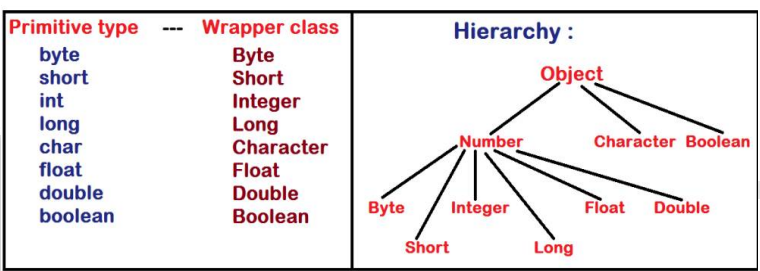
Wrapper Classes:

1. Java is not fully object oriented programming language because it supports primitive data types.
2. Using primitive data types we cannot develop fully object oriented java application.
3. We use wrapping to convert primitive data into objects and object data into primitives.
4. Every primitive data type will have corresponding wrapper classes.



1. All the wrapper classes belong to "lang" package.
2. All the Java API classes belong to a specific pre-defined packages.
3. "java" is the super package of all.
4. Naming conventions of package is "SINGLE WORD IN LOWER CASE"

• Examples... java, lang, io, awt, net, sql.....

Different Conversions by means of Wrapper classes:

In general, we perform six conversions in any given java application:

1) Primitive to Object (Boxing)

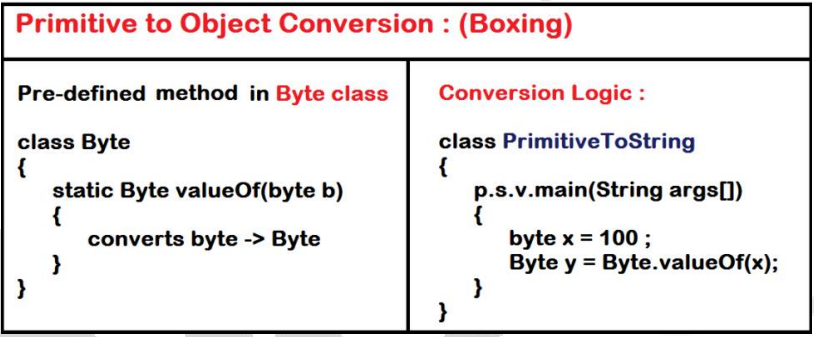
2) Object to Primitive (UN boxing)

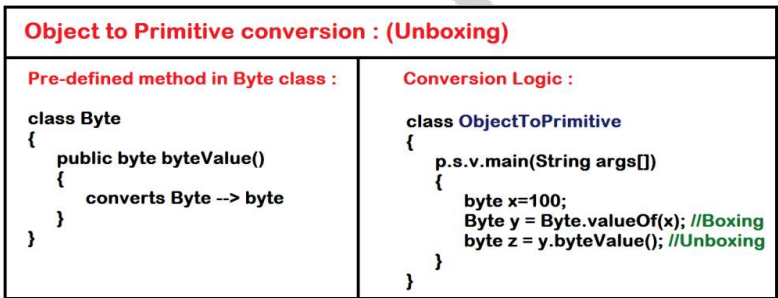
3) Primitive to String

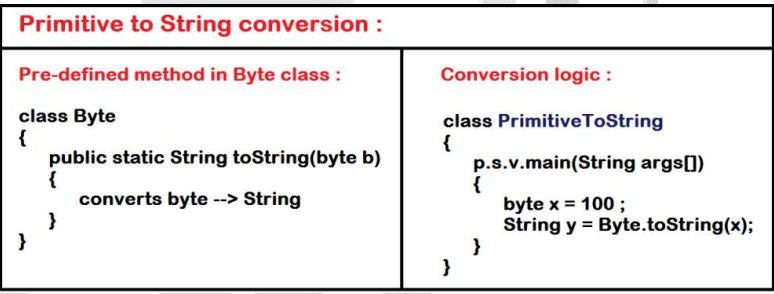
4) String to Primitive

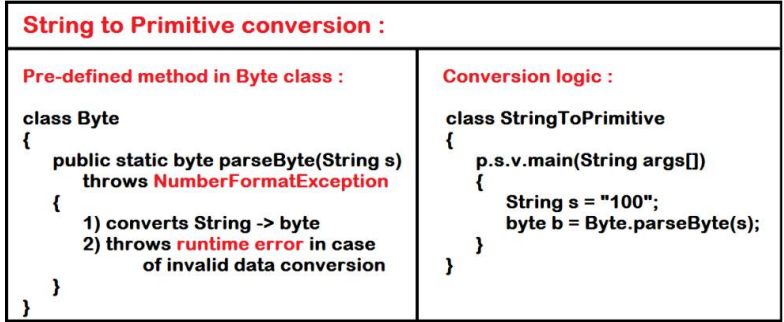
5) Object to String

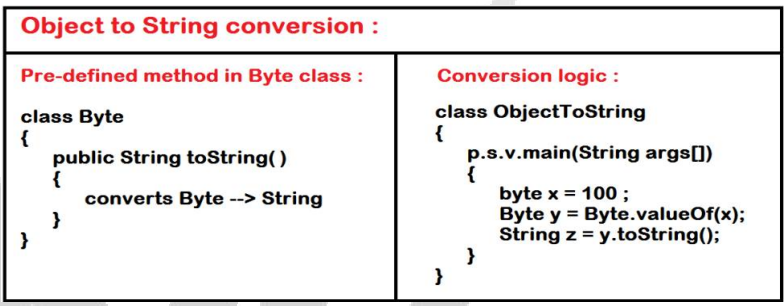
6) String to Object

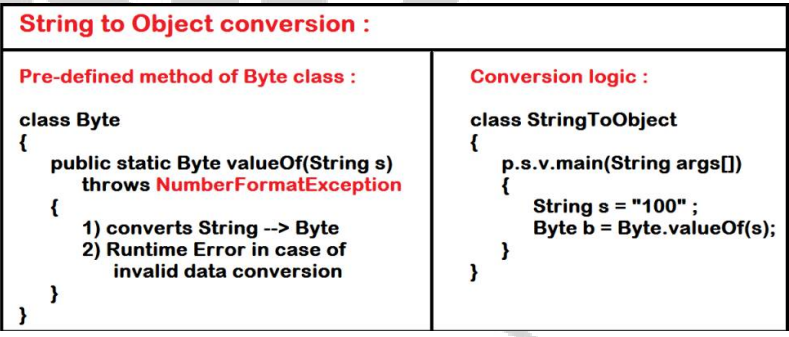












Packages:

1. Package is a java folder or directory.
2. Package is a related group of classes, abstract classes and interfaces.
3. Generally in our System we are creating folders to separate the related information. So that we can access effectively.
4. Java API (library) is having around 5000 pre-defined classes.
5. Every class belongs to a specified pre-defined package.
6. Package naming convention “SINGLE WORD, SMALL CASE“. Examples….. java, lang, nit, io…..
7. “java” is the super package of all.

Creation of Package:

1. “package” is the pre-defined keyword to create user-defined packages in java application.

Syntax : package <Identity>;

Example : package infosys ;

1. Package statement must be the first statement in java application.

Example program:

package infosys ;

import java.lang.System ; //optional

class First

{

public static void main(String args[ ])

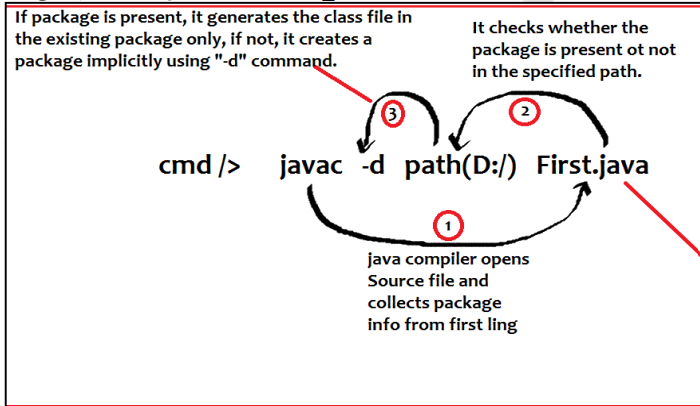
{

System.out.println("welcome");

}

}

Compiling java program including package statement:



Accessing the members of one class from another class of same package:

package infosys ;

import java.lang.System ; //optional

class First {

void fun( ) {

System.out.println("welcome to java world”);

}

}

package infosys;

import java.lang.String;

import java.lang.System;

class Second {

public static void main(String[] args) {

System.out.println("Happy Coding");

First obj = new First();

obj.fun();

}

}

Accessing the members of another package class:

package demo ;

import java.lang.String;

import java.lang.System;

class Second {

public static void main(String[] args) {

System.out.println("Second class main method");

First obj = new First();

obj.fun();

}

}

Accessing the members of another package class:

import:

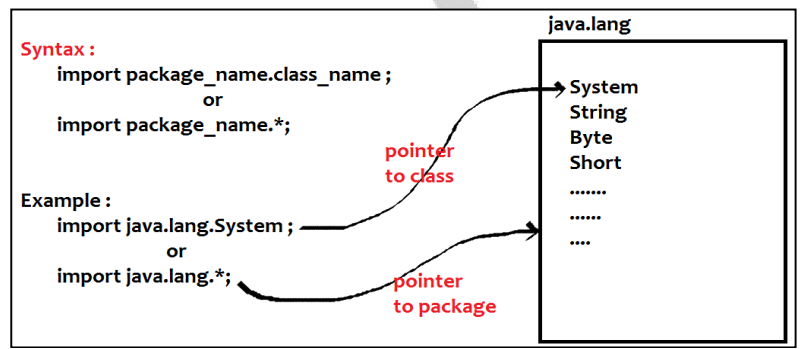
1. It is a pre-defined keyword.

2. Used to connect classes in java application

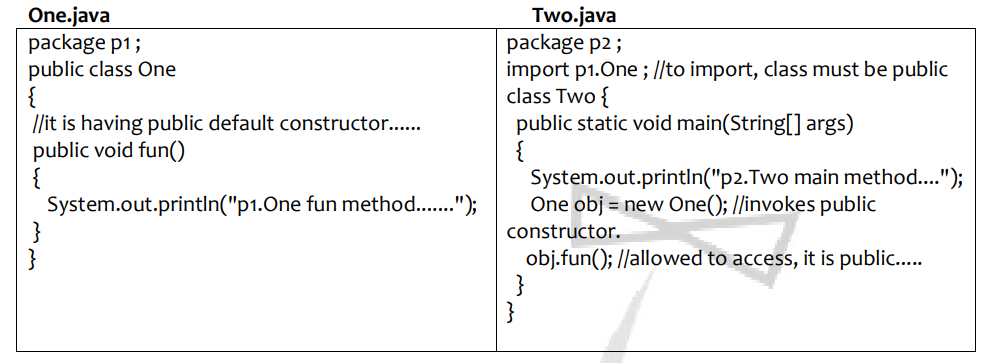
3. Import statement creates a pointer either “to package” or “to class”.

4. Using internal pointer, it loads the class dynamically while application is running.

5. Only “public classes” can be accessed from outer package.



6. If class is public, default constructor is also public.



NOTE:

1.“import” statement doesn’t load the class directly.

2. It creates only internal pointer to package or class.

3. “import” statement only loads the class at runtime, on use.

4. un used classes will not be loaded into JVM.

Fully Qualified Names:

1. Writing full package name while using any class.
2. We can replace “import” statement with fully qualified name.

class Fullyqualifiedname

{

    public static void main(String args[])

    {

        java.util.Scanner sc=new java.util.Scanner(System.in);

        int a;

        System.out.println("enter the value of a");

        a=sc.nextInt();

        System.out.println(a);

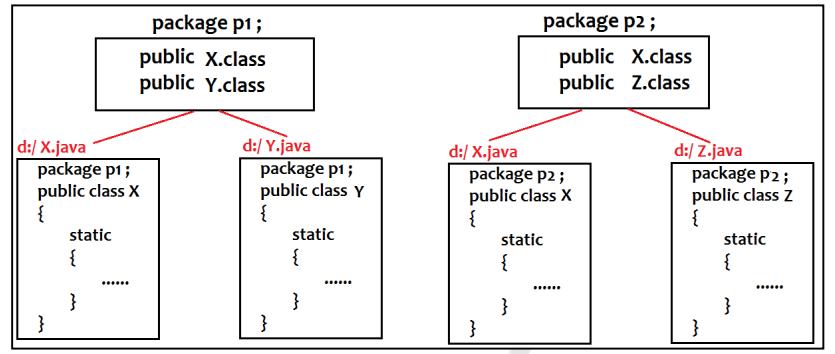
    }

}

Advantages of Packages:

1. The main advantage of packages is, “Avoiding collisions between class names”.
2. Consider, we need to define 2 classes with the same identity in java app, we need to place these 2 classes in 2 different packages.
3. And we can access only by using “Fully Qualified Name”

Task:



package p1 ;

public class X

{

static {

System.out.println("p1.X is loading....");

}

}

package p1 ;

public class Y

{

static

{

System.out.println("p1.Y is loading.....");

}

}

package p2 ;

public class X

{

Static

{

System.out.println("p2.X is loading....");

}

}

package p2 ;

public class Z

{

static { System.out.println("p2.Z is loading.....");

}

}

import p1.\*;

import p2.\*;

class ABC

{

public static void main(String[] args)

{

new Y(); new Z();

}

}

package p3 ;

import p1.\*;

import p2.\*;

class ABC {

public static void main(String[] args) {

new X(); //CE : collisions

}

}

Solution:

package p3 ;

import p1.\*;

import p2.\*;

class ABC {

public static void main(String[] args) {

new p1.X();

new p2.X();

}

}

NOTE: we can define a class with the same name of java-API class.

KEY POINTS:

1. Class loader sub system is the pre-defined program.
2. It loads the class into JVM
3. It gives first priority to Current Working Directory (local drive).
4. If class is not present in CWD, then it searches in API.

Example: class System {

public static void main(String[] args) {

//System.out.println("Hello World!"); //CE :

java.lang.System.out.println("Hello World!");

}

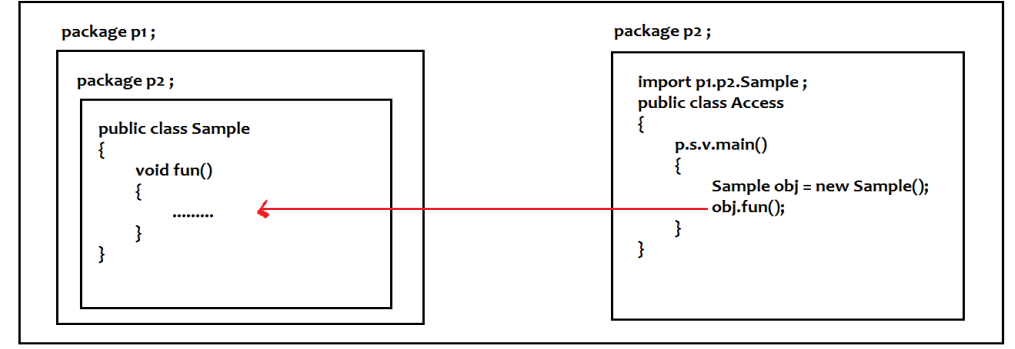
}

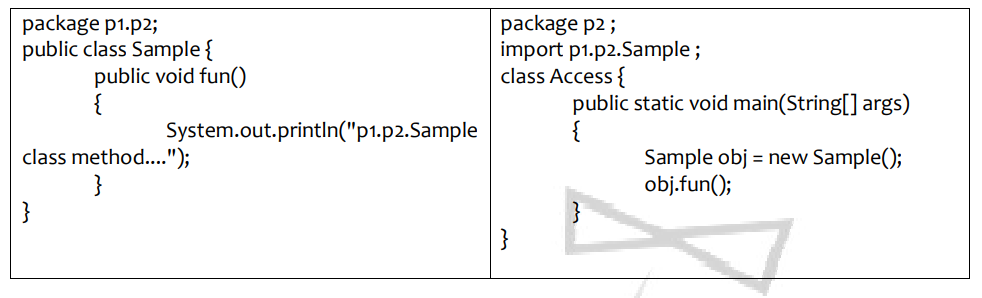
Creation of Subpackages:

1. Using “package” keyword, we can create sub packages.

Syntax : Package Parent\_package.child\_package ;

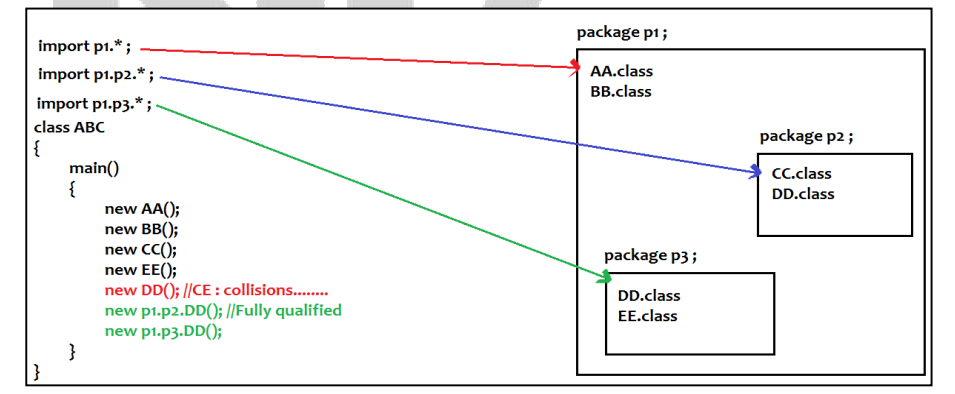
Example : package maths.arithmetic ;





Importing sub packages:

1. If we import only Parent package, internal pointer creates to parent package only then we access only Parent package classes.
2. To access the members of Child package, we must import child package explicitly.



Access Specifiers:

Java supports four Access specifiers :

1. public

2. private

3.<package> or<default>

4. protected.

private:

1. Private member belongs to particular object. Complete hiding from outside world.
2. We can’t apply to class.
3. If we apply to class, complete object will be hidden; hence no one can communicate with that object. Such type of objects is useless.
4. We can apply private functionality only to class members such as variable, methods & constructors.

public:

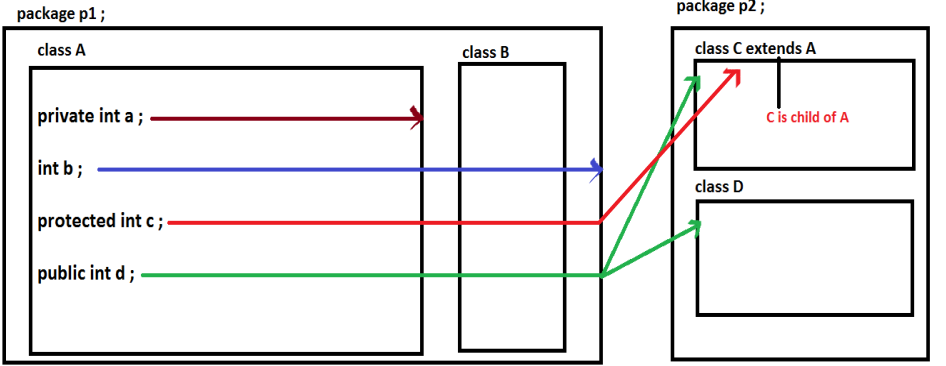
1. We can apply public modifier to class and its members.
2. Public class can be accessed anywhere in java application.
3. In real time applications, all the classes must be defined as public.

<package> :

1. Can be applied to class members.
2. By default every class and its member is of package level.
3. It is the default access modifier in java application.
4. Package level members can be accessed only within the package.

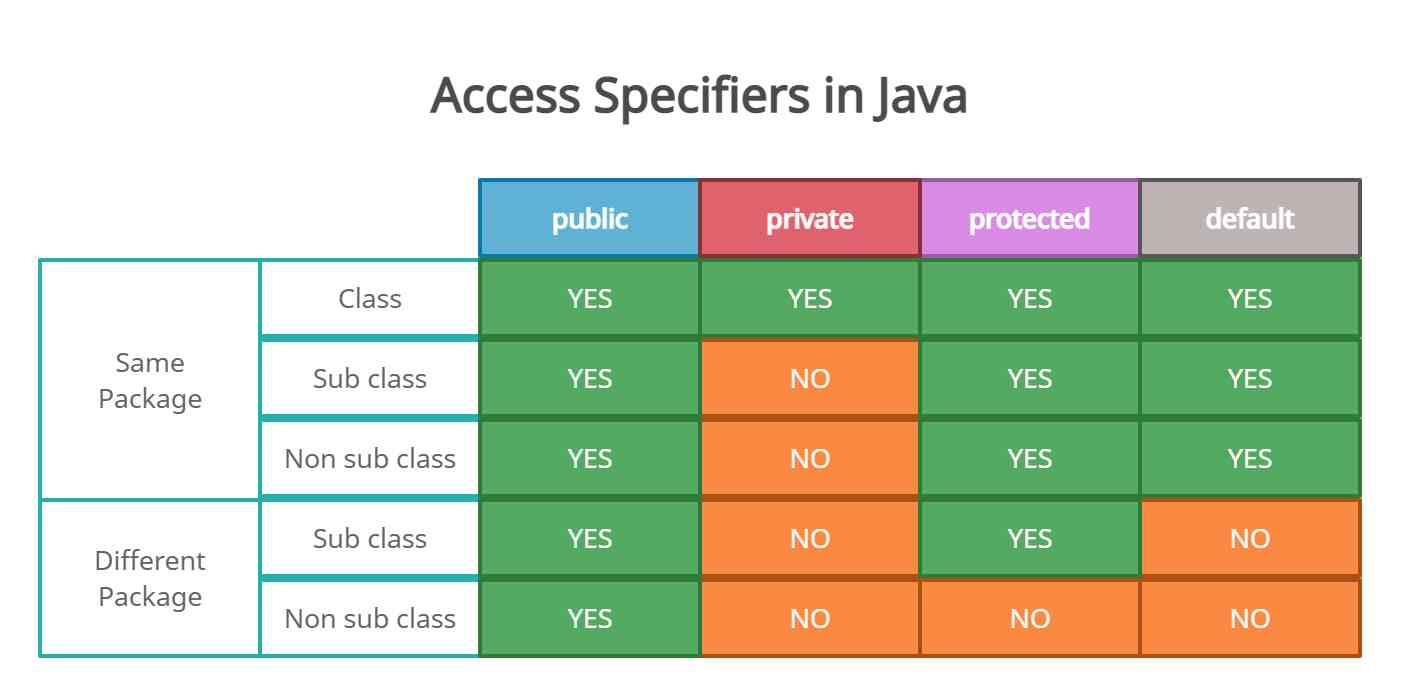
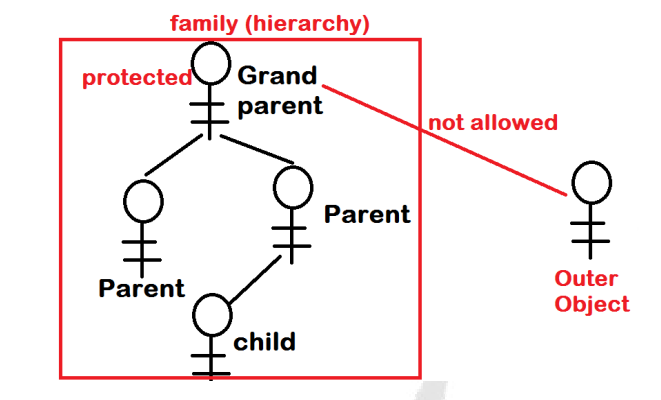
protected:

1. Can be applied only to class members.
2. protected members can be accessed within the package and only from the subclasses of outer package



A class cannot be protected:

1. Protected members can be accessed only in the Hierarchy (family of classes).
2. If a class is protected, the object functionality can be accessed only within the hierarchy in its entire life time.
3. Such type of object cannot exist in the real world.
4. Hence we cannot apply protected to class



Exception Handling:

During program Compilation and Execution there is a chance to get 3 kinds of Errors

Compile time errors:

1. While writing programs using any High level languages, we need to follow set of language specification class syntactical rules.
2. Violation of these rules produces, compile time errors.

Logical errors:

1. Logical error is the compiler error and runtime error.
2. It produces unexpected results when we run the application.

Runtime errors:

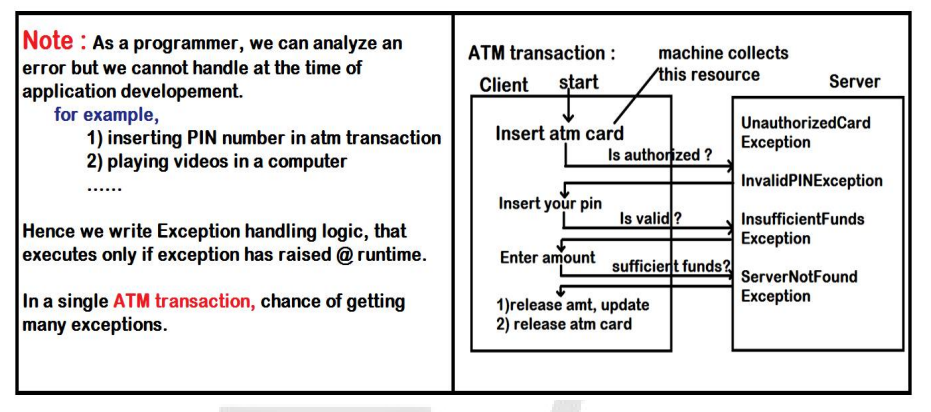
1. Violation of JVM rules.
2. As a programmer some of the errors we can analyze at the time of application development but we cannot handle.
3. As a programmer we can write the logic but that executes only when problem has risen while application is running.

Causes:

1. Abnormal termination of program

2. Informal information to End user

3. Improper shutdown of Resources.



Internal understanding when exception has raised:

1. Exception occurs only either inside the block or method (because every statement must be placed inside the block or method only in java application).
2. When exception has raised, that block or method creates an exception object which contains the complete information of that error including.....

1. Name of error

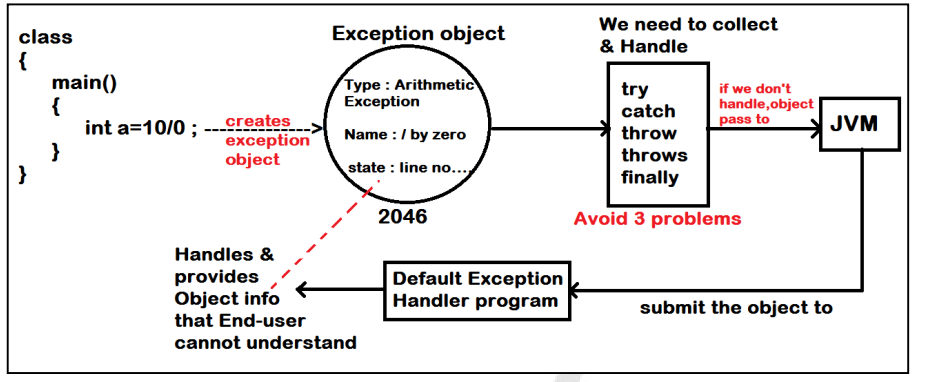
2. Type of error

3. State of error

3. Once object has been created, it passes to the runtime system (JVM) then JVM

handles that exception with the help of Default Exception Handler.

1. Hence mean while, we need to catch that exception and handle before the object reaches JVM, to avoid abnormal termination and continue with remaining statements execution in the application.



Chances to get Exception:

User has entered invalid arithmetic input causes “ArithmeticException”

1. Accessing the data of array which is out of bounds causes “ArrayIndexOutOfBoundsException”.

class Test {

public static void main(String[] args) {

int arr[ ] = new int[5];

arr[9] = 11 ; //Exception :

}

}

1. Accessing the functionality of object using null pointer causes “NullPointerException”.

class Test {

static Test obj = null ;

public static void main(String[] args) {

Test.obj.check(); //Exception

}

void check(){

}

}

1. Invalid data conversion results “NumberFormatException”

class Test {

public static void main(String[] args) {

String s = "abcd";

byte b = Byte.parseByte(s);

}

}

1. Downcasting the Object by providing another class name causes “ClassCastException”
2. Trying to access the data from the file which is not present in the file system “FileNotFoundException”
3. Trying connect with database which is not ready “SQLException”
4. Trying to retrieve IP address of system which is not connection the LAN “UnknownHostException”

Keywords used in Exception Handling:

Java API provides 5 keywords to handle exceptions which are 1. try 2. catch 3. finally 4. throws 5. Throw

try:

1. It is a pre-defined keyword
2. Used to define a block of statements.
3. Doubtful code that raises exception must be placed inside the try-block.
4. if exception raised inside the block, it creates Exception Object and passes to "catch" block instead of JVM.

catch:

1. It is a keyword
2. used to define block which contains Exception handling code that raises in the corresponding try-block.

syntax:

try

{

//doubtful code...

// related info…..

}

catch(<Exception\_type> <identifier>)

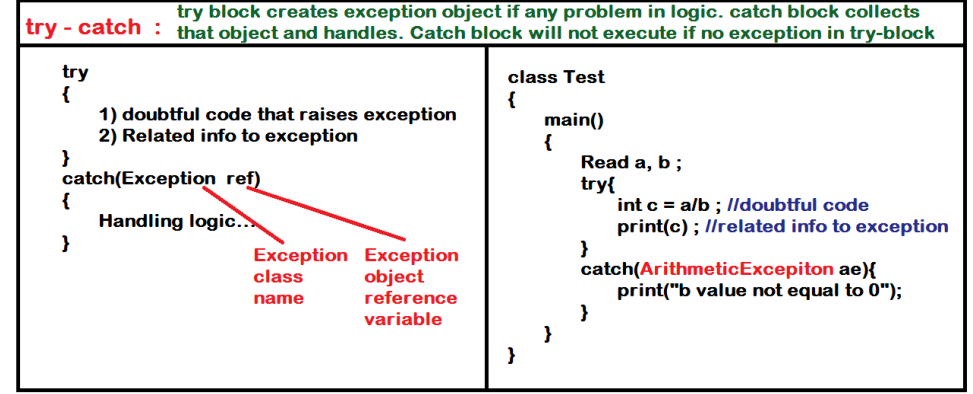
{

//Handling code....

}

<Exception\_type> specifies class name of Exception

<Identifier> specifies variable that hold exception object reference



class ExceptionDemo {

public static void main(String[] args) {

int a,b,c;

System.out.println("Enter two integers :");

java.util.Scanner sc = new java.util.Scanner(System.in);

a = sc.nextInt();

b = sc.nextInt();

try{

c = a/b;

System.out.println("result : "+c);//related information of exception must be

placed inside the try-block

}

catch (ArithmeticException ae){

System.out.println("Denominator should not be zero....");

}

System.out.println("end of program");

}

}

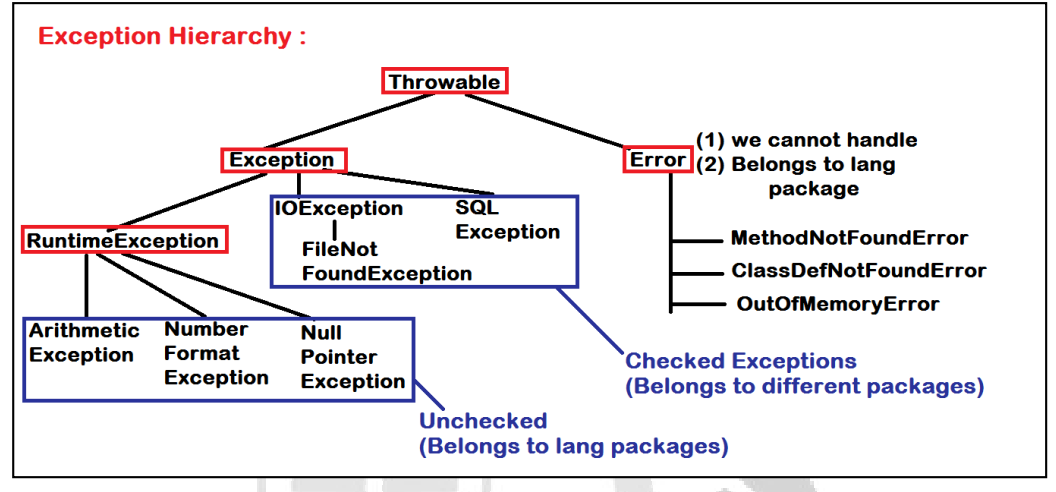
Exception Hierarchy:

1. java.lang.Throwable is the super class of all the Exceptions.
2. It has 2 direct sub classes Exception & Error
3. Errors can't be handled in java application.
4. Sub classes of RuntimeException are called "Unchecked Exceptions" belongs to "java.lang" package.

Sub classes of Exception are called "Checked Exceptions" belongs to corresponding package.

1. IOException (java.io)

2. SQLException(java.sql)



try with multiple catch blocks :

1. We can define set of instructions in one try block.
2. try with multiple catch blocks used to handle different exceptions occurred in different instructions of try block.
3. We use try with multiple catch blocks when one task depends on more than one step of process and in each step there is a chance of getting exception

syntax :

try

{

-------- //chance to get Exception1

……

-------- // chance to get Exception2

}

catch(Exception1 <var>)

{

//Exception1 handling logic...

}

catch(Exception2 <var>)

{

//Exception2 handling logic...

}

Note:

1. If Exception has risen in the try block, control terminates the execution of followed statements in try-block and executes the catch block.
2. After execution of catch block, control will never back to try.
3. If no exception in try block, no catch block will be executed.
4. It is possible to handle one Exception at a time among multiple catch block

Example:

class MultiCatch

{

public static void main(String args[ ])

{

try

{

String input = args[0] ;

System.out.println("Input value : "+input);

byte output = Byte.parseByte(input) ;

System.out.println("Converted value : "+output);

System.out.println("Task completed successfully.....");

}

catch (ArrayIndexOutOfBoundsException ae)

{

System.out.println("No input , task failed...");

}

catch(NumberFormatException ne)

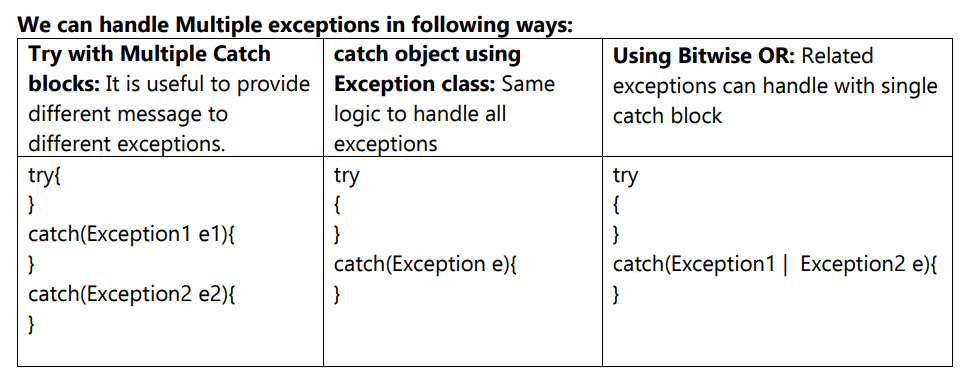
{

System.out.println("Invalid input, task failed...");

}

System.out.println("End....");

}



Unchecked v/s Checked:

Unchecked:

1. These types of exceptions occur in case of logical error.
2. Handling of Unchecked exception takes care by programmer only.
3. Compiler will not report though we don't handle unchecked exception in the application.
4. But handling of unchecked exception avoids abnormal termination of program.

Checked Exception:

1.While working with any outside resource (file, server, database, io devices.....), chance to get Checked Exception.

2. After working with outside resource, it is mandatory to release (shutdown) the resource properly to avoid loss of data.

3. Outside resource is not the property of Java application. Hence we must handle every checked exception in the application.If not, compiler reports error and class file will not be generated

Note:

1. If any method throws Exception, we need to check whether it is Checked or Unchecked.
2. Sub class of RuntimeException is Unchecked.
3. Sub class of Exception is checked.

Example:

import java.io.FileInputStream;

import java.io.FileNotFoundException;

class CheckedException {

public static void main(String[] args) {

try{

FileInputStream fis = new FileInputStream("g:/in.txt");

System.out.println("file opened successfully....");

}

catch (FileNotFoundException f){

System.out.println("File doesn't exist.....");

}

}

}

finally:

1. A block of instructions.
2. Used to release a resource in Exception handling
3. Only using in case Checked exceptions.
4. finally block executes whether or not an Exception has raised in the try-block.
5. Finally block executes in case of abnormal termination of program also.

NOTE:

1. try-block cannot be present without either catch-block or finally-block.

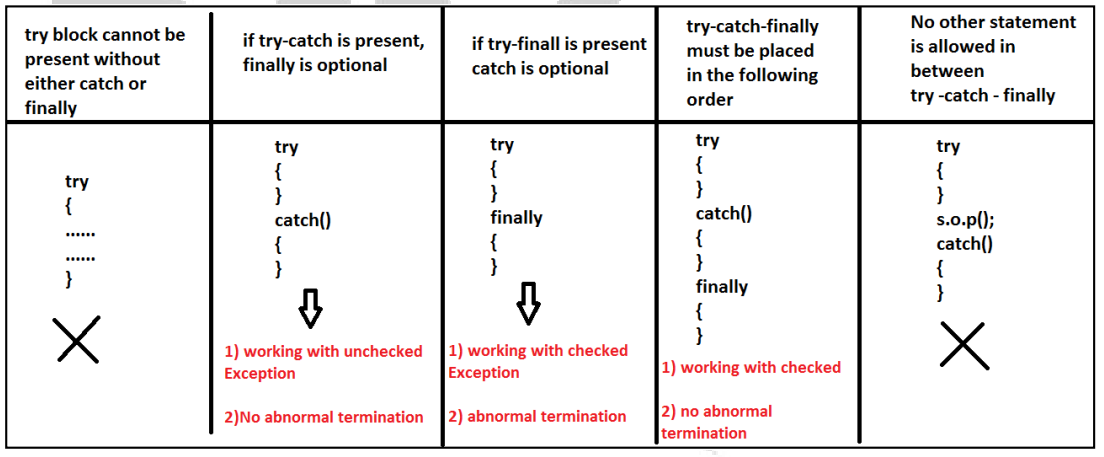
a. if try-catch is present, finally-block is optional

b. if try-finally is present, catch-block is optional

2. finally block gets execute in case of Abnormal termination of program also.

3. The order of 3 block as try-catch-finally.

4. No other statement is allowed in between try-catch-finally.



Examples on finally block:

class Exceptions {

public static void main(String[] args)

{

int a[ ] = {10,20,30,40,50};

System.out.println("Array Elements are. .....");

Try

{

for(int i=0 ; i<=a.length ; i++)

{

System.out.println(a[i]);

}

}

/\*catch (ArrayIndexOutOfBoundsException ae){

System.out.println("out of bounds");

} \*/

finally{

System.out.println("Finally block ");

}

}

}

2. import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

class CheckedException

{

public static void main(String[] args)

{

FileInputStream fis = null;

try

{

fis = new FileInputStream("CheckedException.java");

System.out.println("file opened...");

}

catch (FileNotFoundException fnfe)

{

System.out.println("file doesn't exist...");

}

finally

{

try

{

fis.close();

System.out.println("Connection closed...");

}

catch (IOException io)

{

System.out.println("IO error....");

}

catch(NullPointerException npe)

{

System.out.println("Couldn't close the file....");

}

}

}

}

throws:

If a method is unable to handle an exception, it can throws that Exception.

1. Mostly used in case of Checked Exception.
2. If any method throws Exception, that must be handled by calling method.
3. If we not handle the exceptions in any level, finally JVM pass the exception object to Exception Handler.
4. Exception Handler provides generalized information to end-user.

Example:

import java.io.\*;

class CheckedExceptionDemo

{

public static void main(String[] args) throws Exception

{

FileInputStream fis = new FileInputStream("CheckedExceptionDemo.java");

System.out.println("file opened...");

fis.close();

System.out.println("file closed....");

}

}

throw:

1. Used to throw an exception object explicitly.
2. It is possible to define user defined Exception classes in java programming.
3. If any pre-defined exception has risen when problem, occurred implicitly object will be created for that exception class and thrown....
4. But in case of user-defined exception, if problem occurred, we need to create object explicitly to that user-defined exception class and using "throw" keyword we need to throw....

Creating User-defined Exceptions:

It is allowed to create user-defined exception which is either checked exceptions or un-checked.

class MyException extends RuntimeException

{

//MyException is un-checked

}

class MyException extends Exception

{

//MyException is checked

}

1. Pre-defined Exception Objects will be thrown by the JVM implicitly if raised.
2. But User-defined Exception Objects must be created manually by the Programmer and has to throw explicitly using "throw" keyword.

Example: InsufficientFundsException, InvalidPINException….

Note: Most of the User Exceptions are Checked.

Syntax to throw an Exception:

throw <Throwable\_object>;

Example:

class UserException extends Exception{

String name;

public UserException(String name){

this.name = name ;

}

public String getErrorMessage(){

return this.name;

}

}

class Exceptions

{

public static void main(String[] args) {

try{

UserException obj = new UserException("Error Message");

throw obj;

}

catch (UserException u){

System.out.println("Info : "+u.getErrorMessage());

}

}

}

Example2:

class InsufficientFundsException extends Exception{

int needs;

InsufficientFundsException(int needs){

this.needs = needs;

}

int getNeeds(){

return this.needs;

}

}

class Account{

int balance;

Account(int amount){

this.balance = amount;

}

int getBalance(){

return this.balance;

}

void withdraw(int amount) throws InsufficientFundsException{

System.out.println("trying to withdraw ... : " + amount);

System.out.println("balance in account : "+ this.balance);

if(amount <= this.balance){

System.out.println("pls collect the cash :"+amount);

this.balance = this.balance-amount;

}

else{

int needs = amount - this.balance;

throw new InsufficientFundsException(needs);

}

}

}

class Bank

{

public static void main(String[] args)

{

Account acc = new Account(6000);

System.out.println("initial balance : "+acc.getBalance());

try

{

acc.withdraw(4000);

System.out.println("balance : "+acc.getBalance());

acc.withdraw(3000);

System.out.println("balance : "+acc.getBalance());

}

catch (InsufficientFundsException ife){

System.out.println("transaction failed due to less amount in the account :

"+ife.getNeeds());

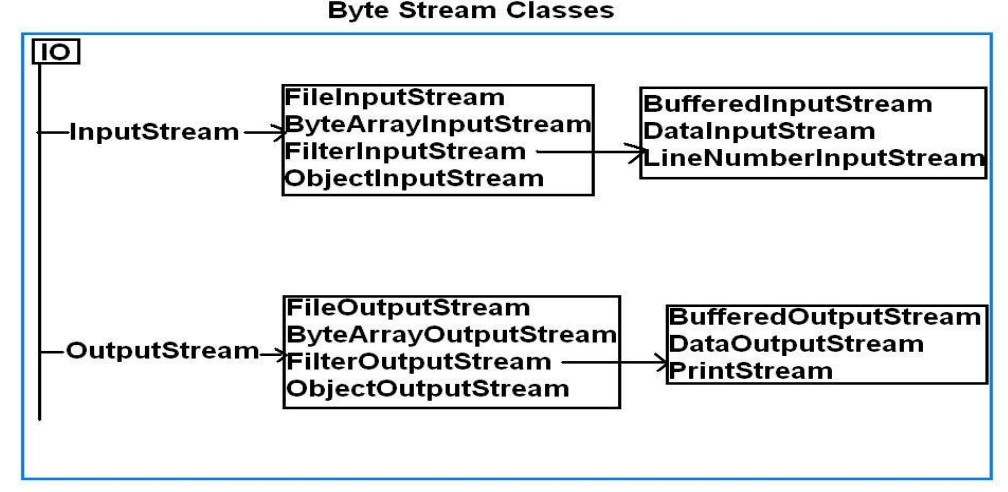
}

System.out.println("end");

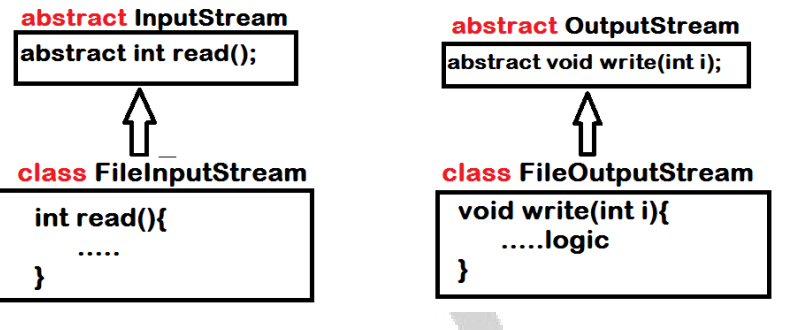
}}

IO Streams:

1. Byte streams are those in which the data will be transferred one byte at a time between primary memory to secondary memory and secondary memory to primary memory either locally or globally.
2. java.io.\* package contains some set of classes and interfaces which will transfer one byte at a time.
3. Programs use byte streams to perform input and output of 8-bit bytes.
4. All byte stream classes are descended from InputStream and OutputStream.



1. The two main classes used to perform Byte stream operations are InputStream & OutputStream.
2. These classes are abstract classes.
3. We use extensions such as FileInputStream & FileOutputStream classes to implements programming.



FileInputStream class:

1. This is the concrete (which we can create an object or it contains all defined methods) subclass of all InputStream class.

public class FileInputStream extends InputStream

1. This class is always used to open the file in read mode.
2. Opening the file in read mode is nothing but creating an object of FileInputStream class. FileInputStream (String fname) throws FileNotFoundException

FileInputStream fis=new FileInputStream (“abc.txt”);

1. If the file name abc.txt does not exist then an object of FileInputStream fis is null and hence we get FileNotFoundException.

FileOutputStream class:

1. This is the concrete sub-class of all OutputStream classes.

public class FileOutputStream extends OutputStream

1. This class is always used for opening the file in write mode is nothing but creating an object of FileOutputStream class.

FileOutputStream(String filePath) FileOutputStream(File fileObj) FileOutputStream(String filePath, boolean append)

1. If the flag is true the data will be appended to the existing file else if flag is false the data will be overlapped with the existing file data.
2. If the file is opened in write mode then the object of FileOutputStream will point to that file which is opened in write mode.
3. If the file is unable to open in write mode an object of FileOutputStream contains null.

Example:

import java.io.\* ;

class ByteStreams

{

public static void main(String[] args) throws IOException

{

FileInputStream fis = null ;

Try

{

fis = new FileInputStream("d:/ByteStreams.java");

System.out.println("file opened.....");

int ch ;

while((ch = fis.read()) != -1){

System.out.print((char)ch);

}

}

catch (FileNotFoundException fnfe)

{

System.out.println("Exception : "+fnfe.getMessage());

}

Finally

{

if(fis != null)

{

fis.close();

System.out.println("file closed....");

}

}

}

}

Writing data into file:

1. We need to represent output file in the construction of FileOutputStream class.
2. If file is present with the specified name, it opens the file and deletes the existing content.
3. If file is not present, it creates the new file with specified name.

Example:

import java.io.\* ;

class WriteInfo{

public static void main(String[] args) throws IOException{

FileInputStream fis = null ;

FileOutputStream fos = null ;

try{

fis = new FileInputStream("d:/WriteInfo.java");

//fos = new FileOutputStream("g:/output.txt");

fos = new FileOutputStream("g:/output.txt" , true);

int ch ;

while((ch = fis.read()) != -1){

fos.write(ch);

}

System.out.println("copied....");

}

finally{

if(fis != null){

fis.close();

}

if(fos != null){

fos.close();

}

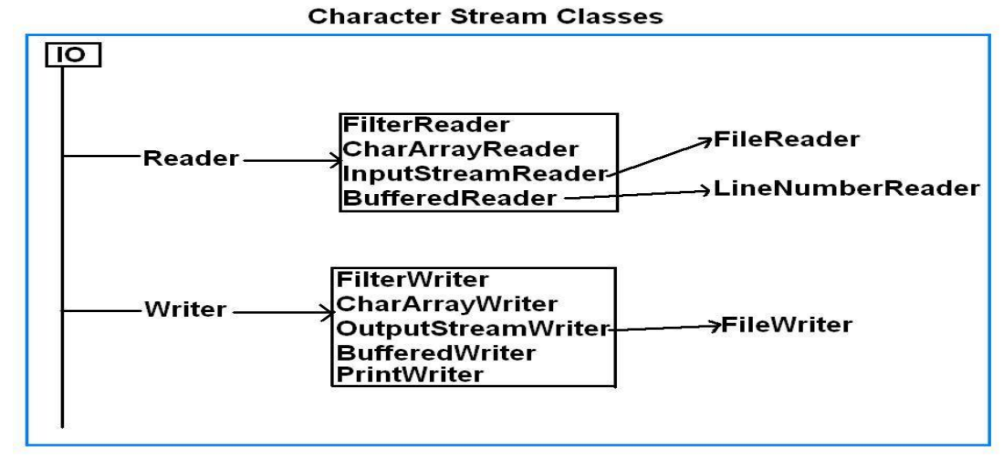
}

}

}

Character Streams:

1. Prior to JDK 1.1, the input and output classes (mostly found in the java.io package) supported only 8-bit "byte" streams.
2. In JDK 1.1 the concept of 16-bit Unicode "character" streams was introduced.
3. While the byte streams were supported via the java.io.InputStream and java.io.OutputStream classes and their subclasses, character streams are implemented by the java.io.Reader and java.io.Writer classes and their subclasses.
4. For example, to read files using character streams, you'd use the java.io.FileReader class; to read using byte streams you'd use java.io.FileInputStream.
5. Unless you're working with binary data such as image and sound files, you should use readers and writers to read and write information for the following reason:
6. Programs that use character streams are easier to internationalize because they're not dependent upon a specific character encoding.
7. To bridge the gap between the byte and character-stream classes, Java provides the java.io.InputStreamReader and java.io.OutputStreamWriter classes.
8. The only purpose of these classes is to convert byte data into character-based data according to a specified (or the platform default) encoding.



FileReader:

1. Inherited from InputStreamReader.
2. Used to read character by character from any underlying stream.
3. We can’t read like images, videos, sound files as these are not come under byte streams.

FileWriter:

1. Inherited from OutputStreamReader.
2. Used to Write character by character to any kind of file or output device

Example:

import java.io.\* ;

class CharacterStreams

{

public static void main(String[] args) throws IOException

{

FileReader fr = null ;

FileWriter fw = null ;

try{

fr = new FileReader("g:/input.jpg");

fw = new FileWriter("g:/output.jpg");

int ch ;

while((ch = fr.read()) != -1){

fw.write(ch);

}

System.out.println("copied....");

}

finally{

if(fr != null){

fr.close();

}

if(fw != null){

fw.close();

}

}

}

}

Object streams (Serialization)

1. Serialization is the concept of converting Object state into Persistent(permanent) state.
2. Generally in Java application, objects will be created in Heap area and will be deleted as soon as execution completes.
3. For every application it is mandatory to maintain the data permanently.
4. Java API providing classes and interfaces to perform serialization.

a. java.io.Serializable(interface)

b. java.io.ObjectInputStream(class)

c. java.io.ObjectOutputStream(class)

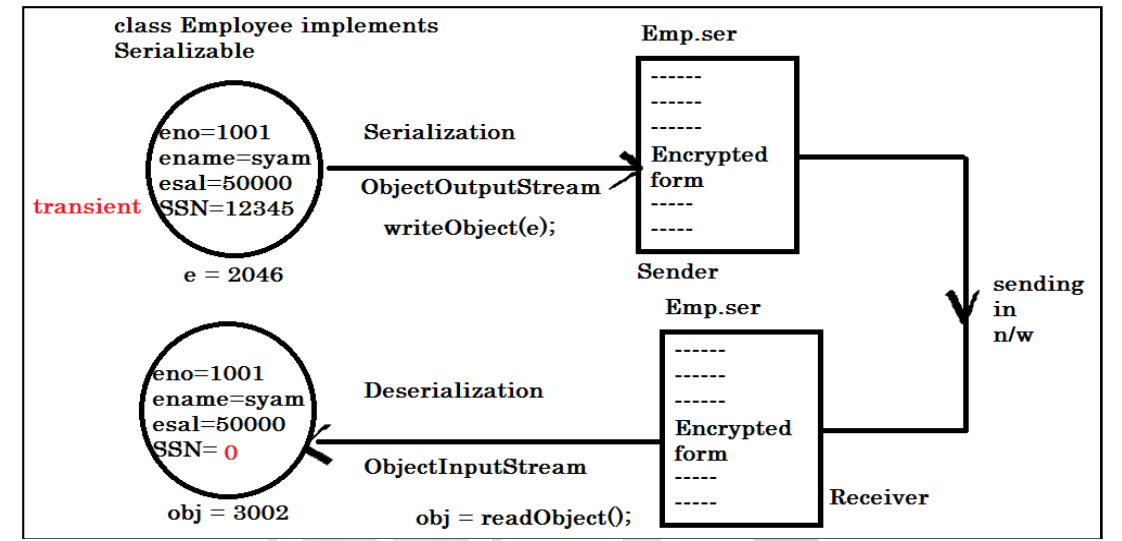
Important Points:

1. class must implements Serializable interface to serialize that object.
2. In the process of serialization, transient variables will not participate.
3. Serializable information must be stored into a file with .ser extension.
4. Serializable file will be in encrypted format, hence we can transfer the file over the network.
5. In the network, receiver should de-serialize the file to check the information.

Serialization : Convert Object ---> File

De-Serialization : Convert File --> Object

Example:



class Employee implements java.io.Serializable{

int eno ;

String ename ;

double esal ;

transient int SSN ;

Employee(int eno, String ename, double esal, int SSN){

this.eno = eno ;

this.ename = ename ;

this.esal = esal ;

this.SSN = SSN ;

}

}

Serialization:

import java.io.\*;

class Serialization

{

public static void main(String[] args) throws Exception

{

FileOutputStream fos = null ;

ObjectOutputStream oos = null ;

Employee e = new Employee(101,"Hari", 50000,12345);

System.out.println("Details : "+"\nEno : "+e.eno+"\nEname : "+e.ename+ "\nEsal :

"+e.esal+"\nSSN : "+e.SSN);

try

{

fos = new FileOutputStream("g:/Emp.ser");

oos = new ObjectOutputStream(fos);

oos.writeObject((Object)e);

System.out.println("Serialized successfully");

}

finally{

if(fos != null){

fos.close();

}

if(oos != null){

oos.close();

}

}

}

}

**Deserialization:**

import java.io.\*;

class DeSerialization {

public static void main(String[] args) throws Exception{

FileInputStream fis = null ;

ObjectInputStream ois = null ;

try{

fis = new FileInputStream("g:/Emp.ser");

ois = new ObjectInputStream(fis);

Employee e = (Employee)ois.readObject();

System.out.println("De-Serialized successfully");

System.out.println("Details : "+"\nEno : "+e.eno+"\nEname : "+e.ename+

"\nEsal : "+e.esal+"\nSSN : "+e.SSN);

}

finally{

if(fis != null){

fis.close();

}

if(ois != null){

ois.close();

}

}

}

}

**transient keyword:**

1. The keyword transient in Java used to indicate that the variable should not be serialized.
2. By default all the variables in the object is converted to persistent state.
3. In some cases, you may want to avoid some variables because you don’t have the necessity to transfer across the network.
4. So, you can declare those variables as transient. If the variable is declared as transient, then it will not be persisted means will not be participated in Serialization process.

**Variables not serialized during java serialization:**

1. static and transient variables.
2. Since static variables belong to the class and not to an object they are not the part of the state of object so they are not saved during Java Serialization process.
3. As Java Serialization only persist state of object and not object itself.
4. Transient variables are also not included in java serialization process and are not the part of the object’s serialized state.

**Buffered Streams:**

1. Buffer is a temporary storage area while processing the information.
2. In some of the applications we use Buffers to process the data instead of fetching information from the secondary memory every time.
3. Processed information will be stored in Buffer temporarily.
4. Once we close(save) the Buffer, the info saved permanently in Secondary memory.

**BufferedReader:**

BufferedReader buffer character to read characters, arrays and lines. It read text from a

character-input stream.

**Example:**

import java.io.\*;

class BufferedReaderDemo {

public static void main(String[] args) {

try {

FileReader f=new FileReader("IN.txt");

BufferedReader br = new BufferedReader(f);

String str;

while ((str = br.readLine()) != null) {

System.out.println(str);

}

} catch (IOException e) {}

}

}

**Example:**

import java.io.\*;

class BufferedStreams {

public static void main(String[] args) throws Exception{

FileReader fr = null ;

FileWriter fw = null ;

BufferedWriter bw = null ;

try

{

fr = new FileReader("d:/BufferedStreams.java");

fw = new FileWriter("g:/buffer.txt");

bw = new BufferedWriter(fw);

int ch;

while( (ch=fr.read()) != -1 ){

bw.write(ch);

}

}

finally{

if(bw != null){

bw.close();

}

if(fr != null){

fr.close();

}

if(fw != null){

fw.close();

} } } }

**StringTokenizer class :**

1. Available in util package.
2. Since jdk 1.0.
3. Used to split a string into Tokens(words).

**Example:**

import java.util.StringTokenizer;

class Tokenizer

{

public static void main(String[] args)

{

String line = "this is a test" ;

StringTokenizer st = new StringTokenizer(line);

while (st.hasMoreTokens())

{

System.out.println(st.nextToken());

}

}

}

**Program to Count the number of words in a file:**

import java.io.\*;

import java.util.StringTokenizer ;

class WordCount {

static int count = 0 ;

public static void main(String[] args) throws Exception{

FileReader fr = null ;

BufferedReader br = null ;

try{

fr = new FileReader("d:/WordCount.java");

br = new BufferedReader(fr);

String line ;

while( (line = br.readLine()) != null){

StringTokenizer st = new StringTokenizer(line);

while (st.hasMoreTokens()) {

System.out.println(st.nextToken());

count++ ;

}

}

System.out.println("Word count is : "+count);

}

finally{

if(br != null){

br.close();

} if(fr != null)

{

fr.close();

} } } }

**DataInputStream**

1. Is used to read java primitive data types from an underlying input stream.
2. It is inherited from FilterInputStream class.
3. It has only one constructor, and it takes byte stream object as an argument.

**Reading and Writing Primitive types:**

import java.io.\*;

public class DataInputStreamDemo

{

public static void main(String[] args)

{

int eno=1001;

String ename="srinivas";

try

{

FileOutputStream fos = new FileOutputStream("emp.txt");

DataOutputStream dos = new DataOutputStream(fos);

dos.writeInt(eno);

dos.writeUTF(ename);

dos.close();

FileInputStream fis = new FileInputStream("emp.txt");

DataInputStream dis = new DataInputStream(fis);

int id= dis.readInt();

System.out.println("Id: " + id);

String name = dis.readUTF();

System.out.println("Name: " + name);

}

catch (IOException e)

{

e.printStackTrace();

}

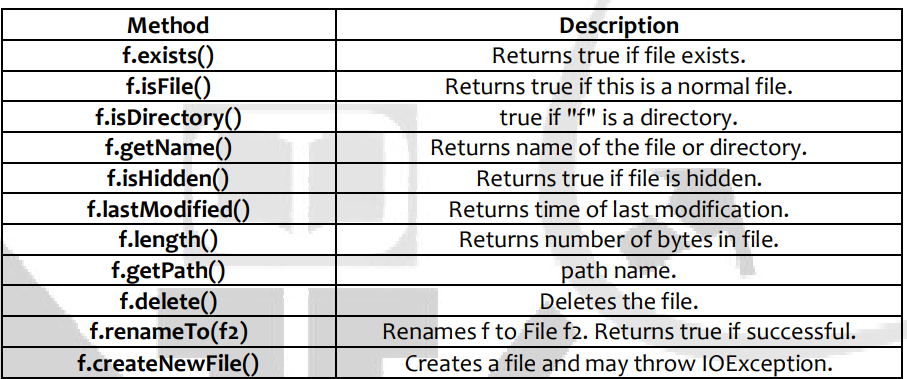
}

}

**Working with Files:**

1. File is a class, available in java.io directory used to work with Files and Directories.
2. The instance of File class represents a File or Directory.
3. To work with files or directories we must create object of file class.
4. When you create object for File class it will not search for that file or directory, externally we should call some pre-defined methods to perform operations on such file/directory.

Thus the statement can be written as: File f = new File("");



**Creating a File:**

import java.io.File ;

class CreateFile {

public static void main(String[] args) throws Exception{

File f = new File("g:/test.txt");

if(f.exists()){

System.out.println("File is present");

}

else{

f.createNewFile();

System.out.println("File created...");

}

}

**Creating directories:**

import java.io.\*;

class CreateDirectory {

public static void main(String[] args) {

String s1 = "d:/test" ;

String s2 = "d:/test1/test2/test3";

File d1 = new File(s1);

File d2 = new File(s2);

if(d1.mkdir())

System.out.println(s1+" created...");

if(d2.mkdirs())

System.out.println(s2+" created...");

}

}

}

**Deleting Files & Directories:**

import java.io.File ;

class Deletion {

public static void main(String list[ ]) {

if(list.length == 0){

System.out.println("no input......");

}

else{

for(int i=0 ; i<list.length ; i++){

Deletion.fileRemove(list[i]); //calling method

}

}

}

static void fileRemove(String name){ //called method

File target = new File(name);

if(target.exists()){

if(target.delete())

System.out.println(name+" deleted...");

else

System.out.println("failed in deletion of "+name);

}

else{

System.out.println(name+" not present");

}

}

}

**Renaming the file:**

import java.io.File;

import java.io.IOException;

public class Rename {

public static void main(String[] argv) throws IOException {

File f = new File("F:/in.txt");

f.renameTo(new File("out.java"));

}

}

**File copy:**

import java.io.\*;

public class Copy {

public static void main(String[] args) throws IOException {

File inputFile = new File("Copy.java");

File outputFile = new File("OutCopy.java");

FileReader in = new FileReader(inputFile);

FileWriter out = new FileWriter(outputFile);

int c;

while ((c = in.read()) != -1)

out.write(c);

in.close();

out.close();

}

}

**File permissions:**

boolean canExecute(); – return true, file is executable; false is not.

boolean canWrite(); – return true, file is writable; false is not.

boolean canRead(); – return true, file is readable; false is not.

**Set the file permission:**

boolean setExecutable(boolean); – true, allow execute operations; false to disallow it.

boolean setReadable(boolean); – true, allow read operations; false to disallow it.

boolean setWritable(boolean); – true, allow write operations; false to disallow it.

import java.io.File;

public class SetWritableTest{

public static void main(String[] args)throws SecurityException {

File file = new File("ddd.txt");

if (file.exists()) {

boolean bval = file.setWritable(false);

System.out.println("set the owner's write permission: "+ bval);

} else {

System.out.println("File cannot exists: ");

}

}

}

**Hidden file: a file is considered to be hidden, if it’s marked as hidden in the file properties.**

import java.io.File;

import java.io.IOException;

public class FileHidden{

public static void main(String[] args) throws IOException{

File file = new File("FileHidden.java");

if(file.isHidden()){

System.out.println("This file is hidden");

}else{

System.out.println("This file is not hidden");

}

}

}

**MultiThreading:**

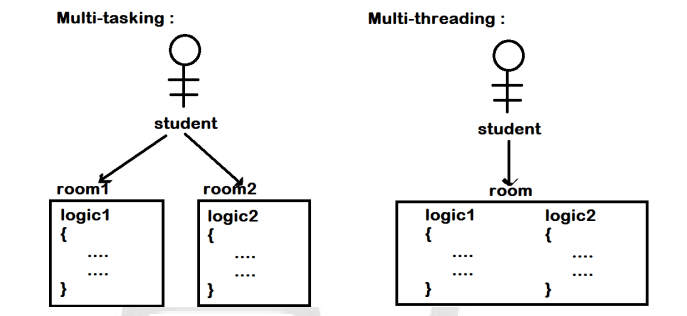
**Multi tasking:**

1. Peforming more than one task simultaneously.
2. Process spaces will be divided into n parts to execute n tasks.

In multi threading, n threads execute in a single process space.

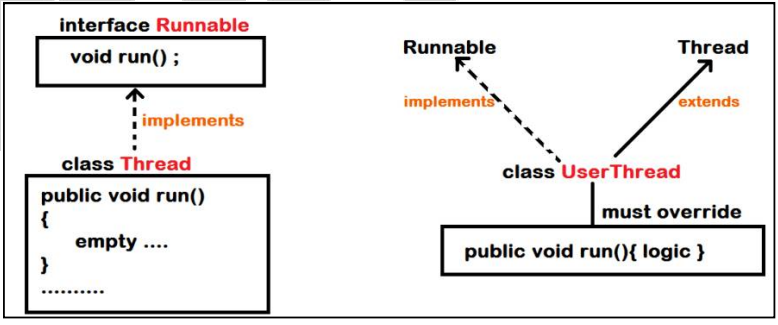
Multi tasking takes care by Operating system. No programming effort is required.

To implement multi threading, we need to write programs.



**Creating a Thread:**

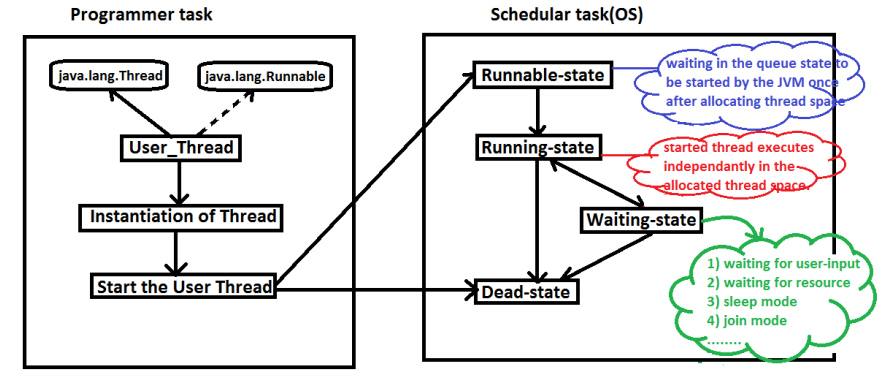
1. java.lang package contains pre-defined class and interface to create User threads.
2. A thread can be created in 2 ways
   1. Extending from java.lang.Thread class
   2. Implementing from java.lang.Runnable interface
3. Every Thread class must override "public void run() " method in its definition.
4. run() method contains the logic that has to execute parallel.
5. Pre-defined run method in the java.lang.Thread class is having empty definition.



**Life Cycle of Thread:**

In the Life cycle process of Thread, Programmer & Scheduler will participate.

1. Creation: Programmer has created Thread class either by extending or by implementing...
2. Instantiation: Thread Object will be created to participate in the communication only when it is instantiated by the programmer.
3. Start: As soon as Thread Object is ready, it has to be started by the Programmer.
4. Runnable state: The started thread by the programmer need to wait in the Queue until Scheduler allocates Thread space in the processor to be executed.
5. Running state: Logic execution state Waiting state: Another Queue state into which thread enters from running state for different reasons....
6. Dead state: The last stage of Thread into which Thread enters either by completing all the thread instruction or in case of abnormal termination



**Types of Thread Applications:**

1. Single Threaded Application

2. Multi Threaded Application

**Single Threaded Application:**

1. When we invoke java application, JVM implicitly creates a thread is called "main thread".
2. In single threaded application, execution starts at main thread and end at the same thread.
3. One Thread space can have “n” number of method spaces to execute the thread logic.
4. All the methods of single thread executes sequentially.

Example:

class SingleThreaded

{

public static void main(String[] args) {

SingleThreaded obj = new SingleThreaded();

obj.printNumbers();

for(int j=1 ; j<=200 ; j++){

System.out.print("j : "+j+"\t");

}

}

void printNumbers()

{

for(int i=1 ; i<=200 ; i++){

System.out.print("i : "+i+"\t");

}

}

}

**Multi threaded Application:**

1. Creating a user thread from main thread referred as Multi threaded application.
2. Multi threaded application execution starts at main thread only.
3. In multi threaded application, main-thread can complete its execution before any child thread.
4. Program execution completes, when all the running threads moved to dead state.
5. All the threads in the application get independent memory allocation and execute parallel

Example:

class MultiThreaded extends Thread{

public static void main(String[] args) {

MultiThreaded obj = new MultiThreaded();

obj.start();

for(int j=1 ; j<=200 ; j++)

{

System.out.print("j : "+j+"\t");

}

}

public void run(){

for(int i=1 ; i<=200 ; i++){

System.out.print("i : "+i+"\t");

}

}

}

**Sleep() method:**

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

Example: class MultiThreaded extends Thread{

public static void main(String[] args) throws InterruptedException{

MultiThreaded obj = new MultiThreaded();

obj.start();

for(int j=1 ; j<=10 ; j++){

System.out.println("j : "+j);

Thread.sleep(500);

}

System.out.println("main exiting...");

}

public void run() // throws InterruptedException{

for(int i=1 ; i<=10 ; i++){

System.out.println("i : "+i);

try{

Thread.sleep(1000);

}

catch (InterruptedException ie){ }

}

System.out.println("Child exiting...");

}

}

**Predefined Thread Identities:**

1. Every thread is having identity.
2. JVM invokes the threads using their identities only.
3. Every thread is initially identified by default name.
4. Default names set by Thread class default constructor in Thread creation process.

The following method is used to check identity of thread.

public final String getName()

**Setting Identity to the Thread Explicitly:**

It is possible to set the identity to thread in 2 ways

1) In the process of Construction of Thread using Thread class constructor.

2) In the process of Execution of Thread, we can rename the thread.

**Using Constructor:**

public Thread(String name)

It sets the name to the current user thread.

class Threads extends Thread {

Threads(String name){

super(name);

}

public void run(){

System.out.println("Name : "+this.getName());

}

public static void main(String[] args) {

Threads t = new Threads("Child");

t.start();

}

}

**Using setName() method :**

Used to rename the thread.

Syntax: public final void setName(String name)

Example:

class Threads extends Thread {

Threads(){

System.out.println("Default thread name : "+this.getName());

}

public void run(){

System.out.println("New Name : "+this.getName());

}

public static void main(String[] args) {

Threads t = new Threads();

t.setName("Child");

t.start();

}

}

**Join() method:**

1. A pre defined method used to implement inter thread communication.
2. We can join the threads using this method.
3. We use join() method in case of one thread execution should resume only when another thread moved to dead state

**Example:**

class JoinDemo extends Thread

{

static int n , sum = 0 ;

public static void main(String[] args) throws InterruptedException

{

java.util.Scanner sc = new java.util.Scanner(System.in);

System.out.println("/\*\*\*SUM OF FIRST N NUBERS\*\*\*/");

System.out.print("Enter n value : ");

JoinDemo.n = sc.nextInt();

JoinDemo child = new JoinDemo();

child.start();

//Thread.sleep(2000);

child.join(); //main thread wait until child moved to dead state.

System.out.println("sum of first "+JoinDemo.n+ " numbers is : "+JoinDemo.sum);

}

public void run(){

System.out.println("Calculation starts...");

for(int i=1 ; i<=JoinDemo.n ; i++){

JoinDemo.sum = JoinDemo.sum + i ;

try{

Thread.sleep(100);

}

catch (InterruptedException ie){ }

}

System.out.println("Calculation end");

}

}

**Execution time of Single Threaded Application:**

class PrintNumbers

{

void print1to50() throws InterruptedException

{

for(int i=1 ; i<=50 ; i++){

System.out.print(i+"\t");

Thread.sleep(100);

}

}

void print50to1() throws InterruptedException{

for(int i=50 ; i>=1 ; i--){

System.out.print(i+"\t");

Thread.sleep(100);

}

}

}

class TimeCheck{

public static void main(String[] args) throws InterruptedException{

PrintNumbers pn = new PrintNumbers();

long start = System.currentTimeMillis();

pn.print1to50();

pn.print50to1();

long end = System.currentTimeMillis();

System.out.println("Time consumed : "+(end-start)/1000+" seconds");

}

}

**Execution time of Multi threaded application:**

class PrintNumbers

{

void print1to50() throws InterruptedException{

for(int i=1 ; i<=50 ; i++){

System.out.print(i+"\t");

Thread.sleep(100);

}

}

void print50to1() throws InterruptedException{

for(int i=50 ; i>=1 ; i--){

System.out.print(i+"\t");

Thread.sleep(100);

}

}

}

class TimeCheck extends Thread{

static PrintNumbers pn = new PrintNumbers();

public void run(){

try{

TimeCheck.pn.print1to50();

}

catch (InterruptedException ie){ }

}

public static void main(String[] args) throws InterruptedException{

TimeCheck t = new TimeCheck();

long start = System.currentTimeMillis();

t.start();

TimeCheck.pn.print50to1();

t.join();//first child should move to dead state, then only we can take the end time

long end = System.currentTimeMillis();

System.out.println("Time consumed : "+(end-start)/1000+" seconds");

}

}

**NOTE:**

**Can we call start() method more than one time on the same thread Object at the same time?**

Ans: not allowed, results IllegalThreadStateException. Violation rule of Synchronization.

class Threads extends Thread

{

public void run()

{

for(int i=1 ; i<=5 ; i++)

{

System.out.println("run is executing.....");

try{

Thread.sleep(1000);

}

catch (Exception e){ }

}

}

public static void main(String[] args){

Threads t = new Threads();

t.run(); //sequential execution

t.start(); //executes independantly

t.start(); //if causes abnormal termination of current Thread(main)

System.out.println("end of main....");

}

}

**Creating Thread using Runnable interface:**

1. When a class is implementing Runnable interface, it has to implement all the specifications of Runnable
2. public void run() is the only specification in Runnable interface....
3. Runnable Object cannot be started directly. It is not the Thread.
4. Hence, it is mandatory to create Thread Object using Runnable Object.

public Thread(Runnable target) : Allocates a new Thread object.

class Threads implements Runnable

{

public void run()

{

System.out.println("Child is created.....");

}

public static void main(String[] args)

{

Threads runnable = new Threads(); //Runnable Object

//runnable.start(); //CE : cannot start....

Thread t = new Thread(runnable);

t.start();

}

}

**Program to set the Identity of Thread that has created using Runnable.....**

public Thread(Runnable target , String name)

class Threads implements Runnable {

static Thread t ;

Threads(String name){

Threads.t = new Thread(this , name);

Threads.t.start();

}

public void run(){

System.out.println("Name of Thread : "+Threads.t.getName());

}

public static void main(String[] args) {

Threads runnable = new Threads("Child");

}

}

**Priority of a Thread:**

1. Every thread can have the priority by which Operating System schedules the thread when it is in waiting (queue) state.
2. Threads having higher priority will be processed first.
3. Every thread is having default priority.
4. java.lang.Thread class contains Fields(pre-defined variables) to describe priorities of Thread.

class ThreadPriorities

{

public static void main(String[] args) {

System.out.println("min priority : "+Thread.MIN\_PRIORITY);

System.out.println("default priority : "+Thread.NORM\_PRIORITY);

System.out.println("max priority : "+Thread.MAX\_PRIORITY);

}

}

**Information of Current Thread:**

class CurrentThreadInfo extends Thread{

public void run(){

System.out.println("Child name : "+this.getName());

System.out.println("Child priority : "+this.getPriority());

}

public static void main(String[] args) {

CurrentThreadInfo t = new CurrentThreadInfo();

t.start();

Thread m = Thread.currentThread();

System.out.println("Parent name : "+m.getName());

System.out.println("Parent priority : "+m.getPriority());

}

}

**Types of Threads:**

Java application is allowed to define 2 types of Threads.

1) Non-Daemon threads

2) Daemon threads.

**Non-Daemon threads:**

1. Every thread is having Non-Daemon behavior as soon as created.
2. Non-daemon threads execute front end logic of application.

**Daemon-threads:**

1. Thread execute behind the application.
2. It is called service thread.
3. It executes back ground logic of application.
4. It is possible to change the behavior of Non-Daemon to Daemon.

public final void setDaemon(boolean on)

1. The JVM implicitly stops all the daemon threads execution if all the Non-Daemon threads moved to dead state.
2. It is possible to change the behavior of Thread into Daemon either in the process of Thread construction or before start.

class Threads implements Runnable {

static Thread t ;

Threads(String name){

Threads.t = new Thread(this , name);

Threads.t.setDaemon(true);//Child thread behaves like Daemon thread....

}

public void run(){

for(int i=1 ; i<=10 ; i++){

System.out.println(Threads.t.getName()+" : "+i);

try{

Thread.sleep(1000);

}

catch (InterruptedException ie){

System.out.println("Exception caught...");

}

}

}

public static void main(String[] args) throws InterruptedException{

Threads runnable = new Threads("Daemon");

Threads.t.start();

Thread m = Thread.currentThread();

m.setName("Non-Daemon");

for(int i=1 ; i<=10 ; i++){

System.out.println(m.getName()+" : "+i);

Thread.sleep(500);

}

}

}

1. It is not possible to change the behavior of Thread once it has been started(in running state).
2. Hence main-thread always behaves like Non-Daemon, because it starts by the JVM implicitly.

class Threads extends Thread

{

public void run()

{

this.setDaemon(true);//Thread behaviour can't be changed while it is running....

}

public static void main(String[] args){

new Threads().start();

}

}

**Thread Synchronization:**

1. The concept of synchronization is, when multiple threads are trying to access a single resource, only one thread is allowed and all the remaining threads moved to queue (waiting) state.
2. Complete class is not possible to synchronize in the java application.
3. A method or block can be synchronized.

class InsufficientFundsException extends Exception

{

String name;

InsufficientFundsException(String name)

{

this.name = name;

}

String getErrorMessage(){

return this.name;

}

}

class Account{

private int balance;

Account(int balance){

this.balance = balance ;

}

public int getBalance(){

return this.balance;

}

synchronized void withdraw(int amount) throws InsufficientFundsException{

System.out.println("Trying to withdraw : "+amount);

System.out.println("Balance in account : "+this.balance);

if(amount <= this.balance){

System.out.println("Collect the cash : "+amount);

this.balance = this.balance - amount ;

}

else{

throw new InsufficientFundsException("No funds.......");

}

}

}

class AccountThread extends Thread{

Account atm ;

int amt ;

AccountThread(Account atm , int amt){

this.atm = atm ;

this.amt = amt ;

}

public void run(){

try{

atm.withdraw(amt);

}

catch (InsufficientFundsException ie){

System.out.println("Error : "+ie.getErrorMessage());

}

}

}

class ATMApplication{

public static void main(String[] args) throws InterruptedException{

Account atm = new Account(6000);

System.out.println("Initial amount in the account : "+atm.getBalance());

AccountThread t1 = new AccountThread(atm, 4000);

AccountThread t2 = new AccountThread(atm, 3000);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println("Final amount in the account : "+atm.getBalance());

}

}