Unit-5 Exception Handling and Multithreaded programming

5.1

- Types of errors, exceptions,
- try...catch statement,
- multiple catch blocks,
- throw and throws keywords,
- finally clause, uses of exceptions,
- user defined exceptions

5.2

- Creating thread, extending Thread class, implementing Runnable interface
- life cycle of a thread
- Thread priority & thread synchronization
- exception handling in threads

Introduction Errors and Exception

- It is common to make mistakes while developing or in typing a program. A mistake leads to an error causing the program to produce unexpected results
- Errors are the wrongs that can make a program to produce unexpected or unwanted output.
- Error may produce
  - An incorrect output
  - may terminate the execution of program abruptly
  - may cause the system to crash
- It is important to detect and manage properly all the possible error conditions in program.

❖ **Types of error**

1. Compile time Errors
   Detected by javac at the compile time
2. Run time Errors
   Detected by java at run time

1. Compile Time Error (syntactical Error)

- Errors which are detected by **javac** at the compilation time of program are known as **<u>compile time</u>** errors.
- Most of compile time errors are due to typing mistakes, which are detected and displayed by **javac**.
- Whenever compiler displays an error, it will not create the **.class** file.
- So it is necessary to fix all the errors before we can compile and run the program.
- A single error may be the source of multiple errors.
- We should solve the earliest error in program. After fix an error, recompile the program and look for other errors.
- Typographical errors are hard to find.

The most common problems:

- Missing *semicolon*
- Missing or mismatch of *brackets* in classes and methods
- Misspelling of identifiers and keywords
- Missing double quotes in strings
- Use of undeclared variables
- Incompatible types in assignments/ Initialization
- Bad references to objects
- Use of = in place of = = operator

And so on…

Other errors are related to directory path. Errors such as

javac :command not found

Means we have not set path correctly where java executables are stored.

Example of compile time Errors

```
1.  class Error1
2.  {
3.          public static void main(String args[])
4.          {
5.                  System .out.println(“Hello”)   // Missing  ;
6.          }
7.  }
```

Javac detects an error and display it as follow:

```
Error1.java:  5: ‘; ‘expected
System.out.println (“Hello)
^
1 error
```

2. Run time Error (Logical Error)

- There is a time when a program may compile successfully and creates a .class file but may not run properly.
- It may produce wrong results due to wrong logic or may terminate due to errors like stack overflow, such logical errors are known as run time errors.
- Java typically generates an error message and aborts the program.
- It is detected by java (interpreter)
- Run time error causes termination of execution of the program.

The most common problems:

- Dividing an integer by zero
- Accessing element that is out of the bounds of an array
- Trying a store a value into an array of an incompatible class or type
- Trying to cast an instance of a class to one of its subclasses
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size of an array
- Using a null object reference as a legitimate object references to access a method or a variable
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string
- And many more

Example of run time errors

```
class Error2
{
public  static void main(String args[])
4.          {
5.                  int  a=10,b=5 ,c=5;
6.                  int x = a / (b-c);        // division by zero
7.                  System.out.println("x=" + x);
8.                  int y = a / (b + c );
9.          System.out.println("y=" + y);
10.         }
11.   }
```

- This code is syntactically correct, therefore does not cause any problem during compilation.
- While executing it displays following message and stops without executing further statement.
  Java.lang. ArithmeticException: / by Zero
  At Error2.main (Error2.java:6)

Difference between Compile time and Runtime Error

Sr. No.	Compile time error(syntax error)	Run time error(logical)
1	Errors which are detected by **javac** at the compilation time of program are known as **compile time** errors.	Errors which are detected by **java interpreter** at the run time are known as **run time** errors.
2	It is detected by javac compiler at the compile time	It is detected by java interpreter at run time
3	Whenever compiler displays an error, it will not create the **.class** file.	A program compiled successfully and creates a .class file but may not run properly.
4	It is also known as syntax error.	It is also called logical error.

Exceptions

- An **Exception** is abnormal condition that is caused by a runtime error in the program.
- When java interpreter encounters an error such as dividing by zero, it creates an exception object and throws it (means informs us that an error has occurred).
- If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program.
- If we want to the program to continue with the execution of remaining code, then we should <u>try to catch exception object thrown by the error condition</u> and <u>then display an appropriate message for taking corrective actions</u>. This is known as **Exception handling**.
- The purpose of exception handling is to provide a means to detect and report an “exceptional circumstances “so we can take proper actions.

- Error handling code consist of two segments

1) Detect errors and to throw exceptions

2) Catch the exceptions and take appropriate actions

<u>This mechanism performs following tasks in sequence</u>

1. Find the problem (**Hit** the Exception)
2. Inform that an error has occurred (**Throw** the Exception)
3. Receive the error information catch the exception)
4. Take corrective actions (**Handle** the Exception)

❖ **Common Java Exceptions** or **system defined exceptions**

No.	**Exception Type**	**Cause of Exception**
1	ArithmeticException	Caused by math error such as divide by zero
2	ArrayIndexOutOfBoundsException	Caused by use of bad array indexes(if index you are trying to access is not available in array)
3	ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
4	FileNotFoundException	Caused by an attempt to access a nonexistent file Caused by general I/O
5	IOException	failures, such as inability to read from a file
6	NullPointerException	Caused by referencing a null object
7	NumberFormatException	Caused when a conversion between strings and number fails
8	OutOfMemoryException	Caused when there's not enough memory to allocate a new object
9	SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security
10	StackOverflowException	Cause when the system runs out of stack space
11	StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Exception handling code (try – catch block)

The basic concept of Exception handling are throwing an exception and catching it.

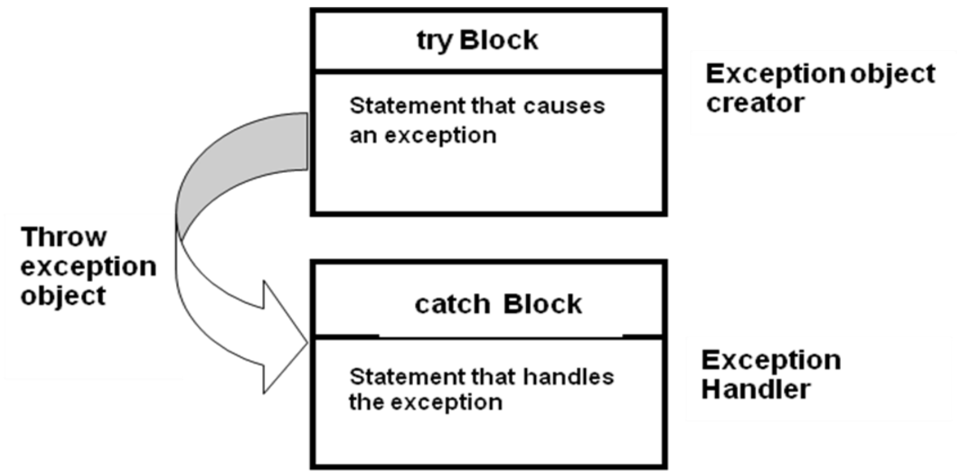


Fig. : Exception handling mechanism

try block

- Java uses a try block defined by keyword **try.**
- **try block contain block of code that causes an error condition and throw an exception.**
- try block can have one or more statements that could be generate an exception.
- If anyone statement generates an exception than remaining statements in block are skipped and execution jumps to the catch block that is placed next to try block.
- Every try statement should be followed by at least one catch statement; otherwise compilation error will occur.

catch block

- Java uses a catch block defined by keyword catch.
- catch block "catches" the exception "thrown" by the try block and handles it appropriately.
- catch block too can have one or more statements that are necessary to process the exception. The catch block is added immediately after the try block.
- catch statement works like a method definition. It is passed as single parameter which is reference to the exception object thrown.
- If catch parameter matches with the type of exception object, then exception is caught and that catch block will be executed.
- If catch parameter does not match with any type of exception then default exception handler will cause the execution to terminate.

Syntax:

```
      ………………………………
```

```
……………………………………
try
 {
Statement;       //generates an exception an throw it
 }
catch (Exception-type e)
{
Statement;   // catch and processes the exception
}
…………………….
…………………….
```

Example 1: using try-catch block

AIM: Write a program which generates **ArithmeticException** & handle it using try catch.

```
class  Error3
{
   public static void main (String args[])
    {
         int a=10,b=5,c=5;
         int x , y;
                  try
                  {
                  x=a / (b-c);                    //Exception here
                  }
                  catch (ArithmeticException e)
                  {
                  System.out.println ( " division by zero");
          }
         y= a /  (b +c );
         System.out.println ("y="+y);

     }

  }
```

Output:

```
Division by zero
y=1
```

Here, The program did not stop at the point of exceptional condition .It catches the error condition ,prints the error message and then continues the execution .

Example 2: handles Exception for invalid command line argument

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

```
      {
        int invalid =0;
        int num , valid=0;
    for ( int i=0; i< args.length ; i++)
      {
          try
              {
                      num = Integer.parseInt (args[i]);
              }
          catch (NumberFormatException  e)
             {
                      invalid =invalid + 1;
                     System.out.println (args[ i] + "  is Invalid Number" );
                     continue;
              }
              valid = valid +1;
         }
       System.out.println ("Total Valid Numbers =" + valid);
       System.out.println ("Total Invalid Numbers="+ invalid );
      }
  }
```

When we run the program with the command line:

```
javac ClineInput.java
java ClineInput  15   28.2   java  &   65
```

Output:

```
28.2 is Invalid Number
Java is Invalid Number
& is Invalid Number
Total Valid Numbers = 2
Total Invalid Numbers = 3
```

- **Multiple catch statements**

We can use more than one catch statements (blocks) with a single try block.

Syntax:

```
 ………………..
try
```

```
{
    Statement; // generates an exception
}
catch ( Exception-type-1 e)
{
Statements  ; // processes exception type 1
}
catch (Exception-type-2 e)
{
Statements ; // processes exception type 2
}
…………..
 catch (Exception-type-N e)
{
Statements; // processes exception type N
}
………….
………….
```

- In case of multiple catch statements they are treated as switch statement.
- The first statement whose parameter matches with the exception object will be executed and remaining statements will skipped.
- Java does not require any processing of the exception at all.
- We can use **catch statement with an empty block** to avoid program abortion. It is default Exception Handler.

```
catch (Exception e);          or

 catch (Exception e)
 {
 }
```

This statement will perform anything, it catch an exception and then ignore it.

Example 3: ArrayIndexOutOfBoundsException using multiple catch block

```
class Error4
{

public static void main(String args[])
{
int a [ ]= {5,10};
int b=5;
    try
          {
                int  x = a[2] / b – a[1];
            }
        catch (ArithmeticException  e)
            {
                System.out.println (“Division by zero”);
            }
        catch (ArrayIndexOutOfBoundsException e)
            {
                System.out.println (“Array index error”);
            }
        catch(ArrayStoreException e)
            {
                System.out.println (“Wrong data type”);
            }
            int y= a[1] / a[0];
            System.out.println (“y= “+y);
        }
}
```

Output:

```
Array index error

Y=2
```

Note: When exception object matches to any catch block, it will **catch** and handle the error. Remaining catch blocks are skipped.

❖ Use of finally statement

- **finally** statement can be used to handle an exception that is not caught by any of the previous catch statements.
- finally block can be used to handle any exception generated within a try block.
- It may be added **immediately after try block** or **after the last catch block.**
- When finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown.
- **finally** statement used to perform operation like closing files and releasing system resources.

```
try
{
.............
.............
}
finally
{
..................
.................
}
```

(1)

```
try
{
..........
}
catch(.......)
{
...............
}
catch (.......)
{
..........
}
.
.
finally
{
..............
}
```

(2)

We can use finally in this two way..

Example 4: try –catch –finally block

```
class Error4
{
public static void main(String args[])
{
    int a [ ]= {5,10};
    int b=5;
    try
    {
            int x = a[2] / b – a[1];
    }
    catch (ArithmeticException  e)
    {
            System.out.println (“Division by zero”);
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
            System.out.println (“Array index error”);
    }
    catch(ArrayStoreException e)
    {
            System.out.println (“Wrong data type”);
    }
    finally
    {
            int y=a [1] /a[0];
            System.out.println (“y =” +y);
    }
  }
}
```

Output:

```
 Array index error
Y=2
```

❖ Nested try-catch block

- Nested try statement means that try statement within a block of another try.
- If an inner try statement does not have a catch handler for a particular exception, then next try statement's catch handlers are inspected for a match.
- This continues until one of catch statements succeeds and until all of nested try { } statements are exhausted.
- If no catch statement matches then the Java-run time system will handle the exception.

Example 5: nested try-catch block

```
class Nested_try
{
    public static void main (String args[])
    {
     try
       {
            int a=2,b=4,c=2,x=7,z;
            int p[]={2};

            try
            {
                    z= x / ((b*b)-(4*a*c));
                    System.out.println("The value of z is =" +z);
            }
            catch (ArithmeticException e)
            {
            System.out.println("Division by zero in Arithmetic expression");
            }
            }
    catch (ArrayIndexOutOfBoundsException e)
    {
            System.out.println("Array index is out-of-bounds");
    }
    }
}
```

Output:

```
Division by zero in Arithmetic expression
```

Type of Exception

- All types of exceptions are subclasses of built in class Throwable.
- Throwable class is contained in java.lang package
- Errors are thrown by any methods of Java API or by java virtual machine.

- Exception is a super class of all types of exceptions.
- When we use multiple catch statements, exception subclasses must come before any of their super classes because catch statement that uses a super class will catch exception of that type plus any of its subclass.
- Thus, a subclass would never be reached if it came after its superbclass, it will give unreachable code.

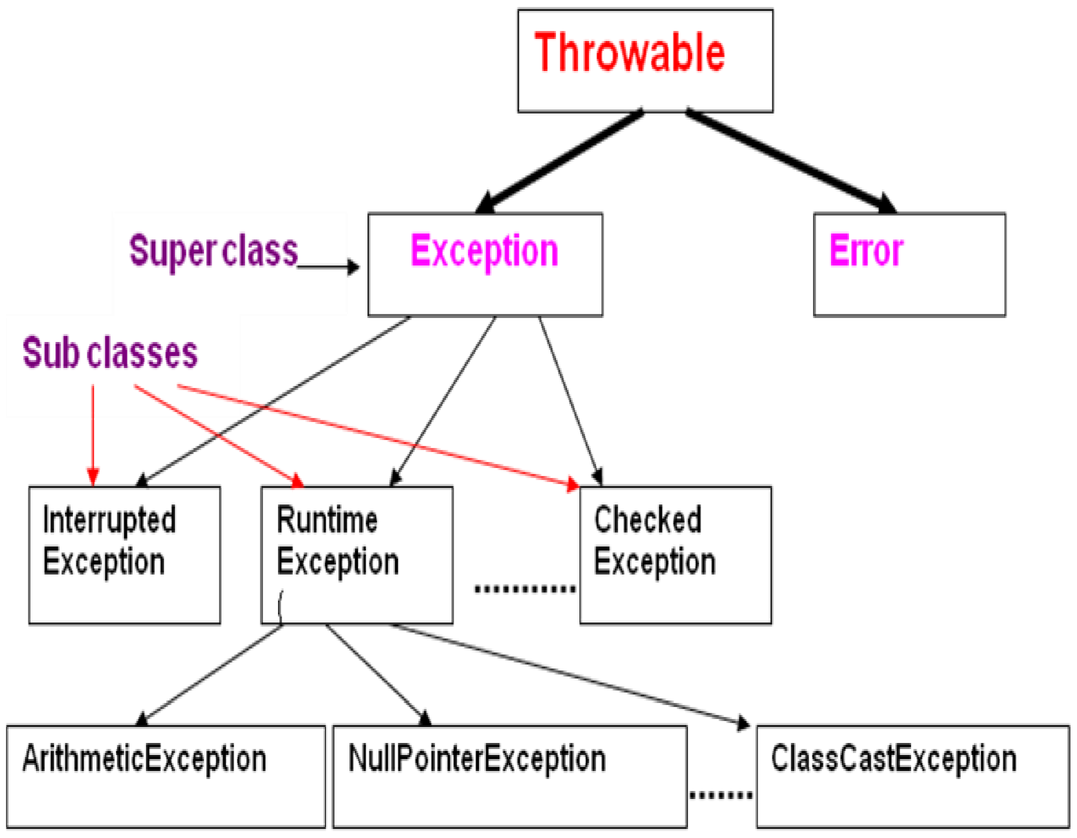


Fig. Java Exception hierarchy

Throwing own exception

- We can make a program to throw an exception explicitly using throw statement.

  throw throwableInstance;

- throw keyword throws a new exception.
- An object of class that extends throwable can be thrown and caught.

  Throwable $\rightarrow$ Exception $\rightarrow$ MyException

- The flow of execution stops immediately after the throw statement ;
- Any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has catch statement that matches the type of exception.
- If it matches, control is transferred to that statement
- If it doesn't match, then the next enclosing try statement is inspected and so on. If no catch block is matched than default exception handler halts the program.

Create java's standard exception objects using **throw keyword**

`Syntax: **throw** new thowable_instance;

Example: throw new ArithmeticException ();

throw new MyException();

- Here, new is used to construct an instance of MyException ().
- All java's runtime exception s have at least two constructors:

  1) One with no parameter

  2) One that takes a string parameter.

In that the argument specifies a string that describes the exception. This string is displayed when object is used as an argument to print () or println ( ).

It can also obtained by a call to getMessage ( ), a method defined by Throwable class

Example 6 : Throwing user defined exception

```
import    java.lang. Exception ;

class MyException extends Exception
{
        MyException (String message )
        {
                super (message);
        }
}
class TestMyException
{
        public static void main (String args[])
        {
                int x=5, y=1000;

                try
                {
                        float z=(float) x/ (float) y;
                        if (z < 0.01)
                        {
                                throw new MyException (“Number is too small”);
                        }
                }

                catch (MyException e)
                {
                        System.out.println (“Caught my exception”);
                        System.out.println (e.getMessage ( ));
                }
                finally
                {
                        System.out.println (“I AM ALWAYS HERE”);
                }
        }
}
```

Output:

```
Caught my exception
Number is too small
I AM ALWAYS HERE
```

Example 7 : Throwing user defined exception

```
import  java.lang. Exception;

class NoMatchFoundException extends Exception
{
        NoMatchFoundException (String message)
          {
                  super (message);
          }
}
class Nested_try
{
          public static void main (String args[])
          {
          String s=args[0];
          try
          {
                  if(!s.equalsIgnoreCase("india"))
                  {
                          throw new NoMatchFoundException (" String doesnt match !!!!
                          enter country name : India");
                  }

                  else
                  {
                          System.out.println(" country Name = India ");
                  }
          }
          catch(NoMatchFoundException e)
          {
                   System.out.println (e.getMessage());
          }

          }
}
```

Output:

```
String doesnt match !!!!  enter country name : India
```

throws keyword

- **throws** keyword declares that a method may throw particular exception.
- If method is capable of causing an exception that it does not handle.
- It must specify this behavior so that callers of that method can guard themselves against that exception.
- We can do this by including throws class in the methods declaration as follows.

SYNTAX:

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

Example 7 : use of throws keyword with method

```
class MyThread
{
        void throwone() throws IllegalAccessException
        {
                System.out.println("exception thrown by method");
                throw new IllegalAccessException();
        }
}
class ThrowsDemo
{       public static void main(String args[])
        {
                MyThread m=new MyThread();
                try
                {
                m.throwone();
                }
                catch(IllegalAccessException e)
                {
                        System.out.println("caught");
                }
        }  }
------------------------------Example-2 ----------------------------------
class MyThread
{
        void throwone() throws ArithmeticException
        {
                System.out.println("exception thrown by method");
                int ans=10/0;
                //throw new ArithmeticException();
        }
}
class ThrowsDemo
{       public static void main(String args[])
        {
                MyThread m=new MyThread();
                try
                {
                        m.throwone();
                }
                catch(ArithmeticException e)
                {
                        System.out.println("caught");
                }
        } }
```

Advantage

Exception handling provides the following advantages over "traditional" error management techniques:

- Separating Error Handling Code from "Regular" Code.
- Propagating Errors Up the Call Stack.
- Grouping Error Types and Error Differentiation.

Using exception for debugging

- Exception handling mechanism can be used to hide errors from rest of the program.
- Exception handling mechanism may be effectively used to locate the type and place of errors.

Sr. 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

❖ Multithreading

- Program is a sequence of statements.
- A Program may be divided into two or more sub programs, this sub programs are called process.
- Many Processes can be implemented at the same time in parallel (parallel Execution).
- Process can be divided into further small parts in its sub process (threads).
- That sub-process is called Thread.
- We can say "**Thread" is a smallest unit of a program.**

Important terms

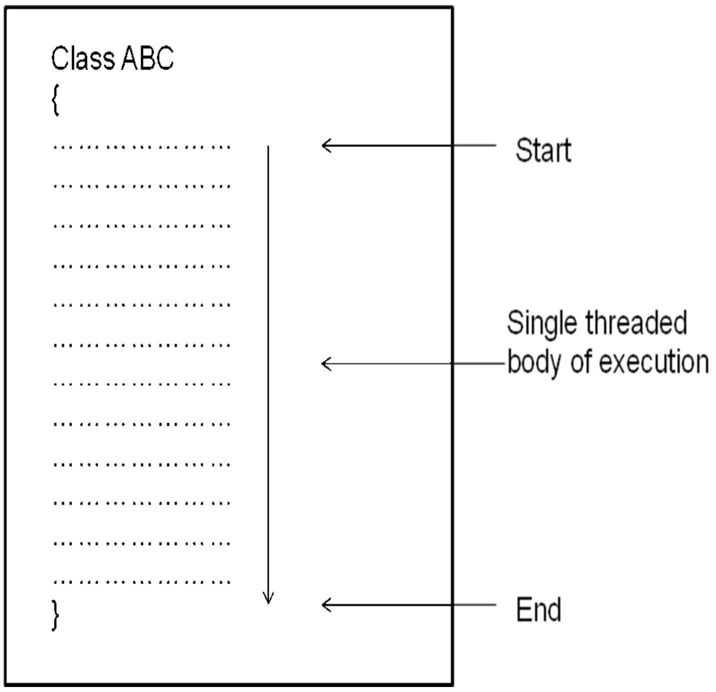
- **Multiprogramming**: More than one program running at same time. That can share same Processor.
- **Multiprocessing**: two or more processes of a program running at same time. They can process on the own (different) memory location.
- **Multithreading**: two or more thread of a process executing in parallel. That can share same memory location in the memory.
- **Multitasking**: multiple task performed at a time

Difference between Thread based and process base Multitasking

NO.	Thread based Multitasking	Process based Multitasking
1	In thread based multitasking thread is the smallest unit of code.	Process is smallest unit of code that can be dispatched by multitasking.
2	Process is dividing into number of threads. (light weight process)	Program is divided into number of processes. (Heavy weight process)
3	Single program can perform two or more task simultaneously.	Process based multitasking allows your computer to run two or more program.
4	Each thread of same process shared the same state, same memory space and can communicate with each other directly. Because they share same variable. So threads are known as Light weight process.	Each process has independent execution units that contain their own state information, address spaces, and interact with each other via IPC.
5	E.g. Text editor can format text a t the same time that it is printing that document. These two actions are being performed by two separate threads	You can run java compiler at the same time that you are using a text editor

❖ Single threaded program

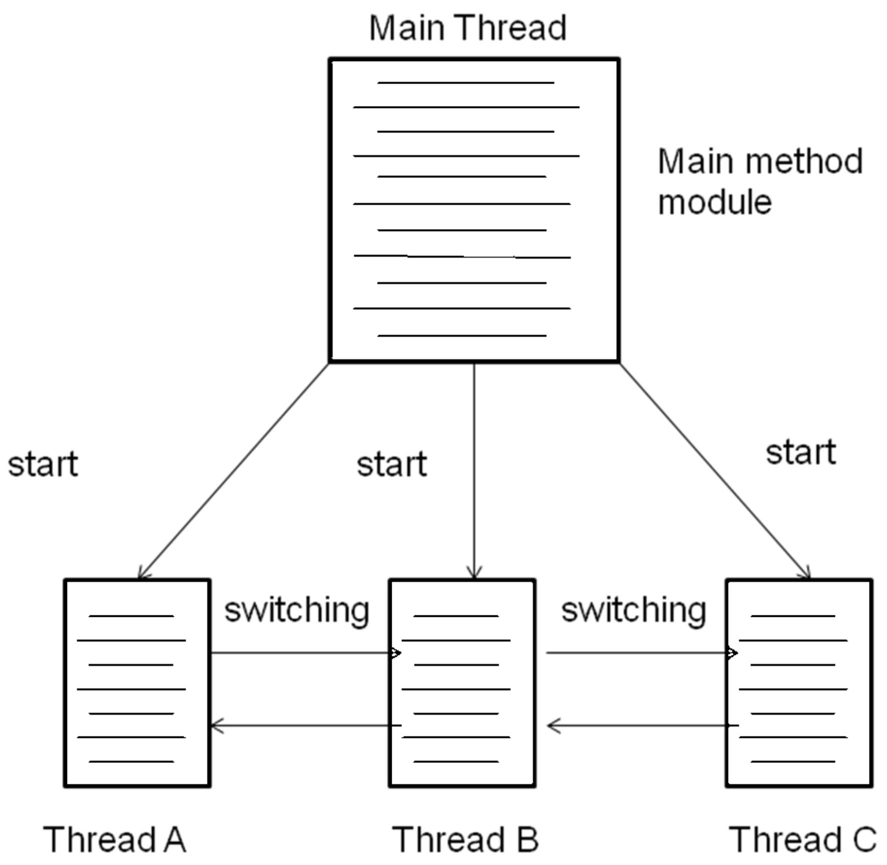
- A program which has single flow of execution are single threaded programs.
- When execute a such program ,program begins, runs through a sequence of execution and finally ends.
- All main programs in our earlier examples are single threaded programs.
- Every program will have at least one thread.



❖ Multithreading Concept

- **Multithreading** is a conceptual programming paradigm where a program (or process) is divided into two or more subprograms(or process),which can be implemented at the same time in parallel.
- Java enables us to use and manage multiple flows of control in developing programs.
- Each flow of control may be thought of as a separate tiny program, which is known as **thread** that runs in parallel to others threads of the same process.
- Threads in java are subprograms of main application program and share the same memory space, they are known as light weight process.
- Once any thread of the process initiated then the remaining threads run concurrently and share the same resources jointly.
- Multithreading is similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously, to achieve a single desire.
- *E.g. In animation, one sub program can display an animation on the screen while another may build the next animation to be displayed*.

- Threads are extensively used in java enabled browsers such HotJava. These browsers can download a file to the local computer, display a web page in the window, and output another web page to a printer and so on.
- If we are working on any application that requires two more things to be done at the same time, then threads are best to use.



Here, Java program with four threads, one main and three others. Here main thread is designed to create and start the other three threads namely A, B and C.

Note:

- Actually, we have only one processor and therefore in reality the processor is doing only one thing at time. However, the processor switches between the processes so fast that it appears to all of them are being done simultaneously.
- Threads' running in parallel does not really mean that they actually run at the same time. Actually, all threads are running on a single processor, the flow of execution is shared between threads.
- Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Use:

- ✓ It enables programmers to do multiple things at one time.
- ✓ We can divided a long program into threads and execute them in parallel, so we can use each and every resources efficiently.
- ✓ We can send tasks into background and continue to perform some other task in the foreground. This increase speed of our program.

❖ Thread life cycle

- During the life time of a thread ,there are many states it can enter, They are:

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

A thread is always in one of these five states. It can move from one state to another via a variety of ways.

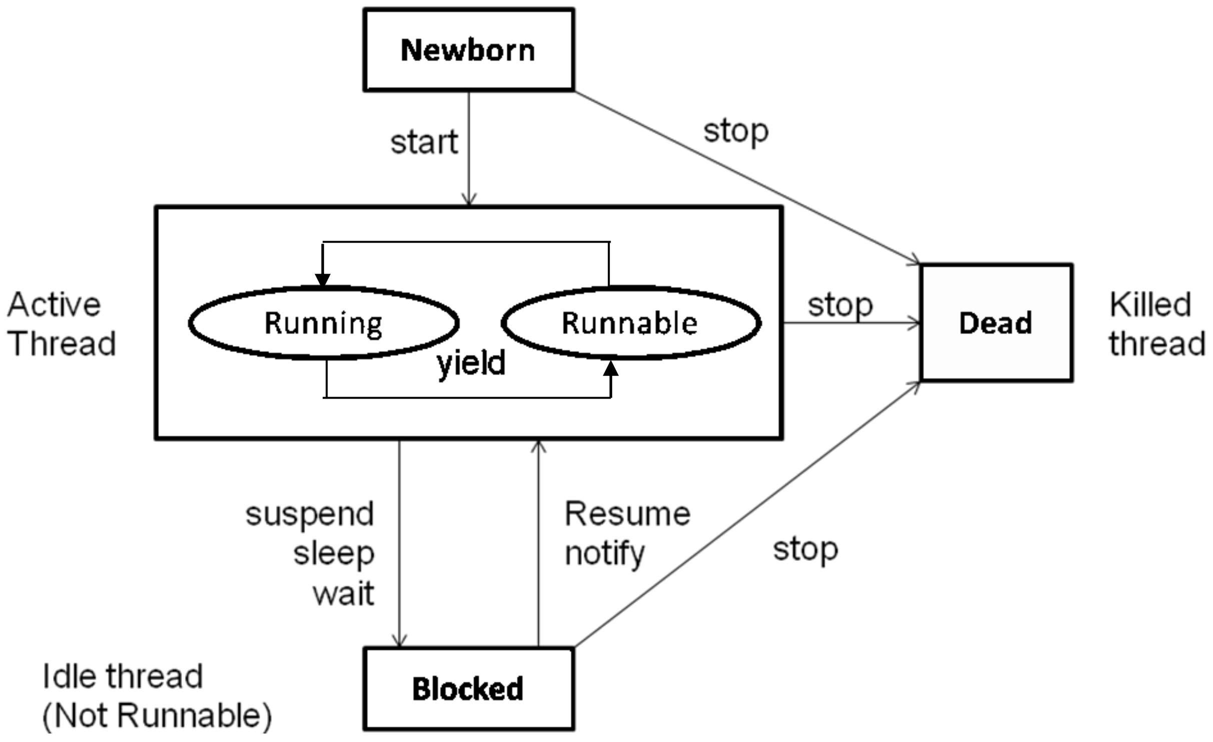


Figure: Thread Life cycle Or **Thread state transition Diagram**

1) New born state

- When we create a thread object, the thread is born and is said to be in ***newborn*** state.
- The thread is not still scheduled for running.
- At this time we can do only one of following with it:
  - ✓ Scheduled it for running using start () method.
  - ✓ Kill it using stop () method.
- If thread scheduled ,it moves to the runnable state
- If we attempt to use any other method at this stage, an exception will be thrown.

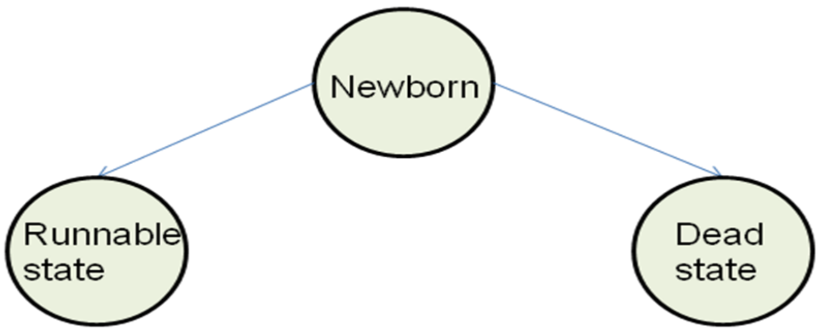
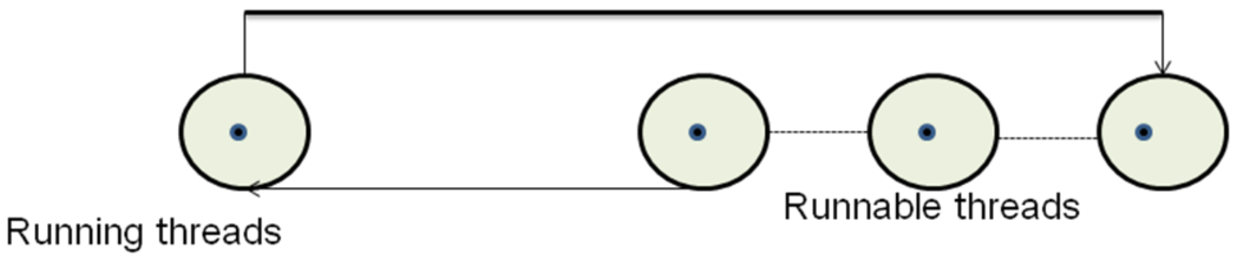


Fig. Scheduling a newborn thread

2) Runnable state

- **Runnable state** means that the **thread is ready for execution** and **is waiting for the availability of the processor**.
- The threads has joined **waiting queue** for execution.
- If all threads have equal priority, then they are given time slots for execution in round robin fashion. i. e. first-come, first serve manner.
- Thread which is relinquishes (leaves) control, joins the queue at the end and again waits for its turn. This process of assigning time to thread is known as **time-slicing.**
- If we want a thread to relinquish(leave) control to another thread of equal priority before its turn comes ,then **yield( )** method is used



Relinquishing control using yield( ) method

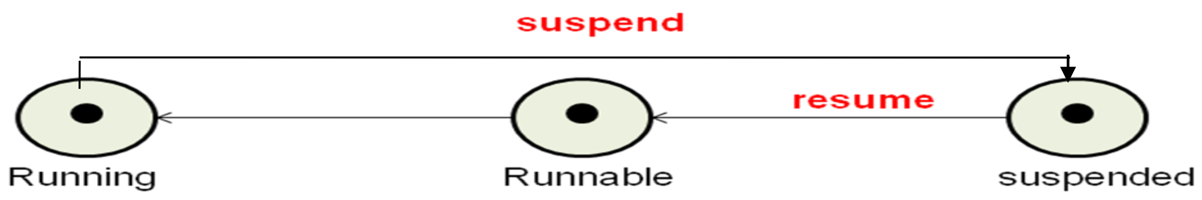
3) Running state

- Running means that processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.
- A running thread may change its state to another state in one of the following situations.

1) **When It has been suspended using suspend ( ) method.**

✓ A suspended thread can be reviewed by **resume ( )** method.

✓ This is used when we want to suspend a thread for some time due to certain reason, but do not want to kill it.



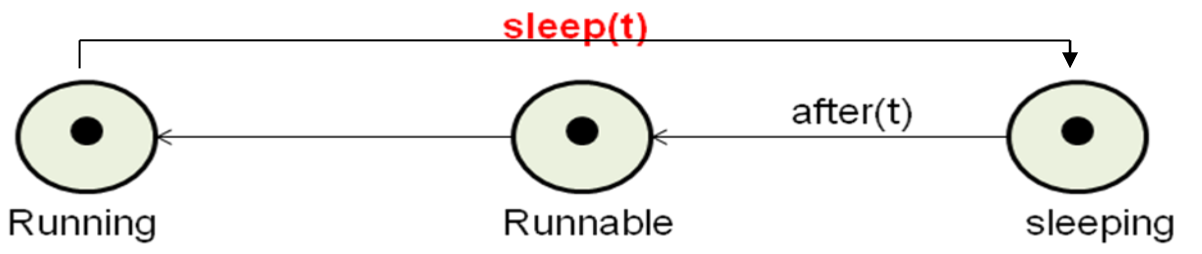
Relinquishing control using suspend( ) method

2) It has been made to sleep ( ).

✓ We can put a thread to sleep for a specified time period using the method **sleep (time)** where time is in **milliseconds**.

✓ This means that the thread is out of the queue during this time period.

✓ Thread is re-enters the runnable state as soon as this time period is elapsed(over).

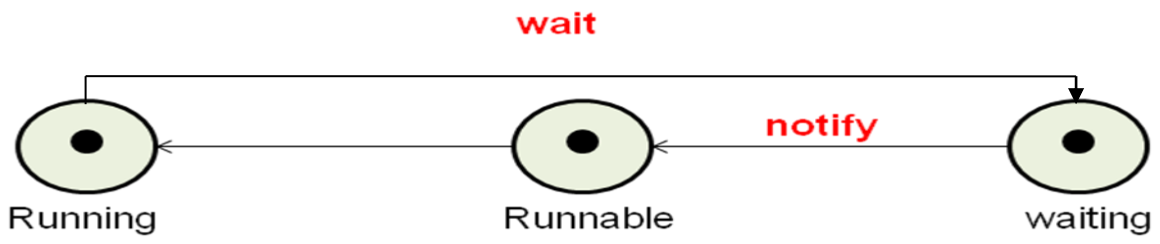


Relinquishing control using sleep( ) method

3) When it has been told to wait until some events occurs.

✓ This done using the **wait ( )** method.

✓ The thread can be scheduled to run again using **notify ()** method.



Relinquishing control using wait( ) method

4) Blocked state

- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state.
- This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.
- A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

5) Dead state

- Every thread has a life cycle. A running thread ends its life when it has completed executing its **run ( )** method.
- It is natural death. However we can kill it by sending the stop message to it as any state thus causing a premature death for it.
- A thread can be killed as soon it is born or while it is running or even when it is in "blocked" condition.

Main thread

- When a java program starts up, one thread begins running immediately**.**
- It is the one that is executed when your program begins, so it is usually called the **main thread** of your program.
- Main thread is important for two reason:
  - ✓ It is the thread from which other "child" threads will be created.
  - ✓ It must be the last thread to finish execution because it performs various shutdown actions.
- Main thread is created automatically when your program is started .it can be controlled through a Thread object.
- We can refer that object by calling the method currentThread() which is **public static** member of Thread.

Syntax: static **Thread currentThread ( )**

- This method returns reference to the thread in which it is called. If we have reference to main thread, we can control it as any other thread.
- By default, the name main thread is **main.**
- ***Thread group*** is a data structure that controls the state of a collection of threads as a whole. It is managed by the particular run-time environment.

Example: controlling the main Thread

```
class CurrentThreadDemo

{

    public static void main(String args[])

    {

            Thread t =Thread. currentThread ( );

            System.out.println (“Current thread:”+ t);

            t. setName (“Mythread”);

            System.out.println (“After name change:” + t);

            t. setPriority(10);

            System.out.println (“After Priority set:” + t);
            try
            {
            for(int n=5; n>0 ;n--)
            {
                    System.out.println (n);
                    Thread. sleep (1000);
            }
            }
            catch (InterruptedException e)
            {
                    System.out.println (“Main thread interrupted “);
            }
    }
}
```

Output:

```
Current thread: Thread [main, 5, main]
After name change: Thread [My Thread, 5, main]
After Priority set: Thread [My Thread, 10, main]
        5
        4
        3
```

2
1

Thread class's methods:

Thread encapsulates a thread of execution. Thread class defines several methods that help manage threads.

No.	**Method**	**Meaning**
1	getName( )	Obtain a thread's name.
2	setName( )	Set a thread's name.
3	getPriority( )	Obtain a thread's priority
4	setPriority( )	Set a thread's priority
5	isAlive( )	Determine if a thread is still running.
6	join( )	Wait for a thread to terminate.
7	run( )	Entry point for the thread.
8	sleep( )	Suspend a thread for a period of time.
9	start( )	Start a thread by calling run method

Creating a thread

- We can create a thread by instantiating an object of type **Thread**.
- The run () method is the heart and soul of any thread.
- run( ) method makes up the entire body of a thread and is the only method in which the thread's behaviour can be implemented.

Syntax: **public void run( )**

```
{…………
  …………   // (statement for implementing thread)
}
```

The run ( ) method should be invoked by an object of concerned thread. For that we have to create thread and initiate it with the help of other thread method called start ().

Java can create a thread by following two ways:

1) You can implement the **Runnable** interface.
2) You can extend the **Thread** class.

- **Which one of above us should use?**

The thread class defines several methods that can be overridden by a derived class .but only one must be overridden is run ( ).

This is, the same method required when we implement Runnable.

Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So it is probably best to implement Runnable instead of extend Thread class.

1) Implementing the ‘Runnable’ interface

The **Runnable interface** declares the run ( ) method that is required for implementing threads in our programs.

Steps to implement Runnable interface:

1. Declare the class as implementing the Runnable interface.
2. Implement the run ( ) method.
3. Create a thread by defining an object that is instantiated from this “runnable” class as the target of the thread.
4. Call the thread’s start( ) method to run the thread

Example of thread creation by implementing Runnable Interface:

```
class X implements  Runnable                                    //step1
{
        public void run( )                                      //step2
        {
        for (int i=0; i<=10; i++)
        {
                System.out.println (“\t ThreadX:” + i );
        }
          System.out.println (“End of ThreadX”);
        }
}
class RunnableTest
{
        public static void main (String args[])
        {
                X obj =new X ( );
                Thread t1= new Thread (obj);                    //step3
                t1.start ( );                                   // step4
                System.out.println (“End of main Thread”);
        }
}
```

Output:

```
End of main Thread
ThreadX:  1
ThreadX:  2
ThreadX:  3
ThreadX:  4
ThreadX:  5
```

```
ThreadX:  6
ThreadX:  7
ThreadX:  8
ThreadX:  9
ThreadX:  10
End of ThreadX
```

2) Extending the thread class

The Thread class declares the run ( ) method that is required for overriding threads in our programs.

We can make our class runnable by extending the class java.lang.Thread.

To extend Thread class, perform following steps:

1. Declare the class as extending the Thread class.
2. Implement the run ( ) method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the start ( ) method to initiate the thread execution .

Step 1: Declaring the class

Thread class can be extended as follows:

```
class MyThread extends Thread
{
        ………..
        ………..
}        now we have new type of thread MyThread.
```

Step 2: Implementing the run ( ) interface

The run ( ) method has been inherited by the class MyThread. We have overridden this method to implement the code executed by our thread.

```
public void run( )
{
………..                    //Thread code here
}
```

When we start the new thread, java calls the thread’s run ( ) method, so it is the **run ( )** where all the actions takes place.

Step3: starting new thread

To actually create and run an instance of our thread class, we must write:

```
MyThread aThread =new MyThread ( );
aThread .start( );                // invokes run() method
                Or
new MyThread( ).start ( );
```

- **First line** instantiate a new object of class MyThread.

It will just create object, the thread that will run this object is not yet running. The thread is in ***newborn*** state.

- **Second line** calls the **start ( )** method causing the thread to move into runnable state.

Then java runtime will schedule the thread to run by invoking its **run ( )** method, Now thread is said to be in the ***running*** state.

Example of thread creation by extending Thread class:

```
class A extends Thread
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println("\tFrom Thread A : i = " +i);
                }
                System.out.println("Exit From A");
        }
}
class SingleThread
{
        public static void main(String args[ ])
        {
                    A thread1=new A( );
                    thread1.start ( );
                    //new A ( ).start( );
        }
}
```

Output:

```
From Thread A: i = 1
From Thread A : i = 2
From Thread A : i = 3
From Thread A : i = 4
From Thread A : i = 5
Exit from A
```

Write a program to create multiple thread.

```
class A extends Thread                                  //step1
{
          public void run ( )                            //step2
          {
          for (int i=1; i<=5;i++)
          {
                System.out.println ("\t from Thread A: i ="+i);
            }
            System.out.println ("Exit from A");
    }
    class B extends Thread
    {
            public void run( )
            {
            for (int j=1; j<=5;j++)
            {
                System.out.println ("\t from Thread B: j ="+j);
            }
            System.out.println ("Exit from B");
    }
    class C extends Thread
    {
            public void run()
            {
            for (int k=1; k<=5;k++)
            {
                System.out.println ("\t from Thread C: k ="+k);
            }
            System.out.println ("Exit from C");
    }
    class ThreadTest
    {
            public static void main(String args[])
            {
            new A( ).start();                                   //step3
            new B( ).start();
            new C( ).start();
            }
    }
```

Output:

First run	Second run
From Thread A : i = 1 From Thread A : i = 2 From Thread B : j = 1 From Thread B : j = 2 From Thread C : k = 1 From Thread C : k = 2 From Thread A : i = 3	From Thread A : i = 1 From Thread A : i = 2 From Thread C : k = 1 From Thread C : k = 2 From Thread A : i = 3 From Thread A : i = 4 From Thread B : j = 1

From Thread A : i = 4 From Thread B : j = 3 From Thread B : j = 4 From Thread C : k = 3 From Thread C : k = 4 From Thread A : i = 5 Exit form A From Thread B : j = 5 Exit from B From Thread C :k = 5 Exit from C	From Thread B : j = 2 From Thread C : k = 3 From Thread C : k = 4 From Thread A : i = 5 Exit form A From Thread B : j = 3 From Thread B : j = 4 From Thread C :k = 5 Exit from C From Thread B : j = 5 Exit from B

Using isAlive ( ) and join( ) methods

In Multi-threading, one thread can know when another thread has ended or not by using two methods **isAlive ( )** and **join ( )** method.

1) isAlive() :

- We can know state of any thread by these methods.
- This method is defined by Thread class .its general form is :

  final boolean **isAlive ( )**

  It returns ***true*** if the thread upon which it is called is still running, It returns **false** otherwise.This method is occasionally useful.

2) Join() :

- The join method causes the current thread to wait until the thread upon which the **join ( )** method is call, gets terminates.
- It s name comes from the concept of calling thread waiting until the specified thread *joins* it.
- **join ( )** also allows to specify a maximum amount of time that user want to wait for the specified thread to terminate.
- This method is defined by Thread class .its general form is :

  final void **join( ) throws InterruptedException**

  Where the join () method either suspends the current thread for timeout milliseconds or until the thread it calls on terminates**.**
- This method is commonly used.

Example of isAlive ( ) and join ( )

```
class display implements Runnable
    {
            public void run( )
            {
                    int i=0;
                    while (i<4)
                            System.out.println ("Hello" + i ++);
            }
    }
    Class Alivejoin_Demo
    {
            public static  void main(String args[])
            {
                    display d=new display();
            Thread  t1=new Thread(d);
            Thread  t2=new Thread(d);
            Thread  t3=new Thread(d);

            t1.start ( );
             t2.start ( );
            t3.start ( );
            try
            {
            System.out.println ("Thread t1 is alive:" + t1.isAlive ( ));
            System.out.println ("Waiting for finishing threads t1");

                    // following line causes main thread to wait until t1
            terminate
            t1.join ( );
            System.out.println ("Thread t1 is alive:" + t1.isAlive ( ));
            }
            catch (InterruptedException e)
            {
            System.out.println ("thread t1 is interrupted ");
            }
        }
    }
```

Output:

```
Thread t1 is Alive :true

Hello : 0
Hello : 0
Hello : 0
Waiting for finishing thread t1
Hello : 1
Hello : 1
Hello : 1
Hello : 2
Hello : 2
Hello : 2
Hello : 3
Hello : 3
Hello : 3
 Thread t1 is Alive: false
```

Threaded priority

- Every thread in java has its own priority.
- Thread priority are used by thread scheduler to decide when each thread should be allowed to run.
- In theory, higher priority threads get more CPU time than lower priority threads.
- When a lower priority thread is running higher priority thread resumes (from sleeping or waiting for on I/O), it will preempt the lower priority thread.
- Threads of equal priority should get equal access to the CPU. For this, thread that share the same priority should yield control once in while.
- Every new thread that has created, inherits the priority of the thread that creates it.
- The priority is an integer value.Priority is in the range of **0** to **10.(Thread.MAX_PRIORITY(0)** to **Thread.MAX_PRIORITY(10))**
- To return thread to **default priority** ,specify **NORM_PRIORITY (5)** .
- These priorities are defined as ***final*** variables within Thread.
- If the value is out of this range than the method throws an exception **IllegalArgumentException**.
- **Most user level processes should use NORM_PRIORITY ,+1 OR -1**
- Background tasks such as Network I/O and screen repainting should use a value very near to lower limit.
- We should be very careful when trying to use very higher priority values.
- By assigning priorities to threads, we can ensure that they are given the attention they deserve.
- Whenever multiple threads are ready for execution, the java system chooses the highest priority thread and executes it. For a thread of lower priority to gain control ,one of the following thing should be happen:
    1. It stops running at the end of run().
    2. It is made to sleep using sleep().
    3. It is told to wait using wait().

If another higher priority thread comes along, the currently running thread will be preempted by the incoming thread and move it to the runnable state.

Example

```
class A extends Thread
{
public void run( )
{
        System.out.println("threadA started");
        for ( int i=1; i<=4 ;i++)
        {
                System.out.println("\t From  thread  A : i"+i);
        }
        System.out.println("Exit from A");
}
}
class B extends Thread
{
public void run( )
{
        System.out.println("threadB started");
        for ( int j=1; j<=4 ;j++)
        {
                System.out.println("\t From  thread  B : j"+j);
        }

        System.out.println("Exit from B");
}
}
class C extends Thread
{
public void run()
{
        System.out.println("threadB started");
        for ( int k=1; k<=4 ;k++)
        {
                System.out.println("\t From  thread  C: k"+k);
        }
        System.out.println("Exit from C");
}
}

class ThreadPriority
{
public static void main(String args[])
{
        A t1=new A ( );
        B t2=new B ( );
        C t3=new C ( );
```

```
                t3.setPriority (Thread.MAX_PRIORITY);                //priority will
be 10
                t2.setPriority (t1.getPriority ( ) + 1);             //priority will
be 5
                t1.setPriority (Thread.MAX_PRIORITY);                //priority will
be 6

                System.out.println (“start thread A”);
                t1.start ( );
                System.out.println (“start thread B”);
                t2.start ( );
                System.out.println (“start thread C”);
                t3.start ( );
                System.out.println (“End of main thread “);
        }

        }
```

Output:

```
        Start  thread A
                Start  thread B
                Start  thread C
                threadB started
                        from thread B: j=1
                        from thread B: j=2
                threadC started
                        from thread C: j=1
                        from thread C: j=2
                        from thread C: j=3
                        from thread C: j=4
                Exit from C
                End of main thread
                        from thread B: j=3
                        from thread B: j=4
                Exit from B
                threadA started
                        from thread A:i=1
                        from thread A: i=2
                        from thread A: i=3
                        from thread A: i=4
                Exit from A
```

Thread Exception

- The call to **sleep ( )** method should enclosed in a try block and followed by a catch block. This is necessary because the sleep ( ) method throws an exception, which should be caught.
- Java run system will throw **IllegalThreadStateException** whenever we attempt to **invoke a method that a thread cannot handle in the given state**.

  Ex: sleeping thread cannot deal with the **resume()** method because a sleeping thread cannot receive any instructions.
- Whenever we call a thread that may throw an exception, we have to supply appropriate exception handler to catch it.
- Here ,the example of different Exception in multithreading are given in next slide:

```
catch (ThreadDeath e)
{
………………
………………      //Killed threads
}
catch( InterruptedException e)
{
………………      //cannot handles it in the current state
………………
}
catch (IllegalArgumentException e)
{
………………       //illegal method argument
………………
}
catch (Exception e)
{
………………
………………
}
```

Synchronization

- When two or more threads need to 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 of achieving this is called ***synchronization.***

  For **example**, one thread may try to read a record from a file while another is still writing to the same file. Depending on situation we may get strange result. We can avoid this by synchronization.
- ***Monitor*** (semaphore) concept is key to synchronization.
- **Monitor** is an object that is used as a mutually exclusive lock or *mutex*.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the *monitor*. *Monitor is like a key and the thread that holds the key can only open the lock.*
- All the other threads attempting to enter the locked monitor will be suspended until the first thread e*xists* the monitor. This other threads are said to be waiting for the monitor.

- *We can synchronize code in two ways:*

1) Using synchronized Methods

2) The synchronized statement

Using synchronized methods

- To create object's monitor, just call a method that uses ***synchronized*** keyword.

- While we declare a method with *synchronized*.java creates monitor and hands it over to the thread that calls the method first time. While a thread is inside a synchronized method, all other threads that try to call it on the same instance have to wait.

```
synchronized void method-name ( )

{

// code here is synchronized

}
```

- Whenever thread has completed its work of using synchronized method(or block or code) ,it will hand over the monitor to the next thread that is ready to use the same resource.

```
class Callme
{
  synchronized void call(String msg)
        {
                System.out.print( " [ " + msg);
                try
                {
                Thread. sleep (1000);
                }
                catch(InterruptedException e)
                {
                System.out.println("thread is interrupted");
                }
                System.out.print(" ] ");
        }
}
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public  Caller(Callme targ,String s)
        {
                 target=targ;
                 msg=s;
                 t=new Thread(this);
                 t.start();
         }
        //synchronize calls to call()
        public void run()
        {
                target.call(msg);
         }
}
class Threadsynchro
{
        public static void main(String args[])
        {
        Callme target =new Callme();
        Caller ob1=new Caller(target,” hello");
        Caller ob2=new Caller(target,"synchronized");
        Caller ob3=new Caller(target,"world");
        try
        {
             ob1.t.join();
             ob2.t.join();
```

```
            ob3.t.join();
        }
        catch(InterruptedException e)
        {
        System.out.println("Interrupted ");
        }
        }
}
```

- **Output:**
  ```
  [Hello] [World] [Synchronized]
  ```

- **without synchronization block**
  ```
  [ Hello [world [synchronized ] ] ]
  ```

The synchronized statement

- Creating synchronized methods within classes that you create will not work in all cases.
- For example, if you want to synchronize access to *objects of a class* that was not designed for multithreading access means that class does not use **synchronized** methods. In addition if this class was created by a third party and you don’t have access to the source code .thus you can’t add synchronized to that class.
- To access a synchronized object of this class, we can use **synchronized block by** putting calling method of a class in that block.

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

- Here, lock-object is a reference to the object being synchronized.
- A synchronized block ensures that a call to a method that is a member of lock-object occurs only after the current thread has successfully entered lock-object’s monitor.

Example

```
class Callme
{
   void call(String msg)
        {
```

```
        System.out.print( " [ " + msg);
        try
        {
                Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
                System.out.println("thread is interrupted");
        }
        System.out.print(" ] ");
        }
}
class Caller implements Runnable
{
         String msg;
         Callme target;
        Thread t;
        public  Caller(Callme targ,String s)
        {
                target=targ;
                msg=s;
                t=new Thread(this);
                t.start();
        }
        //synchronize calls to call()
        public void run()
        {
                synchronized (target)
        {
        target.call(msg);
        }
        }
}
class Threadsynchro
{
public static void main(String args[])
        {
        Callme target =new Callme();
        Caller ob1=new Caller(target,"hello");
        Caller ob2=new Caller(target,"synchronized");
        Caller ob3=new Caller(target,"world");
        try
        {
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
```

```
            }
            catch(InterruptedException e)
            {
                System.out.println("Interrupted ");
            }
            }
    }
```

Output:

```
[ Hello ] [world] [Synchronized]
```

Output without synchronization block

```
[ Hello [world [synchronized ] ] ]
```

Dead lock

- When two or more threads are waiting to gain control of a resources which are already hold by another waiting threads, and no one can further proceed, such situation is known as ***deadlock.***

Ex:

```
Thread A:   synchronized method2()
                {
                    synchronized method1()
                    {
                    ……………..
                    }
                }
Thread B:   synchronized method1()
                {
                    synchronized method2()
                    {

                    ……………..

                    }
                }
```

Interthread communication

- Polling is used to check some condition repeatedly. Once the condition is true appropriate action is taken.
- In polling system, consumer would waste many CPU cycles, while it waited for the producer to produce. Once producer complete it waits for to complete consumer to finish.
- To avoid polling, Java includes inter process communication mechanism via the **wait (), notify( ) and notifyAll ( )** methods.

- These methods are implemented as ***final*** methods in **object. (in package java.lang.Object)**
- All three methods can be called only from within a ***synchronized*** context.

1. **Wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify().**

   Syntax: *final* void **wait ()** throws **InterruptedException**

2. **notify()** wakes up the first thread that called **wait ()** on the same object.
   Syntax: *final* void **notify()**

3. **notifyAll()** wakes up all the threads that called **wait()** on the same object.
   The highest priority thread will run fast.

   Syntax : *final* void **notifyAll()**

Suspending, resuming & stopping threads

1. **Stopping a thread :stop()**

   syntax : threadName.**stop()**;

This causes the thread to move to dead state.A thread will also move to the dead state automatically when it reaches the end of its method.

- The **stop( )** method may be used when the premature death of a read is desired.

2. **Resuming thread**

   syntax : threadName.**resume()**;

- This causes the thread to move to *runnable* state.A thread will also move to the dead state automatically in certain situation.

3. **Suspending thread :sleep() ,wait() and suspend()**

This all causes the thread to move to Blocked state

• threadName.**sleep(t)**;	This makes thread to sleep for *t* time (in milisecond), after time elapsed thread will goes into runnable state.
• threadName.**wait()**;	This tells thread to wait for some event to complete, to gain control **notify()** method is called of that thread.
• threadName.**suspend()**;	This suspends thread for some reason and then using **resume ()** method thread goes into runnable state.