# JAVA

**Haaris InfoTech**
No 10A, Sriram Nagar, South Street
Chennai -18
H/P: 9840135749 / 52040551
mhshoiab@yahoo.co.uk

# JAVA

## INTRODUCTION TO JAVA

An object-oriented, multi-threaded programming language developed by Stanford University Network (SUN) in 1991 by James Gosling, Patrick Naughton, Chris Warth, Mike Sheridon.

Designed to be small, simple & portable across different platforms as well as OS.

## POPULARITY

1. Usage of Applets
2. Powerful Programming language constructs
3. Rich set of significant object classes

# APPLETS & APPLICATIONS

Applications → A program that runs on the computer, under various OS.

Applet → This makes java more important. An applet is an application designed to be transmitted over the internet & executed by a Java-compatible browser.

# FEATURES OF JAVA

1. Simple & Powerful
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multithreaded
7. Architecture-neutral
8. Interpreted & High Performance
9. Distributed
10. Dynamic

# OBJECT-ORIENTED PROGRAMMING

2 Paradigms

a. *Process-oriented Model*

Characterizes a program as a series of steps.

*b. Object-oriented Programming*

Organizes a program around its data & a set of well-defined interfaces to that data.

# OBJECT-ORIENTED PROGRAMMING

## ABSTRACTION

Essential element of OO programming. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows to layer the semantics of complex systems, breaking them into more manageable pieces. Each of them can be called as a Objects which describes its own Unique Behavior.

# OO CONCEPTS

OO concept forms the heart of Java. OO programming is a powerful & natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software projects.

3 CONCEPTS

1. ENCAPSULATION
2. INHERITANCE
3. POLYMORPHISM

# ENCAPSULATION

- Mechanism that binds together code & the date it manipulates & keeps both safe from outside interference & misuse.
- The protective wrapper that prevents from the code & data from being arbitrarily accessed from outside. Access is tightly controlled through a well-defined interface.
- Mechanisms for hiding complexity is to be mark the variable or class as **PRIVATE** OR **PUBLIC**.

# INHERITANCE

- Process by which one object acquires the properties of another object. (information is made manageable by top-down classification)
- By use of inheritance, objects need to define only those qualities that make it unique within its class. It can inherit its general attributes from its parent.

# POLYMORPHISM

- Polymorphism (many-forms) allows one interface to be used for a general class of actions.
- Polymorphism can be expressed as

  "ONE INTERFACE MULTIPLE METHODS"
- It is the compiler's job to select the appropriate method for the situation.
- It is enough to remember & use the interface.

# JAVA FUNDAMENTALS

```
class Sample
{
   public static void main (String args[])
        {
             System.out.println (“Welcome to JAVA”);
        }
}
```

Note: By convention the name of the class should match the filename which hold the program.

# JAVA FUNDAMENTALS

## Compilation

javac <filename>.java

The javac compiler creates a file called <filename>.class that contains the bytecode of the program.

## Execution

java <classname>

# JAVA FUNDAMENTALS

public static void main (String args[])

Public → access specifier, which allows the programmer to control the visibility of class members. Main must be declared as public since it must be called by code outside of its class when the program is started.

Static → allows main to be called without having to instantiate an instance of the class.

Void → returns nothing

Main → the line at which the program begins executing.

Arguments → to receive command line arguments & get executed in the main program.

# BASIC CONSTRUCTS

- Data types & variables
- Operators
- Control Statements
- Arrays
- Strings

# BASIC CONSTRUCTS

Java programs are a collection of whitespace, identifiers, comments, literals, operators, separators, & keywords.

Whitespace → Java is a free-form language. It is not needed to follow any rules. (i.e.) The program can be written in one line. Whitespace is a space, tab, or new line.

Identifiers → class names, method names & variable names. Identifier can be descriptive sequence of LC or UC letters, numbers or anything valid.

Literals → A constant value is created by using a literal representation.

# BASIC CONSTRUCTS

Comments → Display some message related to the program for understanding the program.

Separators → ( ), { }, [ ], ;, , .

; -- is the most commonly used separator in java.

Keywords → abstract, boolean, const, finally, int, public, this, return, throw, throws, implements, package, final, class, catch, byte, etc.

These keywords cannot be used as names for a variable, class or method.

# Type Conversion and Casting

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met
  - The two types are compatible
  - The destination type is larger than the source type.

  To cast incompatible types use (typecasting)

  For ex:

  Int I=257;

  byte b=(byte)I;

  ** b will be equal to 1(257 is divided by 256)and the remainder is assigned.

  Automatic type promotion – for ex: byte b,c;

  int a=b*c;

# Escape Sequences

- \’ – single quote
- \” – double quote
- \\ - backslash
- \r – carriage return
- \n – new line
- \f – form feed
- \t – tab
- \b- backspace

# DATATYPES & VARIABLES

Java is a very strongly typed language.

Java defines 4 types of data.

1. Integers → byte, short, int, & long.
2. Floating-Point → float, double.
3. Characters → char.
4. Boolean → True or False.

# DATATYPES & VARIABLES

**INTEGERS**

| | | |
|---|---|---|
| **long** | **64 bit** | **-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807** |
| **int** | **32 bit** | **-2,147,483,648 to +2,147,483,647** |
| **short** | **16 bit** | **-32,768 to + 32,767** |
| **byte** | **8 bit** | **-128 to +127** |

**FLOATING-NUMBERS**

| | | |
|---|---|---|
| **double** | **64 bit** | **1.7e-308 to 1.7e+308** |
| **float** | **32 bit** | **3.4e-308 to 3.4e+038** |

**Float f=3.7f**

**CHARACTERS**

**range of char is 0 to 65,536**

**BOOLEAN**

**Only 2 possible values. TRUE or FALSE.**

# VARIABLES

Variables are locations in the memory that can hold values.

3 types,

1. Local variables → declared inside blocks as counters or in methods as temp variables.
2. Instance Variables → used to define attributes or the state of a particular object.
3. Class Variables → variables are declared in the class & they are global to the class & to all the instances of the class. A class variable is declared by the keyword “STATIC”.

# OPERATORS

Java provides a rich operator environment. There are 4 groups of operators,

1. Arithmetic Operators
2. Relational Operator
3. Logical Operator

# ARITHMETIC OPERATORS

Types of Arithmetic Operators

| | |
|---|---|
| + | - |
| * | / |
| % | |
| = | |
| ++ | -- |
| += | -= |
| *= | /= |
| %= | |

# OPERATOR PRECEDENCE

| | | | |
|---|---|---|---|
| () | [] | | |
| ++ | -- | - | ! |
| * | / | % | |
| + | - | | |
| > | >= | < | <= |
| == | != | | |
| & | ^ | \| | |
| && | \|\| | | |
| ?: | | | |
| = | | | |

# RELATIONAL OPERATORS

| | |
|---|---|
| == | != |
| > | < |
| >= | <= |

# CONTROL STATEMENTS

**If statement**

General Form

```
if (condition)
        statement
```

**If – else – if**

General Form

```
if (condition)
        statement
else
if (condition)
        statement
….
```

Ternary Operator

```
String s=2>3?"success":"failure";
```

# CONTROL STATEMENTS

**Switch**

It is a branch statement.

General Form

```
switch (choice)
{
case <value 1> :
        statements;
        break;
.....
default :
        default statement;
}
```

# CONTROL STATEMENTS

**While**

General Form

```
While (condition)
{
   body of loop
}
```

**Do-while**

General Form

```
Do
{
   body of loop
} while (condition);
```

# CONTROL STATEMENTS

**For** loop

General Form

```
for (initialization; condition; iteration)
{
        body
}
```

# Break Statements

**Break**
**Continue**

```
public class Hotel
{
public static void main(String args[])
{
      a:{
      for(int i=0;i<10;i++)
      {
                b:{System.out.println(i);
                break b;
                //System.out.println(i);
                }
      }}
      for(int i=0;i<10;i++)
      {
                if(i%2==0)
                {
                          continue;
                }
                System.out.print(i+":");
      }
}}
```

# ARRAYS

Array is an object that stores a list of items of same data type.

In java, a variable to hold the array is declared, & a new object is created & assigned to it.

<data type> <array name> [] = new <data type>
[size of the array];

<data type> <array name> [] = { array elements };

# ARRAYS

```
class sorting
{
public static void main (String args[])
{
int num[] = {21,5,78,56,59,23,19};
int n = num.length;
System.out.println ("The Numbers to be sorted are : ");
for (int i=0;i<n ;i++)
{
   System.out.println (" "+num[i]);
}
```

# ARRAYS

```
System.out.println (" ");
for (int i=0;i<n;i++)
{
   for (int j=i+1;j<n; j++)
{
   if (num[i] < num[j])
   {
         int temp = num [i];
         num [i] = num [j];
        num [j] = temp;
}              }              }
```

# ARRAYS

```
System.out.println ("The Sorted List is as follows : ");
for (int i=0;i<n;i++)
{
   System.out.println (" "+num[i]);
}
System.out.println(" ");
}
}
```

# ARRAYS

Multidimensional Arrays

<data type> <variable name> [] [] = new <data type>
[row size of array] [column size of array];

EX

```
class twod
{
   int I = 0;
   int j = 0;
int matrix [][] = new int [3][3];
```

# ARRAYS

```
void set()
{
matrix [0][0] = 1;
matrix [0][1] = 0;
matrix [0][2] = 0;
matrix [1][0] = 0;
matrix [1][1] = 1;
matrix [1][2] = 0;
matrix [2][0] = 0;
matrix [2][1] = 0;
matrix [2][2] = 0;
}
```

# ARRAYS

```
void show()
{
   for (int I = 0; i<3; i++)
{
   for (int j = 0; j<3; j++)
{
System.out.println (“ “+matrix [i] [j]+” “);
}
System.out.println (“ “);
}
}
```

## Packages & Access Protection

Java's mechanism to provide a solution to partition the class names space into more manageable chunks is packages.

To create a package is quite easy, simply include a package command as the first statement in a java source file. Any classes declared within that file will belong to the specified package.

Ex: package mypack;

To create a package (directory) run this command when compiling

Javac –d <target - dir> <source file>
Ex: javac –d c:\test test.java

**Access Protection ->**

| | Private | No Modifier | Protected | Public |
|---|---|---|---|---|
| **Same Class** | **yes** | **Yes** | **Yes** | **Yes** |
| **Same package sub class** | No | **Yes** | **Yes** | **Yes** |
| **Same package non subclass** | No | **yes** | **Yes** | **Yes** |
| **Different package sub class** | No | No | **yes** | **Yes** |
| **Different package non sub class** | No | No | No | **yes** |

**Package & Access Protection Example ->**

**Example**

```
package mypack2;
public class test2
{
public int i=0;
private int j=0;
protected int k=0;
int m=0;

}
```

```
package mypack1;
import mypack2.*;

public class test1 extends test2
{
        public test1()
                {
                        System.out.println(m);
                }
public static void main(String args[])
        {
                new test1();
        }
}
```

**To Compile**

**C:\dir>javac –d . *.java**

**To Run**

**C:\dir>java mypack1.test1**

**A Study on Classes / Methods/ Constructors / Abstract Classes / Interfaces**

```
Public Class test
{
Test()  // Constructor
{}
Test(int i) // over loaded constructor with primitive data type
{}
Test(test2 obj) // over loaded constructor with comples type
{

}
Public void method() {} // method which dosent return a value
Public String method1(){} //method which return string as return type
Public test method2(){} // method which returns a comples type.
}
```

**Example: A closer look at argument pasing ->**

**Pass by Value**

```
public class pass
{
        public void method(int i,int j)
                {
                        System.out.println(++i);
                        System.out.println(++j);
        }
}
class passbyvalue
{
public static void main(String args[])
        {
        int i=10;
        int j=20;
                pass obj=new pass();
                obj.method(i,j);
                System.out.println(i); // unchanged
                System.out.println(j);}}
```

**Example: pass by reference ->**

## Pass by Reference

```
public class pass
{
        public void method(pr obj)
                {
                        System.out.println(++obj.i);
                        System.out.println(++obj.j);
                }
}
class pr
{       int i=10;
        int j=20;
public static void main(String args[])
        {
                pass obj=new pass();
                pr obj1=new pr();
                obj.method(obj1);
                System.out.println(obj1.i);
                System.out.println(obj1.j);}}
```

**Using Static ->**

**Inheritance**

To inherit you need to use the keyword extends

Public class myclass extends otherclass

Now the child class myclass is substitutable to parent otherclass.

Explain also

Super –ex: super.I / super() / super.method(); Should be the first one in a constructor to call the parent class constructor
This - ex: this.variable_name
Abstract classes – ex: public abstract class A
Interfaces - ex: public interface A

**Understanding static**

To define a class member that will be used independently of any object of that class. When a member is declared as static, it can be accessed before any objects of its class are created, and without any reference to any object.

You can declare both methods and variables as static.
Methods declared as static have several restrictions like

1. They can only call other static methods.
2. They must only access static data.
3. They cannot refer to this or super in any way.
4. Static variables cannot be declared inside methods nor constructors nor inner classes.

You can even create static blocks to initialize values.

**Example Static ->**

```
public class staticdemo  // with flow of execution
{
static int a=10; // the first one to execute
staticdemo() // third one to execute

{    System.out.println("inside constructor");   }

static void method()
{          System.out.println("inside static method");      }
static  // the second one to execute
{          System.out.println("inside static block"+a);     }

public static void main(String a[])
          {
                    new staticdemo();
          }
}
```

**Introducing final ->**

## Introducing final

A variable or method or class can be declared as final. Doing so prevents its contents from being modified.

Ex:

1. final int I=0; // cannot be changed. (constant)
2. Final public class sample // cannot be inherited.
3. Final public void method() // cannot be overriden.

**Exception Handling ->**

# Exception Handling

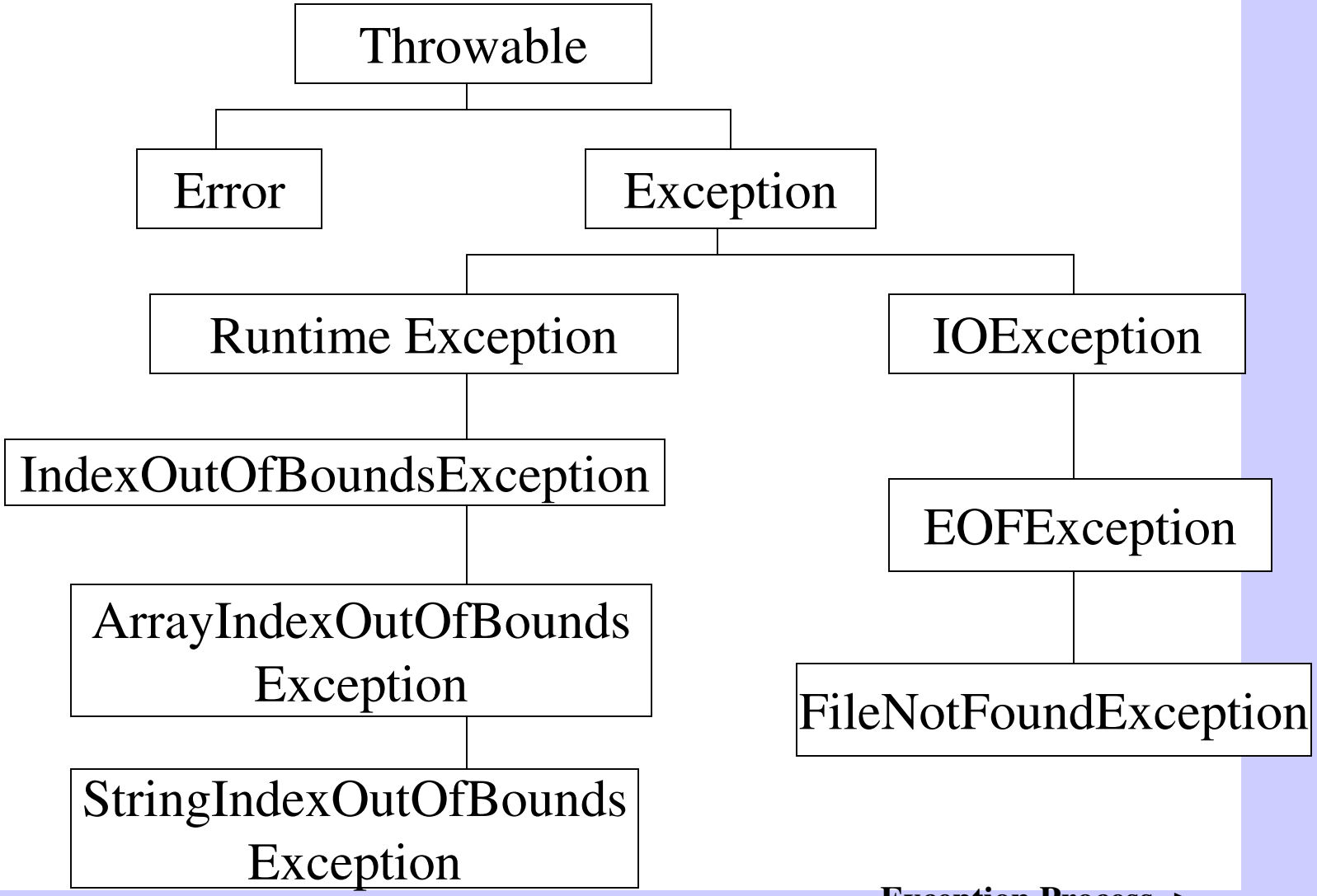
An Exception is an abnormal condition that arises in a code sequence at runtime. In other words exception can be called as a runtime error.

Ex: for Abnormal Error occurrences
- file not found
-class file in a wrong format
- network connection not found

Illegal operators
- Division by zero
- array reference out of boundry
- string reference out of boundry

**Class Hierarchy ->**

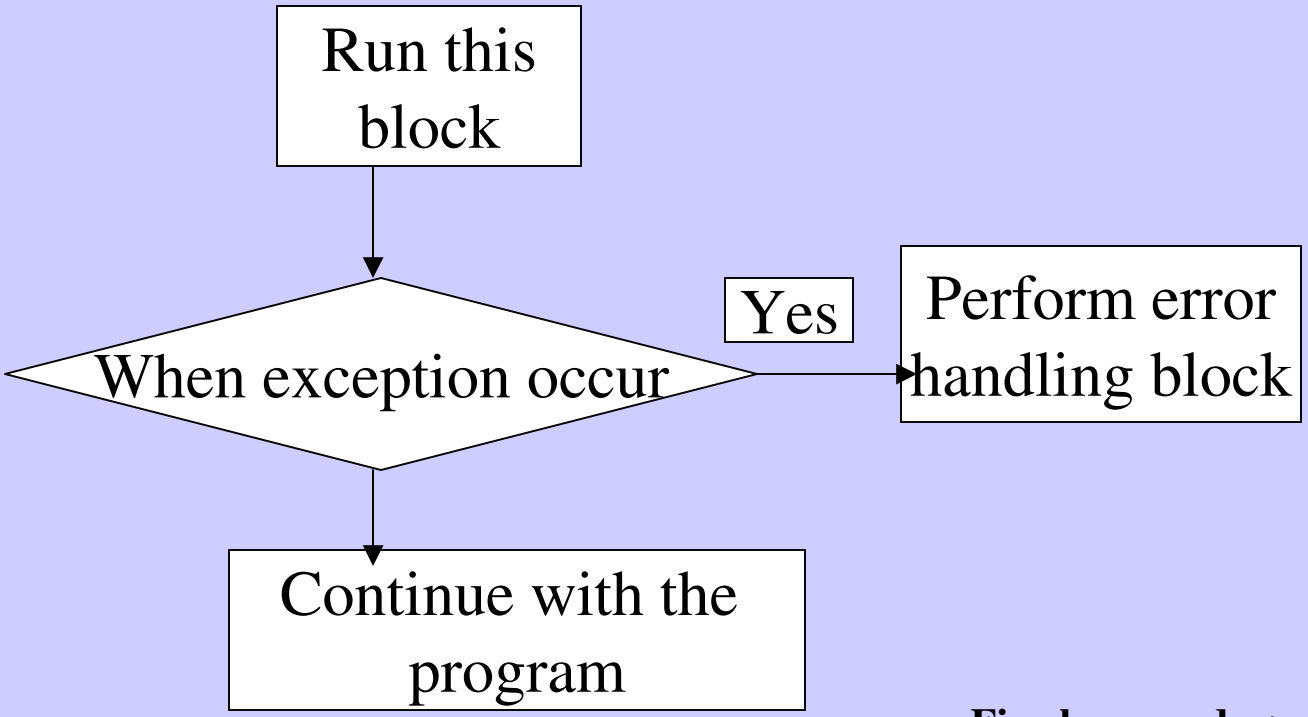
Exception is a class. It is a subclass of Throwable class. Like files the entire exception hierarchy is specified in a class family.




**Exception Process ->**

## Process of Exception handling.

1. Try to execute a block
2. When an error occurs, exception is thrown
3. Catch it and handle appropriately
4. Then continue with the program
5. Finally execute some block if necessary.




**Five key words ->**

Java exception handling is managed via five key words.
1. try, 2. catch, 3. throw,4. throws and 5. finally.

Try - Program statements that you want to monitor for exceptions are contained in a try block.
Catch - To catch the exception and handle in some manner.
Throw - To manually throw an exception, use the keyword throw (System generated exception are automatically thrown by the java run time system)
throws - Any exception that is thrown out of a method must be specified as such by a throws clause.
Finally block - any code that must be executed before a methods returns is put in finally block.

**Example Template ->**

The general form of an exception handling block.

```
try{
// block of code to monitor for errors.
}
catch(Exceptiontype1 e){
//exception handler for type1
}
catch(exceptiontype2 e){
//exception handler for type2
}
//……..
finally{
// block of code to be executed before try block ends
}
```

**Example ->**

Ex:

```
class excepdemo{
public static void main(String args[]){
try{
int I=25;
int z=0;
int j;
j=I/z;
} catch(ArithmeticException e){
System.out.println(“Division by zero error”);
}
finally{
System.out.println(“Out of try block”);
}

} }
```

**Demonstrate multiple catch statements ->**

```
//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[43]=99;
}catch(ArithmeticException e){
System.out.println("Divide by zero error"+e);
}
catch(ArrayIndexOutOfBoundsException e){
System.out.println(" array index oob:"+e);
}
System.out.println("After all the try and catch blocks");
}}
```

**Nested Try Statements ->**

```
// An example of Nested try statements
class nesttry{
public static void main(String args[]){
try{
int a=args.length;
int b=42/a;
System.out.println("a="+a);
        try{   // nested try block
          if(a==1) a=a/(a-a);
          if (a==2)      {
                    int c[]={1};
                    c[23]=99;      }
      }catch(ArrayIndexOutOfBoundsException e){
       System.out.println("Array index error :"+e); }
       } catch(ArithmeticException e){
           System.out.println("divice by 0:"+e);  }   }  }
```

**Demonstrating 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.

There are two ways you can obtain a Throwable object using a parameter into a catch clause, or creating one with the new operator.

**Example throw ->**

```
//demonstrate throw
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);  }  }   }
```

**User Defined exception ->**

## Creating your own Exception Subclasses.

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

**Example ->**

```
// This program creates a custom exception type.


class myexception extends Exception {
private int detail;
myexception (int a){
detail=a;
}
public String toString(){
return "MyException[" +detail+ "]";
}
}
```

```
class ExceptionDemo{
static void compute(int a) throws myexception{
System.out.println("called compute(" +a+")");
if(a>10)
throw new myexception(a);
System.out.println("Normal Exit");
}
public static void main(String args[]){
try{
compute(1);
compute(20);
} catch (myexception e){
System.out.println("caught "+e);
}
}
}
```

**MultiThreading ->**

**MultiThreading**

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

**The Thread Class and the Runnable Interface**

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable.

The thread class defines some of the following methods that help manage threads.

getName
getPriority
IsAlive
Join
Run
Sleep
Start
Wait
notify

## 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 important for two reasons.

1. It is the thread from which other child threads will be spawned.
2. It must be the last thread to finish execution. When the main thread stops your program terminates.

```
class currentthread
{
public static void main(String a[])
        {
                Thread t=Thread.currentThread();
                t.setName("My thread");

                for(int i=0;i<5;i++)
                {
                        try{
                        System.out.println(i);
                        Thread.sleep(300);
                        }catch(InterruptedException e){}
                }
        }
}
```

## Creating a Thread

You create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished.

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

```
class mainthread
{
public static void main(String a[])
                {
                new newthread();
                Thread t=Thread.currentThread();
                t.setName("My thread");

for(int i=0;i<5;i++)
                {
                try{
                System.out.println("main"+i);
                Thread.sleep(300);
                }catch(InterruptedException e){}
                }
                }


}
```

```
class newthread implements Runnable
{
                Thread t;
                newthread()
                {
                t=new Thread(this,"demo thread");
                t.start();
                }
                public void run()
                {
                for(int i=0;i<5;i++)
                {
                try{
                System.out.println("child"+i);
                Thread.sleep(200);
                }catch(InterruptedException e){}
                }

                }

}
```

# To Create multiple threads

```
class mainthread
{
public static void main(String a[])
        {
        new newthread(“first);
        new newthread(“second”);
        Thread t=Thread.currentThread();
        t.setName("My thread");

for(int i=0;i<5;i++)
        {
        try{
        System.out.println("main"+i);
        Thread.sleep(300);
        }catch(InterruptedException e){}
        }
        }


}
```

```
class newthread implements Runnable
{
        Thread t;
        newthread(String name)
        {
        t=new Thread(this,name);
        t.start();
        }
        public void run()
        {
        for(int i=0;i<5;i++)
        {
        try{
        System.out.println("child"+i);
        Thread.sleep(200);
        }catch(InterruptedException e){}
        }

        }

}
```

**isAlive and join**

The isAlive() method return true if the thread upon which it is called is still running.

Join method waits until the thread on which it is called terminates.

For ex

T1.join();

System.out.println(“after join”);  // this statement will be executed only after t1 gets terminated.

**Thread Priorities**

You can set the priority for the thread using the following

setPriority(number)  // number from 1 to 10

Or

Thread.NORM_PRIORITY
Thread.MIN_PRIORITY
Thread.MAX_PRIORITY

**Synchronization**

When two or more threads needs access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.  The process by which this is achieved is called synchronization.

Example…

```
class callme
{
        synchronized void call()
        {
        System.out.println("hello");
        try{
                Thread.sleep(1000);
                System.out.println("world");
        }catch(InterruptedException e){}

        }

}

class mainthread
{
public static void main(String a[])
        {
                callme obj=new callme();
                newthread t1=new newthread(obj);
                newthread t2=new newthread(obj);
        }


}
```

```
class newthread implements Runnable
{
        Thread t;
        callme obj;
        newthread(callme obj)
                {
                        this.obj=obj;
                        t=new Thread(this);
                        t.start();
                }
        public void run()
                {
                        obj.call();

                }

}
```

**The synchronized block**

If you want to synchronize access to objects of a class that was not designed by you and does not use synchronized methods, then the following way is used to make synchronized calls.

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

                }
```

**InterThread Communication**

Java includes an elegant interprocess communication mechanism via the wait(), notify(), and notifyAll() methods. These methods are implemented as final methods in object, so all classes have them.

Note: All three methods can be called only from within a synchronized method

Wait() – tells the calling thread to give up the monitor and go to sleep until other thread enters the same monitor and calls notify()
Notify() wakes up the first thread that called wait() on the same object.
notifyAll() – wakes up all the threads that called wait() on the same object.

# Improper implementation without wait and notify

```
class Q
{
        int n;
        synchronized int get()
        {
                System.out.println("Got: "+n);
                return n;
        }
        synchronized void put(int n)
        {
                this.n=n;
                System.out.println("Put: "+n);
        }
}

class producer implements Runnable
{
        Q q;
        producer(Q q)
        {
                this.q=q;
                new Thread(this,"Producer").start();
        }
        public void run()
        {
                int i=0;
                while(true) {
                        q.put(i++);
                }
        }
}
```

```
class consumer implements Runnable
{
        Q q;
        consumer(Q q)
        {
                this.q=q;
                new Thread(this,"consumer").start();
        }
        public void run()
        {
                int i=0;
                while(true) {
                        q.get();
                }
        }
}

class s
{
        public static void main(String args[])
        {
                Q q=new Q();
                new producer(q);
                new consumer(q);
        }

}
```

# Correct Implementation

```
class Q
{
        int n;int temp=0;
        boolean valueset=false;
        synchronized int get()
        {
                if(!valueset)
                        {

        try{
                        wait();

        }catch(Exception e){}
                        }

                System.out.println("Got:
"+n);
                valueset=false;
                notify();
                return n;
        }
        synchronized void put(int n)
        {
                        if(valueset)
                        {

        try{

                wait();

        }catch(Exception e){}
                        }

                this.n=n;
                valueset=true;
                notify();
                System.out.println("Put:
"+n);
        }
}
```

```
class producer implements Runnable
{
        Q q;
        producer(Q q)
        {
                this.q=q;
                new Thread(this,"Producer").start();
        }
        public void run()
        {
                int i=0;
                for(int j=0;j<5;j++) {
                        q.put(i++);
                }
        }
}
class consumer implements Runnable
{
        Q q;
        consumer(Q q)
        {
                this.q=q;
                new Thread(this,"consumer").start();
        }
        public void run()
        {
                int i=0;
                for(int j=0;j<5;j++) {
                        q.get();
                }
        }
}

class s
{
        public static void main(String args[])
        {
                Q q=new Q();

                new producer(q);
                new consumer(q);

        }
}
```

**Deadlock**

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two thread have a circular dependency on a pair of synchronized objects.

Do this exercise

```
class x
{
		synchronized void met(y obj)
		{

		try{Thread.sleep(1000);}catch(Exception e){}
				obj.last();
		}
		synchronized void last()
		{
				System.out.println("inside x's
last");
		}
}

class y
{
		synchronized void met(x obj)
				{

		try{Thread.sleep(1000);}catch(Exception e){}
					obj.last();
				}
		synchronized void last()
				{
				System.out.println("inside y's
last");
				}
}
```

```
public class mythread implements Runnable
{
		x obj1=new x();
		y obj2=new y();

		mythread()
		{

				Thread t=new Thread(this);

				t.start();
				obj1.met(obj2);
		}
		public void run()
		{
				obj2.met(obj1);
		}

		public static void main(String a[])
		{
				new mythread();
		}
}
```

# Java.lang Package

Java.lang package is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of java programming.

**It includes some of the following classes**

| | | | | |
|---|---|---|---|---|
| Boolean | Long | String | Byte | Character |
| Number | System | Math | StringBuffer | |
| Class | Object | Thread | ClassLoader | |
| Package | ThreadGroup | Compiler | Process | |
| ThreadLocal | Double | Runtime | Throwable | |
| Float | RuntimePermission | | Void | |
| Integer | Short | | | |

**Interfaces:** Cloneable Comparable Runnable

**Simple Type Wrappers**

As you know java uses simple types, such as int and char, for performance reasons.
They are passed by value to methods, if in case you want to refer the instance then you can wrap the simple data type into its corresponding wrapper class.
More over Collections classes don't accept simple datatypes they accept only objects.

So now let us explore few of the wrapper classes and their methods in lang package.

## Number

Number is a abstract class that is implemented by the classes that wrap the numeric types byte, short, int, long, float and double with six concrete subclasses that hold explicit values of each type.

**Double, Float, Byte, Short, Integer, and Long**

Number has abstract methods that return value of the object in each of the different formats.
Viz: byte byteValue(), double doubleValue(), float floatValue(), int intValue(), long longValue(), short shortValue().

Ex:

```
int i=109;
Integer in=new Integer(i);
int j=(int)in;// error
int j=in.intValue(); // correct
```

## Double and Float

```
public class dob
{
        public static void main(String args[])
                {
                        Double d=new Double(1/0.);
                        Double d2=new Double(0/0.);
                        System.out.println(d.isInfinite());
                        System.out.println(d.isNaN());
                }
}
```

**Methods**

isInfinite() - Returns true if the value being tested is infinitely large or small in magnitude.

isNAN() - Returns true if the value being tested is not a number.

**Converting Numbers to and from Strings**

**Integer class**

```
public class convert

{
        public static void main(String a[])
                {
                        int i=Integer.parseInt(a[0]);

                }
}
```

**Character Class**

Declare a char array and check the following methods

Char a[]={ ‘a’, ‘5’, ‘?’ };

Character.isDigit(a[I])
Character.isLetter(a[I])
Character.isWhiteSpace(
Character.isUpperCase
Character.isLowerCase

**Runtime Class**

The Runtime class encapsulates the run-time environment, you cannot instantiate runtime class, but you can get a reference to the current Runtime object by calling the static method Runtime.getRuntime();