MODULE III

Topics:

Exception Handling Concept – Exception Hierarchy- Differences between Multi thread –Muti Tasking-Thread Cycle-Daemon Threads.

Exception in Java:

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. An exception is an error event that can happen during the execution of a program and disrupts its normal flow. Java provides a robust and object-oriented way to handle exception scenarios known as Java Exception Handling.

Exceptions in Java can arise from different kinds of situations such as wrong data entered by the user, hardware failure, network connection failure, or a database server that is down. The code that specifies what to do in specific exception scenarios is called exception handling.

Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes.

try-catch – We use the try-catch block for exception handling in our code. try is the start of the block and catch is at the end of the try block to handle the exceptions. We can have multiple catch blocks with a try block. The try-catch block can be nested too. The catch block requires a parameter that should be of type Exception.

throw – We know that if an error occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometimes we might want to generate exceptions explicitly in our code. For example, in a user authentication program, we should throw exceptions to clients if the password is null. The throw keyword is used to throw exceptions to the runtime to handle it.

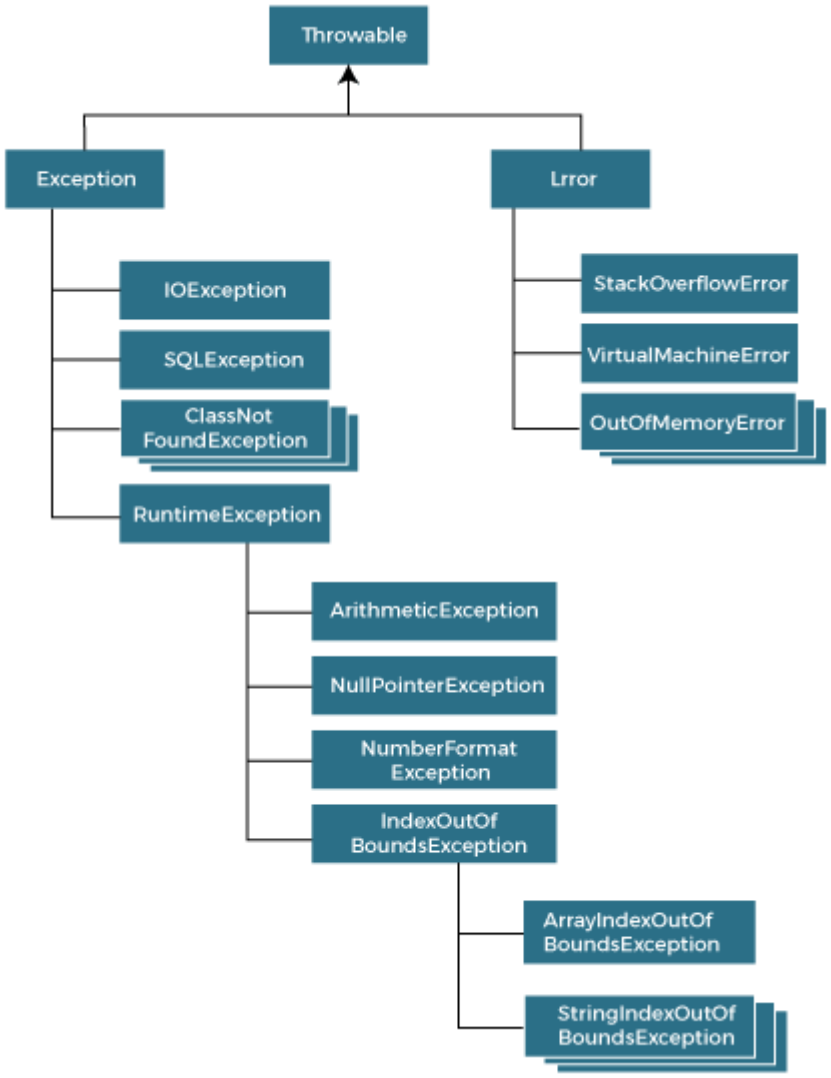
throws – When we are throwing an exception in a method and not handling it, then we have to use the throws keyword in the method signature to let the caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate them to its caller method using the throws keyword. We can provide multiple exceptions in the throws clause, and it can be used with the main() method also.

finally – the finally block is optional and can be used only with a try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use the finally block. The finally block always gets executed, whether an exception occurred or not.

Java Exception Hierarchy

As stated earlier, when an exception is raised an exception object is getting created. Java Exceptions are hierarchical and inheritance is used to categorize different types of exceptions. Throwable is the parent class of Java Exceptions Hierarchy and it has two child objects – Error and Exception. Exceptions are further divided into Checked Exceptions and Runtime Exceptions.

Diagram of the Java Exception Hierarchy.



Throwable is at the top of the diagram. One branch of this tree is Error. Below Error are OutOfMemoryError and IOError. Another branch of this tree is Exception. Exception splits into IOException and RuntimeException. Below IOException is FileNotFoundException. Below RuntimeException is NullPointerException.

Some useful methods of Exception Classes

Java Exception and all of its subclasses don't provide any specific methods, and all of the methods are defined in the base class - Throwable. The Exception classes are created to specify different kinds of Exception scenarios so that we can easily identify the root cause and handle the Exception according to its type. The Throwable class implements the Serializable interface for interoperability.

Some of the useful methods of the Throwable class are:

public String getMessage() – This method returns the message String of Throwable and the message can be provided while creating the exception through its constructor.

public String getLocalizedMessage() – This method is provided so that subclasses can override it to provide a locale-specific message to the calling program. The Throwable class implementation of this method uses the getMessage() method to return the exception message.

public synchronized Throwable getCause() – This method returns the cause of the exception or null if the cause is unknown.

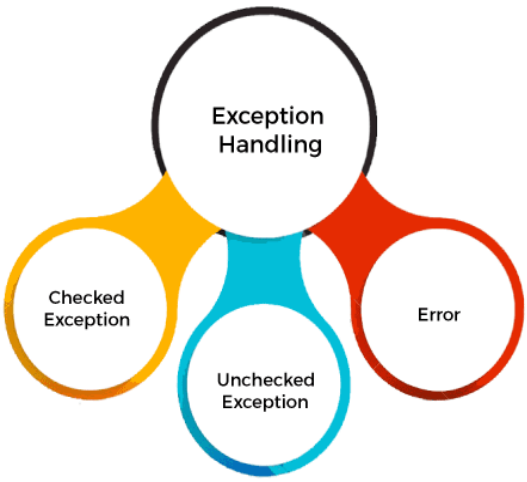
public String toString() – This method returns the information about Throwable in String format, the returned String contains the name of the Throwable class and localized message.

public void printStackTrace() – This method prints the stack trace information to the standard error stream, this method is overloaded, and we can pass PrintStream or PrintWriter as an argument to write the stack trace information to the file or stream.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



Difference between Checked and Unchecked Exceptions

1) **Checked Exception**
The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) **Unchecked Exception**
The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) **Error**
Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Handling Example:

Java try 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 in the try block, the rest of the block code will not execute. So, it is recommended not to keep 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

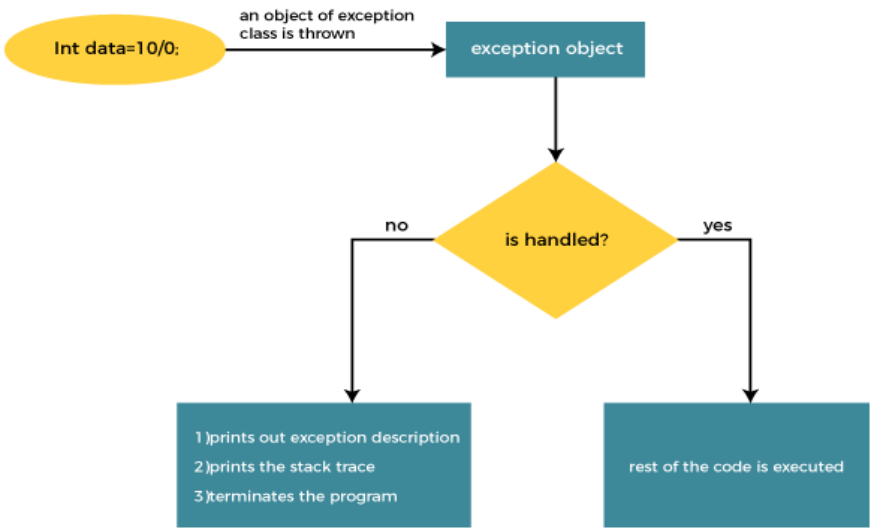
```
try
{
//code that may throw an exception
}
catch(Exception_class_Name ref){}
```

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.



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling:

Here is the example without exception handling

Code:

```
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
```

As displayed in the above example, the rest of the code is not executed (in such case, the rest of the code statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Example for try-catch

```
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
```

As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 2:

Let's see an example to print a custom message on exception.

```
public class TryCatchExample2 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
      }
         // handling the exception
      catch(Exception e)
      {
```

```
            // displaying the custom message
            System.out.println ("Can't divided by zero");
        }
    }
}
```

Example :

Can't divided by zero

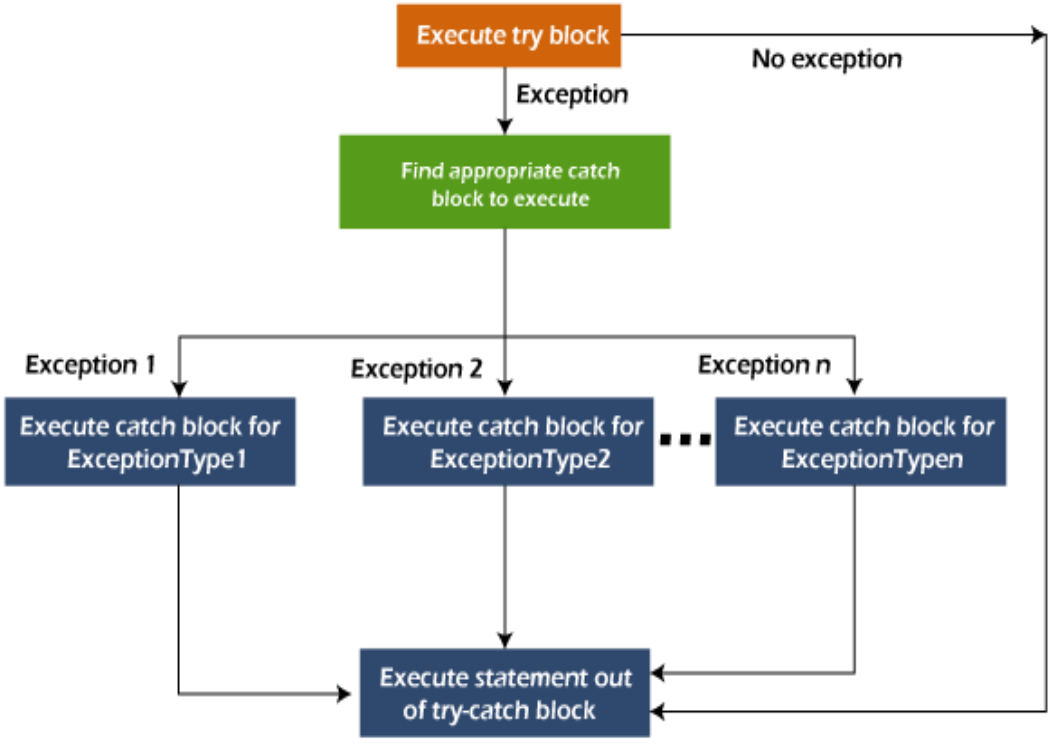
Java Multi-catch block

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.

Flowchart of Multi-catch Block



Example

```
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
```

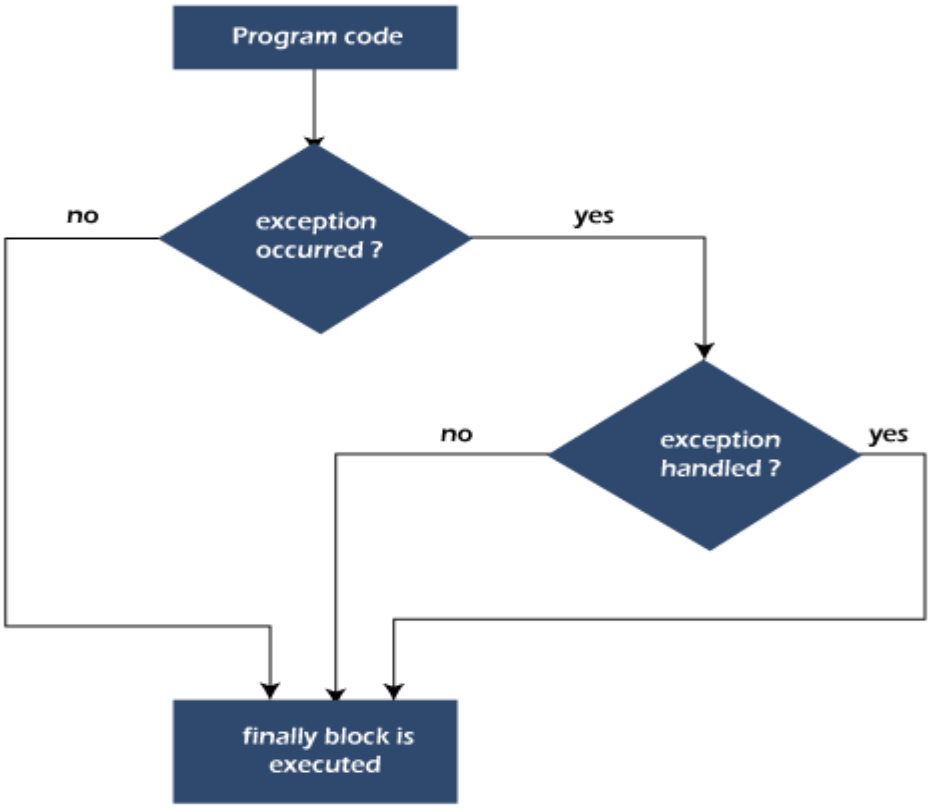
Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block



CASE 1:

```
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
  }
```

}

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock
5
finally block is always executed
rest of the code...
```

CASE 2:

```
public class TestFinallyBlock2{
    public static void main(String args[]){

    try {

      System.out.println("Inside try block");

      //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }

    //handles the Arithmetic Exception / Divide by zero exception
    catch(ArithmeticException e){
      System.out.println("Exception handled");
      System.out.println(e);
    }

    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");
    }
   }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

```
public class TestThrow1 {
   //function to check if person is eligible to vote or not
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
   //main method
   public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
```

}

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

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 the normal flow of the program 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 checking the code before it being used.

Syntax of Java throws

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

Which exception should be declared?

Ans: Checked exception only, because:

unchecked exception: under our control so we can correct our code.

error: beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

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...
```

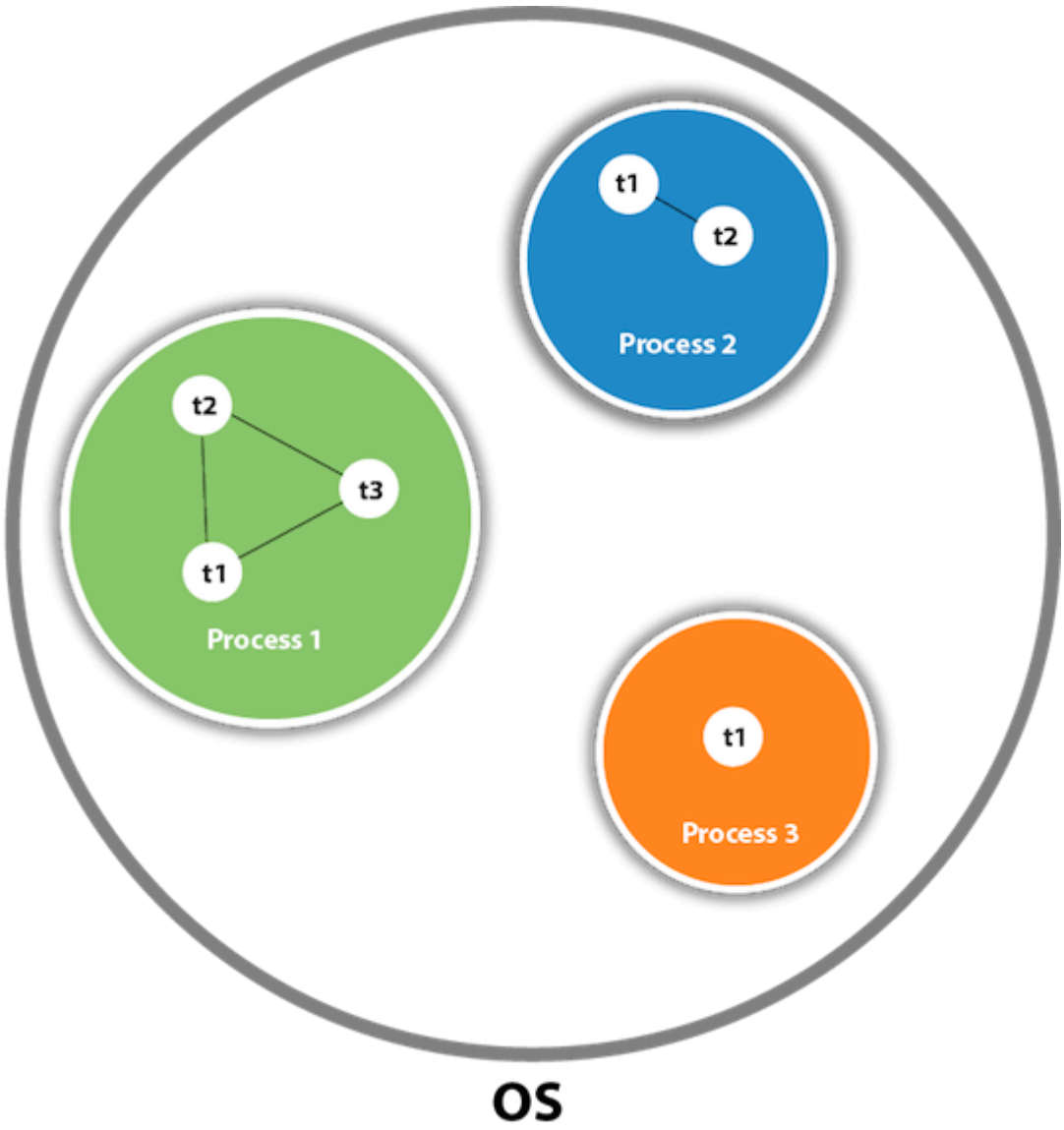
Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Multithreading in Java

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.



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.

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

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

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

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:

- Process-based Multitasking (Multiprocessing)
- 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.

Difference between Multitasking and Multithreading

Parameters	**Multi-tasking**	**Multi-threading**
Basics	The process of multi-tasking lets a CPU execute various tasks at the very same time.	The process of multi-threading lets a CPU generate multiple threads out of a task and process all of them simultaneously.
Working	A user can easily perform various tasks simultaneously with their CPU using multi-tasking.	A CPU gets to divide a single program into various threads so that it can work more efficiently and conveniently. Thus, multi-threading increases computer power.
Resources and Memory	The system needs to allocate separate resources and memory to different programs working simultaneously in multi-tasking.	The system allocates a single memory to any given process in multi-threading. The various threads generated out of it share that very same resource and memory that the CPU allocated to them.
Switching	There is the constant switching between various programs by the CPU.	The CPU constantly switches between the threads and not programs.
Multiprocessing	It involves multiprocessing among the various components.	It does not involve multiprocessing among its various components.
Speed of Execution	Executing multi-tasking is comparatively slower.	Executing multi-threading is comparatively much faster.
Process Termination	The termination of a process takes up comparatively more time in multi-tasking.	The termination of a process takes up comparatively less time in multithreading.

Java Threads | How to create a thread in Java

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

Example:

```
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**

Example:

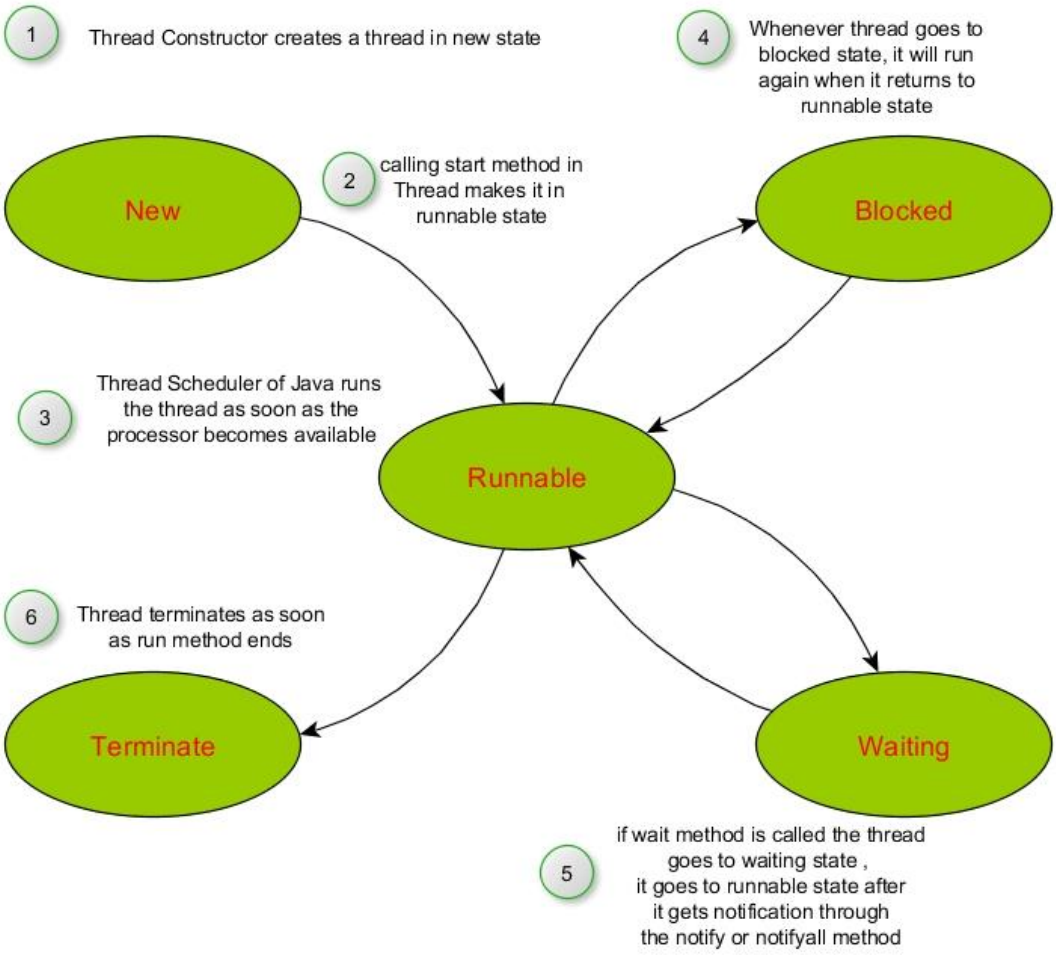
```
class Multi3 implements Runnable{

public void run(){

System.out.println("thread is running...");

}
```

```
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

thread is running...

Thread Life Cycle



In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.

Example:

```
class ABC implements Runnable
{
public void run()
{

// try-catch block
try
{
// moving thread t2 to the state timed waiting
Thread.sleep(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}

  System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -
"+ ThreadState.t1.getState());
```

```
// try-catch block
try
{
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}

// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable
{
public static Thread t1;
public static ThreadState obj;

// main method
public static void main(String argvs[])
{
// creating an object of the class ThreadState
obj = new ThreadState();
t1 = new Thread(obj);

// thread t1 is spawned
// The thread t1 is currently in the NEW state.
System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on
// the thread t1
t1.start();

// thread t1 is moved to the Runnable state
System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getSt
ate());
}

public void run()
{
ABC myObj = new ABC();
Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
t2.start();

// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState
());
```

```
// try-catch block for the smooth flow of the  program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getSt
ate() );

// try-catch block for the smooth flow of the  program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}
catch (InterruptedException ie)
{
ie. printStackTrace();
}
System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getStat
e());
}

}
```

Output:

```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -
TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING
The state of thread t2 when it has completed it's execution - TERM
```

Daemon thread :

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.

Example of Daemon Thread in Java: Garbage collection in Java (gc), finalizer, etc.

Properties of Java Daemon Thread:

2. They cannot prevent the JVM from exiting when all the user threads finish their execution.
3. JVM terminates itself when all user threads finish their execution.
4. If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
5. It is an utmost low priority thread.

Default Nature of Daemon Thread

By default, the main thread is always non-daemon but for all the remaining threads, daemon nature will be inherited from parent to child. That is, if the parent is Daemon, the child is also a Daemon and if the parent is a non-daemon, then the child is also a non-daemon.

Note: Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically.

Methods of Daemon Thread

1. void setDaemon(boolean status):

This method marks the current thread as a daemon thread or user thread. For example, if I have a user thread tU then tU.setDaemon(true) would make it a Daemon thread. On the other hand, if I have a Daemon thread tD then calling tD.setDaemon(false) would make it a user thread.

Syntax:

```
public final void setDaemon(boolean on)
```

Parameters:

on: If true, marks this thread as a daemon thread.

1. **Exceptions:**

IllegalThreadStateException: if only this thread is active.

SecurityException: if the current thread cannot modify this thread.

2. **boolean isDaemon():**

This method is used to check that the current thread is a daemon. It returns true if the thread is Daemon. Else, it returns false.

Syntax:

public final boolean isDaemon()

Returns:

This method returns true if this thread is a daemon thread; false otherwise

Example:

```
public class DaemonThread extends Thread
{
   public DaemonThread(String name){
      super(name);
   }

   public void run()
   {
      // Checking whether the thread is Daemon or not
      if(Thread.currentThread().isDaemon())
      {
         System.out.println(getName() + " is Daemon thread");
      }
       else
      {
         System.out.println(getName() + " is User thread");
      }
   }
```

```
    public static void main(String[] args)
  {
     DaemonThread t1 = new DaemonThread("t1");
     DaemonThread t2 = new DaemonThread("t2");
     DaemonThread t3 = new DaemonThread("t3");
     // Setting user thread t1 to Daemon
     t1.setDaemon(true);
              // starting first 2 threads
     t1.start();
     t2.start();
      // Setting user thread t3 to Daemon
     t3.setDaemon(true);
     t3.start();
  }
}
```

Output:

```
t1 is Daemon thread
t3 is Daemon thread
t2 is User thread
```