Types of Errors in Java

Error is an illegal operation performed by the user which results in the abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

The most common errors can be broadly classified as follows:

1. **Run Time Error:** Run Time errors occur or we can say, are detected during the execution of the program. Sometimes these are discovered when the user enters an invalid data or data which is not relevant. Runtime errors occur when a program does not contain any syntax errors but asks the computer to do something that the computer is unable to reliably do. During compilation, the compiler has no technique to detect these kinds of errors. It is the JVM (Java Virtual Machine) which detects it while the program is running. To handle the error during the run time we can put our error code inside the try block and catch the error inside the catch block. **For example:** if the user inputs a data of string format when the computer is expecting an integer, there will be a runtime error.
1. **Example 1:** Runtime Error caused by dividing by zero

```
// Java program to demonstrate Runtime Error

class DivByZero {
    public static void main(String args[])
    {
        int var1 = 15;
        int var2 = 5;
        int var3 = 0;
        int ans1 = var1 / var2;

        // This statement causes a runtime error,
        // as 15 is getting divided by 0 here
        int ans2 = var1 / var3;

        System.out.println(
            "Division of va1"
            + " by var2 is: "
            + ans1);
        System.out.println(
            "Division of va1"
            + " by var3 is: "
            + ans2);
    }
}
```

RunTime Error in java code:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivByZero.main(File.java:14)
```

Example 2: Runtime Error caused by Assigning/Retrieving Value from an array using an index which is greater than the size of the array

```
class RTErrorDemo {
    public static void main(String args[])
    {
        int arr[] = new int[5];

        // Array size is 5
        // whereas this statement assigns

        // value 250 at the 10th position.
        arr[9] = 250;

        System.out.println("Value assigned! ");
    }
}
```

RunTime Error in java code:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
9

    at RTErrorDemo.main(File.java:10)
```

2. **Compile Time Error:** Compile Time Errors are those errors which prevent the code from running because of an incorrect syntax such as a missing semicolon at the end of a statement or a missing bracket, class not found, etc. These errors are detected by the java compiler and an error message is displayed onto the screen while compiling. Compile Time Errors are sometimes also referred to as **Syntax errors**. These kind of errors are easy to spot and rectify because the java compiler finds them for you. The compiler will tell you which piece of code in the program got in trouble and its best guess as to what you did wrong. Usually, the compiler indicates the exact line where the error is, or sometimes the line just before it, however, if the problem is with incorrectly nested braces, the actual error may be at the beginning of the block. In effect, syntax errors represent grammatical errors in the use of the programming language.
Example 1: Misspelled variable name or method names

```
class MisspelledVar {
    public static void main(String args[])
    {
        int a = 40, b = 60;

        // Declared variable Sum with Capital S
        int Sum = a + b;

        // Trying to call variable Sum
        // with a small s ie. sum
        System.out.println(
            "Sum of variables is "
            + sum);
    }
}
```

Compilation Error in java code:

```
prog.java:14: error: cannot find symbol

            + sum);
```

```
                ^
  symbol:   variable sum
  location: class MisspelledVar
1 error
```

Example 2: Missing semicolons

```
class PrintingSentence {
    public static void main(String args[])
    {
        String s = "GeeksforGeeks";

        // Missing ';' at the end
        System.out.println("Welcome to " + s)
    }
}
```

Compilation Error in java code:

```
prog.java:8: error: ';' expected
        System.out.println("Welcome to " + s)
                                              ^
1 error
```

Example: Missing parenthesis, square brackets, or curly braces

```
class MissingParenthesis {
    public static void main(String args[])
    {
        System.out.println("Printing 1 to 5 \n");
        int i;

        // missing parenthesis, should have
        // been for(i=1; i<=5; i++)
        for (i = 1; i <= 5; i++ {

            System.out.println(i + "\n");
        } // for loop ends

    } // main ends
} // class ends
```

Compilation Error in java code:

```
prog.java:10: error: ')' expected
        for (i = 1; i <= 5; i++ {
                                ^
1 error
```

Example: Incorrect format of selection statements or loops

1.

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

```
        {

            System.out.println("Multiplication Table of 7");
            int a = 7, ans;
            int i;

            // Should have been
            // for(i=1; i<=10; i++)
            for (i = 1, i <= 10; i++) {
                ans = a * i;
                System.out.println(ans + "\n");
            }
        }
    }
```

2. **Compilation Error in java code:**

```
prog.java:12: error: not a statement
        for (i = 1, i <= 10; i++) {
                     ^
prog.java:12: error: ';' expected
        for (i = 1, i <= 10; i++) {
                               ^
2 errors
```

3. **Logical Error:** A logic error is when your program compiles and executes, but does the wrong thing or returns an incorrect result or no output when it should be returning an output. These errors are detected neither by compiler nor by JVM. The Java system has no idea what your program is supposed to do, so it provides no additional information to help you find the error. Logical errors are also called Semantic Errors. These errors are caused due to an incorrect idea or concept used by a programmer while coding. Syntax errors are grammatical errors whereas, logical errors are errors arising out of an incorrect meaning.
   For example: if a programmer accidentally adds two variables when he or she meant to divide them, the program will give no error and will execute successfully but with an incorrect result.
   Example: Accidentally using an incorrect operator on the variables to perform an operation (Using '/' operator to get the modulus instead using '%')

```
public class LErrorDemo {
    public static void main(String[] args)
    {
        int num = 789;
        int reversednum = 0;
        int remainder;

        while (num != 0) {

            // to obtain modulus % sign should
            // have been used instead of /
            remainder = num / 10;
```

```
            reversednum
                = reversednum * 10
                  + remainder;
            num /= 10;
        }
        System.out.println("Reversed number is "
                           + reversednum);
    }
}
```

Output:

```
Reversed number is 7870
```

Example: Displaying the wrong message

```
class IncorrectMessage {
    public static void main(String args[])
    {
        int a = 2, b = 8, c = 6;
        System.out.println(
            "Finding the largest number \n");

        if (a > b && a > c)
            System.out.println(
                a + " is the largest Number");
        else if (b > a && b > c)
            System.out.println(
                b + " is the smallest Number");

        // The correct message should have
        // been System.out.println
        //(b+" is the largest Number");
        // to make logic
        else
            System.out.println(
                c + " is the largest Number");
    }
}
```

Output:

```
Finding the largest number


8 is the smallest Number
```

Exceptions in Java

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Try –Catch Block

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

```
try{

   //statements that may cause an exception

}
```

While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
     //statements that may cause an exception
}
catch (exception(type) e(object))
{
     //error handling code
}
```

Multiple catch blocks in Java

```
class Example2{
   public static void main(String args[]){
     try{
         int a[]=new int[7];
         a[4]=30/0;
         System.out.println("First print statement in try block");
     }
     catch(ArithmeticException e){
        System.out.println("Warning: ArithmeticException");
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("Warning: ArrayIndexOutOfBoundsException");
     }
     catch(Exception e){
        System.out.println("Warning: Some Other exception");
     }
   System.out.println("Out of try-catch block...");
  }
}

o/p
Warning: ArithmeticException
Out of try-catch block...
```

Nested try catch block in Java

When a try catch block is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.

If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don’t handle exception.

```
Syntax:

try
{
```

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

Example

```
import java.util.InputMismatchException;
public class Nested {
    public static void main(String[] args) {
        try {
            System.out.println("Outer try block starts");
            try {
                System.out.println("Inner try block starts");
                int res = 5 / 0;
            } catch (InputMismatchException e) {
                System.out.println("InputMismatchException caught");
            } finally {
                System.out.println("Inner final");
            }
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught");
        } finally {
            System.out.println("Outer finally");
        }
    }
}
```

The output is

```
Outer try block starts
Inner try block starts
Inner final
ArithmeticException caught
Outer finally Outer finally
```

Java Finally block

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

Syntax of Finally block

```
try {
    //Statements that may cause an exception
}
catch {
   //Handling exception
}
finally {
   //Statements to be executed
}
```

A Simple Example of finally block

Here you can see that the exception occurred in try block which has been handled in catch block, after that finally block got executed.

```
class Example
{
   public static void main(String args[]) {
      try{
        int num=121/0;
        System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("Number should not be divided by zero");
      }
      /* Finally block will always execute
       * even if there is no exception in try block
       */
      finally{
        System.out.println("This is finally block");
      }
      System.out.println("Out of try-catch-finally");
   }
}
```

Output:

```
Number should not be divided by zero
This is finally block
Out of try-catch-finally
```

Chained exception in Java

Chained exception helps to relate one exception to other. Often we need to throw a custom exception and want to keep the details of an original exception that in such scenarios we can use the chained exception mechanism. Consider the following example, where we are throwing a custom exception while keeping the message of the original exception.

```
import java.io.IOException;

public class ChainedException
{
    public static void divide(int a, int b)
    {
      if(b == 0)
      {
        ArithmeticException ae = new ArithmeticException("top layer");
        ae.initCause(new IOException("cause"));
        throw ae;
      }
      else
      {
        System.out.println(a/b);
      }
    }

    public static void main(String[] args)
    {
      try
      {
        divide(5, 0);
      }
      catch(ArithmeticException ae) {
        System.out.println( "caught : " +ae);
        System.out.println("actual cause: "+ae.getCause());
```

```
        }
    }
}
```

```
caught:java.lang.ArithmeticException: top layer
actual cause: java.io.IOException: cause
```

Example 2

```
public class ChainedDemo1
{
    public static void main(String[] args)
    {
        try
        {

NumberFormatException a = new NumberFormatException("====> Exception");

a.initCause(new NullPointerException("====> Actual cause of the exception"));

            throw a;
        }

catch(NumberFormatException a)
        {

System.out.println(a);
System.out.println(a.getCause());
        }
    }
}
```

User-defined Custom Exception in Java

Java provides us facility to create our own exceptions which are basically derived classes of Exception. For example MyException in below code extends the Exception class.
We pass the string to the constructor of the super class- Exception which is obtained using "getMessage()" function on the object created.

```
// A Class that represents use-defined expception
class MyException extends Exception
{
```

```
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("My Mesaage for Excetion");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
o/p

Caught
My Mesaage for Excetion
```

throws keyword

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 has to handle the exception using a try-catch block.Syntax

To declare the list of exceptions, use throws keyword along with exception class names.

Syntax

```
public static void method( ) throws FileNotFoundException,
ConnectionException {

    //code

}
```

Example

```
class ThrowsExecp
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalAccessException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

```
Inside fun().
caught in main.
```

throw keyword

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

```
// Java program that demonstrates the use of throw
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

Output

```
Caught inside fun().
Caught in main.
```

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

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.

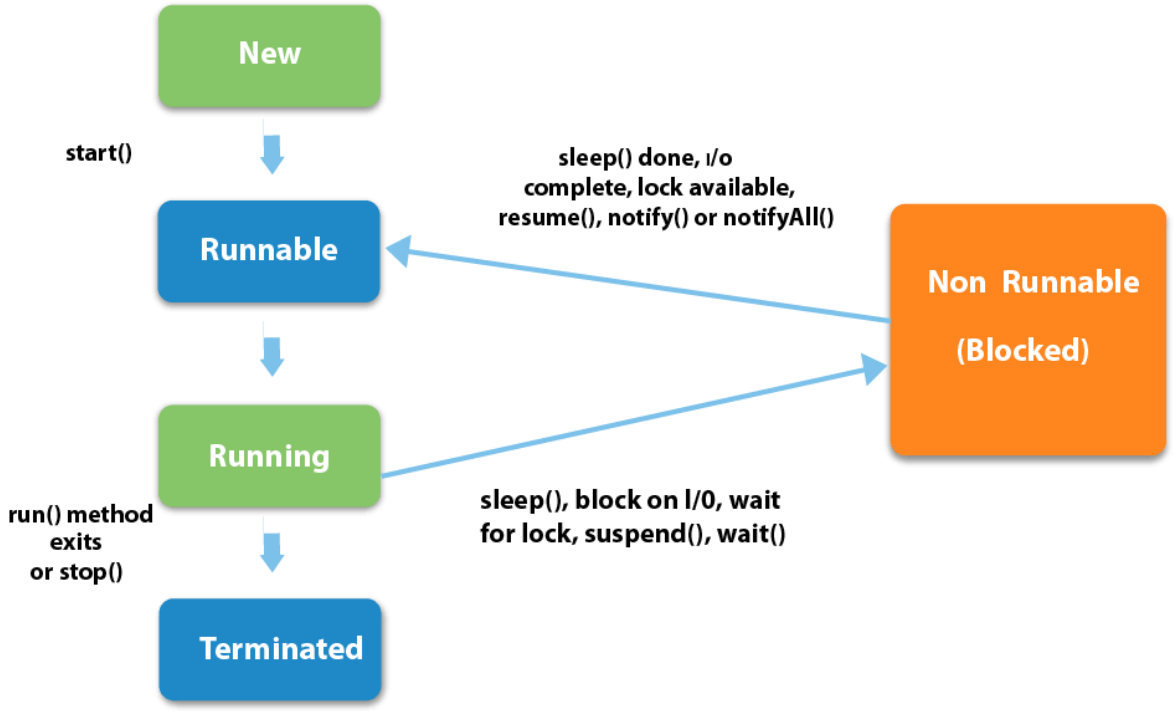
Thread

A **thread** in computer science is short for a **thread** of execution. **Threads are** a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks.
... **Threads are** lightweight, in terms of the system resources they consume, as compared with processes.

Threads in Java

Java threads facility and API is deceptively simple:
Every java program creates at least one thread [main() thread].
Additional threads are created through the Thread constructor or by instantiating classes that extend the Thread class.

Life cycle of a Thread (Thread States)



1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it's code is yet to be run and hasn't started to execute.
2. **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
3. **Blocked/Waiting state:**When a thread is temporarily inactive, then it's in one of the following states:
   - Blocked
   - Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

4. **Timed Waiting:** A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
5. **Terminated State:** A thread terminates because of either of the following reasons:
   - Because it exists normally. This happens when the code of thread has entirely executed by the program.
   - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

   A thread that lies in a terminated state does no longer consumes any cycles of CPU.

A thread can be in one of the five states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

Thread creation in Java

Thread implementation in java can be achieved in two ways:

1. **Extending the java.lang.Thread class**
2. **Implementing the java.lang.Runnable Interface**

Note: The Thread and Runnable are available in the java.lang.* package

1) By extending thread class

- The class should extend Java Thread class.

- The class should override the run() method.
- The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.
void run(): The new thread begins its life inside this method.

Example:

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

2) By Implementing Runnable interface

- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method

Example:

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

Thread Example Imp

```
class PrintEvenTask implements Runnable{
    Printer printer;
    int max;
    PrintEvenTask(Printer printer, int max){
        this.printer = printer;
        this.max = max;
    }
    @Override
    public void run() {
        for(int i = 2; i <= max; i+=2){
            printer.printEven(i);
        }
    }
}

class PrintOddTask implements Runnable{
    Printer printer;
```

```
    int max;
    PrintOddTask(Printer printer, int max){
        this.printer = printer;
        this.max = max;
    }
    @Override
    public void run() {
        for(int i = 1; i <= max; i+=2){
            printer.printOdd(i);
        }
    }
}

public class Printer {
    boolean evenFlag = false;
    Semaphore semaphoreEven = new Semaphore(0);
    Semaphore semaphoreOdd = new Semaphore(1);
    //Prints even numbers
    public void printEven(int num){
        try {
            semaphoreEven.acquire();
        } catch (InterruptedException e) {
            System.out.println("Thread Interrupted" + e.getMessage());
        }
        System.out. println(Thread.currentThread().getName() + " - " + num);
        semaphoreOdd.release();
    }

    //Prints odd numbers
    public void printOdd(int num){
        try {
            semaphoreOdd.acquire();
        } catch (InterruptedException e) {
            System.out.println("Thread Interrupted" + e.getMessage());
        }
        System.out. println(Thread.currentThread().getName() + " - " + num);
        semaphoreEven.release();
    }
    public static void main(String[] args) {
        Printer printer = new Printer();
        // creating two threads
        Thread t1 = new Thread(new PrintOddTask(printer, 10), "Odd");
        Thread t2 = new Thread(new PrintEvenTask(printer, 10), "Even");
        t1.start();
        t2.start();
    }
}
```

```
Output
Odd - 1
Even - 2
Odd - 3
Even - 4
Odd - 5
Even - 6
Odd - 7
Even - 8
Odd - 9
Even - 10
```

Example program to understand life cycle of thread

```
public class ThreadMethodsDemo extends Thread {
public void run() {
        for(int i=0;i<10;i++) {
        System.out.println(" thread methods
demo");
        try {
                System.out.println("thread is going to sleep");
                ThreadMethodsDemo.sleep(1000);
                System.out.println("thread wake up");
        } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
        }
}
public static void main(String[] args) throws InterruptedException {
        ThreadMethodsDemo de = new ThreadMethodsDemo();

System.out.println("getstate1"+de.getState());
        de.start();

System.out.println("getstate2"+de.getState());
        System.out.println("getstate3"+de.getState());
        System.out.println("getstate4"+de.getState());
        System.out.println("thread Name"+de.getName());
        System.out.println("thread Priority"+de.getPriority());
        System.out.println("getstate5"+de.getState());

}
}
```

Running state

Waiting state

New State

Runnable state

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Threads and exceptions

All uncaught exceptions are handled by code outside of the run() method before the thread terminates. The default exception handler is a Java method; it can be overridden.This means that it is possible for a program to write a new default exception handler. The default exception handler is the uncaughtException() method of the ThreadGroup class. It is called only when an exception is thrown from the run() method. The thread is technically completed when the run() method returns, even though the exception handler is still running the thread.

How to catch an Exception from a thread

```
class MyThread extends Thread{
   public void run(){
      System.out.println("Throwing in " +"MyThread");
      throw new RuntimeException();
   }
}
public class Main {
   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();
      try {
         Thread.sleep(1000);
      } catch (Exception x) {
         System.out.println("Caught it" + x);
      }
      System.out.println("Exiting main");
   }
}
Output :
Throwing in MyThread
Exception in thread "Thread-0" java.lang.RuntimeException
        at testapp.MyThread.run(Main.java:19)
Exiting main
```

Thread Priority

Every thread in Java has a priority that helps the thread scheduler to determine the order in which threads scheduled. The threads with higher priority will usually run before and more frequently than lower priority threads. By default, all the threads had the same priority, i.e., they regarded as being equally distinguished by the scheduler, when a thread created it inherits its priority from the thread that created it.

However, you can explicitly set a thread's priority at any time after its creation by calling its setPriority() method. This method accepts an argument of type int that defines the new priority of the thread. Its syntax is. **final void setPriority(int priority)**

For example: To set the thread to the maximum priority, the following statements should be used.

```
Thread t1 = new thread(task);
t1.setPriority(Thread.Max_Priority);
t1.start();
```

You can also determine the current thread priority by calling the getPriority() method. **final int getpriority()**

This method returns an in t value which indicates the current priority of the thread.

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

```
// Java program to demonstrate getPriority() and setPriority()
import java.lang.*;

class ThreadDemo extends Thread
{
    public void run()
    {
        System.out.println("Inside run method");
    }

    public static void main(String[]args)
    {
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        System.out.println("t1 thread priority : " +
                           t1.getPriority()); // Default 5
        System.out.println("t2 thread priority : " +
                           t2.getPriority()); // Default 5
        System.out.println("t3 thread priority : " +
                           t3.getPriority()); // Default 5

        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw IllegalArgumentException
```

```
        System.out.println("t1 thread priority : " +
                           t1.getPriority());  //2
        System.out.println("t2 thread priority : " +
                           t2.getPriority()); //5
        System.out.println("t3 thread priority : " +
                           t3.getPriority());//8

        // Main thread
        System.out.print(Thread.currentThread().getName());
        System.out.println("Main thread priority : "
                       + Thread.currentThread().getPriority());

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println("Main thread priority : "
                       + Thread.currentThread().getPriority());
    }
}
Output :

t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Main thread priority : 5
Main thread priority : 10
```

Java - Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Syntax

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Multithreading Example without Synchronization

```
class PrintDemo {
   public void printCount() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
      } catch (Exception e) {
         System.out.println("Thread  interrupted.");
      }
   }
}
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;
   ThreadDemo( String name,  PrintDemo pd) {
      threadName = name;
      PD = pd;
   }
      public void run() {
      PD.printCount();
      System.out.println("Thread " +  threadName + " exiting.");
   }
   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
public class TestThread {
   public static void main(String args[]) {

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
         try {
         T1.join();
         T2.join();
      } catch ( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
//This produces a different result every time you run this program
```

```
Output :
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1
exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2
exiting.
```

Multithreading Example with Synchronization

```
class PrintDemo {
   public void printCount() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
      } catch (Exception e) {
         System.out.println("Thread  interrupted.");
      }
   }
}
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;
   ThreadDemo( String name,  PrintDemo pd) {
      threadName = name;
      PD = pd;
   }
   public void run() {
      synchronized(PD) {
         PD.printCount();
      }
      System.out.println("Thread " +  threadName + " exiting.");
   }
   public void start () {
      System.out.println("Starting " +  threadName );
      if (t == null) {
         t = new Thread (this, threadName);
         t.start ();
      }
   }
}
public class TestThread {
   public static void main(String args[]) {
      PrintDemo PD = new PrintDemo();
      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );
      T1.start();
      T2.start();
      // wait for threads to end
      try {
         T1.join();
```

```
Output :
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1
exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2
exiting..
```

```
         T2.join();
      } catch ( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

Interthread Communication

Interthread communication is important when you develop an application where two or more threads exchange some information.

Java provide benefits of avoiding thread pooling using inter-thread communication. The **wait()**, **notify()**, and **notifyAll()** methods of Object class are used for this purpose. These method are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context.

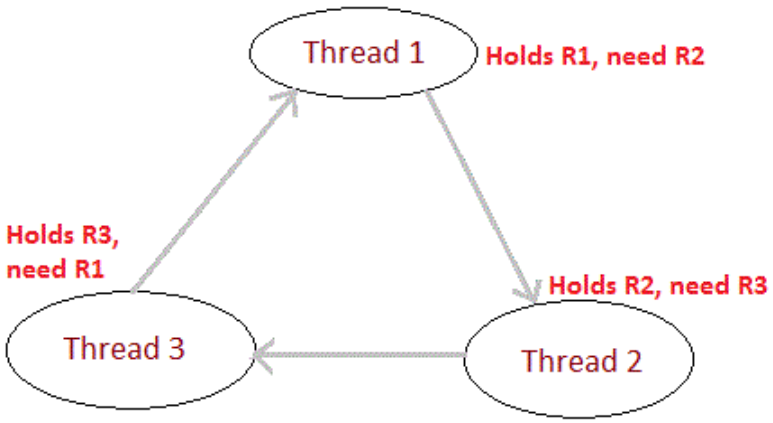
- **wait()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- **notify()** wakes up a thread that called wait() on same object.
- **notifyAll()** wakes up all the thread that called wait() on same object.

Example

```
class Chat {
   boolean flag = false;
   public synchronized void Question(String msg) {
      if (flag) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = true;
      notify();
   }
   public synchronized void Answer(String msg) {
      if (!flag) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      System.out.println(msg);
      flag = false;
      notify();
   }
}
class T1 implements Runnable {
   Chat m;
   String[] s1 = { "Hi", "How are you ?", "I am also doing fine!" };

   public T1(Chat m1) {
```

```
      this.m = m1;
      new Thread(this, "Question").start();
   }
   public void run() {
      for (int i = 0; i < s1.length; i++) {
         m.Question(s1[i]);
      }
   }
}
class T2 implements Runnable {
   Chat m;
   String[] s2 = { "Hi", "I am good, what about you?", "Great!" };

   public T2(Chat m2) {
      this.m = m2;
      new Thread(this, "Answer").start();
   }
   public void run() {
      for (int i = 0; i < s2.length; i++) {
         m.Answer(s2[i]);
      }
   }
}
public class TestThread {
   public static void main(String[] args) {
      Chat m = new Chat();
      new T1(m);
      new T2(m);
   }
}

Output :
Hi
Hi
How are you ?
I am good, what about you?
I am also doing fine!
Great!
```

Thread Deadlock in Java



Deadlock Condition

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2,

which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

Example of deadlock

```
class Pen{
}
class Paper{
}

public class Write {
  public static void main(String[] args)
  {
     final Pen pn =new Pen();
     final Paper pr =new Paper();

     Thread t1 = new Thread() {
        public void run()
        {
            synchronized(pn)
            {
                System.out.println("Thread1 is holding Pen");
                try{
                    Thread.sleep(1000);
                }
                catch(InterruptedException e){
                  // do something
                }
                synchronized(pr)
                {
                  System.out.println("Requesting for Paper");
                }
            }
        }
      };
      Thread t2 = new Thread() {
          public void run()
          {
              synchronized(pr)
              {
                  System.out.println("Thread2 is holding Paper");
                  try {
                      Thread.sleep(1000);
                  }
                  catch(InterruptedException e){
                      // do something
                  }
                  synchronized(pn)
                  {
                      System.out.println("requesting for Pen");
                  }
              }
          }
      };
      t1.start();
      t2.start();
  }
}
```

```
Output :
Thread1 is holding Pen
Thread2 is holding Paper
```