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.

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;
```

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:

1. Checked Exception
2. Unchecked Exception
3. Error

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

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.

```
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...
```

ArithmeticException

```
int a=50/0;//ArithmeticException
```

NullPointerException

```
String s=null;
1.System.out.println(s.length());//NullPointerException
```

NumberFormatException

```
String s="abc";
1.int i=Integer.parseInt(s);//NumberFormatException
```

ArrayIndexOutOfBoundsException

```
int a[]=new int[5];
1.a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Multi-catch block

```java
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
```

```java
public class MultipleCatchBlock4 {


    public static void main(String[] args) {

        try{
            String s=null;
            System.out.println(s.length());
           }
           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:

```
Parent Exception occurs
rest of the code
```

finally:

```
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:

```
Output:finally block is always executed
```

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

throw:

```
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

throws: The **Java throws keyword** is used to declare an exception.

```
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...

Multithreading in Java

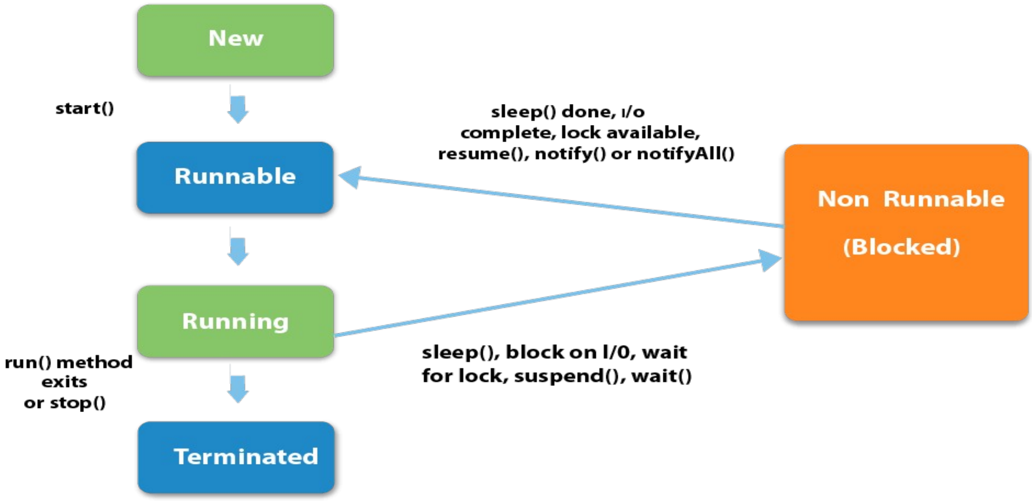
Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Process	Thread
1. Process has its own main memory for execution. 2. Process is considered as heavyweight component. 3. One process can have multiple threads. 4. Context switch time is more.	1. Thread use process's main memory for execution and share it with other threads. 2. Thread is considered as lightweight component. 3. One thread can't have multiple process. 4. Context switch time is less.

Life cycle of a Thread

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated


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

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.

How to create thread

There are two ways to create a thread:

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

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...

sleep():

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e)
{System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

output:

```
1
1
2
```

```
2
3
3
4
4
```

Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

public class TestSynchronization3{
public static void main(String args[]){
final Table obj = new Table();//only one object

Thread t1=new Thread(){
public void run(){
obj.printTable(5);
}
};
Thread t2=new Thread(){
public void run(){
obj.printTable(100);
}
};

t1.start();
t2.start();
}
}
```

```
Output: 5
10
15
20
25
100
200
```

```
300
400
500
```

Synchronized Block in Java

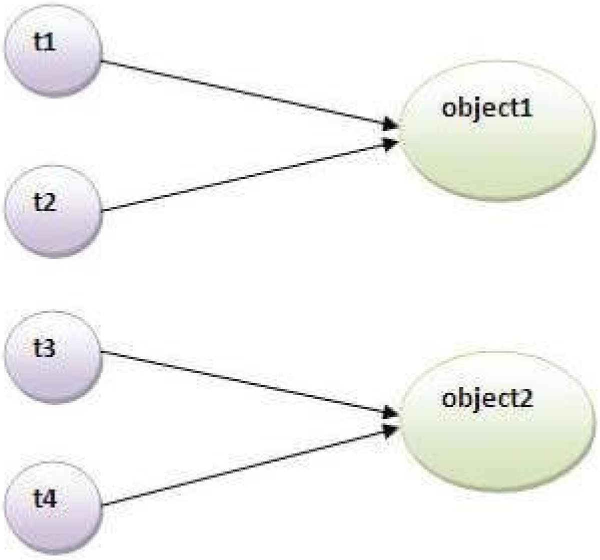
Synchronized block can be used to perform synchronization on any specific resource of the method.

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}
```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

```
class Table{

 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}

class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}

class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}




class MyThread4 extends Thread{
public void run(){
```

```
Table.printTable(1000);
}
}

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

```
Output: 1
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
```

300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

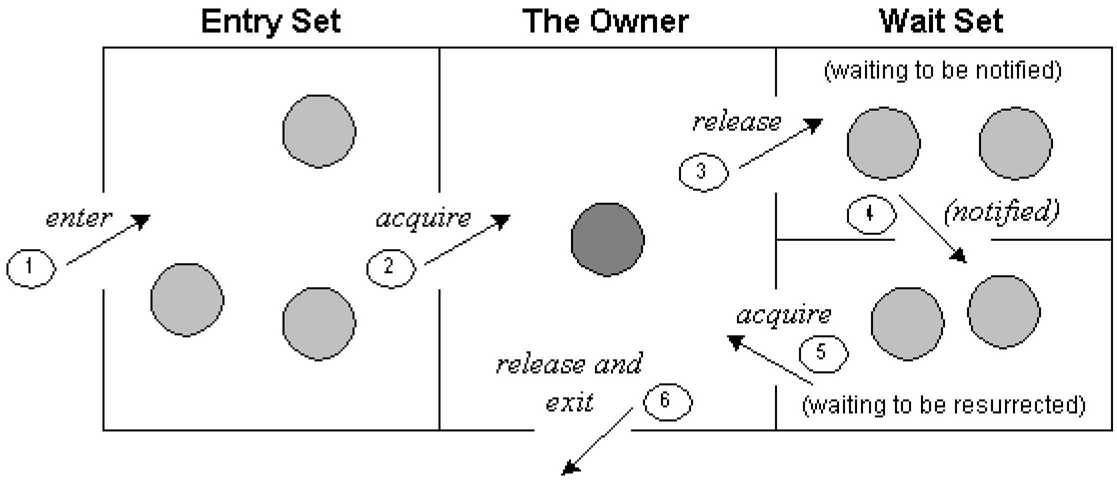
wait()

- notify()
- notifyAll()

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	**sleep()**
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.



```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed
```