MODULE 4

Exception Handling

Exception Handling

An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error.

An error produces an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage all the possible error conditions in the program.

There are two types of errors –

- Compile-Time errors – detected during the compilation and can be corrected(syntax errors). Eg – missing semicolon, use of undeclared variables, missing double quotes in strings etc.
- Run-Time errors – Occurs during the execution of program. It is detected at run time only. Eg – divide by zero, array out of bounds, file not found, IO failures etc. Such errors are called exceptions.

If the exception is not handled properly, the interpreter will display an error message and will terminate the program. If the execution of the remaining program has to be continued, then the exceptions thrown by error conditions must be caught and appropriate message for taking corrective actions must be displayed. This process is called exception handling.

//A program that shows an unhandled divide-by-zero exception.

```
class Exc0
{
        public static void main (String args [])
        {
                int d = 0;
                int a = 42 / d;
                System.out.println (“This line is not printed in output”);
        }
}
```

When the interpreter detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, as the exception is not handled. As this exception that is not caught and handled in the program, this will be ultimately processed by the default handler. The default handler displays a string describing the exception and terminates the program.

Output of the above program is –

java.lang.ArithmeticException: / by zero

at Exc0.main(Exc0.java:6)

The class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **6**, are all included in the display.

Exception-Handling Fundamentals

When an exceptional condition arises in the program, an object representing that exception is created and *thrown* in the method that caused the error. Then, the exception is *caught* and processed.

Exceptions are thrown by Java because of fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

Java exception handling is managed via five keywords: **try, catch, throw, throws, and finally**. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some manner. To manually throw an exception, the keyword **throw** is used. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

General format of exception-handling

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, *Exception Type* is the type of exception that has occurred.

Note: All the different types of exception class are the subclass of the base class **Exception.**

Using try and catch

To guard the automatic abrupt termination of program and handle the run-time error, the code to be monitored is put inside a **try** block. Immediately following the **try** block, there must be a **catch** clause that specifies the exception type that is expected to occur.

To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **Arithmetic Exception** generated by the division-by-zero error:

```
class Exc2
{
        public static void main(String args[])
        {
```

```
            int d, a;
            try // monitor a block of code.
            {
                  d = 0;
                  a = 42 / d;
                  System.out.println("This will not be printed.");
            }
            catch (ArithmeticException e) // catch divide-by-zero error
            {
                  System.out.println("Division by zero.");
            }
            System.out.println("After catch statement.");
      }
}
```

Output –
Division by zero.
After catch statement.

Here **println( )** inside the **try** block is never executed, as exception occurs before that statement and the exception is thrown. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Once the **catch** statement has executed, program control continues with the next line in the program after the entire **try**/**catch** mechanism. As the **try** block throws a divide by zero, the **Arithmetic Exception** type is used in the catch block.

//A program to show the handling of exception and continue the execution of program.

Here for each iteration of the **for** loop obtains two random integers are taken. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
import java.util.Random;
class HandleError
{
      public static void main(String args[])
      {
            int a=0, b=0, c=0;
            Random r = new Random();
            for(int i=0; i<32000; i++)
      {
            try {
            b = r.nextInt();
```

```
                c = r.nextInt();
                a = 12345 / (b/c);
                }
                catch (ArithmeticException e)
                {
                        System.out.println("Division by zero.");
                        a = 0; // set a to zero and continue
                }
                System.out.println("a: " + a);
                }
        }
}
```

To display the description of an Exception, the preceding program can be rewritten as follows:

```
catch (ArithmeticException e)
{
        System.out.println("Exception: " + e);
        a = 0; // set a to zero and continue
}
```

For the above program, if there is a divide-by-zero error, then the following message is displayed–

Exception: java. lang.ArithmeticException: / by zero

Multiple catch Clauses

There can be cases that more than one exception can be raised by a single piece of code.

To handle this type of situation, two or more **catch** clauses can be written for a single **try** block, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block. The following example traps two different exception types:

```
class MultiCatch
{
        public static void main(String args[])
        {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;  // arithmetic exception may arise if no cmd-line arguments
                                                //are given
                        int c[] = { 1 };
                        c[42] = 99;       // array is of size 1, so array out of bound exception
                }
                catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                }
                catch(ArrayIndexOutOfBoundsException e) {
```

```
                System.out.println("Array index oob: " + e);
            }
            System.out.println("After try/catch blocks.");
        }
}
```

This program will cause a division-by-zero exception if there is no command-line parameters, since **a** will equal zero. It will survive the division if command-line argument are provided.

But it will cause an **Array Index Out Of Bounds Exception** since the **int** array **c** has a length of 1, but the program attempts to assign a value to **c[42]**.

Output –

```
C:\\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException: 42
After try/catch blocks.
```

Some Built-in Exceptions

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element a incompatible type variable
IllegalArgumentException	Illegal argument used to invoke a method.
NegativeArraySizeException	Array created with a negative size.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
ClassNotFoundException	Class not found.
InstantiationException	Attempt to create an object of an abstract class or interface.

throw

It is possible for a program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. There are two ways of obtaining a **Throwable** object: using a parameter of the **catch** clause, or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest **catch** statements are inspected to see if it matches the type of the exception. If a match is found, 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 message.

```
// Demonstrate throw.
class ThrowDemo
{
	public static void main(String args[])
	{
		try {
			throw new ArthimeticException();
		}
		catch(ArthimeticException e)
		{
			System.out.println("Caught an ArthimeticException.");
		}
	}
}
```

Output –
Caught an ArthimeticException.

In this program, an exception thrown from the **try** block explicitly is caught and handled in the **catch** block.

throws

If a method is capable of causing an exception, but does not handle that exception, then it must specify this behavior so that callers of the method can guard themselves against such exceptions.

Such conditions are handled by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw separated by comma.

Syntax –

```
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.

```
class ThrowsDemo
{
	static void throwOne() throws IllegalAccessException, ArithmeticException
	{	int d=0;
		System.out.println("Inside throwOne.");
		int a= 45/d;
		throw new IllegalAccessException);
	}
	public static void main(String args[])
	{
		try {
		    throwOne();
		}
		catch (IllegalAccessException e) {
```

```
            System.out.println("Caught " + e);
        }
        catch (ArithmeticException e) {
            System.out.println("Caught " + e);
        }

    }
}
```

Output –

```
Inside throwOne
Caught java.lang.ArithmeticExceptio
```

In this program the **throwOne( )** method invokes **IllegalAccessException** and **ArithmeticException,** but it is not handled in that method. So it is needed to declare that method **throwOne( )** throws **IllegalAccessException** and **ArithmeticException**. The calling method **main( )** must define a **try**/**catch** statement that catches this exception.

finally

finally creates a block of code that will be executed after a **try**/**catch** block has completed. 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 executed just before the method returns to the called function. The method may return to the called function due to return statement or due to unhandled exception. Whatever might be the case finally clause is executed.

This block is useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method.

The **finally** clause is optional. Each **try** statement requires at least one **catch** or a **finally** clause.

```
// Demonstrate finally.
class FinallyDemo
{
    static void procA() throws RuntimeException  // Through an exception out of the method.
    {
        try {
            System.out.println("inside procA");
            throw new RuntimeException( );
        }
        finally {   // executed before the function returns even if exception is thrown.
            System.out.println("procA's finally");
        }
    }

    static void procB()// Return from within a try block.
    {
        try {
            System.out.println("inside procB");
            return;
```

```
            }
            finally {  // executed before the function returns even if return statement is used.
                System.out.println("procB's finally");
            }
        }

        static void procC()   // Execute a try block normally.
        {
            try {
            System.out.println("inside procC");
            }
            finally {
            System.out.println("procC's finally");
            }
        }

        public static void main(String args[])
        {
            try {
                procA();
            }
            catch (Exception e) {
                System.out.println("Exception caught");
            }
            procB();
            procC();
        }
}
```

Output –

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

In this program, **procA( )** stops its execution by throwing an exception. The **finally** clause is executed before the method exits. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Multithreaded Programming

What is Java Multithreading?

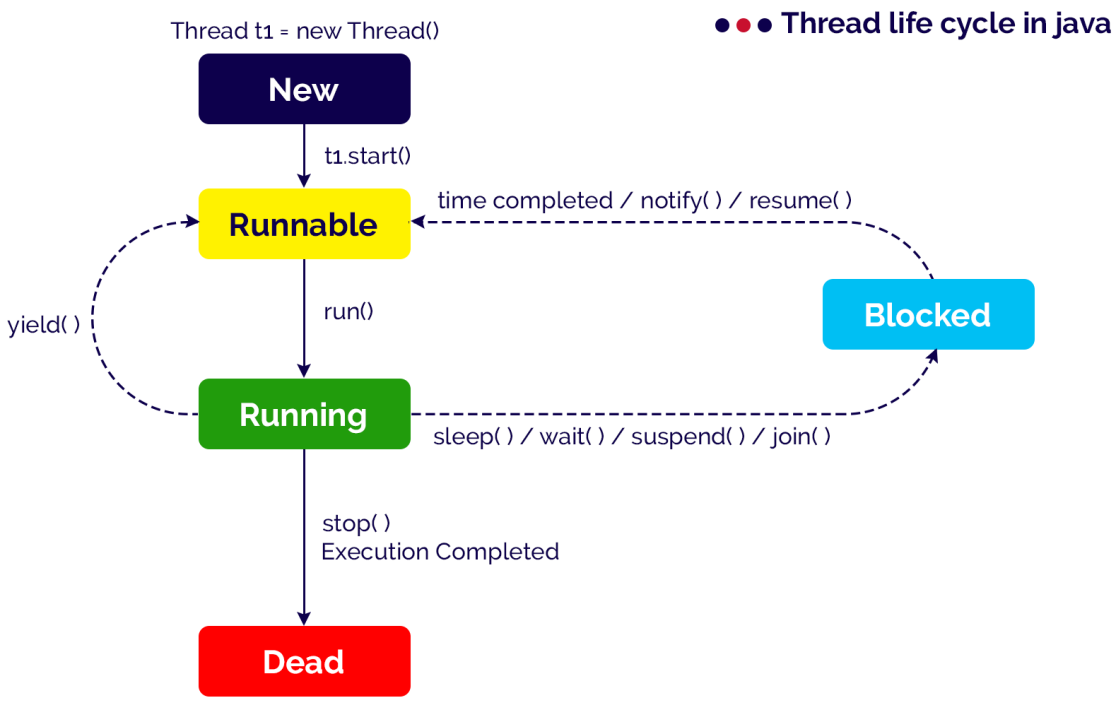
In Java, multithreading allows a program to execute multiple parts (threads) concurrently, maximizing CPU utilization and enabling efficient handling of tasks like background processes or interactive elements.

What is Java Thread Model?

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority.

Life cycle of a thread in java

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



New

When a thread object is created using new, then the thread is said to be in the New state. This state is also known as Born state.

Example:

```
Thread t1 = new Thread();
```

Runnable / Ready

When a thread calls start( ) method, then the thread is said to be in the Runnable state. This state is also known as a Ready state.

Ex:

```
t1.start( );
```

Running

When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

Blocked / Waiting

A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.

When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

Example:

```
Thread.sleep(1000);
wait(1000);
wait();
suspened();
notify();
notifyAll();
resume();
```

Dead / Terminated

A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called. The dead state is also known as the terminated state.

In java, a thread is a lightweight process. Every java program executes by a thread called the main thread. When a java program gets executed, the main thread is created automatically. All other threads called from the main thread.

The java programming language provides two methods to create threads, and they are listed below.

- **Using Thread class (by extending Thread class)**
- **Using Runnable interface (by implementing Runnable interface)**

Extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

- **Step-1**: Create a class as a child of Thread class. That means, create a class that extends Thread class.
- **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
- **Step-3**: Create the object of the newly created class in the main( ) method.
- **Step-4**: Call the start( ) method on the object created in the above step.

Look at the following example program.

```
class SampleThread extends Thread{

	public void run() {
		System.out.println("Thread is under Running...");
		for(int i= 1; i<=10; i++) {
			System.out.println("i = " + i);
		}
	}
}

public class My_Thread_Test {

	public static void main(String[] args) {
		SampleThread t1 = new SampleThread();
		System.out.println("Thread about to start...");
		t1.start();
	}
}
```

Implementing Runnable interface

The java contains a built-in interface Runnable inside the java.lang package. The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.

- **Step-1**: Create a class that implements Runnable interface.
- **Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.
- **Step-3**: Create the object of the newly created class in the main( ) method.
- **Step-4**: Create the Thread class object by passing above created object as parameter to the Thread class constructor.
- **Step-5**: Call the start( ) method on the Thread class object created in the above step.
  Look at the following example program.

```
class SampleThread implements Runnable{

	public void run() {
		System.out.println("Thread is under Running...");
		for(int i= 1; i<=10; i++) {
			System.out.println("i = " + i);
		}
	}
}

public class My_Thread_Test {

	public static void main(String[] args) {
		SampleThread threadObject = new SampleThread();
		Thread thread = new Thread(threadObject);
		System.out.println("Thread about to start...");
		thread.start();
	}
}
```

creating multiple threads in Java to achieve multitasking. In all the previous thread programs so far, we have created only two threads: main thread, and one new thread (often known as child thread).

When we run a Java program, the JVM automatically creates the main thread, which is responsible for executing the main() method. When we create child threads within the program, they execute concurrently with the main thread.

Let's understand the benefits of creating multiple threads in Java program with the help of a real-time example.

Suppose when we go to watch a movie in a theatre, generally, a person stay at door to check and cutting the tickets. When we enter the hall, there is one more person that stays to show seats to us.

Example:

```
// Two threads performing two tasks at a time.
public class MyThread extends Thread
{
// Declare a String variable to represent task.
  String task;

  MyThread(String task)
  {
   this.task = task;
  }
  public void run()
  {
   for(int i = 1; i <= 5; i++)
   {
     System.out.println(task+ " : " +i);
     try {
       Thread.sleep(1000); // Pause the thread execution for 1000 milliseconds.
     } catch(InterruptedException ie) {
         System.out.println(ie.getMessage());
       }
   } // end of for loop.
  } // end of run() method.
public static void main(String[] args)
{
// Create two thread objects to represent two tasks.
// Passing task as an argument to its constructor.
  MyThread th1 = new MyThread("Cut the ticket");
  MyThread th2 = new MyThread("Show your seat number");
```

```
// Create two objects of Thread class and pass two objects as parameter to constructor of Thread class.
   Thread t1 = new Thread(th1);
   Thread t2 = new Thread(th2);
   t1.start();
   t2.start();
  }
}
```

Explanation:

- In the preceding example program, we have created two threads on two objects of MyThread class. Here, we created two objects to represent two tasks.
- When we will run the above program, the main thread starts running immediately. Two threads will generate from the main thread that will perform two different tasks.
- When t1.start(); is executed by JVM, it starts execution of code inside run() method and print the statement “Cut the ticket” on the console.
- When JVM executes Thread.sleep(1000); inside the try block, it pauses the thread execution for 1000 milliseconds. Here. sleep() method is a static method that is used to pauses the execution of thread for a specified amount of time.
- For example, Thread.sleep(1000); will pause the execution of thread for 1000 milliseconds (1 sec). 1000 milliseconds means 1 second. Since sleep() method can throw an exception named InterruptedException, we will catch it into catch block.
- Meanwhile, JVM executes t2.start(); and second thread starts execution of code inside the run() method almost simultaneously. It will print the statement “Show your seat number”. Now, the second thread will undergo to sleep for 1000 milliseconds.
- When the pause time period of the first thread is elapsed, it will re-enter into the running state and starts the execution of code inside run() method. The same process will also happen for the second thread. In this manner, both threads will perform two tasks almost simultaneously.

Joining threads in Java

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

Ex:

final boolean isAlive()

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

Ex:

final void join() throws Interrupted Exception

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of the **join()** method, which allows us to specify a time for which you want to wait for the specified thread to terminate

Ex:.

final void join (long milliseconds) throws Interrupted Exception

Ex: is Alive() Method

```
public class MyThread extends Thread
{
	public void run()
	{
		System.out.println("r1 ");
		try {
		Thread.sleep(500);
		}
	catch(InterruptedException ie)
	{
		// do something
	}
	System.out.println("r2 ");
	}
	public static void main(String[] args)
	{
		MyThread t1=new MyThread();
		MyThread t2=new MyThread();
		t1.start();
		t2.start();
		System.out.println(t1.isAlive());
		System.out.println(t2.isAlive());
	}
}
```

Output:
r1
true
true
r1
r2
r2

Example of thread with join() method

In this example, we are using join() method to ensure that thread finished its execution before starting other thread. It is helpful when we want to executes multiple threads based on our requirement.

```
public class MyThread extends Thread
{
	public void run()
	{
		System.out.println("r1 ");
		try {
		Thread.sleep(500);
		}catch(InterruptedException ie){ }
		System.out.println("r2 ");
	}
	public static void main(String[] args)
	{
		MyThread t1=new MyThread();
		MyThread t2=new MyThread();
		t1.start();
		try{
			t1.join();	//Waiting for t1 to finish
		}catch(InterruptedException ie){}
		t2.start();
	}
}
```

Output:
r1
r2
r1
r2

Priority of a Thread in Java

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. You can get and set the priority of a Thread. Thread class provides methods and constants for working with the priorities of a Thread.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform-dependent.

Built-in Property Constants of Thread Class

Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

MIN_PRIORITY: Specifies the minimum priority that a thread can have.

NORM_PRIORITY: Specifies the default priority that a thread is assigned.

MAX_PRIORITY: Specifies the maximum priority that a thread can have.

Thread Priority Setter and Getter Methods

Thread.getPriority() Method: This method is used to get the priority of a thread.

Thread.setPriority() Method: This method is used to set the priority of a thread, it accepts the priority value and updates an existing priority with the given priority.

Example of Thread Priority in Java

```
package com. thread;
public class TestThread
{
public void printName()
{
System.out.println("Thread Name: " + Thread.currentThread().getName());
System.out.println("Thread Priority: " +Thread.currentThread().getPriority());
}
 public static void main(String args[])
{
 TestThread thread = new TestThread();
thread.printName();
 } }
```

Output:

Thread Name: main

Thread Priority: 5

Ex: In this example, we've created a ThreadDemo class which extends Thread class. We've created three threads. Each thread is assigned a priority. In run() method, we're printing the priorities and in output, it is reflecting in threads execution.

```
class ThreadDemo extends Thread {
  ThreadDemo( ) {
  }
  public void run() {
    System.out.println("Thread Name: " + Thread.currentThread().getName()
      + ", Thread Priority: " +Thread.currentThread().getPriority());
    for(int i = 4; i > 0; i--) {
      System.out.println("Thread: " + Thread.currentThread().getName() + ", " + i);
    }
    try {
      Thread.sleep(50);
    } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
  public void start () {
    super.start();
  }
}
public class TestThread {
  public static void main(String args[]) {
    ThreadDemo thread1 = new ThreadDemo();
    ThreadDemo thread2 = new ThreadDemo();
    ThreadDemo thread3 = new ThreadDemo();
    thread1.setPriority(Thread.MAX_PRIORITY);
    thread2.setPriority(Thread.MIN_PRIORITY);
    thread3.setPriority(Thread.NORM_PRIORITY);
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

Output:

Thread Name: Thread-2, Thread Priority: 5
Thread Name: Thread-1, Thread Priority: 1
Thread Name: Thread-0, Thread Priority: 10
Thread: Thread-1, 4
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-0, 4
Thread: Thread-1, 2
Thread: Thread-2, 3
Thread: Thread-0, 3
Thread: Thread-0, 2
Thread: Thread-0, 1
Thread: Thread-2, 2
Thread: Thread-2, 1
Thread: Thread-1, 1

Thread Synchronization in Java

Java programming language provides a very handy way of creating threads and synchronizing their task by using synchronized blocks. You keep shared resources within this block. Following is the general form of the synchronized statement.

Syntax

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

the **object identifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents.

```
class PrintDemo {
  public void printCount() {
    try {
      for(int i = 5; i > 0; i--) {
        Thread.sleep(50);
        System.out.println("Counter   ---   "  + i );
      }
    } catch (Exception e) {
      System.out.println("Thread  interrupted.");
    }
```

```
   }
}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo printDemo;

   ThreadDemo( String name, PrintDemo pd) {
      threadName = name;
      printDemo = pd;
   }

   public void run() {
      synchronized(printDemo) {
         printDemo.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 printDemo = new PrintDemo();

      ThreadDemo t1 = new ThreadDemo( "Thread - 1 ", printDemo );
      ThreadDemo t2 = new ThreadDemo( "Thread - 2 ", printDemo );
```

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

      // wait for threads to end
      try {
        t1.join();
        t2.join();
      } catch ( Exception e) {
        System.out.println("Interrupted");
      }
   }
}
```

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.
```