An Exception in java signals an error or an unusual event that happens during the execution of a program. One important point to note is that all errors in your program are not signaled by exceptions. Exceptions should be reserved for the unusual or catastrophic situations that can arise. For example a user entering incorrect input to your program should be handled without recourse to exceptions.

The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling. Exception handling separates the code that deals with errors from the code that is executed when program is normally executing. It also provides a way of enforcing a response to particular errors.

An exception in java is an object that is created when an abnormal situation arises in your program. This exception object has fields that store information about the nature of the problem. When the error situation is encountered we will say an exception is thrown.,-That is the object identifying the exceptional circumstances is tossed as an argument to the program code that has been written specifically to deal with that kind of problem. The code receiving the exception object as a parameter is said to catch it.

The situations that cause exceptions, fall into four broad categories:

Code or data Errors: In this situation, exceptions happen, due to error in your program code or data. For example,an invalid cast of an object or trying to use an array index that is outside the limits for the array,or an arithmetic expression that has a zero divisor.

Standard Method Exceptions: There are situations in which the standard methods defined by the java classes may cause exceptions. For example,if you use the **substring()** method in the **String** class can throw **StringIndexOutofBoundException**

Throwing your own Exceptions: Java provides facility for the user to throw their own exceptions.

Java Error: There are situations in which the errors can be caused by the Java Virtual Machine. This usually arises as a consequence of an error in your program

Classification of Exceptions

Exceptions fall into two categories:

- **Checked Exceptions**
- **Unchecked Exceptions**

Checked Exceptions

Checked Exceptions are frequently considered "non-fatal" to program execution. They represent invalid conditions in areas outside the immediate control of the program (invalid user input, database problems, network outages, absent files)These exceptions are checked at

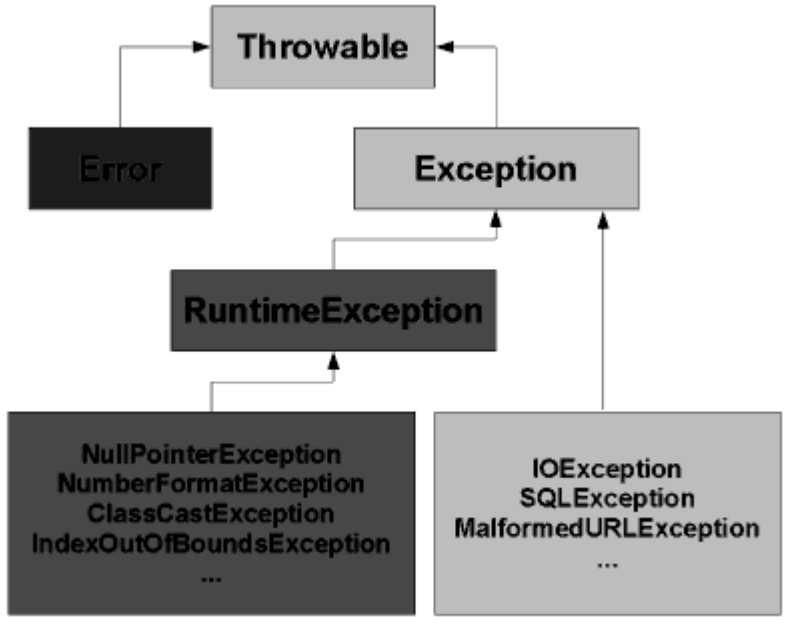
compilation time by the java compiler. Any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause. Java compiler checks if the program either catches or lists the occurring checked exception. If not, compiler error will occur.

Unchecked Exceptions

Unchecked exceptions represent error conditions that are considered "fatal" to program execution. They represent defects in the program (bugs) - often invalid arguments passed to a non-private method. Unchecked runtime exceptions represent conditions that, generally speaking, reflect errors in your program's logic and cannot be reasonably recovered from at run time. The programmer cannot do anything when they occur. The programmer can write a java program with unchecked exceptions and errors and can compile the program. These exceptions are not subject to compile-time checking for exception handling. While running the program it terminates with an appropriate error message. They are checked by JVM.

Exception Hierarchy

Exceptions in Java are actual objects, instances of classes that inherit from the class Throwable.which is the base class for an entire family of exception classes, declared in **java.lang** package as **java.lang.Throwable**. When an exception is thrown, an instance of a Throwable class is created.. The following figure shows exception hierarchy



Error class

The exceptions defined by error and its subclasses are characterized by the fact that they all represent conditions that you aren't expected to do anything about. They are unchecked exceptions representing serious issues which we are not able to solve. They are ignored by the compiler.

Error has 3 important subclasses-ThreadDeath, LinkageError and VirtualMachineError

ThreadDeath: A ThreadDeath exception is thrown whenever an executing thread is deliberately stopped.

LinkageError: LinkageError class has subclasses that record serious errors with the classes in your program. Incompatibilities between classes or attempting to create an object of a non-existent class type are the sort of things that cause these exceptions to be thrown.

VirtualMachineError: Its subclasses specify the exceptions that will be thrown when the failure of java virtual machine occurs.

When these exceptions occur ,we can do little or nothing to recover from them during the execution of the program.

Exception class

Almost all the exceptions that are represented by the subclasses of exceptions are checked exceptions, except **RuntimeException.** So code must be provided in the program to handle those exceptions if there are chances that your program may throw those exceptions. So if a method in your program has the potential to generate an exception of a type that has **Exception** as super class, you must either handle those exceptions within the method or mention that your code may throw such an exception.

RuntimeException class

It is a subclass of Exception class which is treated differently. The exceptions of this class are unchecked exception which arise because of serious errors in your program. So the compiler allows you to ignore them. In most of the cases there is nothing you can do to recover the situation. However in some contexts we may want to deal with some of the exceptions which belong to the RuntimeException class.

For example Exceptions like **ArithmeticException** and **ArrayIndexOutOfBoundsException** which are subclasses of RuntimeException turns up quite easily in your program. In these cases we may be writing code to handle these exceptions.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them.

public class UncaughtExe

```
{
 public static void main(String args[])
{  int i=1;
  int j=0;

  System.out.println(i/j);
}
}
```

Here When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of ***UncaughtExe*** to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. 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.

Compiling the above program will not cause any errors.

On running the program the Output obtained will be:

Exception in thread "main" java.lang.ArithmeticException: / by zero → (**string describing the exception**)
at UncaughtExe.main(UncaughtExe.java:9)-→(**StackTrace**)

Here type of the exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

The stack trace will always show the sequence of method invocations that led up to the error. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Exception Handling in Java

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

This is the general form of an exception-handling block:
try {

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

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

Following are the **advantages** of Exception-handling in Java:

- Exception provides the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.

- One of the significance of this mechanism is that it throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

- With the help of this mechanism the working code and the error-handling code can be disintegrated. It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Exception Handling 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 two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to **catch.**

```
public class TryCatch
{
 public static void main(String args[])
{  int i=1;
  int j=0;
  try
 {
   System.out.println("Inside Try");
```

```
   System.out.println(i/j);
  System.out.println("Ending Try");
}
catch(ArithmeticException e)
{
   System.out.println(" Arithmetic exception caught");
}
  System.out.println("After try catch");
}
}
```

Here while compiling the above program won't raise any error. But on running the program **ArithmeticException** is thrown while evaluating the arithmetic expression (i/j). Once an exception is thrown, program control transfers out of the **try** block skipping the line "***Ending Try***". It continues execution with the code in the catch block. So the line "***Arithmetic exception caught"*** will be printed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

The Output is:

Inside Try
Arithmetic exception caught
After try catch

In the above example set the value of **j to 1**.In that case, all the statements written inside the try block will be executed. The catch block will be skipped and execution will continue from the statement immediately after catch block.

The Output will be:

Inside Try
1
Ending Try
After try catch

A **try** and its **catch** statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot **catch** an exception thrown by another try statement (except in the case of nested try statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

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

```
import java.util.Random;
class HandleError
{
  public static void main(String args[])
  {
    int a=0, b=0, c=0;
    Random r = new Random();
   for(int i=0; i<32000; i++) {
 try
  {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
 }
 catch (ArithmeticException e)
 {
    System.out.println("Division by zero.");
    a = 0; // set a to zero and continue
}
     System.out.println("a: " + a);
}
}
}
```

Displaying a Description of an Exception

When an exception occurs an exception object is created which has fields that store information about the details of the exception. It is possible to display this description in a println( ) statement by simply passing the exception as an argument.

For Example:
```
catch (ArithmeticException e)
        {
           System.out.println("Exception: " + e);
        }
```

Here each divide-by-zero error displays the following message:
Exception: java.lang.ArithmeticException: / by zero

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, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

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

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

First Scenario

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

Second Scenario

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

When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes. This is because a catch statement that uses a super class will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its super class.So unreachable code will be created and a compile-time error will result.

```
Eg:classSuperSubCatch
 {
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 you try to compile this program, you will receive an error message 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** statementwill 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

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
Eg:classNestTry
{
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 arg is used, then a divide-by-zero exception will be generated by  the
   following code. */
if(a==1) a = a/(a-a); // division by zero
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 by 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 onto 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 Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException
```

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry( )**:

```
Eg: classMethNestTry
    {
static void nesttry(int a)
{
try
            { // nested try block
/* If one command-line arg is used, then a divide-by-zero exception will be generated by the
following code. */
if(a==1) a = a/(a-a); // division by zero
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);
}
}
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);
nesttry(a);
 }
 catch(ArithmeticException e)
{
  System.out.println("Divide by 0: " + e);
}
}
}
```

The output of this program is identical to that of the preceding example.

finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this problem.

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 statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit **return** statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources 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 try statement requires at least one catch or a finally clause.

```
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
```

```
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed. If a finally block is associated with a try, the **finally** block will be executed upon conclusion of the **try** .

output

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

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter into a catch clause, or creating one with the **new** operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the 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.
Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
class ThrowDemo {
static void demoproc()
```

```
{
try
{
  throw new NullPointerException("demo");
}
 catch(NullPointerException e)
{
   System.out.println("Caught inside demoproc.");
    throw e; // rethrow the exception
}
}
public static void main(String args[])
{
  try
 {
    demoproc();
 }
 catch(NullPointerException e)
{
   System.out.println("Recaught: " + e);
}
}
}
```

First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown.

Output

Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

Note: All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**

The **throw** clause is useful in software testing in the following way:

Throw clause is used in the software testing to test whether a program is handling all exceptions as claimed by the programmer. For example a programmer who has written the application claims that he had written code for handling 5 exceptions. The tester needs to test and certify that whether the program is handling all the 5 exceptions or not. For this the tester needs to provide data that will cause exception. But causing exceptions intentionally like this is very difficult .In this cases

the tester can make use of **throw** clause to check whether a particular exception is handled by the application or not.

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-separated list of the exceptions that a method can throw.

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

The above program 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

To make this example compile, you need to make two 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);
}
}
}
```

output

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Java's Built-In Exceptions

In 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. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they 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. java.lang also defines exceptions 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.

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.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in 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.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Unchecked RunTimeException subclasses

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 thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Java's Checked Exceptions Defined in java.lang

User Defined Exceptions

Sometimes the built-in exceptions are not able to describe a certain situation. In such cases it is possible for the user to create his own exceptions which are called User-Defined Exceptions. The following steps are followed in the creation of User-Defined Exceptions.

- Since all Exceptions are subclasses of Exception class, So user should create a sub class of exception.

Class MyException extends Exception

- User can create an empty constructor in his own exception class. he can use it ,in case he doesn't want to store any exception details. If the user doesn't want to create an empty object to his exception class ,he can eliminate writing the default constructor.

```
MyException(){ }
```

- The user can create a parameterized constructor with a string as a parameter. He can use this to store exception details. He can call super class(Exception) constructor from this and send the string there.

```
MyException(String str)
 {
      super(str);
   }
```

- When the user want to raise his own exception, he should create an object to his exception class and throw it using throw clause as:

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

eg:

Threads

Introduction

A thread represents a separate path of execution of a group of statements. The way statements are executed are classified into two types:

1) Single Tasking
2) Multi Tasking

Multi Tasking

Suppose the processor starts at the first job. Then it will spend exactly ¼ milliseconds for the first job. If the processor is not able to complete the job within this time, it will store the intermediate results in a temporary memory and it goes to the second task .It spends exactly ¼ milli seconds for the second task. After that it proceeds in the same manner for 3rd and 4th tasks. After executing the fourth task for ¼ milli seconds,it will come back to the first task,in a circular manner. This is called 'round robin' method.

The processor after returning to first task resumes it from the point where it has left the first task earlier and executes it for ¼ milli seconds and proceeds in the 'round -robin' manner. Here the processor is executing your job for ¼ milli seconds and keeping you waiting for another ¾ millisecond, while it is going round executing the other tasks. After ¾ milliseconds it will again execute your job for ¼ milli seconds. But you will not be aware that you are kept waiting for ¾

milliseconds as this time is very small. You will feel that the processor spends time executing your job only.

So in multitasking, there will be a single processor performing multiple tasks. It is achieved by providing each process a time-slice and executing them in round robin manner ,so that you will get a feel that the processor is spending time for your task only.

The main advantage of multi taking is to user processor time in a better way. Here most of the processor time is engaged and it is not sitting idle. In this way we can complete several tasks at a time, and thus achieve good performance.

Multi tasking are of two types:

a) Process-based multi tasking

b) Thread based multi tasking

Process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Inter-process communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Inter thread communication is inexpensive, and context switching from one thread to the next is low cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is controlled by java. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Uses of Thread

- Threads are used in server-side programs to serve the needs of multiple clients on a network or internet. If we use threads in the server, they can do various jobs at a time, and there by handle several clients.
- Threads can also be used to create games and animation. Animation means moving the objects from one place to another. In many games we have to perform more than one task simultaneously. This can be achieved by using threads.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. To create a new thread, your program will either extend Thread or implement the Runnable interface. The Thread class defines several methods that help manage threads:

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

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is created automatically when your program is started, it can be controlled through a Thread object by calling the method **currentThread( )**. This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often it must be the last thread to finish execution because it performs various shutdown actions.

```
// a simple demo of main thread ...

class mainthread
{
   public static void main(String args[])
   {
      Thread t = Thread.currentThread();
       System.out.println("The current thread :   " + t);
       try
       {
           for(int i=0;i<5;i++)
           {
              System.out.println("The value of i :   " + i);
              Thread.sleep(1000);
           }
```

```
        }
        catch(InterruptedException ie)
        {
           System.out.println("Thread interrupted");
        }
    }
}
```

Creating a Thread

A thread can be created by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- **By extending the Thread class.**
- **By implementing the Runnable interface**.

Extending Thread

The steps to create a new thread are:

- Extend the java.lang.Thread class.
- Override the run() method in this subclass of Thread.
- Create an instance of this new class.
- Invoke the start() method on the instance

Extending the java.lang.Thread Class

. An instance of the java.lang.Thread class is associated with each thread running in the Java VM. These Thread objects serve as the interface for interacting with the underlying operating system thread. It is possible to extend the Thread class in our class. Then creating objects of class acts as threads Through the methods in this class, threads can be started, stopped, interrupted, named, prioritized, and queried regarding their current state.

Overriding the run() Method

After extending Thread, the next step is to override the run() method because the run() method of Thread does nothing:

public void run() { }

When a new thread is started, the entry point into the program is the run() method. The first statement in run() will be the first statement executed by the new thread. Every statement that the thread will execute is included in the run() method or is in other methods invoked directly or

indirectly by run(). The new thread is considered to be alive from just before run() is called until just after run() returns, at which time the thread dies. After a thread has died, it cannot be restarted.

Create an instance of the newly created thread class.

Once the class which extends the thread class is defined we need to create an instance of the newly created class in order to execute that thread.

Invoke the start() method

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. The start() method causes and not actually starts execution.It schedules the thread and when CPU scheduler picks this thread for execution then JVM calls the run() method to actually start execution.

Thread can call start() method only once in a program. A thread will throw 'IllegalStateException' if we try to call start method on already started thread instance.

```
class NewThread extends Thread
 {
    public void run()
     {
          System.out.println("this thread is running…");
    }
}
class ExtendThread
{
  public static void main(String args[])
 {
   NewThread t=new NewThread();
  t.start();
}
}
```

Implementing the Runnable Interface

The steps for create a thread using Runnable Interface are:

1. Create a class that implements the interface and override the run() method

```
class MyThread implements Runnable
 {
    …….
  public void run(){
 //thread body of execution }
}
```

2. Create an object of the class

MyThread myobj=new MyThread();

3. Create Thread Object
   Thread thr1=new Thread(myobj);

4. Start execution of the thread using start()
   thr1.start();

```
class NewThread implements Runnable
 {
    public void run()
     {
          System.out.println("this thread is running…");
    }
}
class ImplementThread
{
  public static void main(String args[])
  {
    Thread t=new Thread(new MyThread());
   t.start();
}
}
```

Thread Class versus Runnable Interface

By extending the thread class, the derived class itself is a thread object and it gains full control over the thread life cycle. Implementing the Runnable Interface doesnot give developers any control over the thread itself. It simple defines the unit of work that will be executed in a thread. Another important point is that when extending the Thread class, the derived class cannot extend any other base classes because java only allows single inheritance. But by implementing Runnable Interface, the class can still extend other base classes if necessary.

To summarize, if the program needs a full control over the thread life cycle, extending the Thread class is a good choice, and if the program needs more flexibility of extending other base classes, implementing the Runnable interface is preferred.