❖ Multithreading in Java:

- The process of executing multiple threads simultaneously is known as Multithreading.
- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program. Each part of such program is called a thread
- Thread is basically a lightweight process.
- Multiple threads share a common memory area for their execution.
- They don't allocate separate memory space for each thread. That's via it is called as light weight process.
- Process is a heavy weight so we use Multithreading than Multiprocessing.
- Context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation etc.
- Due to Multi-threading, multiple activities can proceed concurrently in the same program.
- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU.
- Multitasking can be achieved by two ways:
  1. Process-based Multitasking (Multiprocessing)
  2. Thread-based Multitasking (Multithreading)
- Below diagram shows the Java Multithread program

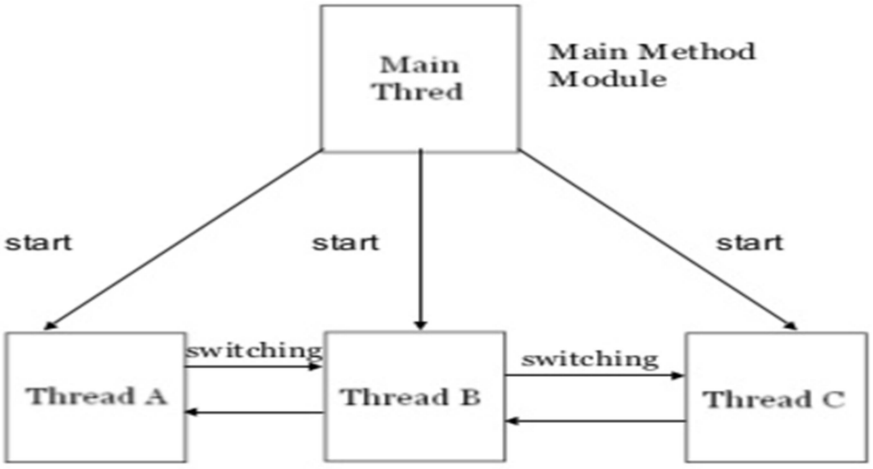


Fig. Java Multithread Program

Advantages of Java Multithreading:

- It doesn't block the user because threads are independent and you can perform multiple operations at same time.
- You can perform many operations together so it saves time.
- Threads are independent so it doesn't affect other threads if exception occurs in a single thread.

❖ Difference between Multithreading and Multiprocessing:

Multithreading	Multiprocessing
1) The process of executing multiple threads simultaneously is known as Multithreading	The process of executing multiple process simultaneously is known as Multiprocessing
2) Thread is lightweight	Process is heavyweight
3) Threads share the same address space	Each process allocates separate memory space
4) Cost of communication between the thread is low	Cost of communication between the process is high
5) Same job is divided into multiple threads and executed simultaneously.	Multiple jobs can execute simultaneously.
6) Creation of thread is not time-consuming	Creation of a process is time-consuming
7) Multithreading is not classified.	Multiprocessing can be symmetric or asymmetric.
8) It saves processor time and memory.	It does not save processor time and memory.

VJTECH Academy

❖ Life cycle of a Thread:

- A thread goes through various stages in its life cycle. The life cycle of the thread in java is controlled by JVM.
- The following diagram shows the complete life cycle of a thread.

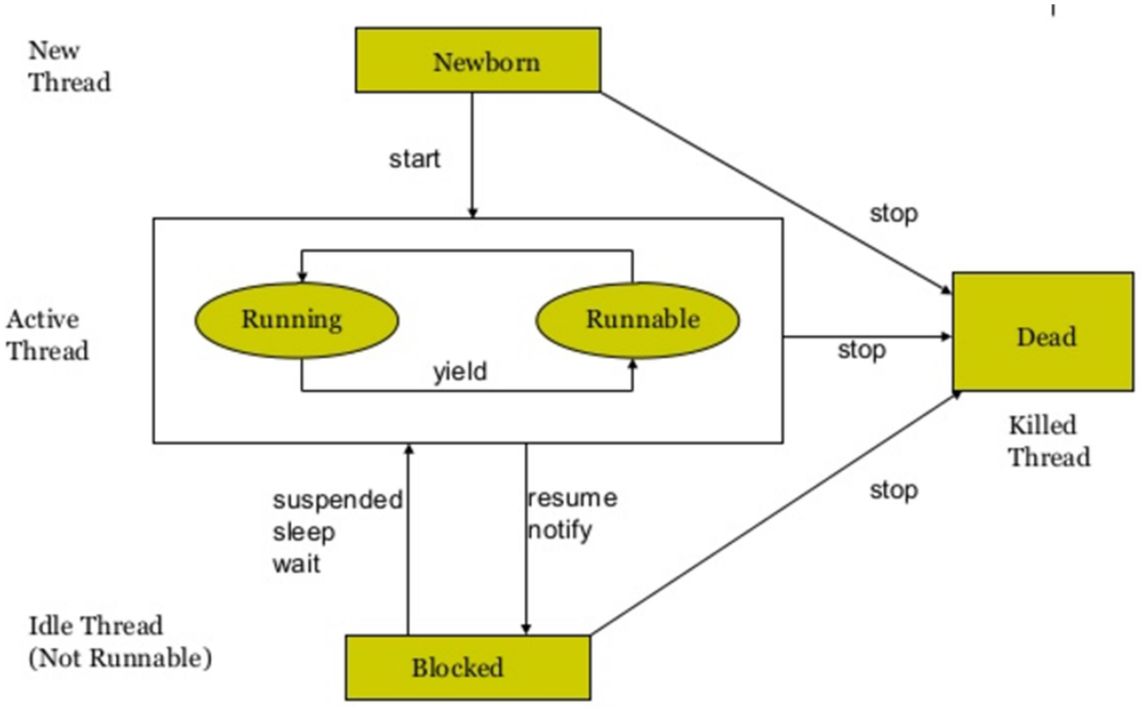


Fig. Life Cycle of Thread.

Following are the stages of the life cycle of a Thread.

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

1. Newborn state:

- When a thread object is created, the thread is born then called as thread is in Newborn state.
- A new thread begins its life cycle in this state.
- When thread is in Newborn state then you can pass to the Running state by invoking start() method or kill it by using stop() method.

2. Runnable state:

- The thread is in runnable state after invocation of start() method.
- The Runnable state of thread means that the thread is ready for the execution and waiting for the availability of the processor.
- The threads which are ready for the execution are managed in the queue.
- The same priority threads are processed on the basis of First-come-First-Serve.

3. Running state:

- The Running state means that the processor has given it's time to thread for their execution
- When thread is executing, its state is changed to Running.
- A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

4. Blocked state:

- A thread can be temporarily suspended or blocked from entering into runnable or running state.
- Thread can be blocked due to waiting for some resources to available.
- Thread can be blocked by using following thread methods:
  I. suspended()
  II. wait()
  III. sleep()
- Following are the methods used to entering thread into Runnable state from Blocked state.
  I. The resume() method is invoked in case of suspended().
  II. The notify() method is invoked in case of wait().
  III. When the specified time is elapsed in the case of sleep().

5. Dead state:

- The thread will move to the dead state after completion of its execution. It means thread is in terminated or dead state when its run() method exits.
- Also Thread can be explicitly moved to the dead state by invoking stop() method.

VJTECH Academy

❖ Creating a Thread in Java:

There are two different ways of creating the Thread in Java programming language.

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

1. By extending the Thread class:

We can create the thread by extending the Thread class. Thread class provide constructors and methods to create and perform operations on a thread.

Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Following steps are required to create the thread in Java by using Thread class:

I. Declare the class by extending the Thread class.

```
class ThreadX extends Thread
{
    //body of ThreadX class
}
```

II. Implement the run() method.

```
public void run()
{
    //thread code
}
```

III. Create the object of a thread class.

```
ThreadX t1=new ThreadX();
```

IV. Invoke the start() method.

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

Example:

```
//*********************************************************************************************
class ThreadX extends Thread
{
    public void run()
    {
          for(int i=1;i<=5;i++)
          {
                    System.out.println("From ThreadX:i="+i);
          }
    }
}

//*********************************************************************************************
class ThreadY extends Thread
{
    public void run()
    {
          for(int j=1;j<=5;j++)
          {
                    System.out.println("From ThreadY:j="+j);
          }
    }
}

//*********************************************************************************************
class ThreadDemo
{
    public static void main(String args[])
    {
          ThreadX t1=new ThreadX();
          ThreadY t2=new ThreadY();
          t1.start();
          t2.start();
    }
}
```

VJTECH Academy

2. By implementing Runnable interface.

We can create the thread by implementing the Runnable interface. Following steps are required to create the Thread in Java.

I. Declare the class by implementing the Runnable interface.

```
class RunnableX implements Runnable
{
    //body of ThreadX class
}
```

II. Implement the run() method.

```
public void run()
{
    //thread code
}
```

III. Create the object of a Thread class by passing object of class as argument which is implemented from the Runnable interface.

```
RunnableX r1=new RunnableX ();
Thread t1=new Thread(r1);
```

IV. Invoke the start() method.

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

Example:

```
//*************************************************************************************
class RunnableX implements Runnable
{
    public void run()
    {
          for(int i=1;i<=5;i++)
          {
                    System.out.println("From RunnableX:i="+i);
          }
    }
}

//*************************************************************************************
class RunnableY implements Runnable
{
    public void run()
    {
          for(int j=1;j<=5;j++)
          {
                    System.out.println("From RunnableY:j="+j);
          }
    }
}

//*************************************************************************************
class RunnableDemo
{
    public static void main(String args[])
    {
          RunnableX r1=new RunnableX();
          RunnableY r2=new RunnableY();

          Thread t1=new Thread(r1);
          Thread t2=new Thread(r2);
```

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

❖ **Important points between Thread Class vs. Runnable Interface**

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

❖ **Thread priority:**

- Each thread have a priority, Priorities are represented by a number between 1 and 10.
- Thread priorities are the integers which decide how one thread should be treated with respect to the others.
- Thread priority decides when to switch from one running thread to another, process is called context switching
- A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread below setPriority() method is used which is a method of the class Thread.

```
ThreadName.setPriority(int number);
```

Where, number is an integer value which between 1 to 10.

- getPriority() is used to retrieve the priority of the thread.
- In place of defining the priority in integers, we can use below three constants:
  - **MIN_PRIORITY**
  - **NORM_PRIORITY**
  - **MAX_PRIORITY**
- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

VJTECH Academy

Example:

```
//*********************************************************************************************
class ThreadX extends Thread
{
    public void run()
    {
         for(int i=1;i<=5;i++)
         {
                  System.out.println("From ThreadX:i="+i);
         }
    }
}

//*********************************************************************************************
class ThreadY extends Thread
{
    public void run()
    {
         for(int j=1;j<=5;j++)
         {
                  System.out.println("From ThreadY:j="+j);
         }
    }
}

//*********************************************************************************************
class ThreadZ extends Thread
{
    public void run()
    {
         for(int K=1;K<=5;K++)
         {
                  System.out.println("From ThreadZ:K="+K);
         }
    }
}


//*********************************************************************************************
class ThreadDemo
{
    public static void main(String args[])
    {
         ThreadX t1=new ThreadX();
         ThreadY t2=new ThreadY();
         ThreadZ t2=new ThreadZ();

         t1.setPriority(Thread.MIN_PRIORITY);
         t2.setPriority(Thread.NORM_PRIORITY);
         t3.setPriority(Thread.MAX_PRIORITY);

         t1.start();
         t2.start();
         t3.start();
    }
}
```

VJTECH Academy

❖ Exception handling in Java:

- Exception handling is one of the most important features of Java programming.
- It handles the runtime errors caused by exceptions.

Exception:

- An Exception is an unwanted event that interrupts the normal flow of the program.
- Exception may or may not be occur in program.
- When an exception occurs program execution gets terminated. In this case we get a system generated error message.
- When Java interpreter found the error such as divide by zero, the interpreter creates an exception object and throws it to inform that exception is occurred.
- The good thing about the exceptions is that they can be handled in Java.
- By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message.
- There can be several reasons that can cause a program to throw exception.
- For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user, divide by zero etc.

Exception Handling:

- Exception Handling is a mechanism to handle the runtime errors.
- If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user.
- For example look at the system generated exception below:

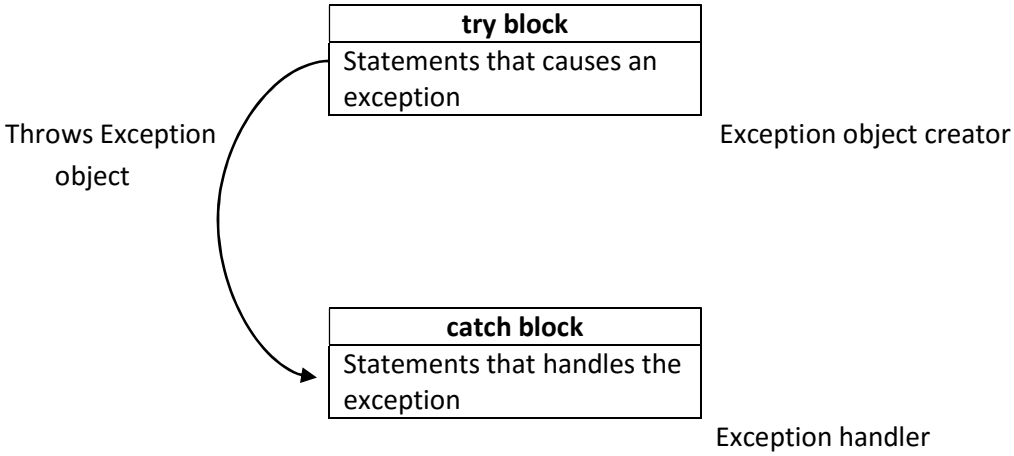
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo.main(ExceptionDemo.java:7)
```

- This message is not user friendly so a user will not be able to understand.
- So by using Exception handling mechanism, we handle an Exception and then print a user friendly warning message to user.
- The basic idea of exception handling mechanism is to find the exception, Throws the exception, Catch the exception and handle the exception.
- There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

VJTECH Academy

- Following diagram shows the exception handling mechanism:



- **Advantages of Exception Handling:**
  1. The main advantage of exception handling is to maintain the normal flow of the application.
  2. Exception normally disrupts the normal flow of the application that via we use exception handling. Let's take a scenario:

```
Statement 1;
Statement 2;
Statement 3;
Statement 4; //Exception occurs
Statement 5;
Statement 6;
```

Suppose there is 6 statements in your program and there occurs an exception at statement 4, rest of the code will not be executed i.e. statement 5 to 6 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

- **There are 5 keywords used in java exception handling.**

1. **try** – The statements which cause the exception are given in try block. The try block contains set of statements where an exception can occur. If exception is occurred then exception object is created and it is thrown. Syntax of try block is given below:

```
try
{
        //statements that may cause an exception
}
```

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

2. **catch** – The statements which handles the exception are given in catch block. This block catches the exception which is generated by the try block and handles it. This block must write after the try block. A single try block can have several catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. Syntax of catch block:

```
try
{
	//statements that may cause an exception
}
catch
{
	// Statements which handles the exception
}
```

3. **throw** – System generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception 'throw' keyword is used. The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.

```
throw ExceptionObject;
```

4. **throws** - throws is a keyword in Java which is used to indicate that this method might throw one of the listed type exceptions. Below is the general form of a method which includes a throws clause:

```
returntype method_name(parameters) throws exception_list
{
	//method body
}
```

5. **finally** – finally is a keyword in Java. finally block is always executed whether exception is handled or not. We can write the finally block after the try block or catch block. Finally block is optional.

```
try
{
	//statements that may cause an exception
}
catch
{
	// Statements which handles the exception
}
finally
{
	//Statements to be executed
}
```

- **Example:**

```
class ExceptionDemo
{
  public static void main(String args[])
  {
          int x,y,a=10,b=5,c=5;
          try
          {
                  System.out.println("try block execution begin");
                  x=a/(b-c);
                  System.out.println("try block execution end");
          }
          catch(ArithmeticException e)
          {
                  System.out.println("Divide by zero error is occurred");
          }
          finally
          {
                  System.out.println("Statements always executed");

          }
          y=a/(b+c);
          System.out.println("Y="+y);
  }
}
```

Output

```
try block execution begin
Divide by zero error is occurred
Statements always executed
Y=1
```

❖ **Multiple catch blocks:**

- A single try block can have several catch blocks. When an exception occurs in try block, then corresponding catch block which handles that particular exception is executed.
- Below syntax is used to write the multiple catch block:

```
try
{
  //statements that may cause an exception
}
catch
{
  // Statements which handles the exception
}
catch
{
```

```
  // catch block Statements 1
}
catch
{
  // catch block Statements 2

}
---
---
```

- **Example:**

```
class ExceptionDemo
{
  public static void main(String args[])
  {
          int x,y,a=10,b=5,c=5;
          try
          {
                  System.out.println("try block execution begin");
                  x=a/(b-c);
                  System.out.println("try block execution end");
          }
          catch(NullPointerException e)
          {
                  System.out.println("NullPointer exception is occurred");
          }
          catch(ArithmeticException e)
          {
                  System.out.println("Arithmetic exception is occurred");
          }
          catch(ArrayIndexOutOfBoundsException e)
          {
                  System.out.println("ArrayIndexOutOfBoundsException is occurred");
          }

          y=a/(b+c);
          System.out.println("Y="+y);
  }
}
```

Output

```
try block execution begin
Arithmetic exception is occurred
Y=1
```

❖ throw keyword in Java:

- System generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception 'throw' keyword is used.
- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.
- **Syntax** of throw keyword is given below

```
throw ExceptionObject;
```

- **Example:**

  In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException by using throw keyword otherwise print a message 'You are eligible for voting'.

```
class ThrowDemo
{

  void validate(int age)
  {
          if(age<18)
          {
                  throw new ArithmeticException("Not valid Age for voting");
          }
          else
          {
                  System.out.println("You are eligible for voting");
          }
  }
  public static void main(String args[])
  {
          ThrowDemo d1=new ThrowDemo();
          d1.validate(16);
  }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Not valid Age for voting
        at ThrowDemo.validate(Testtrycatch1.java:8)
        at ThrowDemo.main(Testtrycatch1.java:18)
```

Note: - No need to write the output in exam

❖ throws keyword in Java:

- throws is a keyword in Java which is used to indicate that this method might throw one of the listed type exceptions.
- Below is the general form of a method which includes a throws clause:

```
returntype method_name(parameters) throws exception_list
{
  //method body
}
```

- The throws do the same thing that try-catch does but there are some cases where you would prefer throws instead of try-catch. For example: Let's say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use try-catch as shown below:

```
void myMethod()
{
  try
  {
            // Statements that might throw an exception
  }
  catch (ArithmeticException e)
  {
            // Exception handling statements
  }
  catch (NullPointerException e)
  {
            // Exception handling statements
  }
}
```

- But suppose you have several such methods that can cause exceptions, in that case it would be difficult to write these try-catch for each method. The code will become unnecessary long and will be less-readable.

- One way to overcome this problem is by using throws keyword: declare the exceptions in the method using throws and handle the exceptions where you are calling this method by using try-catch.

```
void myMethod()throws ArithmeticException, NullPointerException
{
            // Statements that might throw an exception
}
```

- **Example**

```
class ThrowsDemo
{
  void myMethod(int num)throws ArithmeticException, NullPointerException
  {
     if(num==1)
        throw new ArithmeticException("ArithmeticException Occurred");
     else
        throw new NullPointerException("NullPointerException Occurred");
  }
  public static void main(String args[])
  {
           try
           {
                ThrowsDemo obj=new ThrowsDemo();
                obj.myMethod(1);
           }
           catch(Exception e)
           {
                System.out.println(e);
            }
  }
}
```

Output

```
java.lang.ArithmeticException: ArithmeticException Occurred
```

❖ **Difference between throw and throws in java:**

1. Throws keyword is used to declare an exception which works similar to the try-catch block. throw keyword is used to throw an exception explicitly.

2. throw keyword is followed by an instance of Exception class and throws is followed by exception class names.

3. throw keyword is used in the method body to throw an exception and throws keyword is used with the method to declare the exceptions that can occur in statement present in the method.

4. By using throw keyword we can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.

❖ User defined exception in Java:

- We know the different exception classes such as ArithmeticException, NullPointerException etc. but all these exception classes are predefined which will be executed when particular condition is occurred.
- For example, when you divide a number by zero it executes the ArithmeticException.
- In Java we can create our own exception class and throw that exception using throw keyword.
- These exceptions are known as user-defined exceptions.
- User-defined exception must extend Exception class.
- The exception is thrown by using throw keyword.

Example:

```
class MyException extends Exception
{
  Myexception(String msg)
  {
           super(msg);
  }
}

class ExceptionDemo
{

  public static void main(String args[])
  {
           try
           {
                //MyException m1=new MyException("Used defined Exception");
                //throw m1;
                  throw new MyException("Used defined Exception");

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

Output:

```
MyException: Used defined Exception
```