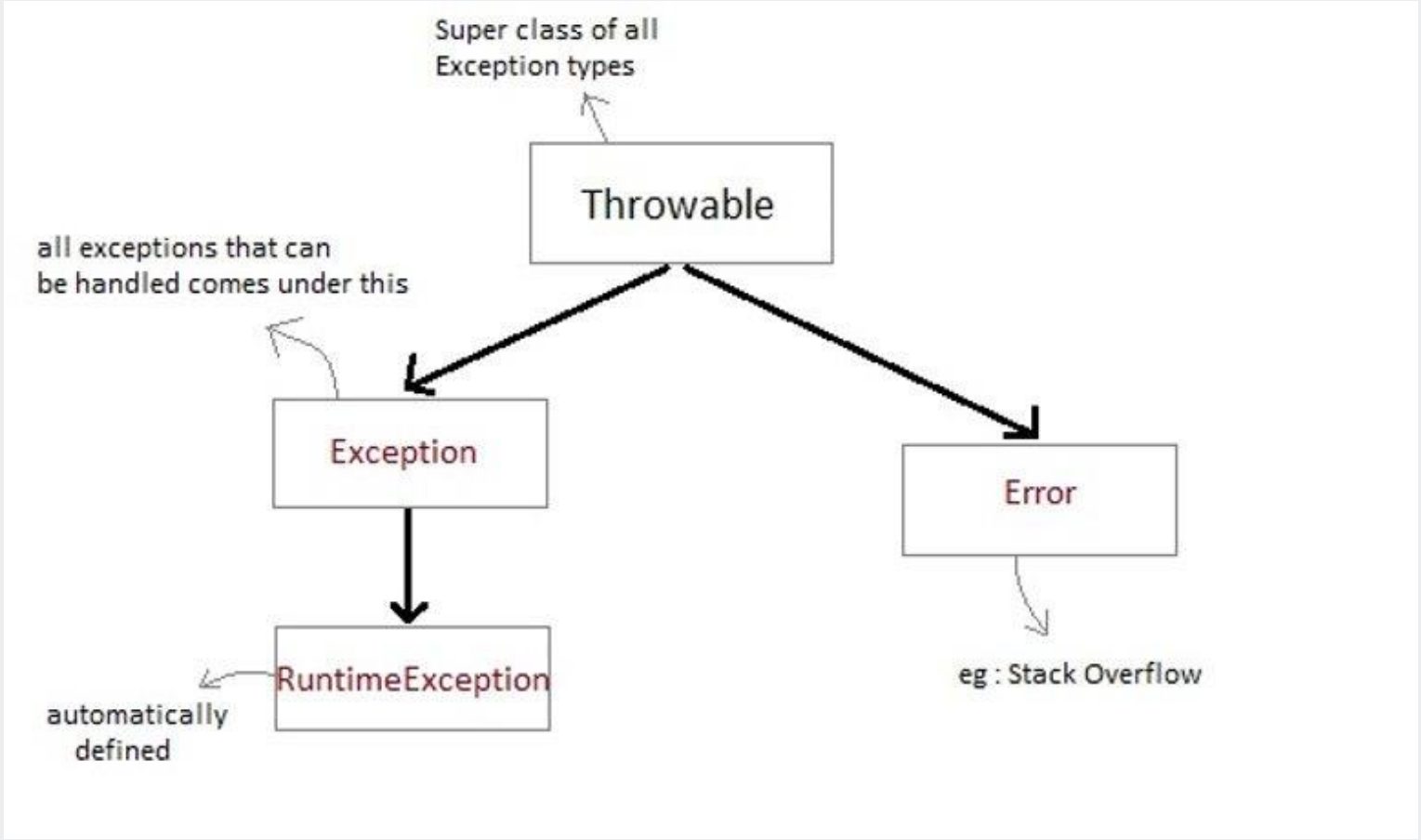
Problems Javadoc Declaration Search Console Terminal Data Source Explorer

<terminated> JavaFehlerBehandlung [Java Application] C:\Java_Home\bin\javaw.exe (17.03.2017, 17:55:06)

```
Fach 0: 1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
        at JavaFehlerBehandlung.main(JavaFehlerBehandlung.java:9)
```

- Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.

Exception Class Hierarchy



All exception types are subclasses of the built-in class **Throwable**.

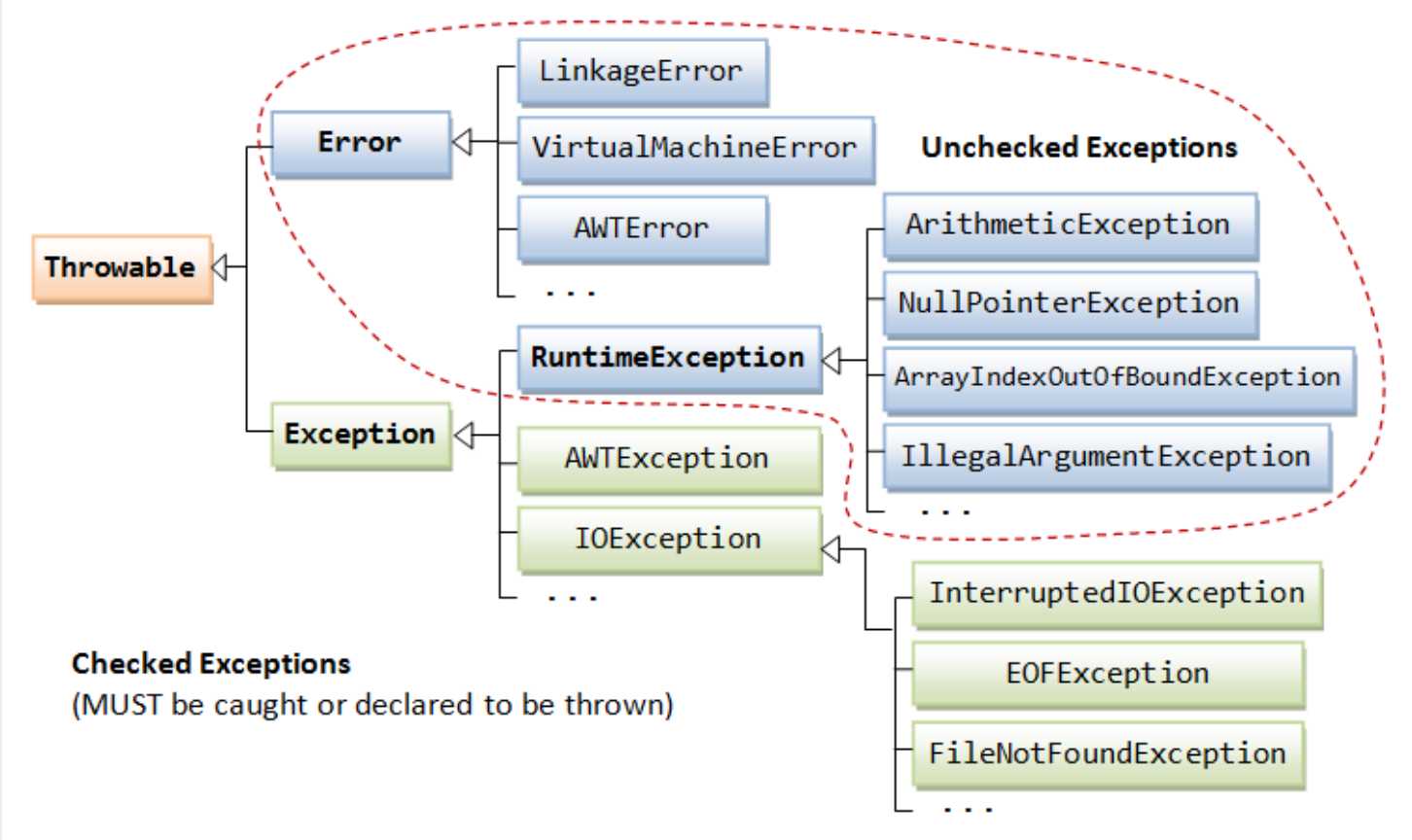
Throwable is at the top of the exception class hierarchy.

This class is used for exceptional conditions that user programs should catch.

Exception handling of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing

- **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are indicating errors having to do with the run-time environment itself.

Categories of Exceptions



Checked Exception:

- This type of Exception must be checked at compile time.
- This type of exception must be included in a method's **throws** list if that method can generate one of these exception and does not handle itself or it must be caught by a programmer.

Unchecked Exception

- This type of exception are not checked at compile time.(ignored at compile time)
- Exception classes that extends **RuntimeException** fall under this category.

Errors

- Such error are not predictable in code at time of program development.
- stackoverflow , OutofMemoryError,

Uncaught Exception

- When Java run time system detects an attempt to divide by zero ,it create a new exception object and throws this exception.
- this causes execution of program to stop ,because once an exception has been thrown it must be caught by an exception handler and dealt with immediately.
- here, we have-not provide our own exception handler, so exception is caught by default handler provided by java runtime system.
- this default handler displays string that describes the exception and terminates the program.

```
class A
{
    public static void main(String args[])
    {
        int d=0;
        int a =5/d;
    }
}
```

Note :

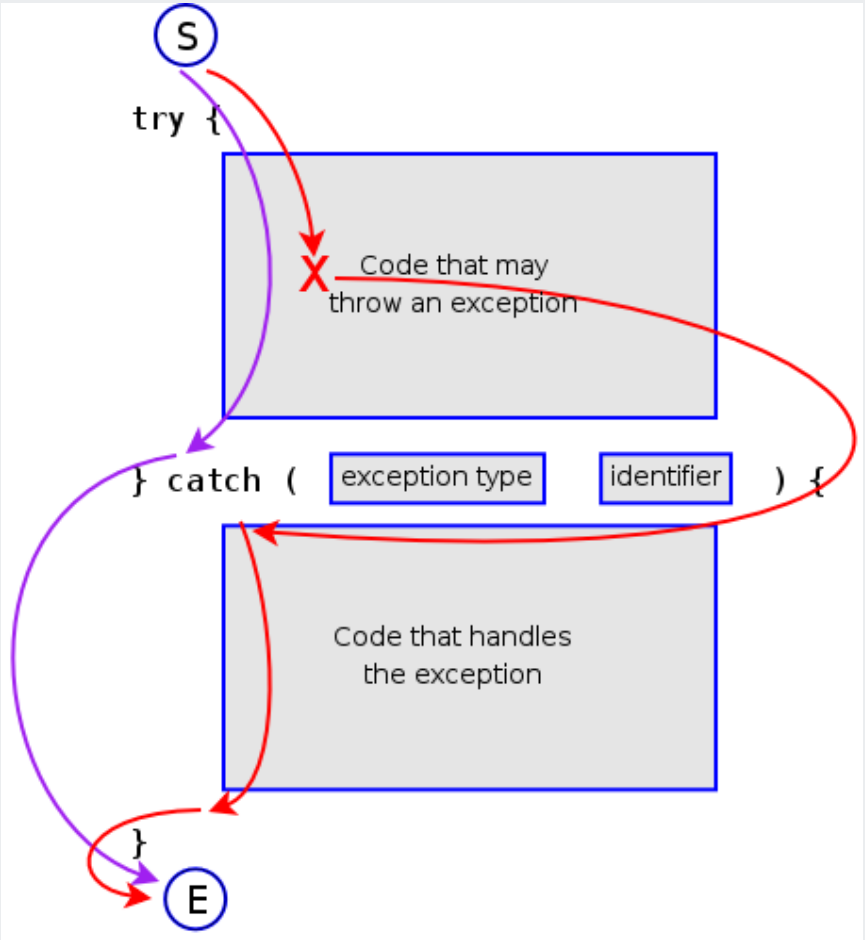
Though default exception handler provided by the java runtime system is useful for debugging , you should handle the exception yourself so that you can prevent abnormal termination of your program.

Exception Handling

- Exception handler is a set of code that handle an exception occurred at runtime.
- The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors to prevent abrupt termination of program.
- Exception handling can be achieved in Java using following
  - Try
  - Catch
  - Throw
  - Throws
  - finally

Exception Handling using try catch

- As default exception handler provided by java runtime system is used for just debugging you will want to handle an exception yourself.
- To guard against and handle a runtime error , simply enclose the code that you want to monitor inside a t**ry block.**
- immediately following the try block , include catch clauses that specifies the exception type that you wish to catch.



Note:

- The "try" block must be followed by either catch or finally.
- We cannot use try block alone.
- catch clause must be precedes by try block ,we cannot use catch block alone.
- catch block can be followed by finally later.

Example :

```
class ExceptionDemo2
{
    public static void main(String[] args)
    {
        String s = new String("Hello");
        System.out.println("Length of String="+ s.length());

        s = null;

        System.out.println("Length of String="+ s.length());

        System.out.println("Message After Exception");

    }
}
```

```
1  class ExceptionDemo2
2  {
3      public static void main(String[] args)
4      {
5          String s = new String("Hello");
6          System.out.println("Length of String="+ s.length());
7
8          s = null;
9
10         try
11         {
12             System.out.println("Length of String="+ s.length());
13         }
14         catch(Exception e)
15         {
16             System.out.println("We caught the Exception="+ e);
17         }
18
19         System.out.println("Message After Exception");
20
21     }
22 }
```

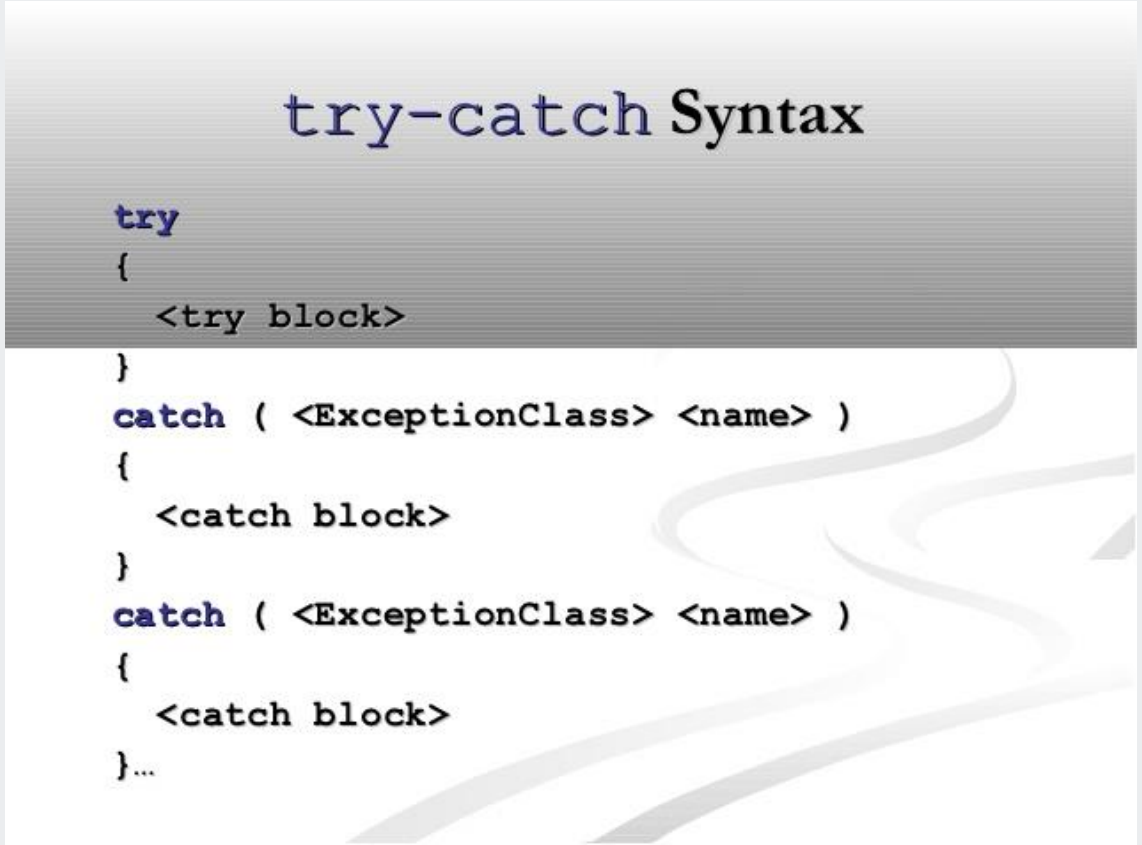
Note :

- Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.
- Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**.
- Thus, the line “Hello World” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try**/**catch** mechanism.

```
class ExceptionDemo1
{

    public static void main(String[] args) {
        int a =5, b;
        Scanner sc = new Scanner(System.in);
        b = sc.nextInt();
        try
        {
            int c = a/b;
            System.out.println("Hello World");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }

        System.out.println("Hello");
    }
}
```

Multiple catch clauses

- Sometime more than one exception could be raised by a single piece of code .
- to handle such type of situation you can specify two or more catch clauses ,each catching different type of exception.
- When an exception is thrown , each catch statement is inspected in order ,and first one whose type matches that of exception is executed.
- Once one catch statement executes , the others are bypassed, and execution continues after try/catch block.



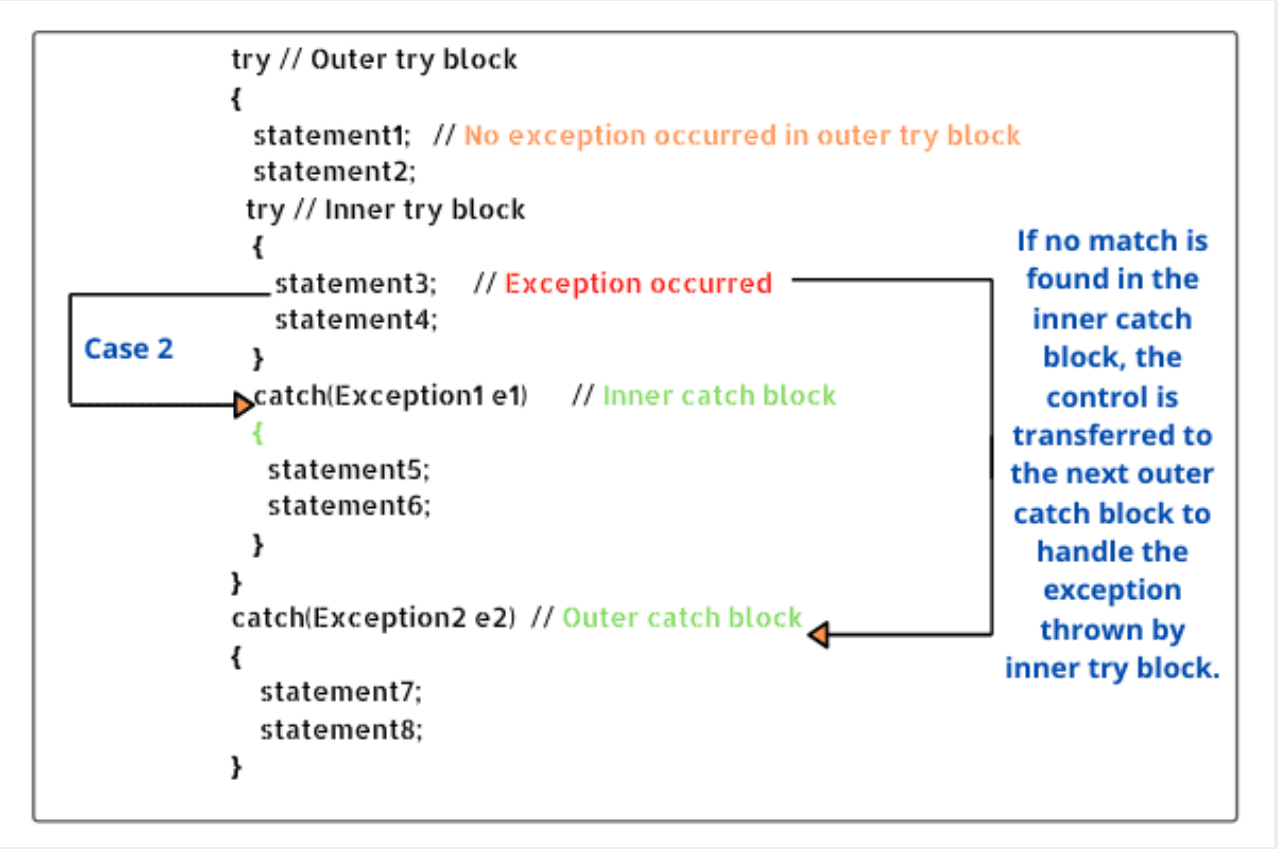
Example :

```
class ExceptionDemo3
{

    public static void main(String[] args) {
        int a =5, b;
        b = args.length;
        int x[] = new int[5];
        try
        {
            int c = a/b;
            x[6] = 12;
            System.out.println("Hello World");
        }
        catch(ArithmeticException e)
        {
            System.out.println("ArithmeticException = "+ e);
        }
        catch(ArrayIndexOutOfBoundsException e1)
        {
            System.out.println("Aray Index = "+ e1);
        }



        System.out.println("Hello");
    }
}
```

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Nested try Statements

- The **try** statement can be nested. That is, a **try** block can be used inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound, and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

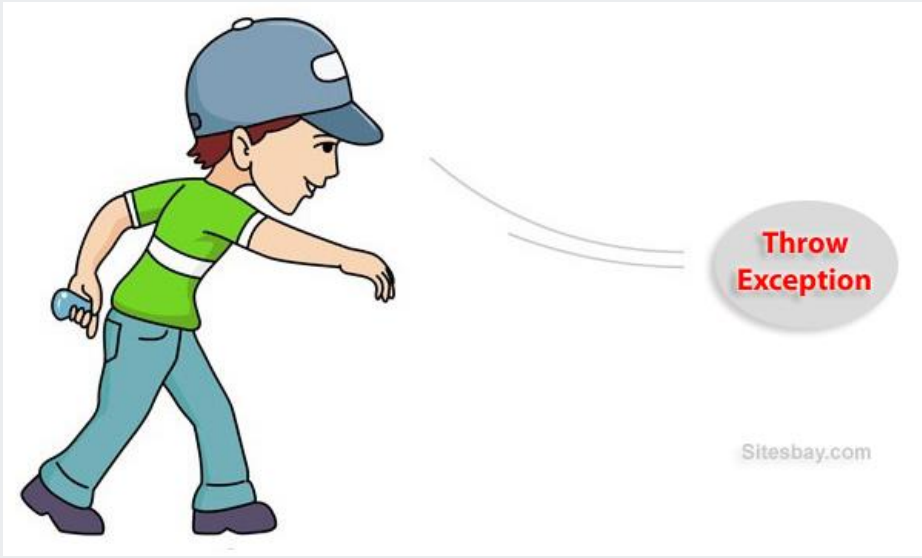


Example :

```
class ExceptionDemo4
{

    public static void main(String[] args) {
        int a =5, b;
        b = args.length;
        int x[] = new int[5];
        try
        {
            int c = a/b;

            try
            {
                int d = a / (b-1);
                x[5]=12;
            }
            catch(ArrayIndexOutOfBoundsException e1)
            {
                System.out.println("Aray Index = "+ e1);
            }
            System.out.println("Inner try ---Hello World");
        }
        catch(ArithmeticException e)
        {
            System.out.println("ArithmeticException = "+ e);
        }
        System.out.println("Outer try----------Hello");
    }
}
```

throw keyword in Java

- it is possible for your program to throw an exception explicitly from a method or any block of code, using the **throw** statement.
- throw keyword is mainly used to throw custom exception.
- The general form of **throw** is shown here:
  - **throw *ThrowableInstance;***
  - Here, ***ThrowableInstance*** must be an object of type **Throwable** or a subclass of **Throwable**.
- There are two ways you can obtain a **Throwable** object:
  - *using a parameter in a **catch** clause, or*
  - creating one with the **new** operator.



How it works?

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception.
- If it does find a match, control is transferred to that statement.
- If not, then the next enclosing **try** statement is inspected, and so on.
- If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace

```
class Vote
{
    void validate(int age)
    {
        if(age > 18)
        {
            System.out.println("You are elligible for Vote...");
        }
        else
        {
            try
            {
                throw new ArithmeticException("Not elligible for vote");
            }
            catch(NullPointerException e)
            {
                System.out.println("Caught validate :" + e);
            }
        }
    }
}
class ThrowDemo1
{
    public static void main(String[] args)
    {
        Vote v = new Vote();
        try{
            v.validate(13);
        }
        catch(ArithmeticException e1)
        {
            System.out.println("caught in main :" + e1);
        }

        System.out.println("Hello World");
    }
}
```

Note :

- Many of Java’s builtin run-time exceptions have at least two constructors:
  - one with no parameter and
  - one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception.
- This string is displayed when the object is used as an argument to **print( )** or **println( )**.
- It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**

Throws Keyword

- If a method can cause an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw.

- This is the general form of a method declaration that includes a **throws** clause:
- *type method-name*(*parameter-list*) throws *exception-list*

```
{
        // body of method
}
```

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Note :

- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

Example :

```
class Throws1
{
    void call()
    {
        throw new InterruptedException("Hello");
    }
}
class Throws1Demo
{
    public static void main(String[] args)
    {
        Throws1 th=new Throws1();
        th.call();

    }
}
```

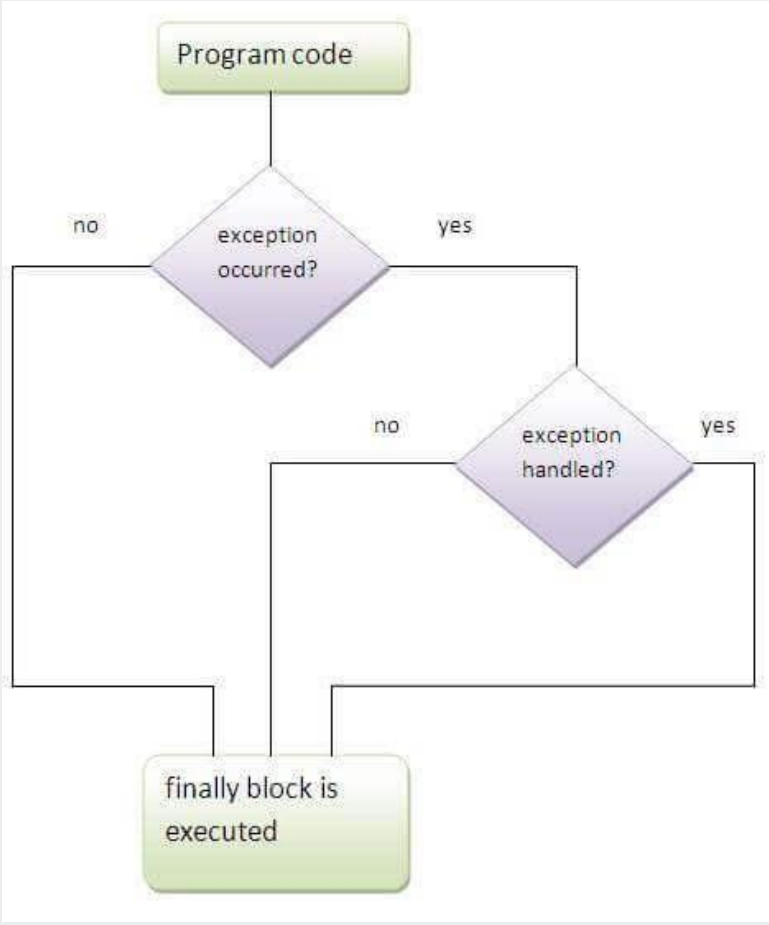
```
class Throws1
{
    void call() throws InterruptedException
    {
        throw new InterruptedException("Throws Demo");
    }
}
class Throws1Demo
{
    public static void main(String[] args)
    {
        Throws1 th=new Throws1();
        try
        {
            th.call();
        }
        catch(InterruptedException e)
        {
            System.out.println("Caught the Exception="+ e);
        }

    }
}
```

Note :

- throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions.
- The caller to these methods must handle the exception using a try-catch block.
- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.

Finally Block

- **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.



Where We need to use it?

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

Example:

```java
class FinallyDemo1
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Hello");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is executed.....");
        }
    }
}
```

```java
class FinallyDemo1
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Hello");
            int a = 5/0;
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is executed.....");
        }
    }
}
```

```java
class FinallyDemo1
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Hello");
            int a = 5/0;
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("finally block is executed.....");
        }
    }
}
```

Difference Between final, finally and finalize in java

- final is the modifier applicable to variable , method and classes
- finally is a block always associated with try/catch.
- cleanup activities or resource de-allocation activity code can be put in finally block.
- finalize() is a method present in Object class.
- finalize() method is called by garbage collector before destroying the object to perform cleanup activities.
- finally meant for cleanup activities related to try block.
- finalize() meant for cleanup activities related to objects.

User Defined Exception

- **Though** Java's built-in exceptions handle most common errors, but you will probably want to create your own exception types to handle situations specific to your applications.
- **Java** provides us facility to create our own **exceptions** which are basically derived classes of **Exception.**
- This is quite easy to do: just define a subclass of **Exception.**
- The **Exception** class does not define any methods of its own.
- It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

How to create User defined Exception?

Create your custom exception class that **extends Exception** class

```
class MyException extends Exception
{

}
```

How to create User defined Exception?

Define a single argument constructor which describes the exception.

```
class MyException extends Exception
{
    String msg ;

    MyException(String msg)
    {
        this.msg = msg;
    }
}
```

How to create User defined Exception?

Override **toString()** which is used to display exception related description.

```
class MyException extends Exception
{
    String msg ;

    MyException(String msg)
    {
        this.msg = msg;
    }

    public String toString()
    {
        return "MyException" + msg;
    }
}
```

Example:

```java
import java.util.Scanner;
class DivideByZero extends Exception
{
    String msg;

    DivideByZero(String msg)
    {
        this.msg = msg;
    }

    public String toString()
    {
        return msg;
    }
}
class DBZDemo
{
    public static void main(String[] args)
    {
            int a=5;
            Scanner sc =new Scanner(System.in);
            int b = sc.nextInt();

            if(b==0)
            {
                try
                {
                    throw new DivideByZero("/ by Zero");
                }
                catch(DivideByZero e)
                {
                    System.out.println("Caught= "+ e);
                }

            }
            else
            {
                System.out.println("Answer:"+ (a/b));
            }
    }
```

Example :

```
class MyException extends Exception
{
    String msg ;

    MyException(String msg)
    {
        this.msg = msg;
    }

    public String toString()
    {
        return "MyException" + msg;
    }
}
class UDExceptionDemo
{
    public static void main(String[] args) {

        try
        {
            throw new MyException("Check");
        }
        catch(MyException e)
        {
            System.out.println("Caught:" + e);
        }

    }
}
```