EXCEPTION HANDLING

EXCEPTION HANDLING FUNDAMENTALS

- An Exception is run time error (or) An exception is abnormal condition that arises in a code sequence at the run time.
- All exception occur only at runtime but some exception are detected at compile time and some others at run time
- The exception that are checked at compilation time by the java compiler are called "**checked exception**" while the exception that are checked by the jvm are called **"unchecked exception"**
- ***When the jvm encounter an error such as division by zero, it creates an exception object and thrown it.If the exception object is not caught and handled properly, the interpreter will display an error message and will terminates the program.***
- If we want the programmer to continue with the execution of the remaining code then ,an exceptional condition arises an object representing that exception is created and thrown in the method that caused the error.
- The method may choose to handle the exception itself, or pass it on. Either way, at some point the exception is caught and processed.
- Java exception handling is managed via five keywords
  1) Try
  2) Catch
  3) Throw
  4) Throws
  5) Finally
- Statements that need to be monitored for exceptions should be placed within a **try** block
- If an exception occurs within the try block, it is thrown and Your code can catch this exception using **catch** block and handles it in some rational manner.

- System generated exception are automatically thrown by the jvm. To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of method must be specified as such by a **throws** clause.
- Any code that obsolutely must be executed after a try block completes is put in a **finally** block.
- Java supplies several built in exception types that match the various sorts of run time error that can be generated.

RUNTIME ERROR	**Corresponding Built in Exception**
Dividing an Integer by Zero	ArithmeticException
Accessing an element that	ArrayIndexOutOfBoundsException
Trying to store a value into an array of an incompatible class or type	ArrayStoreException

- 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 2 benefits
  1) first, it allows you to fix the error
  2) second, it prevents the program from automatically terminating.

Example:

```
class Exception
{
        Public static void main(String args[])
        {
                int d,a;
                try
                {
                        d=0;
                        a=42/d;
                        System.out.println("this will  not be printed");
                }
                Catch("ArthmeticException e)
                {
                        System.out.println("division bt zero");
                }
}
```

- This program generates the following output

```
 Division by zero
After catch statement
```

- Notice that the contents of 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 in the next line in the program following the entire try/catch block mechanism.

General Form Of An Exception Handling Block

```
Try
{
        Block of code to moniter for errors
}
Catch(Exception type1 exod)
{
        Exception handler for Exception type1
}
Catch(Exception type2 exob)
{
        Exception handler for Exception tpye2
}
Finally
{
        Block of code to be executed after
        Try block ends
}
```

EXCEPTION TYPES

- The throwable class, which is an immediate subclass of object, is at the root of the exception hierarchy.
- Throwable has two immediate subclass
  - -Exception
  - -Error

{

UNCAUGHT EXCEPTIONS

Let us see what happens when exceptions are not caught. The below program includes an expression that intentionally causes a divide-by-zero error:

```
Class Exc0
{
	Public static void main(String args[])
	{
		int d = 0;
```

```
            int a = 42/d;
        }
    }
```

- When the java run-time systems detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. Since the exception is not caught by the program, it is caught by the default handler provided by the Java run-time system, ***which causes the Execution of Exc0 to Stop.***
- Any Exception that is not caught by the program will ultimately be processed by the default handler. 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.

  Here is the Exception generated when the above example is executed

```
java.lang.ArithmeticException: /by Zero
        at Exc0.main(Exc0.java:4)
```

  the class name, ***Exc0***; the method name, ***main***; the filename,***Exc0.java***;and the line number,***4***, are all included in the ***simple stack trace.***

USING TRY AND CATCH

- 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 2 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 wish to catch.

Example:

```
    class Exception
    {
        Public static void main(String args[])
        {
            int d,a;
            try
            {
                d=0;
                a=42/d;
                System.out.println("this will  not be printed");
            }
            Catch("ArthmeticException e)
            {
                System.out.println("division bt zero");

            }
    }
```

- This program generates the following output

```
   Division by zero
  After catch statement
```

- Notice that the contents of 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 in the next line in the program following the entire try/catch block mechanism

Try block

- The try block can have one or more stmts that could generate an exception.
- If any one stmt generates an exception the remaining stmts in the block are skipped and exception jumps to the catch block that is placed next to the try block.
- The catch block too can have one or more stmts that are necessary to process the exception.
- Remember that every try stmt should be followed by atleast one catch stmt Otherwise compilation error will occur.

Catch method works like a method definition

- The catch stmt is passed a single parameter, which is reference to the exception object thrown.
- If the catch parameter matches with the type of exception object, then the exception is caught and stmt in the catch block will be executed.
- Otherwise the exception is not caught and the default exception handler will cause the execution to terminate.

MULTIPLE CATCH CLAUSES

- In some cases, ***more than one exception*** could be raised by a single piece of code.
- To handle this type of situation, 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.***
- Eample

```
//Demonstrate Multiple Catch Statements
class MultiCatch
```

```
{
        public static void main(String args[])
        {
                try
                {
                        int a = args.length;
                        System.out.println("a = "+a);
                        int b = 42/a;
                        int c[] = {1};
                        c[42] = 99;
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Divide by 0:" +e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array index out of bound:"+e);
                }
                System.out.println("After try/catch block");
        }
}
```

- This program will cause a division-by-zero exception, if it is started with ***no command-line arguments***, since **a** will equal zero.
- It will cause an ArrayIndexOutOfBoundsException, if we provide a command-line arguments, since setting **a** to something larger than zero and since the ***int array c*** has a length of 1, yet the program attempts to assign a value to c[42].

  Here is the output generated by running in both ways

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

After try/catch block
C:\>java MultiCatch TestArg
a = 1
Array index out of bound: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch block
```

- When multiple catch statements are used, it is important to remember that ***exception subclasses must come before any of their superclasses.***

- This is because a catch statement that uses ***a superclass will catch exceptions of that type plus any of its subclasses.***
- Thus, a subclass would never be reached if it came after its superclass. Further, in java, ***unreachable code is an error.***

```
/* this program contains an error
A subclass must come before its superclass in a series of catch statements. If not,
unreachable code  will be created and a compile-time error will result.
*/
class SuperSubCatch
{
   Public static void main(String args[])
   {
           try
           {
                   int  a = 0;
                   int b = 42/a;
           }
           catch(Exception e)
           {
                   System.out.println("generic Exception catch");
           }
           /* This catch is never reached because ArithmeticException is a subclass
   of
           Exception */
           Catch(ArithmeticException e)
           {
                   System.out.println("This is never Reached");
           }
}
}
```

If the above program is compiled, an error message is displayed stating that the ***second catch statement is unreachable because the exception has already been caught.*** since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException. This means that the second catch statement will never execute.***To fix the problem , reverse the order of the catch statements***

NESTED TRY STATEMENTS

So far we have been using just single try statement. However, it is possible to nest a try statement inside another try statement. **If** one try block does not have a corresponding catch block that handles the exception, Java will search the next outer try block for a catch block that will handle the exception, back through successive nesting. If the Java cannot find the catch block for the exception, it will pass the exception to its default exception handler.

Often nested try statements are used to allow different categories of errors to be handled in different ways. Many programmers use an inner try block to catch the less severe errors and outer try block to catch the more severe errors.

//An example of nested try statements

```
class NestTry
{
        Public static void main(String args[])
        {
        try
        {
                int a=args.length;
                /*If no command-line args are present, the following statement will generate a
                divide-by-zero exception. */
                int b = 42/a;
                System.out.println("a = " + a);
                try     //Nested Try block
                {
                        /* If one command line args are use, then a divide-by-zeroexception will be
                        generated by the following code. * /
                        if(a==1)    a=a/(a-a);//Division by zero
                        /*If two command line args are use, then generate an out-of-bounds
                        exception.  */
                        if(a==2)
                        {
                                int c[ ]={1};
                                c[42] = 99;    //generate an out-of-bounds exception
                        }
                }
                Catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Array Index out-of-bounds:" + e);
                }
        }
        Catch(ArithmeticException e)
        {
                System.out.println("Divide by 0:" + e);
        }
}
}
```

As you can see, this program nests one try block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block. Execution of the program with one command-line-argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on the outer try block, where it is handled. If you execute the program with two command-line-arguments, an array boundary exception is generated from within the inner try block. Here are sample runs that illustrate each case.

```
c:\>java NestTry
Divide by 0: java.lang.ArithmeticException: /  by zero
c:\>java NestTry one
a=1
Divide by 0: java.lang.ArithmeticException: /  by zero
c:\>java NestTry one
a=2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:42
```

THROW

- There is also a ***throw*** statement available in java to throw an exception explicitly and catch it.
- Till now we have seen catching exceptions that are ***thrown*** by the ***java run-time system***. However, it is possible for our program to ***throw an exception explicitly, using the throw statement***.
- The general form of throw is shown here.
  Throw *ThrowableInstance*
- 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 a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on.
- If no matching catch is found, then the default exception Handler halts the program and prints the stack trace.

Eg:

```
Class Sample
{
        static void demo()
        {
                try
                {
                        System.out.print(“inside demo());
                        Throw new NullPointerException(“Exception data”);
                }
                catch(NullPointerException ne)
                {
                        System.out.print(ne);
                }
        }
}
Class ThrowDemo
{
        public static void main(String args[])
        {
                Sample.demo();
        }
}
C:\>javac ThrowDemo.java

C:\>java ThrowDemo

Inside demo()

Java.lang.NullPointerException:Exception data
```

- In this program we are using throw clause to throw NullPointerException class object. Then it is caught and its details are displayed in catch block.

1) **Throw clause is used in software testing to test whether a program is handling all the Exception as claimed by the programmer**.

Suppose a programmer has written a program to handle some 5 exception properly. Now the software tester has to test and certify whether the program is handling all the 5 exception as said by the programmer or not.

For this, the tester should plan the input data such that if he provides that input, the said exception will occur. Causing the exceptions intentionally like this will be at times very difficult. In this case the tester can take the help of the throw clause.

Suppose the tester wants to test whether IOException is handled by the program or not, he can introduce a statement in the source code as

```
throws new IOException (“my IOException”);
```

The tester compiles and runs the program. Tester checks whether the exception is handled or not and gives feedback to the programmer. This is the way throw will help the tester to test a program for exceptions.

2) **Throw clause can be used to throw our own Exception also**

Just like the exception available in java we can also create our own exceptions which are called "user defined exceptions". We need the throw clause to throw there user defined exceptions.

THROWS

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.

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-seperated list of the exceptions that a method can throw.*

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 to declare this fact, the program will not compile.

//This program contains an error and will not compile.

```
class ThrowsDemo
{
        static void throwOne( )
        {
                System.out.println("inside throwone");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[ ])
        {
                throwOne( );
        }
}
```

To make this example compile, you need to make changes. First, you need to declare that throwOne( ) throws **IllegalAccessException.** Second, main( ) must define a try/catch statement that catches this exception.

The corrected example is shown here:

```
//This is now correct
class ThrowsDemo
{
        static void throwOne( )throws IllegalAccessException
        {
                System.out.println("inside throwone");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[ ])
        {
                try
                {
                        throwOne( );
                }
                catch(IllegalAccessException e)
                {
                        System.out.println( "caught " + e);
                }
        }
}
```

FINALLY

- when exceptions are thrown execution in a method takes a rather abrupt, nonlinear path that alters the normal flow thro's 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 method.
- For Eg. 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 bypassed by the exception handling mechanism.
- The finally keyword is designed to address this contingency.
- Finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown the finally block will execute even if no catch stmt matches the exception.

- This can be useful for
  -closing file handles and
  -freeing up any other resource that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional.
- However each by stmt requires at least one catch or a finally clause.
- Any time a method is about to return to the uncaught exception or an explicit return stmt the finally clause is also executed just before the method return.
- Here is an example program that shows three methods that exit in various ways none without executing their finally clause.

```
class FinallyDemo
{
        static void procA()
        {
                try
                {
                        System.out.print(“inside procA”);
                        Throw New RuntimeException(“Demo”);
                }
                finally
                {
                        System.out.print(“procA’s finally”);
                }
        }
        static void procB()
        {
                try
                {
                        System.out.print(“inside procB”);
                        return;
                }
                finally
                {
                        System.out.print(“procB’s finally”);
                }
        }
         static void procC()
        {
                try
                {
                        System.out.print(“inside procC”);
                }
                finally
                {
                        System.out.print(“procC’s finally”);
                }
        }
        public static void main(String args[])
        {
                try
                {
                        procA();
                }
                catch(Exception e)
                {
                        System.out.print(“Exception caught”);
                }
                procB();
                procC();
        }
}
```

Output:

```
Inside procA
ProcA’s finally
Exception caught
Inside procB
ProcB’s finally
Inside procC
Proc’s finally
```

JAVA’S BUILT-IN EXCEPTION

Inside the standard package **java.lang,** java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException.** As previously explained, these exceptions need not be included in any method’s throws list. In the language of java, these are called **unchecked exceptions** because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in following tables those exceptions defined by **java.lang** that must be included in a method’s throws list if that method can generate one of these exceptions and does not handle it itself. These are called **checked exceptions**. Java defines several other types of exceptions that relate to its various class libraries.

Java’s unchecked RuntimeException subclasses defined in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero
ArrayIndexOutOfBoundsException	Array index is out-of-bounds
ArrayStoreException	Assignment to an array element of an incompatible type
ClassCastException	Invalid cast

EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke amethod
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread
IllegalStateException	Environment or application is incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOFBoundsException	Some type of index is out-of-bounds
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumbeFormatException	Invalid conversion of a string to a numeric format
securityException	Attempt to violate security
StringIndexOutOfBounds	Attempt to index outside the bounds of a string
TypeNotPresentException	Type not found
UnsupportedOperationException	An unsupported

Java's Checked Exceptions defined in java.lang

Exception	**Meaning**
ClassNotFoundException	Class not found
CloneNotSupportedException	Attempt to clone an object that does not implement the cloneable interface.
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface
InterruptedException	One threadd has been interrupted by another thread
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist

CREATING YOUR OWN EXCEPTION SUBCLASSES

- Sometimes, the built in exception in java are not able to describe a certain situation.
- In such cases like the built in exception, the user can also create his own exception which are called "user defined exception.
- The following steps are followed in creation of user defined exceptions
  1. The user should create an exception class as a sub class to exception class. Since all exception are sub classes of exception class the user should also make his class a sub class to it. This is done as

     Class MyException extends Exception

2. The user can write a default constructor in his own exception class. He can use it, In case he does not want to store any exception details.

   MyException
   {
   }

3. The user can create a parameterized constructor with a strong as a parameter. He can use this to store exception details as

   MyException
   {
   Super(str)
   }

4. When the user wants to raise his own exception, he should create an object to his exception class and throw to using throw clause, as

   MyException me=new MyException("Exception details")
   throw me;

5. To understand the user defined exception, we are taking the details of account numbers, customer names, and balance amounts in the form of three arrays.Then in main() method, we display these details using a for loop.

This program checks if the balance amount is less than the minimum balance , then MyException is raised and a message is displayed "balance amount is less".

Example

```
//user defined exception//
//to throw whenever balance amount is below rs.1000//
class MyException extends Exception
{
        Private statcic int accno[]={1001,1002,1003,1004,1005};
        Private static string name[]={"raja rao",rama rao","appa rao","subba rao","laxmi
devi"};
        Private static double bal[]={10000,12000,5600,999,1100};
        MyException()
        {
        }
        MyException(String args[])
        {
        }
        MyException(String arg[])
        {
                Super(str);
        }
        Public static void main(String args[])
        {
                try
                {
                        System.out.print(“accno”+”/t”+”customer”+”\t”+”balance”);
                        for(int i=0;i<5;i++)
                        {
                                if(bal[i]<1000)
                                {
                                MyException  me=new  Myexception(“balance  amount  is
                         less”);
                                throw me;
                                }
                        }
                }
                Catch(Myexception me)
                {
                        me.printstackTracec();
                }
        }
}
C:\>javac Myexception.java
C:\>java MyException
ACCNO          CUSTOMER        BALANCE
1001          RAJAROA           10000.00
1002          RAMAROA           12000.00
1003          SUBBAROA           5600.00
1004          APPA ROA           999.00
MyException: balance amount is less at MyException , main(MyException.java:39)
```

Here , we are throwing our own exception when the balance amount in a bank account is less than Rs. 1000.

Here , we created our own exception since there is no exception class available to describe situation where the balance amount in a bank account is less than the prescribed minimum.

NOTE:

Throws clause is used when the programmer does not want to handle the exception and throw it out of a method.

Throw clause is used when the programmer wants to throw an exception explicitly and wants to handle it using catch block.

MULTITHREADED PROGRAMMING

1) **Introduction**
2) **Thread States**
3) **Creating a Multiple Threads**
4) **Thread Priorities**

INTRODUCTION

Java is a *multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading enables to write in a way where multiple activities can proceed concurrently in the same program.

THREAD STATES

During the life time of a thread there are many states it can enter. They are

A. NewBorn state
B. Runnable state
C. Running State
D. Blocked state
E. Dead state

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

Fig

A. **New Born state :**

when we create a thread object, the thread is born and is said to be in new born state. The thread is not yet scheduled for running .At this state, we can do only one of the following things with it.

Schedule it for running using start() method.
Kill it using stop() method.

Fig

If scheduled ,it moves to the runnable state.

B. **Runnable State:**

- The runnable state means that the thread is ready for execution and is waiting for availability of the processor.i.e the thread has joined the queue of threads that are waiting for execution.
- If all threads have equal priority, then they are given time slots for execution in Round Robin fashion,i.e FCFS manner.
- The thread that relinguishes control joins the queue at the end & again waits for its turn.this process of assigning time to threads is known as time slicing.

Fig

C. **Running State:**

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it relinguishes its control in one of the following situations.
- It has been suspended using suspend().a suspended thread can be received by using the resume() method.

Fig

- 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.
- Hence the thread is out of queue during this time period.
- The thread re-enters the runnable state as soon as this time period is elapsed.

Fig

- It has been told to wait until some event occurs. This is done using the wait() method.
- The thread can schedule to run again using the notify() method.

Fig

D. **Blocked state:**

- A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods.

  Sleep()   // blocked for a specified time

  Suspended()   // blocked until further orders

  Wait()   // blocked until certain condition occurs.

- These methods cause the thread to go into the blocked sate. The thread will return to runnable state when the specified time is elapsed in the case of sleep(),the resume() method is invoked in case of suspend(),and notify() method is called in case of wait().

E. **Dead State:**

- Every thread has a life cycle .A running thread ends its life when it has completed executing its run().it is natural death.
- However, we can kill it by using stop message to it at any stage.Thus causing premature death to it.

CREATING A MULTIPLE THREADS

- Creating threads in java is simple. Threads in java can be created in two ways
  1) By extending the thread class.
  2) By implementing the runnable interface.

1) **Creating threads by extending the thread class:**

- Define a class that extends thread class and override its run()with the code required by the thread. Steps to create thread by extending thread class are
  a) Declaring the class
  b) Implementing the run() method.
  c) Starting New Thread.

a) Declaring the class:

- declare the class by extending the thread class as:

```
Class  MyThread extends Thread
{
     ---------
     ---------
}
```

b) Implementing the run() method:

- the run method is the heart and soul of any thread.
- We have to override this method in order to implement the code to be executed by our thread.

- It makes up the entire body of a thread and is the only method in which the threads behavior can be implemented.
- The basic impolementation of run() will look like

```
 Public void run()
 {
           Thread code
  }
```

- When we start new thread ,java calls the threads run() method.

C)Starting New Thread:

- create a thread object and call the start() method to initiate the thread execution.
- To create and run an instance of our thread class, we must write the following:

```
    MyThread   t1=new   MyThread();
    T1.start();
```

- The first line instantiates a new object of class MyThread.
- The second line calls start() causing the thread to move into runnable state.
- Then, the java runtime will schedule the thread to run by invoking its run().Hence the thread is said to be in Running state.

Ex:

```
Ex:   Class A Extends Thread
        {
               Public void  run()
               {
                  Try
                  {
                       For(int i=1;i<=5;i++)
                       {
                                 S.o.p("from thread A:"+i);
                                 Thread.sleep(100);
                        }
                    }
                    Catch(InterruptEx  ception  e)
                    {
                              S.o.p(e);
                    }
                   }
          }

                Class B extends Thread
                    {
                 Public void  run()
{
   Try
   {
        For(int i=1;i<=5;i++)
         {
          S.o.p("from thread B :"+i);
            Thread.sleep(100);
```

```
                }
        }
Catch(InterruptException  e)
{
   S.o.p(e);
}
                            }
                        }
                Class ThreadDemo
                {
                Psvm(String    args[  ] )
                {
                  A   t1=new A();
                  B   t1=new B();
                    T1.start();
                   T2.start();
               }
          }
```

```
Z:\> javac ThreadDemo.java
Z:\> java ThreadDemo
From thread B:1
From  thread A  : 1
Frrom  thread B  :2
From  thead  A:2
From thread B  : 5
From  thread B  :3
From  thread  A : 3
From  thread B   :4
From  thread  A  :4
From thread   B   : 5
From  thread  A: 5
```

2.<u>**Creating threads by implementing runnable interface**</u>:

Define a class that implements Runnable interface .the runnable interface has only one method ,run() that is to be defined in method with the code to be executed by thread.

Steps to create thread by implementing runnable interface.

a.Declaring the class
b.Implementing the run() method.
c. Starting New Thread.

a.Declaring the class:

The easiest way to create a thread is to create a class that implements the runnable interface.

Declare a class as

```
        Class  MyThread implements Runnable
           {
                      --------
                      --------
           }
```

b.Implementing the run():to implement runnable , which is declared as

```
                public   void run()
                   {
                                   --------
                                   --------
                   }
```

C.Starting New thread:after creating a class that implements runnable ,instantiate an object oif type thread from within that class .

Thread (Runnable threadob, string threadName).

Ex:

```
Class A implements   thread
{
   Public void  run()
   {
       Try
        {
              For(int i=1;i<=5;i++)
                {
                  S.o.p("from thread A:"+i);
                     Thread.sleep(100);
                }
       }
Catch(InterruptException  e)
{
    S.o.p(e);
}
                             }
                        }
                        Class B implements  Thread
                        {
     Public void  run()
   {
       Try
        {
              For(int i=1;i<=5;i++)
                {
                  S.o.p("from thread B :"+i);
                     Thread.sleep(100);
                }
       }
Catch(InterruptException  e)
{
    S.o.p(e);
}
                             }
                        }
```

```
            Class ThreadDemo
           {
        Psvm(String    args[  ] )
        {
                A   t1=new A();
                B   t1=new B();
        Thread     th1= new    Thread(t1);
        Thread    th2=new    Thread(t2);
                  Th1.start();
                  Th2.start();
            }
         }
Z:\> javac ThreadDemo.java
Z:\> java ThreadDemo
From thread B:1
From  thread A  : 1
Frrom  thread B  :2
From  thead  A:2
From thread B  : 5
From  thread B  :3
From  thread  A : 3
From  thread B   :4
From  thread  A  :4
From thread   B   : 5
From  thread  A: 5
```

Choosing an approach:

So, if you will not be over riding any of threads other methods ,it is probably best simply to implement runnable.

In short,if we want to override only run() method better to use Runnable interface .If it requires to extend we have no choice but to implement the runnable interface since java classes cannot have two super classs.

THREAD PRIORITIES AND SCHEDULING

- In java, each thread is assigned a priority,which affects the order in which it is scheduled for running.
- The thread of the same priority are given equal treatment by java scheduler and therefore they share the processor on a first come, first serve basis.
- Java permits us to set and get priority of a thread using setPriority() and getPriority() as follows

  ThreadName.setPriority(int Number)
  ThreadName.getPriority()
- Thread class constants are

  MIN_PRIORITY=1
  MAX_PRIORITY=10

NORM_PRIORITY=5

- NORM_PRIORIty is the default priority.
- Java run time system usually applies one of the two following strategies
- Preemptive scheduling:if the thread has a higher priority then the current running thread leaves runnable state and higher priority enter to runnable state.
- Time-sliced(Round robin):A running thread is allowed to be execute for the fixed time, after completion of time , current thread indicates to the another thread to enter it in the runnable state.
- Example

```
Class A extends Thread
{
            Public void run()
            {
                    S.o.p("thread A started");
                    For(int i=1;i<=4;i++) {
                    S.o.p("\t from thread A:i=  "+i);
            }
            S.o.p("exit from A");
 }
Class B extends Thread{
{
            Public void run

                    S.o.p("thread   B started");
                    For(int j =1   j<=4;  ++) {
                    S.o.p("\t from thread B:j=  "+j);
            }
            S.o.p("exit  from B ");
    }
Class  C extends Thread
{
            Public  void  run
            {
                    S.o.p("thread   C  started");
                    For(int  k=1   k<=4;  k++) {
                    S.o.p("\t from thread C:k=  "+k);
            }
            S.o.p("exit  from  c ");}
    }
Class ThreadPriority
{
    psvm(String   args[ ])
    {
            A   t1=new A();
```

```
                B   t2=new B();
                C   t3=new C();
                t3.setPriority(Thread.MAX_PRIORITY);
                t2. setPriority(t1.getPriority()+1);
                t1. setPriority(Threag.MIN_PRIORITY);
                S.o.p("start Thread A");
                t1.start();
                S.o.p("start Thread B");
               t2.start();
                S.o.p("start Thread C");
                t3.start();
                ZS.o.p("end of main thread ");}
        }
}
```

THREAD SCHEDULING

Java run time system usually applies one of the two following strategies

1.<u>Preemptive scheduling</u>:if the thread has a higher priority then the current running thread leaves runnable state and higher priority enter to runnable state.

2.**<u>Time-sliced(Round robin):</u>**A running thread is allowed to be execute for the fixed time, after completion of time , current thread indicates to the another thread to enter it in the runnable state.