UNIT – V

EXCEPTION HANDLING

&

MULTITHREADED PROGRAMMING

5.1 Exception Handling

- **Exception:** The term exception means a problem that may arise during the execution of program. In other words, an exception is a run-time error.

- A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource busy (database, printer, fonts etc…).

- Exception Handling is the mechanism to handle runtime problems.

- We need to handle such problems to prevent abrupt termination of program.

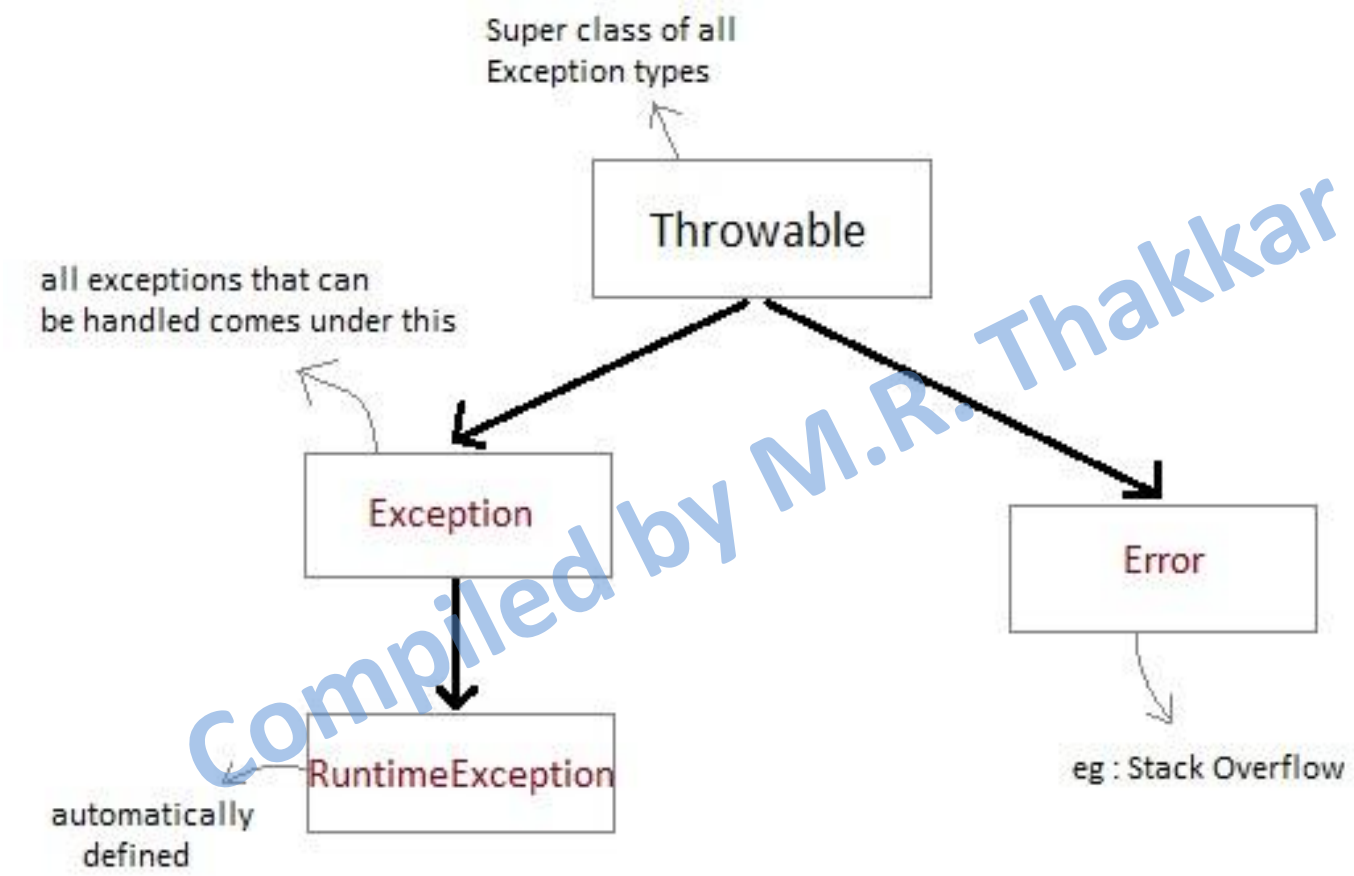
5.1.1 Types of Errors

- All exception types are subclasses of the built-in class **Throwable**. Thus, Throwable is at the top of the exception class hierarchy.

- Immediately below Throwable, are two subclasses that partition exceptions into two distinct branches:

1. **Exception class.**
2. **Error class.**

5.1.1 Types of Errors



5.1.1 Types of Errors

1. **Exception class:** This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.

— There is an important subclass of **Exception, called RuntimeException.** Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

2. **Error class :** This class defines exceptions that are not expected to be caught under normal circumstances by your program.

— Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.

— Stack overflow is an example of such an error.

5.1.2 Exceptions

- Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error.

```
class Example
{
    public static void main(String args[])
    {
        int a = 0;
        int b = 4 / a;
        System.out.println("b=" + b);
    }
}
```

5.1.2 Exceptions

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws this exception.*

- This causes the execution of **Example** to stop, because once an exception has been thrown, it must be *caught by an exception* handler and dealt with immediately.

- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

- Any exception that is not caught by your program will ultimately be processed by the default handler.

5.1.2 Exceptions

- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

```
java.lang.ArithmeticException: / by zero
        at Example.main(Example.java:6)
```

- Notice that the type of the exception thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error (/ by zero) happened.

- As discussed earlier, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

- Notice how the class name, **Example; the method name, main; the filename, Example.java;** and the line number, 6**, are all included in the simple stack trace.**

5.1.3 try..catch statement

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.

- Doing so provides two benefits:
  1. First, it allows you to fix the error
  2. Second, it prevents the program from automatically terminating.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

5.1.3 try..catch statement

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

```
class Example
{
        public static void main(String args[])
        {
                try  // monitor a block of code.
                {
                        int a = 0;
                        int b = 4 / 0;
                        System.out.println("b=" + b);
                }
                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.

5.1.3 try..catch statement

- Notice that the call to **println( ) inside the try block is never executed. 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 following the entire **try/catch mechanism.**

5.1.4 Multiple catch blocks

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more **catch clauses, 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.**

5.1.4 Multiple catch blocks

```
public class Example
{
    public static void main(String args[])
    {
        try
        {
            int a = 0;
            int c[] = { 1, 2 };

            int b = 4 / a;
            c[5] = 99;
        }
```

5.1.4 Multiple catch blocks

```
        catch(ArithmeticException e)
         {
            System.out.println(e);
        }
         catch( ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }

        System.out.println("After try/catch blocks.");
    }
}
```


Output:

```
java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

5.1.4 Multiple catch blocks

```
public class Example
{
    public static void main(String args[])
    {
        try
        {
            int a = 2;
            int c[] = { 1, 2 };

            int b = 4 / a;
            c[5] = 99;
        }
```

5.1.4 Multiple catch blocks

```
        catch(ArithmeticException e)
        {
           System.out.println(e);
        }
        catch( ArrayIndexOutOfBoundsException e)
        {
           System.out.println(e);
        }

        System.out.println("After try/catch blocks.");
    }
}
```


Output:

```
java.lang.ArrayIndexOutOfBoundsException: 5
After try/catch blocks.
```

5.1.5 throw & throws keywords

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- However, it is possible for your 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.**
- Non-Throwable classes, such as String and Object, cannot be used as exceptions.

5.1.5 throw & throws keywords

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

5.1.5 throw & throws keywords

```
class Example
{
	public static void main(String args[])
	{
		try
		{
		   throw new NullPointerException("demo");
		}
		catch(NullPointerException e)
		 {
		   System.out.println("Caught NullPointerException ");
		}

	}
}
```

5.1.5 throw & throws keywords

```
class Example
{
        static void  display()
        {
                try
                {
                    throw new NullPointerException("demo");
                }
                catch(NullPointerException e)
                 {
                    System.out.println("Caught NullPointerException.");
                }

        }

        public static void main(String args[])
        {
            display();
        }
}
```

5.1.5 throw & throws keywords

- **throws keyword:**

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

- You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw.

- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

5.1.5 throw & throws keywords

- **throws keyword:**

- This is the general form of a method declaration that includes a **throws clause:**

```
type method-name(parameter-list) throws exception-list
{
   // body of method
}
```


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

5.1.5 throw & throws keywords

- **throws keyword:**

- Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause, the program will not compile.

```
class ThrowsDemo
{
        static void display()  //give compile time error
        {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
           display();
        }
}
```

5.1.5 throw & throws keywords

- **throws keyword:**

- To make this example compile, you need to make two changes.

1. You need to declare that display( ) throws IllegalAccessException.
2. main( ) must define a try/catch statement that catches this exception.

- The corrected example is:

```
class ThrowsDemo
{
        static void display() throws IllegalAccessException
        {
                System.out.println("Inside Display method.");
                throw new IllegalAccessException("demo");
        }
```

5.1.5 throw & throws keywords

- **throws keyword:**

```
        public static void main(String args[])
        {
              try
              {
                 display();
              }
               catch (IllegalAccessException e)
              {
                 System.out.println("Caught " + e);
              }
        }
}
```

5.1.6 finally clause

- When exceptions are thrown, execution in a method takes non-sequential path that alters the normal flow through the method.

- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.

- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be by passed by the exception-handling mechanism.

- The **finally** keyword is designed to address this contingency.

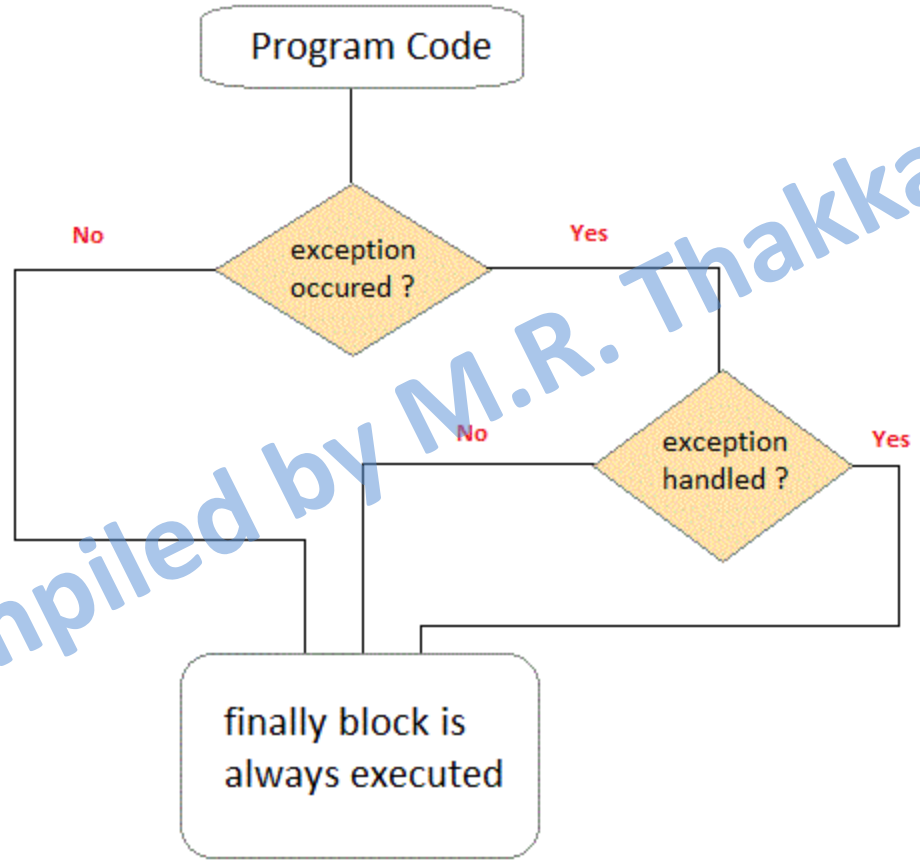
5.1.6 finally clause

- "finally" clause creates a block of statements that will be executed regardless of whether any exception occurred or not.

- A finally block appears at the end of try..catch block. The code of finally block always executes whether or not exception has occurred.

- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

5.1.6 finally clause



5.1.6 finally clause

```
public class Example
{
        public static void main(String args[])
        {
                try
                {
                        int a=4;
                        int b=0;
                        int c= a/b;
                }
                catch (ArithmeticException  e)
                {
                        System.out.println (“Exception: Divide by zero”);
                }
                finally
                {
                        System.out.println (“Inside Finally clause.");
                }
        }
}
```

Output:

Exception: Divide by zero

Inside Finally clause

.

5.1.6 finally clause

```
public class Example
{
        public static void main(String args[])
        {
                try
                {
                        int a=4;
                        int b=2;
                        int c= a/b;
                }
                catch (ArithmeticException  e)
                {
                        System.out.println (“Exception: Divide by zero”);
                }
                finally
                {
                        System.out.println (“Inside Finally clause.");
                }
        }
}
```

Output:

Inside Finally clause

5.1.6 finally clause

```
public class Example
{
        public static void main(String args[])
        {
                try
                {
                        int a=4;
                        int b=2;
                        int c= a/b;
                }
                finally
                {
                        System.out.println (“Inside Finally clause.");
                }
        }
}
```

Output:

Inside Finally clause

5.1.7 User defined exceptions

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.

- This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).

- The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable.**

- Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

- You may also wish to override one or more of these methods in exception classes that you create.

5.1.7 User defined exceptions

```
class MyException extends Exception
{
        String detail;

        MyException(String s)   //constructor
         {
            detail = s;
        }
        public String toString()
        {
            return  detail;
        }
}
```

5.1.7 User defined exceptions

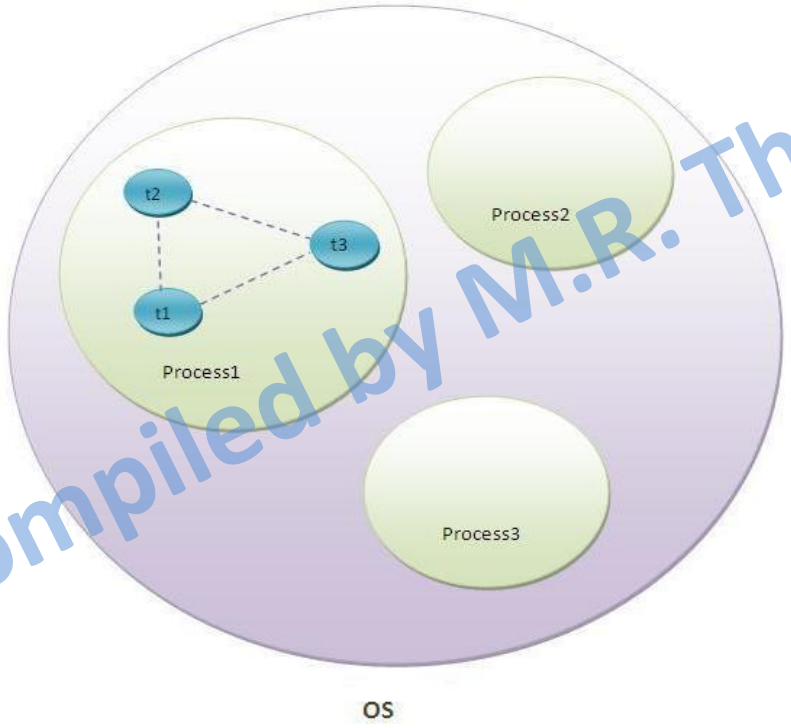
```
public class Example
{
    public static void main(String args[])
    {
        try
        {
            throw new MyException(“User defined Exception.");
        }
        catch (MyException e)
        {
            System.out.println("Caught: " + e);
        }
    }
}
```


Output:

Caught: User defined Exception.

5.2 Multithreaded programming

- **Thread:** A thread is a lightweight subprocess, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process.



- Thread is executed inside the process. There can be multiple processes inside the OS and one process can have multiple threads.

5.2 Multithreaded programming

- Java provides built-in support for multithreaded programming.

- A multithreaded program contains two or more threads that can run concurrently. Each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

5.2.1 Creating thread

- Java's multithreading system is built upon the Thread class and Runnable interface.
- Java defines two ways to create thread:
  1. **extend the Thread class**
  2. **implement the Runnable interface**
- The Thread class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

5.2.2 Extending Thread class

- The easiest way to create a thread is to create a new class that extends **Thread class,** and then to create an instance (object) of that class.
- The extending class must override the **run( ) method,** which is the entry point for the new thread.
- Inside run( ), you will define the code that will be executed by the new thread.
- It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.
- **It must also call start( ) to begin execution of the new thread.**

5.2.2 Extending Thread class

```
class NewThread extends Thread
{
        public void run()
        {
                System.out.println(“Inside Child thread.");
        }
}
```

5.2.2 Extending Thread class

```
public class Example
{
        public static void main(String[] args)
        {
                NewThread t1=   new NewThread();

                t1.start();

                System.out.println(“Inside Main thread.");
        }
}
```

5.2.2 Extending Thread class

Output:

Inside Main thread.

Inside Child thread.

5.2.3 implementing Runnable interface

- The second way to create a thread is to create a class that implements the **Runnable** interface.
- You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

```
public void run( );
```

- Inside run( ), you will define the code that will be executed by the new thread.
- It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.

5.2.3 implementing Runnable interface

- After you create a class that implements Runnable, you will instantiate an object of type Thread within that class.
- Thread defines several constructors. The one that we will use is shown here:

  Thread(Runnable *threadOb*)

- In this constructor, *threadOb is an instance of a class that implements the* ***Runnable*** interface.
- After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. In essence, start( ) executes a call to run( ).
- The start( ) method is shown here:

  void start( )

5.2.3 implementing Runnable interface

```
class NewThread implements Runnable
{
        public void run()
        {
                System.out.println(“Inside Child thread.");
        }
}
```

5.2.3 implementing Runnable interface

```
public class Example
{
        public static void main(String[] args)
        {
                NewThread t= new NewThread();
                Thread t1 = new Thread(t);  // create a new thread
                T1.start();

                System.out.println(“Inside Main thread.");
        }
}
```

5.2.3 implementing Runnable interface

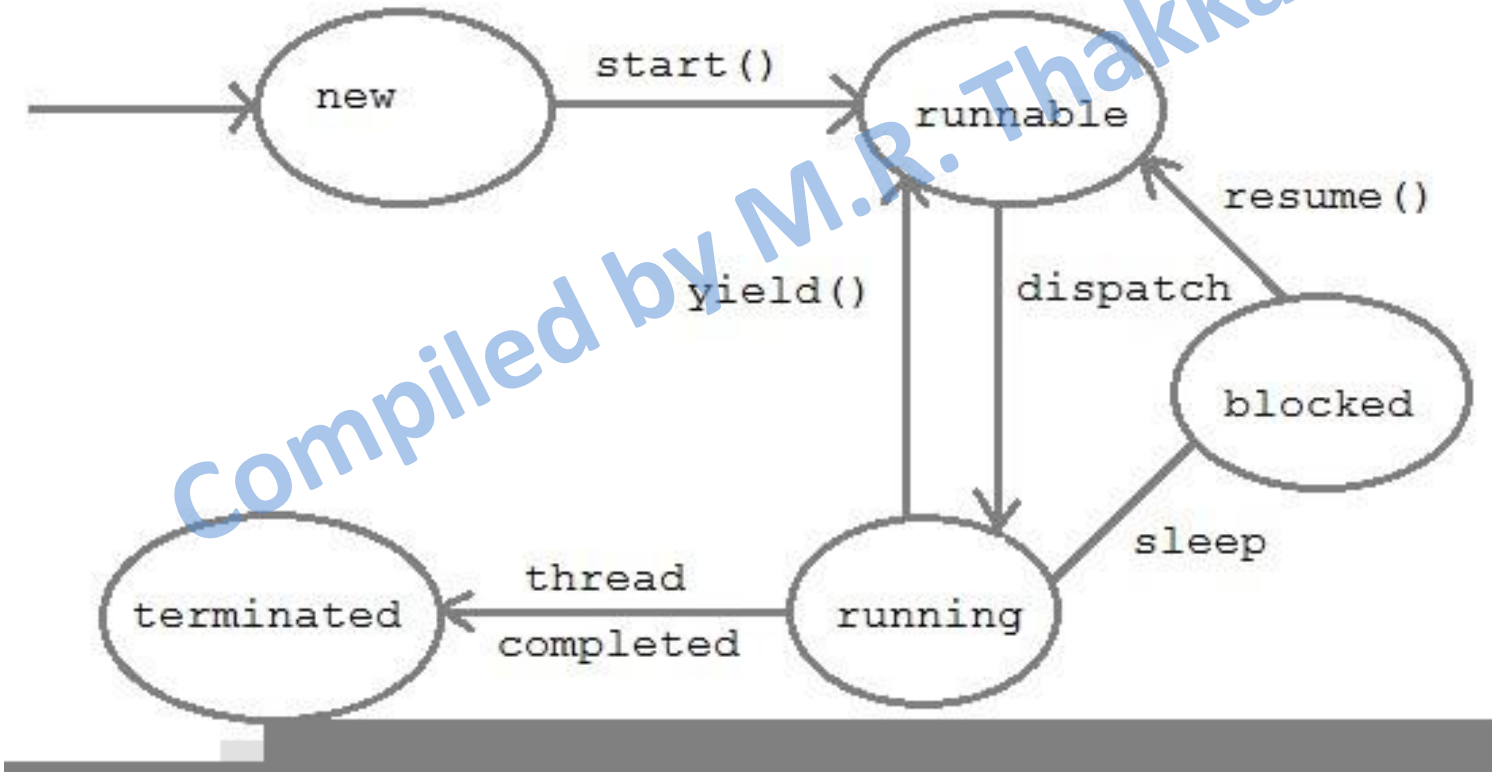
Output:

Inside Main thread.

Inside Child thread.

5.2.4 Life cycle of a thread

- When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states.



5.2.4 Life cycle of a thread

- **New**: A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

- **Runable**: After invocation of start() method on new thread, the thread becomes runable.

- **Running**: A method is in running thread if the thread scheduler has selected it.

- **Blocked**: A thread is waiting for another thread to perform a task. In this stage the thread is Blocked.

- **Terminated**:A thread enters the terminated state when it completes its task.

5.2.5 Thread Priority

- Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In Java thread priority ranges between,

- MIN-PRIORITY (a constant of 1)
- MAX-PRIORITY (a constant of 10)

Constant	Description
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 1)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

- By default every thread is given a NORM-PRIORITY(5). The main thread always has NORM-PRIORITY.

5.2.5 Thread Priority

- The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

5.2.5 Thread Priority

```
class NewThread implements Runnable
{
    public void run()
    {
        System.out.println("Inside Child thread.");

    }
}
```

5.2.5 Thread Priority

```
public class Example
{
    public static void main(String[] args)
    {
        NewThread t= new NewThread(); // create a new thread
        Thread t1 = new Thread(t);
        t1.start();

        t1.setPrioority(3);
        Thread mt = Thread.currentThread();

        System.out.println("Main thread Priority:" + mt.getPriority());
        System.out.println("Child thread Priority:" + t1.getPriority());
    }
}
```

5.2.5 Thread Priority

Output:

Main thread Priority:5

Inside child thread.

Child thread Priority:3

5.2.6 Thread Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization. As you will see, Java provides unique,* language-level support for it.

- You can synchronize your code in either of two ways. Both involve the use of the **synchronized keyword, and both are examined here.**

- Using Synchronized Methods
- Synchronized Block

5.2.6 Thread Synchronization

- **Using Synchronized Methods**
- Any method is specified with the keyword synchronized is only executed by one thread at a time.
- If any thread wants to execute the synchronized method, firstly it has to obtain the objects lock.
- If the lock is already held by another thread, then calling thread has to wait.

5.2.6 Thread Synchronization

- **Using Synchronized Methods (Example)**

- To understand the need for synchronization, let's begin with a simple example that does not use it—but should.

- The following program has three simple classes. The first one, Counter , has a single variable named **c** and single method named **increment()**.

- The increment() method increment the value of variable c by value 1 and then display resulting value.

- The interesting thing to notice is, after incrementing the variable by value 1 it calls **Thread.sleep(1000)**, which pauses the current thread for one second and then it display the resulting value.

5.2.6 Thread Synchronization

- **Using Synchronized Methods (Example)**

- The constructor of the next class, **NewThread**, takes a reference to an instance of the Counter class and stored in reference variable of class Counter.

- Finally, the **Example** class starts by creating a single instance of Counter, and three instances of NewThread, each of which create thread and the same instance of **Counter** is passed to each thread.

- Each thread calls the increment method of counter class which increments the value of c and display resulting value.

5.2.6 Thread Synchronization

- **Using Synchronized Methods**

```
class Counter
{
        int c=0;

        void increment()
        {
                c++;

                try
                {
                    Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                    System.out.println("caught: " + e);
                }

                System.out.println("C = " + c);
        }
}
```

5.2.6 Thread Synchronization

- **Using Synchronized Methods**

```
class NewThread extends Thread
{
        Counter obj;

        NewThread(Counter o)
        {
           obj = o;
        }

        public void run()
        {
           obj.increment();
        }
}
```

5.2.6 Thread Synchronization

- **Using Synchronized Methods**

```
public class Example
{
        public static void main(String[] args)
        {
                Counter c1 = new Counter();

                NewThread t1 = new NewThread(c1);
                t1.start();

                NewThread t2 = new NewThread(c1);
                t2.start();

                NewThread t3 = new NewThread(c1);
                t3.start();


        }
}
```

Output:

C = 3
C = 3
C = 3

5.2.6 Thread Synchronization

- **Using Synchronized Methods (Example)**

- Here, we were expecting the output to be:

  C = 1

  C = 2

  C = 3

- But, by calling **Thread.sleep( ),** the increment( ) method allows execution to switch to another thread.

- This results in the unexpected output as:

  C = 3

  C = 3

  C = 3

5.2.6 Thread Synchronization

- **Using Synchronized Methods (Example)**
- This issue can be resolved by synchronizing the execution of the increment method.
- To make the increment method synchronized, it is required to add the synchronized keyword at the method definition as shown below:

```
synchronized void increment()
{
        c++;
        try
        {
           Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
           System.out.println("caught: " + e);
        }
        System.out.println("C = " + c);
}
```

Output:

C = 1
C = 2
C = 3

5.2.6 Thread Synchronization

- **Using Synchronized Block (statements)**
- Another way of handling synchronization is Synchronized Blocks (Statements).
- Synchronized block must specify the object that provides the native lock.
- This is the general form of the **synchronized block:**

```
synchronized(object)
{
    // statements to be synchronized
}
```

5.2.6 Thread Synchronization

- **Using Synchronized Block (statements)**

- This earlier problem can also be resolved by synchronizing the execution of the increment method, using synchronized block as shown below:

```
public void run()
{
   synchronized (obj)
   {
            obj.increment();
   }
}
```

Output:

C = 1
C = 2
C = 3

5.2.7 Exception handling in threads

```
class NewThread extends Thread
{
    public void run()
    {
        try
        {
            throw new NullPointerException("Demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught Inside Run.");
        }
    }
}
```

5.2.7 Exception handling in threads

```
public class Example
{
	public static void main(String[] args)
	{
		NewThread t1 = new NewThread();
		 t1.start();
	}
}
```


Output:

Caught Inside Run.

****