

JavaUnit-IOverview

- Java the new programming language developed by sun microsystems in 1991.
- Invented by the team called the green team. One of the inventor was "James Gosling".
- Originally java was called oak later called HOT-Java.
- In 1995, Internet & web was just emerging so, sun turned it into a language of Internet programming.
- Java was not designed for Internet. The objective behind development of java was to create a common development environment for consumer electronic devices which was easily portable from one device to another.

Advantage of Java over C or C++

- C & C++ (and most other languages) they are designed to be compiled for a specific target.

- Compilers are expensive and time-consuming to create.
- But, Java is platform independent i.e., it could be used to produce code that would run on a variety of CPU's under differing environment.

Eg' of Java

```

class Simple // class
{
    public static void main (String args[])
    {
        System.out.println ("Hello Java");
    }
}

```

command line argument

Package, class, Method.

Compile Java Simple.java
Run Java Simple.

(main is the first method that is seen).

②. Web - Application :

→ An application that will produce / load Dynamic pages and run on the server side, is called a web-application.

→ Eg: javatpoint.com,

→ Servlet and Jsp are used to create web-application.

③. Enterprise Application :

→ An application that is used in distributed system such as banking application (system).

→ EJB is used to create Enterprise Application.

④. Mobile Application :

→ An application that will run on any mobile is called mobile application.

→ Such as whatsapp etc

→ Android. ~~Java~~ JME used to create mobile applications.

Characteristics of Java

- ① Java is Simple
- ② Object-oriented
- ③ per the distributed
- ④ Architectural-neutral
5. portable
6. Robust
7. secure
8. dynamic
9. High-performance
10. platform Independent
11. Interpreted
12. Multithreaded

Java is Simple.

- Java has no pointers and has automatic garbage collector.
- Java is partially modeled on C++ but is quite simple than C++.
- Java is C++-++.
It has is like C++ language with more functionality and lesser negative aspects.
- It fixes some of clumsy features of C++.

Java is Object - Oriented.

- Java is strictly object-oriented.
- An Object oriented language provides greater flexibility, reusability and modularity through Encapsulation, Inheritance and polymorphism.
- Reusability is the central issue in software development.

Java is distributed.

Java program on compilation produces an output which is called Byte-code. Now, this bytecode can run on any machine with a JVM Interpreter. In it, Java is distributed as anyone can have access to thousands of programs with internet.

- EJB is used to create distributed programs in Java.

Java is Architectural Neutral.

Neutral means change in architecture, design will not have any effect on the execution of program.

Jana follows "Write Once Run anywhere" notion. With a Jana virtual machine, you can write any program that will run on any ~~more~~ platform.

Jana is portable.

Jana is portable because it is Architectural-neutral and distributed. Jana programs can be run on any platform with JVM installed.

Jana is Robust.

Robust means strong.

Jana is a strong language as it uses high memory management.

It has Run-time Exception Handling feature.

All these points makes Jana Robust.

Automatic garbage collector

Jana is secure,
Jana has multilayered system of security.

Bytecode verifier and security managers add security to Java programs.

Jana creates fire-wall between Java programs and remaining of the computer system that neither allows computer system to interfere with Java program nor does it allow Java program to interfere with remaining of computer system.

Jana is dynamic.

One can access thousands of Java program and other computers using Internet.

Jana loads dynamic pages when it is run on server side.

Jana's High Performance.

Jana has high performance as it is architectural neutral and portable.

Java is platform Independent.

platform is any hardware or software environment that is needed to run a program.

Java compilation provides bytecode which is neutral. Java has its own run-time environment (JRE) and API, hence it is platform independent.

Java is Interpreted.

You need an interpreter to run java program. Java program produces Bytecode on compilation in JVM. This Bytecode is platform independent and can be run on any machine which has interpreter installed in it.

Java is multithreaded.

Multithreading is integrated smoothly in Java. whereas in other languages we have to call procedures specific to the operating system.

⑧. Jana Environment.

→ Jana environment includes a large number of development tools and hundreds of classes and methods.

⑩. Jana Development Kit

→ JDK comes with collection of tools that are used for developing and running Jana programs.

Javac → Jana compiler

Jana → Jana interpreter

Javah → produces header files

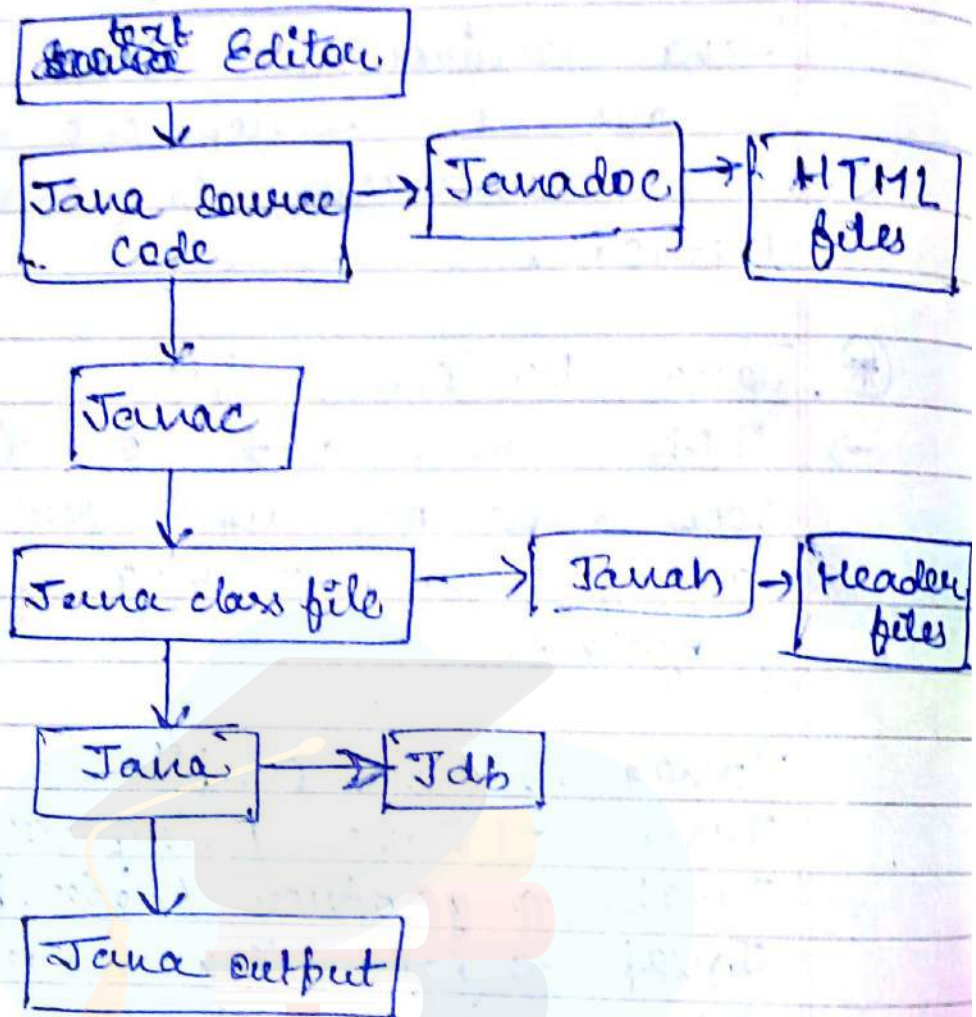
Javap → Jana disassembler

Jdb → Jana debugger.

Applet viewer → Enables to run Jana applets.

Janadoc → creates HTML format documentation.

Flow of Java program.



Explain Each - One

JDK, JRE and JVM

JDK → Java Development tool kit
 JRE → Java run-time Environ-
 -ment
 JVM → Java Virtual Machine,



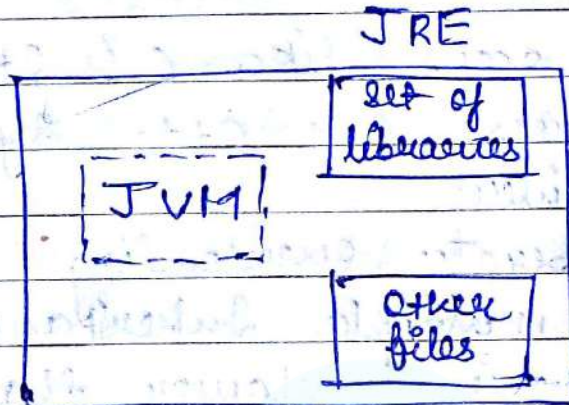
Date :

Page No.

JRE

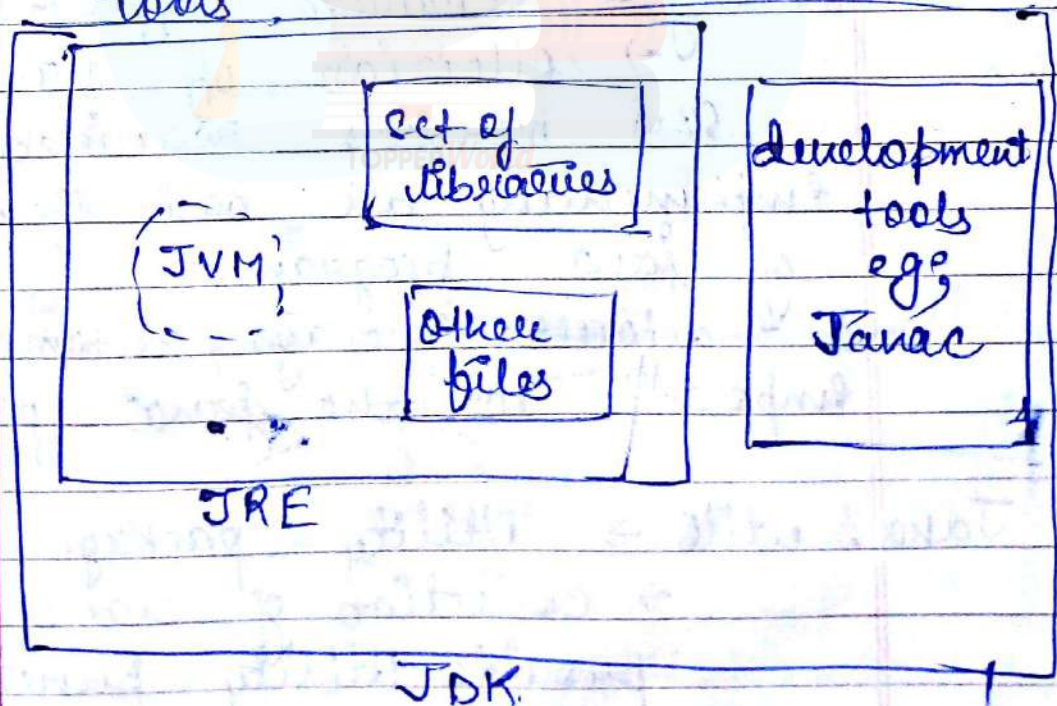
Java Run-time Environment,
used to provide run-time
environment.

→ It is an implementation of JVM,

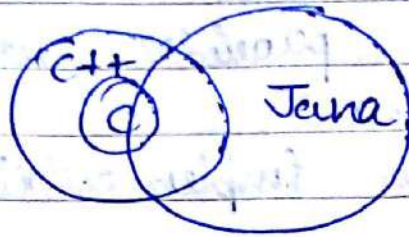


JDK → development tool kit,

→ It contains JRE + Development
tools



Overview of C, C++ & Java



- Java seems like C & C++ as it uses C & C++ syntax
- No pointers
- No operator overloading
- No multiple inheritance
- Java is slower than C++.

Packages in Java program

Java.lang → Language support package
 → collection of classes and methods required for implementing all basic features of java program.
 → default package; automatically imported to the java program.

Java.util → Utility package.
 → collection of classes to provide utility functions such as date & time functions

Java , Io &

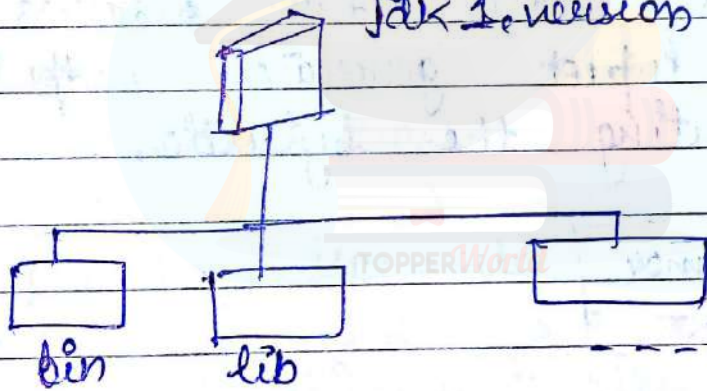
- Input / Output Package →
- collection of classes required for I/O manipulation.

Java.net → Networking package.

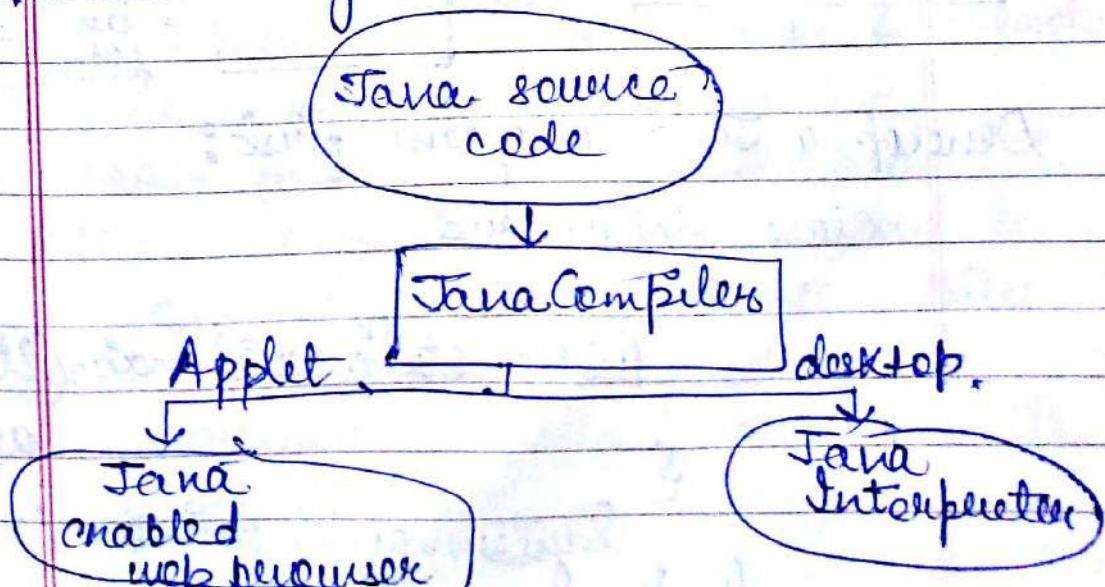
- collection of classes required for communicating with other computers via internet.

AWT package = Java.Awt

#. JDK directory structure & JDK version



#. Java Program Structure

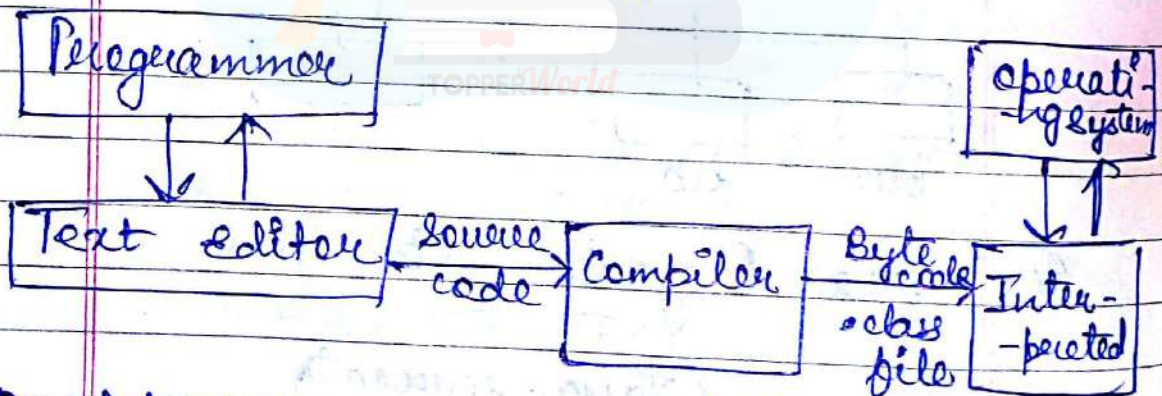


*. Java Program : compilation and execution process:-

Compilation and execution of a java program is a two-step process. During compilation, Java compiler compiles the source code and generates bytecode.

This intermediate Bytecode is saved in the form of .class file.

In second phase, the Java interpreter takes this .class file as input generates outputs by executing the bytecode.



Develop a Java program first:

```

class HelloWorld
{

```

```

    public static void main(String args[])
    {

```

```

        System.out.println("Hello");
    }
}

```


Compile the Java program

Once the Java program is ~~comp~~ written and saved, it needs to be compiled. To compile a Java program, we need to invoke the Java compiler by supplying `javac` command.

Java Compiler comes with JDK i.e. Java development kit. JDK is collection of tools needed for development & running of Java programs. It includes JRE and set of API classes.

`javac HelloWorld.java`

Java compiler creates a .class file i.e. bytecode which contains instructions that Java interpreter will execute.

Main function of compiler is to convert source code to bytecode or source files to class file that the virtual machine can execute.

Run → Java virtual machine
↓
Interpreter

Date :

Page No.

Run Java Program.

After successful compilation of Java program from HelloWorld.java to HelloWorld.class to actually run the program we need the Java interpreter "java". To do so, we pass the command line argument as follows:

Java HelloWorld.

→ The message HelloWorld will be printed on the screen as a result of above command.

The JVM takes the following steps to run a program:

① loading of classes and Interfaces
→ finding the bytecode

② Linking of classes and Interfaces
→ taking the bytecode and combining it to the run-time state of Java virtual machine, so that it can be executed.

Ques Why main method is declared as public and static?

Ans main() method is declared as "public" to provide access to the JVM so that the JVM can call it without being member of that class. main() method is declared as "static" to allow JVM to call main() method without ~~affecting~~ ~~the~~ creating the object.

In Java, methods can be called only with the help of objects and main is the starting point of any java program.

And that instance, ~~there~~ no object can be created.

Hence, main is declared as static as only static ^{methods} ~~members~~ can be called without objects.

General structure

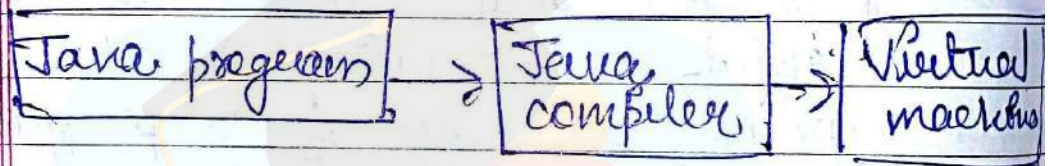
General str. of Java Program

Document Section	→	Suggested
Package statement	→	optional
Import statement	→	optional
Interface statement	→	optional
class Definition	→	optional
main method ^{class} {main method definition }	→	Essential

Java Virtual Machine

Java Compiler generates intermediate code called Bytecode for a machine that doesn't exist.

This machine is called virtual machine and it exists only inside the computer memory. Byte code is also referred to as virtual machine code.

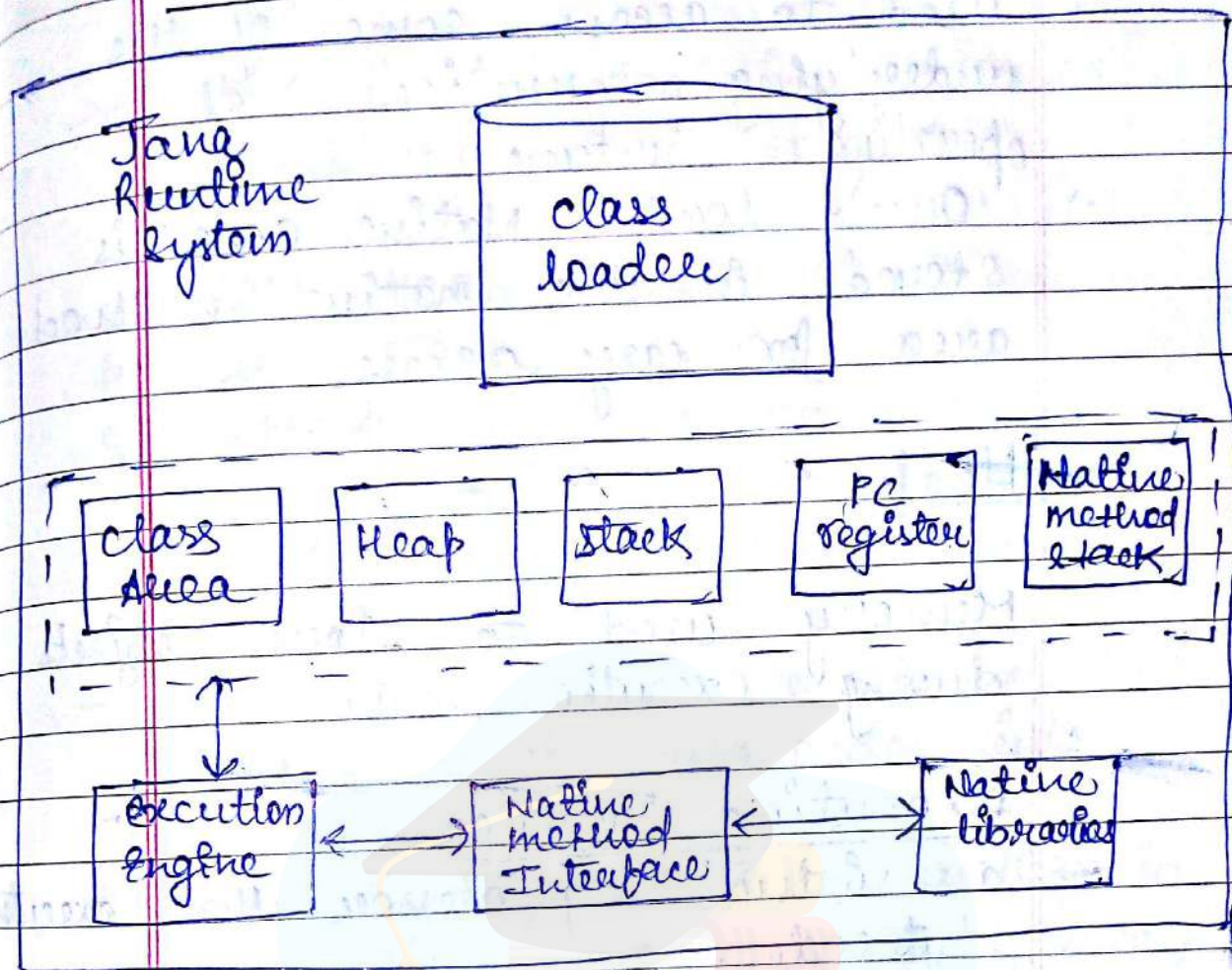


Virtual machine code is not machine specific code.

The machine specific code is generated by the Java interpreter by acting as the intermediary between the virtual machine and real machine.



Internal Architecture of JVM



class loader

- locates and loads classes into the JVM.
- loads java classes into the run-time environment.
- loads trusted class file.
- loads untrusted classes from the local file system and passes them to class file verifier.

Native - Method loader

- used to access some of the underlying functions of operating systems.
- Once loaded, Native code is stored in the native method area for easy access.

Heap

Memory used to store objects during execution.

Execution Engine

- a virtual processor that executes bytecode.
- has virtual registers, stack etc.
- contains JIT
- performs memory management etc.

JIT - Just In Time compiler

- translates bytecode into machine code at run-time.
- performance overhead due to interpreting bytecode.
- works on a method-by-method basis.

Organisation of JVM

→ JVM is divided into three conceptual data spaces:

- a. Class Area
- b. Java stack
- c. Heap

Class Area

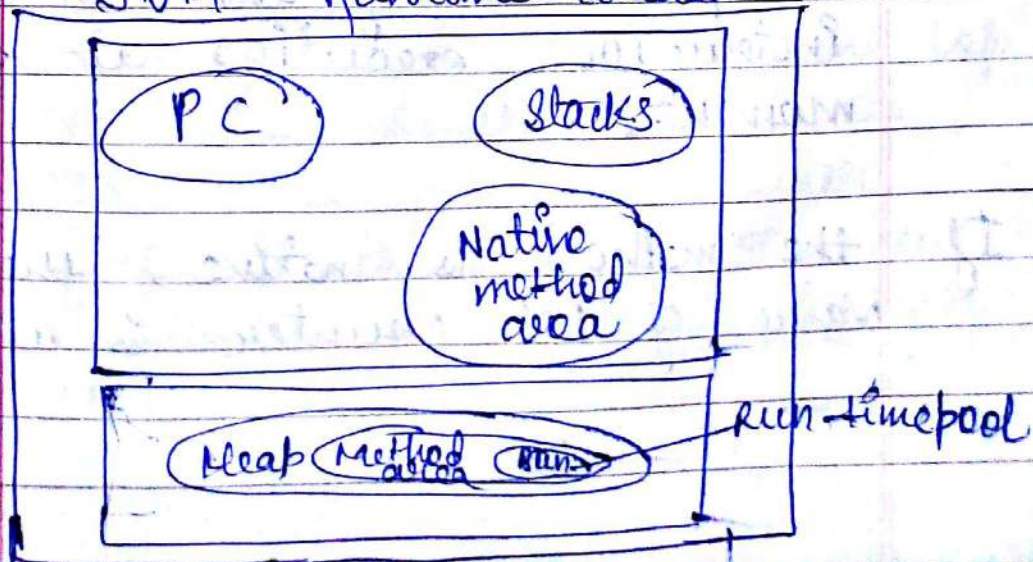
- The class Area stores the classes that are loaded into the system.

→ Method implementations are kept in the space called method area, and constants are kept in a space called constant pool.

⇒ objects are stored in the heap.

~~A class~~

JVM runtime areas



Classes have several properties

- Superclass
- List of interfaces
- List of fields
- List of methods
- List of constants

- All properties of classes are immutable. i.e., there's no way to change any property of class once it has been brought to the system. This makes the machine more stable.

→ In Java, ~~multithreaded~~ we have multithreaded architecture as Java supports multithreading. So, a program counter 'PC' is created every time a new thread is created.

PC → Keeps track of the current instruction executing at any moment.

If the method is native, then the value of 'PC' counter is undefined.

Jana Stack

- JVM stacks are used to ~~create~~ store JVM frames. The JVM frame on the top of the stack is called the active stack frame.
- JVM frames are created; every time a method is invoked. Each stack/JVM frame has an operand stack, an array of local variables and a PC. PC → points to the current instruction being executing.
- Only the operand stack & local variable array in the active stack frame is used.
- The program counter. Each time, the method is invoked, a stack frame is created and becomes the top of Jana stack. The PC is saved as part of older Jana stack frame. The new frame has its own new PC.

Heap

Stack-overflow errors This error occurs in fixed JVM stacks.

→ If the memory size is not sufficient, during the program execution.

Out of Memory - Errors When dynamic sized stacks, when they try to expand for more memory need & there is no memory available to allocate and, we get out of memory error.

Heaps

- Objects are stored in the Heap.
- Each object is associated with a class in class Area.
- Heap areas are used to store objects of data array and classes.

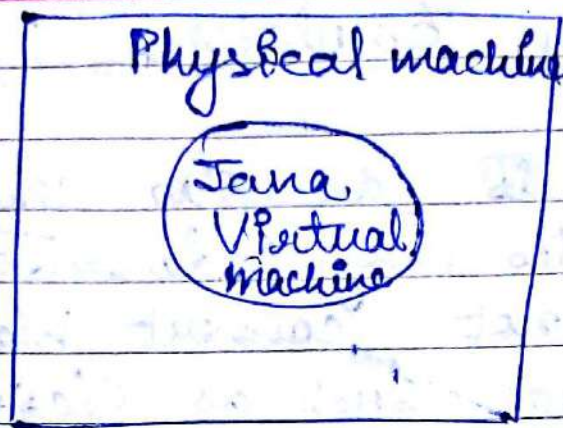
JVM as an Emulator

JVM is an ~~also~~ emulator that emulates bytecode. Java Compiler does not convert high-level language such as C/C++ to the machine level language. It converts the Java language to the Java bytecode that the JVM understands.

Since, Java bytecode has no platform-dependent code, it is executed on the hardware on which JVM is installed, even when the CPU or OS is different.

The difficult part of creating Java bytecode is that the source code is compiled for the machine that doesn't exist. This machine is called Java virtual machine, and it exists only in computer's memory.

Emulator : A hardware or software that enables one computer system (host) to behave like another computer system (emulator). Typically



The Java virtual machine is responsible for interpreting byte code and translating this into actions or Operating System calls.

JVM as an Interpreter

An Interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle.

→ In order to execute a program, an Interpreter reads an instruction, decides what is necessary to carry out the instruction and performs the required command to do so.

→ One use of interpreter is to use high-level language.

Source Code —————> byte code —————> Interprets —————> same

The Java virtual machine Instructions set,

Java virtual machine instruction consists of an opcode specifying the operations to be performed.

Types of Java Virtual Machine Instructions set

The JVM is stack-oriented interpreter that creates a local stack frame of fixed size each time a method is invoked. The size of the local stack is to be computed by the compiler. The values may also be stored intermediary in a frame area containing local variables. These variables are numbered from 0 to 65535 i.e., we have a maximum of 65536 of local variables per method.

- The instruction set can be roughly grouped as follows:

①. Stack Operations &

In stack, we have load & store instructions.

load takes something from the arguments and pushes it to the top of the stack area.

different kinds of "load" instructions

i load → Integer load
 f load → float load
 l load → long load
 d load → double load.

→ Each instruction has different "forms" for different kinds of operands.
 Eg; different forms of i load instruction.

Eg

i load - 0	26	
i load - 1	27	
i load - 2	28	
i load - 3	29	
i load - n	21	n

Store pop something from the top of operand stack and places it into args / local area.

②. Arithmetic operations

The instruction set of JVM distinguishes its operand type using different instructions.

Add : ladd, ladd, fadd, dadd

Subtract : isub, lsub, fsub, dsub

Multiply : imul, lmul, fmul, dmul.

③. Control flow There are branch instructions like if-icmpg which compares if two integers are equal.

another ex: jse instruction
(Jump to sub-routine)

④. Field access

→ In java, different kinds of memory can be accessed by different kind of instructions.

1. Accessing locals and arguments:
load & store instruction.
in object

2. Accessing fields : getfield, putfield.

3. Accessing static field : getstatic, putstatic.

AG

5. Method Invocations

1. invokevirtual is usual instruction for calling a method on an object.
2. invokeinterface is same as invokevirtual but used when a called method is declared in an interface.
3. invokespecial is also called invokestatic
→ used for calling things such as "constructor".
4. invokestatic is used for calling method that have the "static" modifier.

6. Conversion

- i2l → Integer to long
- i2f → Integer to float
- i2d → Integer to double
- l2f → long to float
- l2d → long to double
- f2d → float to double.

④, Class file format

My Very Cute Animal Turns Savage
In Full Moon Areas.

Magic no.	version
Constant pool	
Access flag	
This class	
Super class	
Interfaces	
Fields	
Methods	
Attributes	

order
is
strictly
defined.

1. magic number. → 0x CAFE BABE.
2. version → minor and major versions of class file.
3. Constant pool → pools of constants for the class.
4. Access flag → whether the class is ^{abstract} or ^{static}.
5. This class → name of current class.
6. Super class → name of super class
7. Interfaces → Any interfaces in the class
8. Fields → Any fields in the class.
9. Methods → Any methods in the class
10. Attributes → Any attributes in the class.

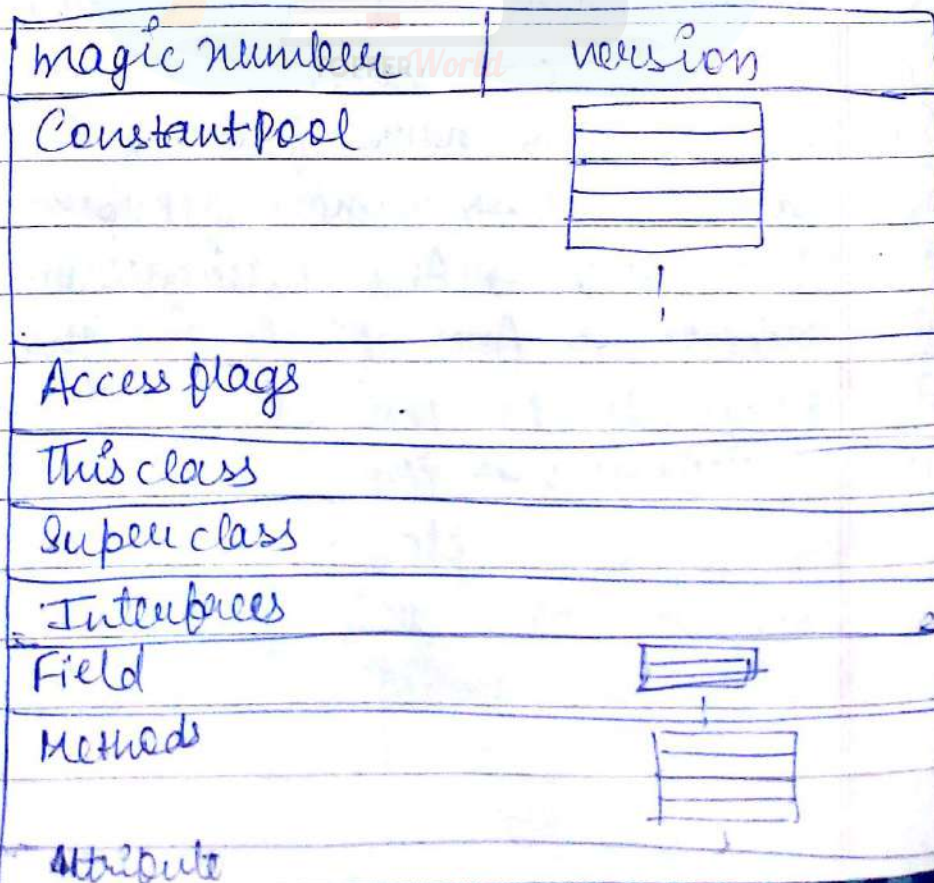
Ex 1

```

ClassFile {
    u4 magic;
    u2 minor-version;
    u2 major-version;
    u2 constant-pool-count;
    u2 access-flags;
    u2 this-class;
    u2 super-class;
    u2 interfaces-count;
    u2 fields-count;
    u2 methods-count;
    u2 attributes-count;
}

```

- length of class is not fixed.
 → variable length sections are constant pools, methods etc.



* Verification

- The purpose of verification is to ensure that programs follow a set of rules that is designed for the security of the JVM.
- Program's can try to subvert the security of JVM by stack overflowing, corrupting the memory they are not allowed to access.
- The verification algorithm ensures that this does not happen by checking that each object in the class is used according to their proper type. It traces the code to do so.
- The verification algorithm is applied to the classes ~~as before~~ before it is loaded, before its instances are created. This ensures the JVM ~~to no~~ implementation to assume that classes have certain safety properties.

→ The verification algorithm makes it possible to safely download the jar applets from the Internet.

⇒ How verification algorithm works?

The Java virtual machine specification contains a large ^{set} of rules that programs have to follow if they ~~have~~ want to run in the machine. It is the job of the verification algorithm to prove that these rules are followed by each class.

The verification algorithm works by asking a set of questions about class files.

The questions can be answered by just looking at the bytecode without executing the program.

→ The enables java virtual machine to stop badly behaving programs before they start.

→ It permits faster ~~verification~~ execution.

base example,

int $i = 70$;
 float $j = 111.1$;
 double $k = i + j$;

Initialize
 ldc 70 i // loads i
 dstore 1 ;
 ldc 111.1 ; // initialize j
 fstore 1 ;
 fload 1 ; // loads i
 fload 2 ; // loads j
 fadd ;
 dstore 3 ;

→ give error because at the top of stack, there is float so fadd → will create error.

✓

Jana Garbage Collection

The allocation and deallocation of memory for objects is done by garbage collection in an automated way by Java.

The developer not need to write ~~for~~ code for garbage collection process as in C.

It is an automatic process to manage memory at run time used by programs.

Garbage collector takes into consideration the memory space no longer available for use & empties it for future references.

Jana Garbage Collection

- GC Initiation

Developers need not to initialize garbage collection explicitly in code.

`System.gc()` & `request.gc()` are the

methods that are used to call garbage collection process.

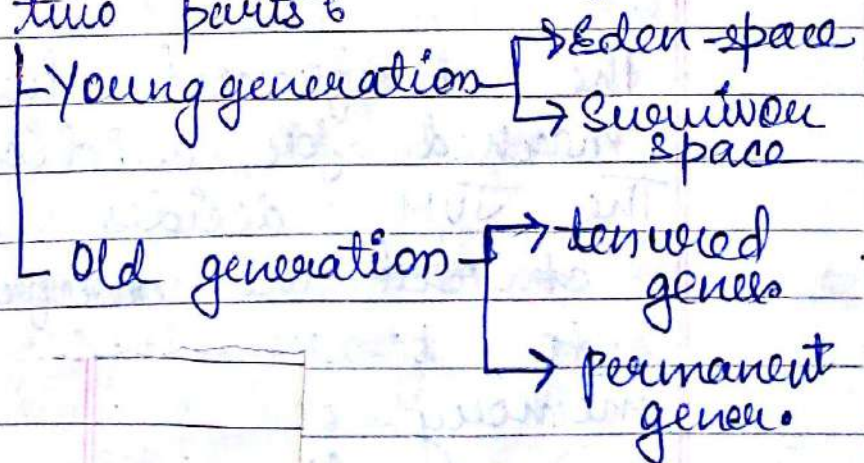
JVM can also choose to reject these ~~processes~~ requests so it is not guaranteed that ~~calling~~ these request can call the garbage collection process.

~~Hotspot~~ ~~JVM~~ ~~will~~

Decision is taken by the JVM based on the "eden" space availability in the heap memory.

Hotspot Virtual machine's garbage collector uses generational garbage collection process.

It separates JVM memory into ~~three~~ two parts:



Eden Space : When an instance is created it is firstly stored in Eden-space, or in young generation of heap area.

Survivor Space (S0 & S1) : Instances stored in Eden spaces are scanned; The instances that are live (still referenced) are move to S0 from Eden Space. And instances which are not live (not referenced) are marked for garbage collection (eviction process).

Similarly, ~~then~~ garbage collector scans the move all the live instances to S1 from S0.

The dereferenced instances (not live) marked for eviction process. The JVM decides either to ~~choose~~ the dereferenced instances to be removed from the memory or go on the eviction process will be done in the separate process.

Old generation : The live instances in the S1 are further promoted to old generation.

- Tenured generation is the ~~best~~ logical part of old generation in heap memory area.
- This process is called minor GC process.

Major GC : Old generation is the last phase of the instance cycle.

Major GC is the process that scans the old generation part of heap memory.

Memory fragmentation : Once the instances are deleted from the heap memory it empties the memory space to be used ~~for~~ in future for live instances.

This memory ~~area~~ ^{spaces} are fragmented across the memory area.

Finalisation of Instances is done by calling the `final()` method just before the eviction process.

Jana Security 6

The need of Jana security 6

- Downloaded executable content can be dangerous.
- The term security might lead us to expect that Jana programs would be
 - programs should not be allowed to harm the user's computing environment (Trojan horses, viruses)
 - programs should be prevented from discovering private information on the host computer.
 - Data sent and received by the program should be encrypted.
 - The identity of parties involved in the program must be verified
 - Rules of operations should be set & verified.

Security promises of the JVM

There are some security promises that JVM made ~~after~~ about programs that passed verification.

1. Every object is created ^{exactly} once, before it is used.
2. Every object is an instance of exactly one class, which can't be change through the life of object.
3. If field or method, is marked as private, then it can only be accessed within its class itself.
4. If field or method is marked protected, then it can only be accessed through the code which is an implementation of that class.
5. Every field is initialized before it ~~initialized~~ used.
6. Every ~~field~~ local variable is initialized, before it is used.
7. It is impossible to underflow or overflow ~~to~~ stack.
8. An attempt to use a null reference as a receiver of method invocation always throws a null pointer exception.

9. Final methods cannot be overridden and final class cannot be made subclass.
10. It is impossible to change the length of the array, ~~before~~ once it is created.
11. It is impossible to write past the end of the array or at the beginning of the array.

Security architecture and security policy

The Java platform builds a security architecture on the top of precautions premised by JVM.

~~The~~ A security architecture is the way of organising a platform that makes up java platform so that potentially harmful operations are isolated from unprivileged code and only available to the privileged code.

Most code are ~~up~~ unprivileged; security architecture is responsible for ensuring that unprivileged

code does not masquerade the privilege code.

~~The core of Java~~

There are three pillars of Java security :

1. Security Manager.
2. Class loader.
3. Bytecode Verifier.

The core of Java platform security architecture is security manager. This class decides which piece of code can perform certain operations and which cannot.

These decisions are called security policies.

Security manager ensures that the permission specified in policy file cannot be overridden.

It uses a `checkPermission()` method.

- This method uses permission object as an input and returns a 'yes' or 'no' (based on the source code & permissions)

checkpermission () is called from trusted classes.

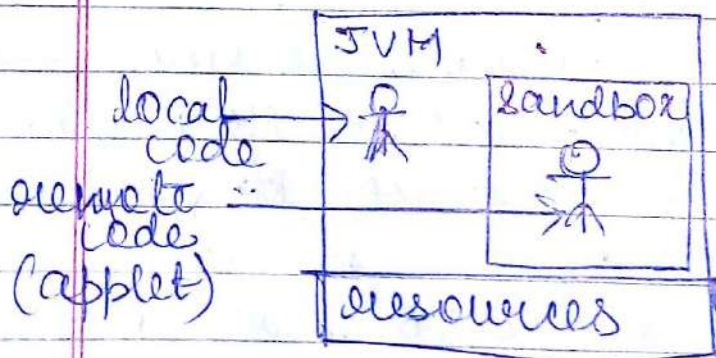
There can only be one security manager active at a time.

JDK has its own security manager. (default SM).

Java programs can ~~have their~~ replace the default security manager by their own security manager if they have permission to do so.

Sandbox Security Model

Idea is limit the access to system resources of by applet (remote code).



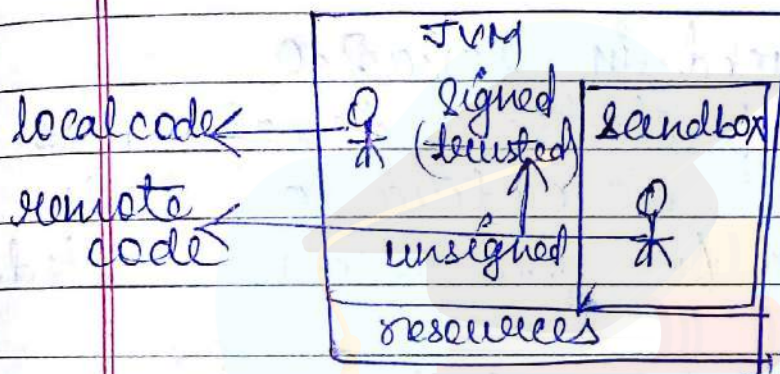
- Introduced in Java 1.0
- local code has unrestricted access to system resources.
- remote code (applet) has

restricted access to system resources.

→ remote code (applet) is limited to sandbox.

The Concept of trusted code

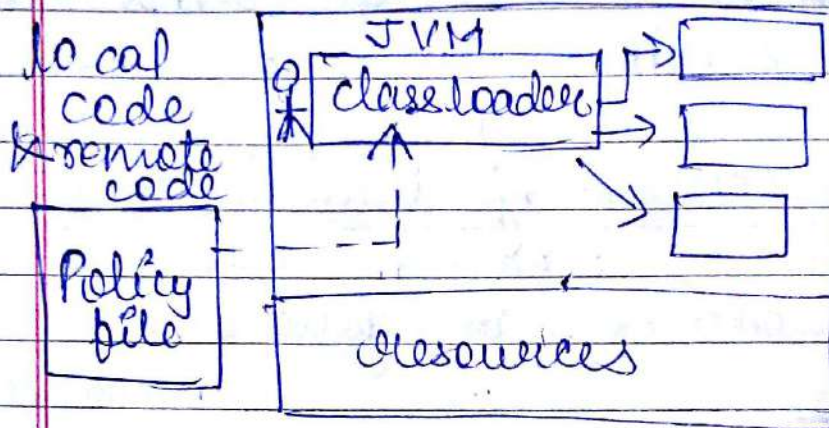
→ introduced in java 1.1



→ Based on the idea, that local code ~~can~~ has unrestricted ^{access} to the system resources.

→ Remote code (applet) signed by unauthorized principals
unsigned applet has restricted access to system-resources
→ limited to sandbox.

File - gran access control



- Introduced in Java 2.0
- The protection domain was created according to Code source & permission granted.
- Every Code (local & remote) has unrestricted access to the system resources based on the permissions written in policy file.
- Code source is consists of URL.
- permissions granted to the code source are specified in the policy file.

UNIT-2

#. Why 'main' is made static
→ Because to call main function we need to create object which will consume memory. for the class, so, in order to save memory, ~~we~~ avoid to create object we make main as static

new keyword is not used with primitive datatype because 'new' creates references whether primitive datatype dynamically stores / assigns value.

#. General structure of a Java program.

Document Section	→ suggested
package statement	→ optional
Import "	→ "
Interface "	→ "
class Definition	→ "
Main method class { main method Definitn }	→ Essential

If we access a variable with class name \rightarrow it is static.

String concatenation \rightarrow left to right.

Page:

Date: / /

Scope of variable \rightarrow accessibility or life-time of a variable.

* String \rightarrow sequence of characters. Anything written in double quotes " " String is inbuilt class in java. Hence it is not object but it is a class in java.

To access first character - `selection.charAt(0);`

* Wrapper classes:-

- \rightarrow For conversion.
- \rightarrow For converting string input to ~~value~~ objects & vice-versa for accessing them.

In Java, we have 8 primitive data types.

Syntax :- \rightarrow (are not classes)

`Wrapper.valueOf()`

\downarrow
wrapper class name.

Abstract^{ly} → Data Hiding.

Page:

Date: / /

Collections - getting all items

Simple Example.

```
list mylist = new ArrayList();  
// we add items.
```

```
Iterator iterator = mylist.iterator();  
while (iterator.hasNext())
```

```
{
```

String

Syntax for class

```
class classname { extends sub-class
```

```
}
```


Ex: Java class

④

class Box

{

double w, l, h;

double area()

{

return (w * l * h);

}

void setvalue(double w, double l, double h)

{

this.width = w;

this.length = l;

this.height = h;

}

class

BoxDemo

{

public static void main (String [] args)

{

Box b = new Box ();

Box c;

b.setvalue (10.2, 11.4, 12.1);

c = b;

d = b.addvalue (c);

↓

Invoking
objct.

Primitive type \rightarrow call by value
 , objects \rightarrow call by reference

Box add volume (Box b)

{

width = width + b.width

length = length + b.length

return this;

}

Sum took up for

- ①. name of the method
- ②. no. of arguments.
- ③. Type of arguments.

private protected \rightarrow accessible
 to each & every sub-class

All Instance of ~~this~~ class
 share the same static
 variable.

Reverse no. = reverse no * 10 + temp,
no. = no / 10.

Page: /

Date: / /

Restrictions on static blocks

- ①. static methods can only call other static methods.
- ②. They can only access static data.
- ③.

Final &

Syntax

final . type name .

Vararg → alternate for function overloading
→ concept in java through which we can pass multiple ~~different~~ arguments

→ Vararg is done with the ... operator

Example 6

lang. package

↓
math { → class }↓
sqrt, pow, sin etc { → functions
or
methods }

Eg. - Class Power

```
{
    public static void main (String arg
    []).
```

```
{
    double a = 5;
    double b = 2;
    double c;
    c = math. pow (5, 2);
    System.out. print ("power = " + c + "ans")
}
```

Output :- power = 25.0 Ans.

String

Round

$$\frac{4}{3} = 1.33 \rightarrow \text{round } 1.4$$

```

class Abc
{
    public static void main (String
                        arg[])
    {
        double a = 3.533;
        double b;
        b = Math. Round (3.533);
        System.out.print ("rounded" + b +
                        "ans");
    }
}

```

O/P → rounded = 4.0 Ans

```

class Xyz
{
    public static void main (String arg[])
    {
        double a = 2.6;
        double b;
        b = Math. floor (2.6);
        System.out.print ("reqd + b +
    
```


* Conditional statements:

If, else, nested if, ladder, switch.

(*)

```
import java.util.*;
class compare
{
    public static void main (String arg[])
    {
        Scanner sc = new Scanner(System.in);
        int a, b;
        System.out.print("enter a = ");
        a = sc.nextInt();
        System.out.print("enter b = ");
        b = sc.nextInt();
        {
            if (a > b)
            {
                System.out.print("a" + " is " + "greater");
            }
            if (b > a)
            {
                System.out.print("b" + " is " + "greater");
            }
        }
    }
}
```


Even/odd

```
import java.util.*;
class Compare1
{
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        int n;
        System.out.print("Enter n = ");
        n = sc.nextInt();
        {
            if (n % 2 == 0)
            {
                System.out.print("n" + " is " + "even");
            }
            if (n % 2 != 0)
            {
                System.out.print("n" + " is " + "odd");
            }
        }
    }
}
```


①

```
import java.util.*;
```

```
class grade
```

```
{
```

```
public static void main (String arg[])
```

```
{
```

```
Scanner sc = new Scanner (System.in)
```

```
int m1, m2, m3, m4, m;
```

```
System.out.print ("Enter m1 = ");
```

```
m1 = sc.nextInt();
```

```
System.out.print ("Enter m2 = ");
```

```
m2 = sc.nextInt();
```

```
System.out.print ("Enter m3 = ");
```

```
m3 = sc.nextInt();
```

```
m4 m4 = m1 + m2 + m3;
```

```
System.out.print ("Total = " + m4 + "ans");
```

```
m = m1 + m2 + m3 m4 / 3;
```

```
System.out.print ("Average = " + m + "ans");
```

```
{
```

```
if (m <= 100 && m >= 70)
```

```
{
```

```
System.out.print ("A");
```

```
}
```

```
elseif (m < 70 && m <= 60)
```

```
{
```

```
System.out.print ("B");
```



```
}  
else if (m < 60 && m <= 50)  
{  
    System.out.print("C");  
}  
else  
{  
    System.out.print("Fail");  
}  
}  
}  
}
```

Vowels / consonent

or

→ This program was to find the total marks of a student and to determine the result i.e., fail or pass.

Q Review

vowel / consonant

Date :

Page No.

#

```
import java.util.*;
class Letters
{
    public static void main(String
                                arg[])
    {
        Scanner sc = new Scanner(System.in);
        String N;
        System.out.print("Enter any character");
        N = sc.next();
        {
            if (N == 'a' || 'e' || 'i' || 'o' || 'u')
            {
                System.out.print("Vowel");
            }
            else
            {
                System.out.print("Consonant");
            }
        }
    }
}
```


Import java.util.*

FI/R by Scanner class.

m, n

obj = Input ()

obj . Output ()



TOPPERWorld

#. Adapter class

- used to simplify process of Event Handling in Java.
- As we know, when we implement any interface, ~~we~~ all the methods defined in that interface ~~are~~ needs to be override in the class, which is not desirable in the case of Event Handling.
- Adapter classes are useful as they provide an empty implementation of all the methods defined in an event listener interface.

Anonymous Inner class

~~class~~ → class i.e., not assigned any name

Example

class Anony - test extends Frame
{

Frame f;

Button b;

TextField t;

Anony - test ()

{

frame

f = new Frame();

f.setResizable(true);

f.setLayout(new FlowLayout());

f.setSize(200, 300);

t = new TextField();

b = new Button("click me");

f.add(b);

f.add(t);

b. Add Action Listener (new ActionListener)

{

public void actionPerformed (ActionEvent e)

{

~~String~~ tf.setText(~~add~~ "welcome");

}

}

};

};

~~Multi~~ Inheritance & Interface

Subclass are specialised version of Super class.

Benefits of Java's Inheritance

- ①. Reusability of code
- ②. code sharing
- ③. consistency in using an interface,

Types ①. Single Inheritance



class employee

{

private int eno;

private String ename;

public void set emp(int no, String ename)

{

eno = no;

ename = name;

}

public void put emp()

{

System.out.println("Emp no is " + eno);

System.out.println("Ename is " + ename);

}

}

class department extends employee

{

private int dno;

private String dname;

public void set dept (but no, String name)

```
{
    dno = no;
    dname = name;
}
```

public void put dept()

```
{
    System.out.println("deptno = " + dno);
    System.out.println("deptname = " + dname);
}
```

public static void main (String args[]) {

department d = new department();

d.setemp(100, "aas");

d.setdept(20, "sales");

d.putemp();

d.putdept();

}

Q. multilevel

A

↓

B

↓

C

Ex 6

```
class student
{
```

```
    private int stno;
```

```
    private String stname;
```

```
    public void set Stud(int no, int name)
```

```
{
```

```
    stno = no;
```

```
    stname = name;
```

```
}
```

```
public void put Stud()
```

```
{
```

```
    Sys O.P ("Student " + stno)
```

```
    S.O.P ("Student " + stname);
```


}

class marks extends Student

{

~~private~~ ~~int~~ go

protected int marks1, marks2;

public void set marks (int m1, int m2)

{

marks1 = m1;

marks2 = m2;

}

public void put marks()

{

S.O.P ("marks 1" + marks1);

S.O.P ("marks 2" + marks2);

}

}

class Finaltot extends marks

{

private int total;

public void calc()

{

total = marks1 + marks2;


```

}
public void putTotal()
{
    s.o.p ("Total B " + total);
}

```

```

P. s.v.m. (stmt args[]);
{

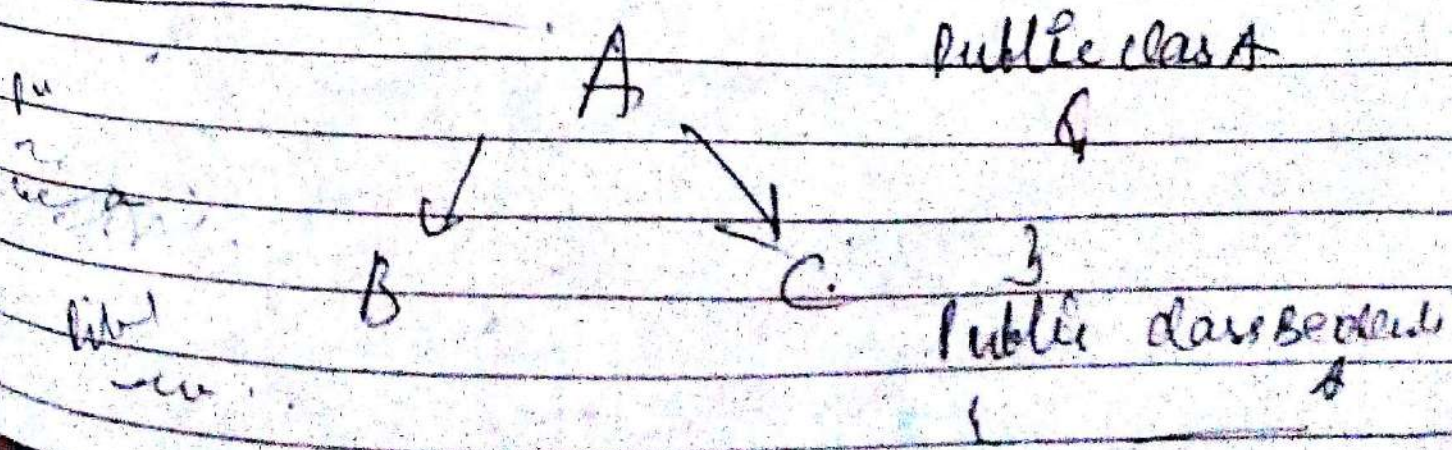
```

```

final tot f = new finaltot();
f.setstud(70, "sunny");
f.setmarks(78, 89);
f.calc();
f.putstud();
f.putmarks();
f.putTotal();
}
}

```

Hierarchal



Construction In Inheritance

```
class Parent {
```

```
    Parent()
```

```
    {
```

```
        S.O.R ("S1");
```

```
    }
```

```
class child extends Parent {
```

```
    child()
```

```
    { S.O.P ("S2");
```

```
    }
```

```
}
```

```
public class Test5 {
```

```
    P.S.V.M ( )
```

```
        child child = new child();
```

```
    }
```

```
}
```


④. Using "Super" keyword

⇒ Super is a reference variable that is used to refer immediate parent class object,

Uses of Super keywords are as follows:

- ① Super() is used to invoke immediate parent class constructor.
- ② Super is used to invoke immediate parent class method.
- ③ Super is used to refer immediate parent class variable.

Example

```
class base
{
    int a = 100;
}

class sup2 extends base
{
    int a = 200;
    void show()
    {
        // ...
    }
}
```


Example B

```
class A
```

```
{
```

```
    int i, j;
```

```
    A(int a, int b)
```

```
{
```

```
        i = a;
```

```
        j = b;
```

```
}
```

```
void show()
```

```
{
```

```
    S.o. P. ("i and j = " + i + " + j");
```

```
}
```

```
}
```

```
class B extends A {
```

```
    int k;
```

```
    B(int a, int b, int c)
```

```
{
```

```
        super(a, b);
```

```
        k = c;
```



```
void show() {
    S.O.P. ("K" + K);
}
```

```
public class Overload {
    P.S.V.M. ( )
    {
        B new b1 = new B(1, 2, 3);
        b1.show();
    }
}
```

O/R \Rightarrow K % 3

①. Overloading vs Overriding

method overloading is
compile-time polymorphism

\rightarrow way to provide ^{more} ~~one~~ than one method in one class with same name but diff. arg. to distinguish b/w them.
_{same name diff. arg.}

run time polymorphism

\rightarrow same name
same ~~arg~~ args
same return type

3. Method overloading forms the basis of dynamic method dispatch →

dynamic method dispatch It is an mechanism by which a call to an overridden method is resolved at run time, rather than the compile time.

Dynamic method dispatch is RUP because it is how Java implements run-time polymorphism.

* A superclass reference variable can refer to a subclass object.

→ Java uses this fact to resolve calls to overridden methods at run time.

```

class Parent
{
    int i = 10;
    void
    
```


Abstract class

An abstract class is a class that is declared abstract

It may or may not include abstract methods

→ cannot be instantiated

→ An abstract method has no body.

→ cannot create objects

→ An abstraction method is a method that is declared without an implementation.

Syntax:

```
abstract void moveTo (double  
deltaX,  
double  
deltaY);
```

```
Public abstract class GraphicsObject {  
    abstract void draw();  
}
```


In order to use abstract-classes,
they have to be sub-classed.

Examples

```

Abstract class parent-class
{
    [abstract void meth1();
    void meth2();
    }
    S.O.P ("method of abstract class");
}

```

```

class child extends parent
{
    void meth1() ←
    {
        S.O.P ("method of child class");
    }
}

```

```

class abstract
{
    P. S. V. M. —
    {
        child c1 = new child();
        c1.meth1();
    }
}

```


c2.meth2()

}
}



Interface

An Interface is a collection of abstract methods.

A class implements an interface, thereby implementing the abstract methods of the interface.

An interface is not a class.

A class describes the attributes and behaviours of an object.

An interface contains behaviour that a class implements.

- Interface is not extended by a class it is implemented by a class.
- An Interface can extend multiple Interfaces.

Declaring Interface

```

public Interface emp
{
    public void salary();
    int increment = 10;
}

```

Implementing Interface

```

class InterfaceTest implements emp
{
    public void salary()
    {
        S.O.P("salary of emp is "
            + (increment
            + 5000))
    }

    public static void main(String args[])
    {
        InterfaceTest P1 = new InterfaceTest();
        P1.salary();
    }
}

```


How Interface provides multiple Inheritance in java

Interface emp

Bj

```
public void salary()
    int increment = 10;
```

}

Interface director

{

```
public void salary()
    int increment = 1000000;
```

}

Class InterfaceTest implements emp, director

{

```
System.out.println("Salary of emp: "
    + (emp.increment + 100));
```

```
System.out.println("Salary of director: "
    + (director.increment + 500000));
```

}

```
public static void main (String args[])
    InterfaceTest I1 = new InterfaceTest
```


I 1. salary () ;

}

}

Package 6

Package action-forma;

public class add

{

int a;

public int b

public add-math()

{

S.O.P (sum is $a + b$)

}

}

}

Use of "final" 6

- ①. Final variable → Once a variable is declared as final, its value cannot be changed during the scope of the program.
Final int i=10;
- ②. Final method → cannot be overridden.
Final void method() overridden
- ③. Final class → cannot be inherited.
Final class FinalMethod

Exception Handling

Exception → problem arises during execution.

scenarios → invalid ip

→ open file to be opened not found

→ lost of network connection

&

Handling

try to catch the exception object throw by the error condition.

Types

Checked Exception

↳ ~~IO~~ IO exception, SQL Exception

Unchecked Exception → Runtime Exception

↳ Null pointer Exception

↳ Arithmetic Exception

Error → not exceptions at all

Hierarchy of Exception &

Object

Throwable

Exception

IO Exception

SQL Exception

Runtime Exception

Arithmetic Exception

Null pointer Exception

NumberFormatException

Error

Virtual machine error

Assertion Error

TOPPERWorld

Following five keywords are used to handle exception in Java

1. try
2. catch
3. ~~throw~~ finally
4. throws
5. throws

try-catch block

- placed around the code that might cause exception.
- code within this block is referred to as protected code.

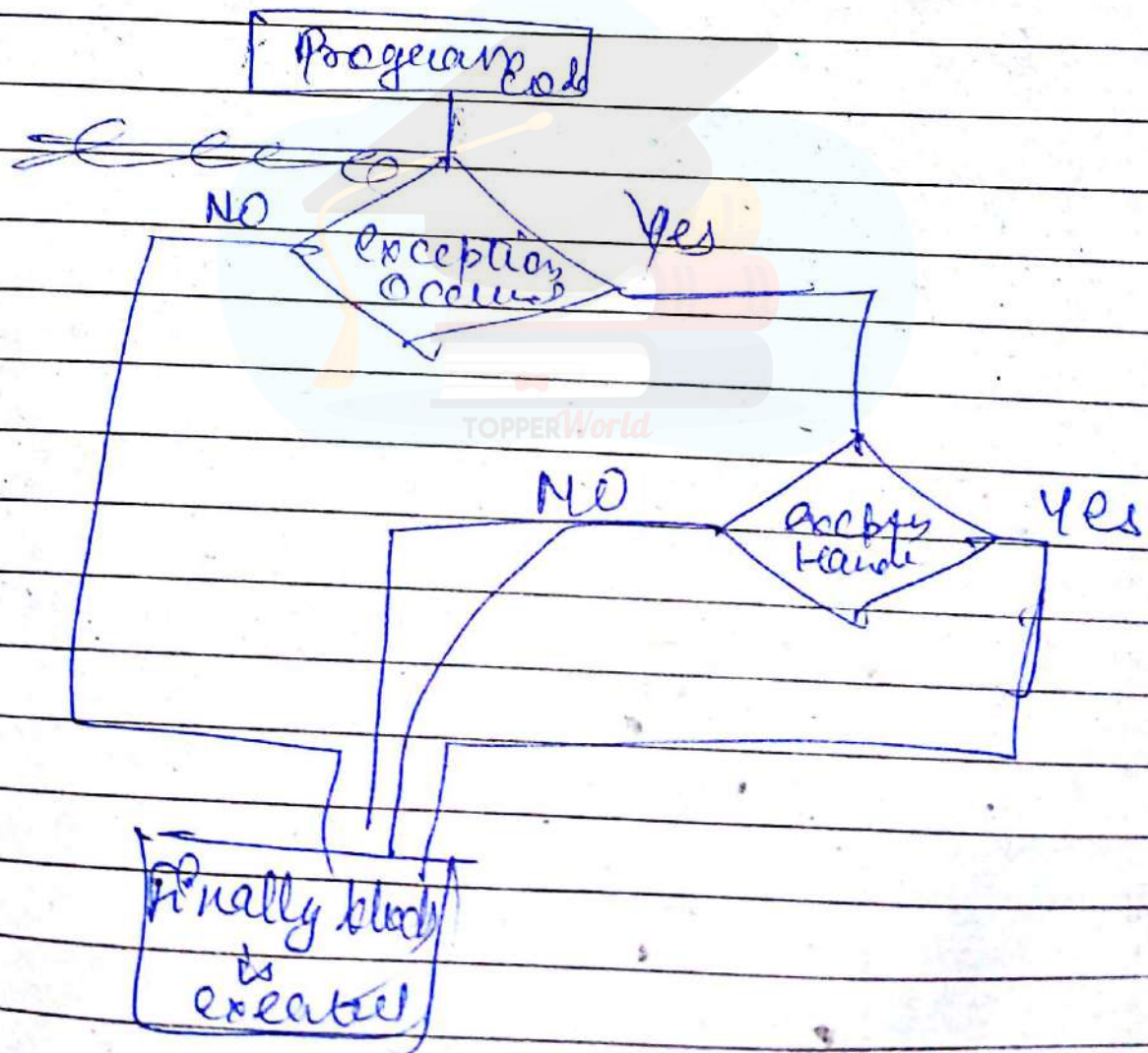
Syntax

```
try
{
    // protected code
} catch (exception e)
{
    // catch block.
}
```


Catch Statement \Rightarrow declaring the type of exception we are trying to catch.

Finally

\rightarrow always executed whether or not an exception has occurred.



toy

{

}

cath

{

}

finally

{

}

↓ sequence

}

}

TOPPERWorld

Applets

Applet is a special type of program that is embedded in the webpage to generate the dynamic content.

- It runs inside the browser.
- Applets are used to provide interactive features to web application that cannot be provided by HTML alone.
- A Java applet extends the Java Applet class.
- The class must override the methods from the applet to set up a user interface.
- An applet is automatically loaded and executed when you open a web-page that contains it.
- Environment of the applet is known as the context of the applet.

→ Use javac to compile and appletviewer to execute them.

→ All applets are subclasses of Applet.

→ Applets are not standalone programs.

They run within either a web-browser or an applet viewer.

Execution of applet does not begin at main().

→ Use AWT methods.

→ To use an applet, it is specified in an HTML file.

#. Java Applet Advantages :

1. It is simple to make it ~~work~~ on any platform.

2. ~~It~~ It is fast - can ~~to~~ even have similar performance as native performed software.

3. It works at client side, so less response time.

4. Secured.

Java Applet disadvantages

- requires Java plug-in.
- Java applets with specific requirements are impossible to run unless Java is manually updated.

Difference between Applet & Application

<u>Application</u>	<u>Applet</u>
we cannot make web-pages by using application.	we can make web-pages using applets.
cannot use HTML tags in application.	we can use HTML tags in Applet.
for execution of application, there is no need for web-browser.	for executing applet, web-browser is required.
To execute application, Java interpreter is needed.	To execute applet, Java - web-enabled browser is needed.

In application program execution begins with `main()` method.

In applet, execution does not begin with `main()` method.

No necessarily to use inheritance in a `Applet` program.

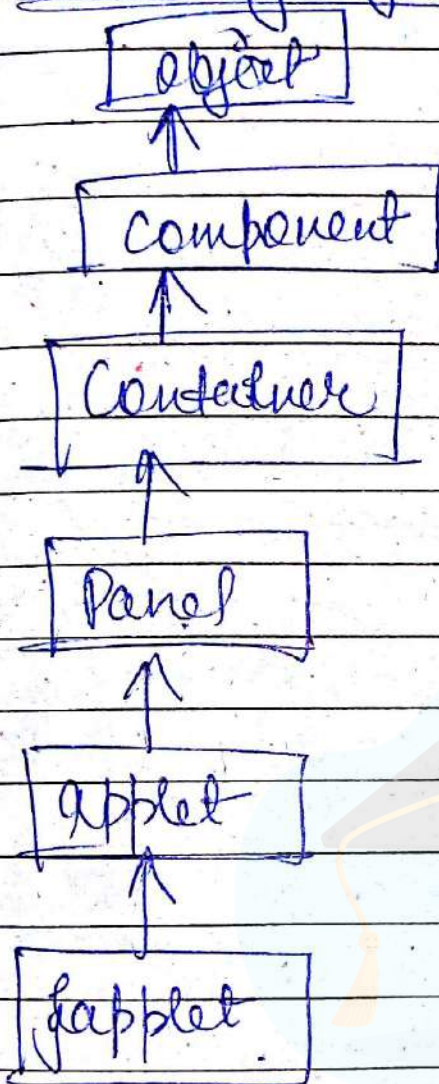
Applet is inherited from `Applet`.

Ex of Applet

```
import java.awt.*;  
import java.applet.Applet;  
public class First extends Applet {  
    <applet code =
```

```
    public void paint(Graphics g) {  
        g.drawString("welcome", 100, 100);  
    }  
}
```


⑦ Hierarchy of Applet



- Applet class extends Panel
- Panel class extends Container
- Container is a sub-class of Component.

→ Applet interact with the user through the AWT not console-based I/O classes

→ Applet derived from AWT

steps to incorporate and run an applet

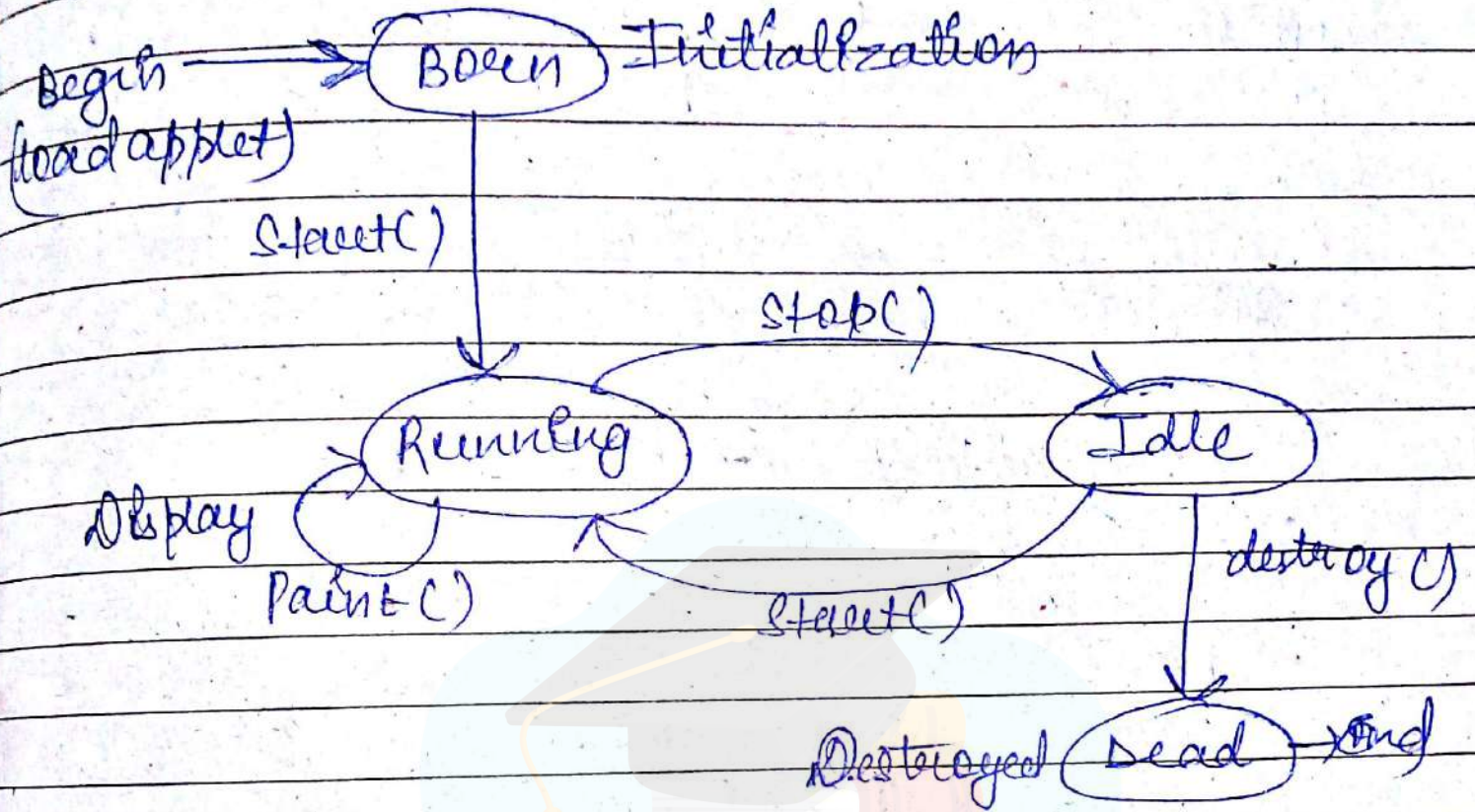
1. Name Myapplet.java
2. javac Myapplet.java
3. Name Myapplet.class
4. Create Myapplet.HTML.

<applet code = "Myapplet.class" width = 200
height = 200 > </applet>

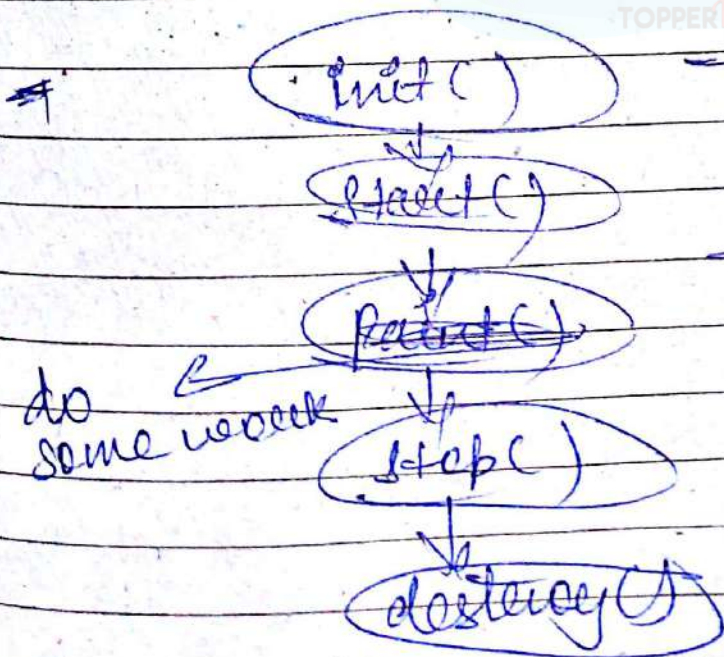
→ Applets do not have main
Instead they have:

init()
start()
paint()
stop()
destroy()
update()
repaint()

#. Applet Life - Cycle



Exit of browser



→ `init()` and `destroy()` are only called once

→ `start()` and `stop()` are called whenever a browser enters or leaves the page

→ do some work is code called by your listeners.

Applet methods

```
public void init()
public void start()
public void stop()
public void destroy()
public void paint(Graphics)
```

④, Why an applet works?

- we create an applet by extending Applet
- Applet defines methods— `init()`, `start()`, `paint(Graphics)`, `stop()`, `destroy()`
- These methods are do nothing—they are stub.
- we make the applet do something by overloading these methods

⑧. The init() method

→ executes only one time in the life of an applet.

→ ~~red~~ init() method is called ~~where~~ the first time applet is loaded into the memory of the computer.

→ Initialize variables & components like buttons and ^{checkbox} boxes.

⑧. start() method

start() method is called immediately after the init() method and every time the applet receives focus as a result of scrolling in the active window.

→ We use this method, if we want to restart a process.

⑧. stop() method

→ used to reset variable and stop the threads that are running.

⑧. destroy() method

→ destroy() method is called by the browser when the user moves to another page.

→ this method is used to perform cleanup operations like closing a file.

⑩. Maintaining the applet:

⑩. update() method

→ takes Graphics class object as a parameter

→ update() method is called to clear the screen and call the paint() method.

→ screen is then refreshed by the system.

⑩. Paint() method

→ draws the graphics of an applet in the drawing area.

The method is automatically called the applet is displayed on the screen is every time applet receives the focus.

→ The paint() method can be triggered by invoking the repaint() method.

⑩. repaint() method

→ called when applet area to be redrawn.

repaint() method calls the update() method that the applet has to be redrawn. The default action of update()

method is to ~~call the~~ refreshes the screen & call the paint() method.

drawstring () method,

↳ member of graphic class,
syntax;

void drawstring (String message, int x, int y)

Ex 6 import java.awt.*;
import java.applet.*;

// <applet code = "applet2.class", width = 200,
// height = 200 > </applet>

public class applet2 extends Applet {

int in = 0, st = 0, st0 = 0, del = 0;

public void init()

{
in++;
repaint();
}

public void start()

{
st++;
repaint();
}


```
public void stop()
{
```

```
    sto = 0;
    repaint();
}
```

```
public public void destroy()
{
```

```
    des = 0;
    repaint();
}
```

```
public void paint(Graphics g)
{
```

```
    g.drawString("init has invoked" + String.valueOf(sto),
    10, 20);
```

```
    g.drawString("start has invoked" + String.valueOf(st),
    10, 35);
```

```
    g.drawString("stop has invoked" + String.valueOf(sto),
    10, 50);
```

```
    g.drawString("destroy has invoked" + String.valueOf(
    10, 65));
```

```
}
```


Advantages of Java

Thread // Applet

Date :

Page No.

Threads

→ Java has built in thread support for multithreading.

→ Synchronisation.

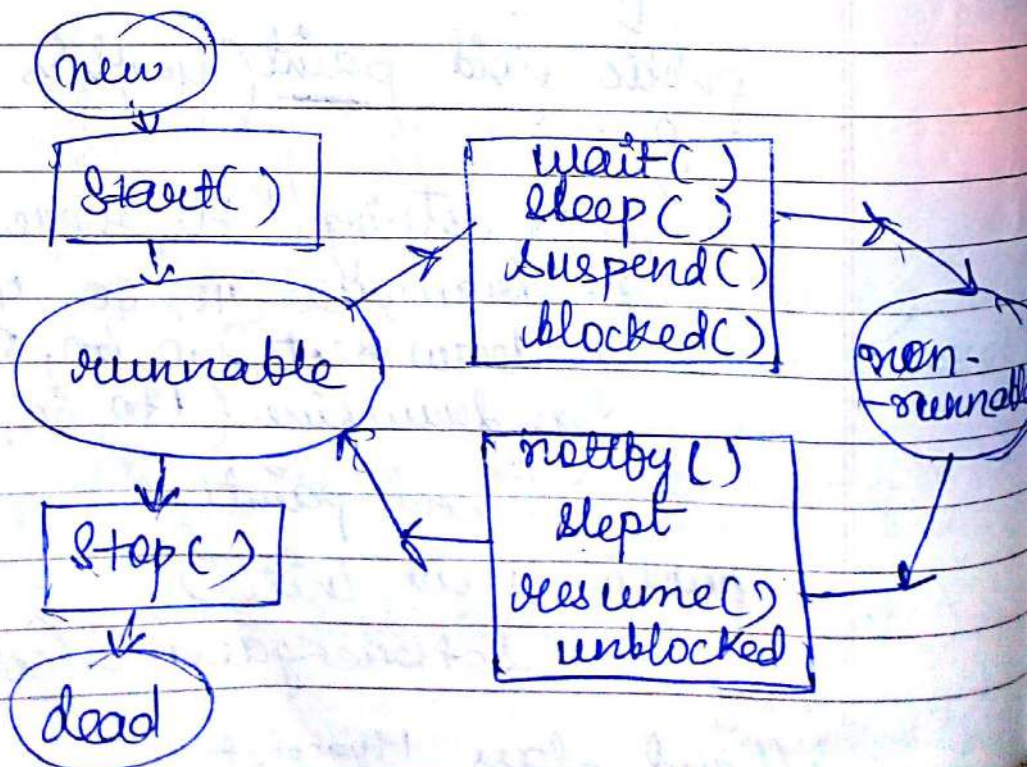
→ Thread scheduling.

→ Inter thread Communication

Current thread	start	setpriority
yield	run	getpriority
sleep	stop	suspend
		Resume

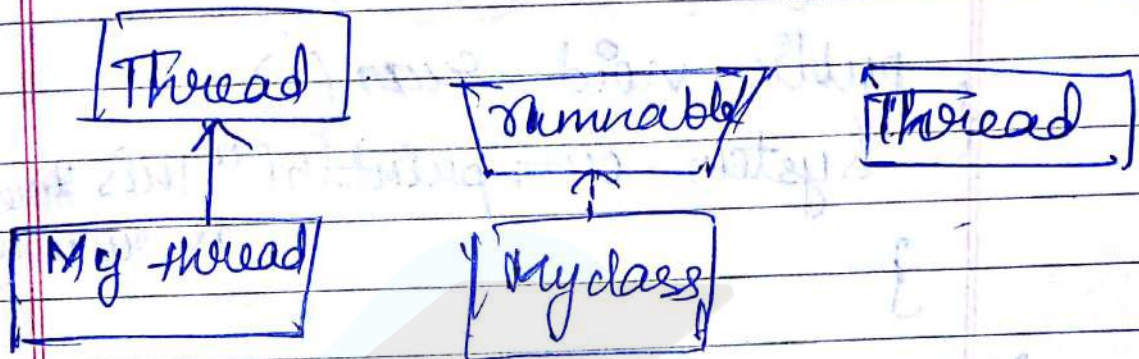
→ Java - garbage collector is a low priority thread.

Thread States



⑧. Threading mechanisms :-

- Create a class that extends the thread class.
- Create a class that implements the runnable interface



⑧. 1st method :- Extending the thread class.

```

class MyThread extends Thread
{
    public void run()
    {
        // thread body execution
    }
}
  
```

⇒ Creating Thread :
 MyThread thr1 = new MyThread();

Start Execution :
thr1.start();

Example 6-

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("This thread
        is running")
    }
}
```

//end class MyThread

Example 7 // utilization of a thread.

class ThreadEx2 {

```
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
```

t.start();

```
    }
}
```


2nd method — for Runnable Interf-
-ace.

class MyThread implements Runnable

{

public void run()

{

// - - -

}

}

⊕ Creating object :-

MyThread

myobject = new

MyThread
()

ps)

Creating Thread object.

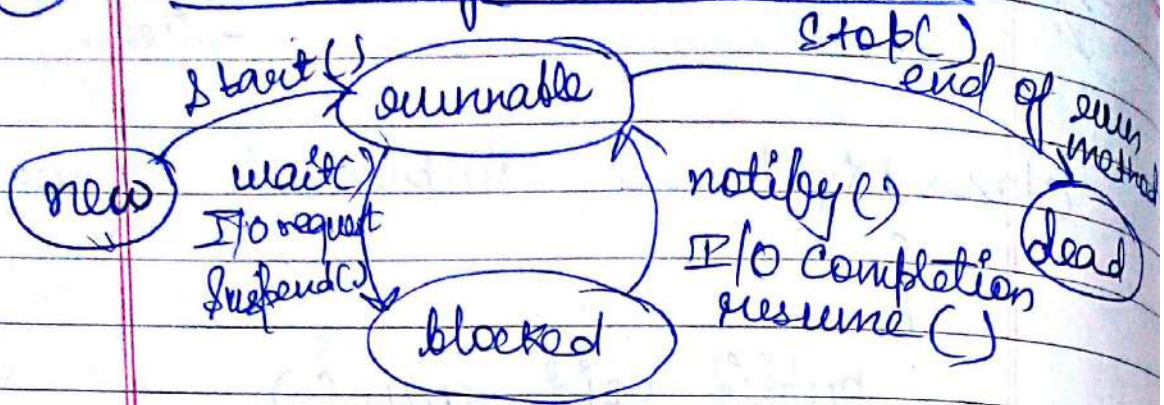
Execution → top to bottom.

Date :

Page No.

⑪

States of Java Threads



⑫

Creating Threads Example 6-

```
class Outputs extends Thread {
    private String today;
    public Output (String st) {
        today = st;
    }
```

```
    public void run() {
        try {
```

```
            for (i=0; i<10; i++) {
```

```
                System.out.println(today);
                sleep(1000);
            }
```

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



```

class program {
    public static void main(String[] args)
    {
        Output thr1 = new Output("Hello");
        Output thr2 = new Output("There");
        thr1.start();
        thr2.start();
    }
}

```

⑧. class Output implements Runnable

```

{
    private String tosay;
    public Output(String st) {
        tosay = st;
    }
    public void run() {
        try {
            for(;;) {
                System.out.println(tosay);
                Thread.sleep(1000);
            }
        }
    }
}

```



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

```
}
```

```
}
```

```
class program {
```

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

```
        Output out1 = new Output ("Hello");
```

```
        Output out2 = new Output ("There");
```

```
        Thread thr1 = new Thread (out1);
```

```
        Thread thr2 = new Thread (out2);
```

```
        thr1.start();
```

```
        thr2.start();
```

```
    }
```

```
}
```


Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. There are two distinct types of Multitasking i.e. Processor-Based and Thread-Based multitasking.

Q: What is the difference between thread-based and process-based multitasking?

Ans: As both are types of multitasking there is very basic difference between the two. Process-Based multitasking is a feature that allows your computer to run two or more programs concurrently. For example you can listen to music and at the same time chat with your friends on Facebook using browser. In Thread-based multitasking, thread is the smallest unit of code, which means a single program can perform two or more tasks simultaneously. For example a text editor can print and at the same time you can edit text provided that those two tasks are performed by separate threads.

Q: Why multitasking thread requires less overhead than multitasking processor?

Ans: A multitasking thread requires less overhead than multitasking processor because of the following reasons:

Processes are heavyweight tasks where threads are lightweight

- Processes require their own separate address space where threads share the address
- space Interprocess communication is expensive and limited where Interthread
- communication is inexpensive, and context switching from one thread to the next is lower in cost.

Benefits of Multithreading

1. Enables programmers to do multiple things at one time.
2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution
3. Improved performance and concurrency
4. Simultaneous access to multiple applications

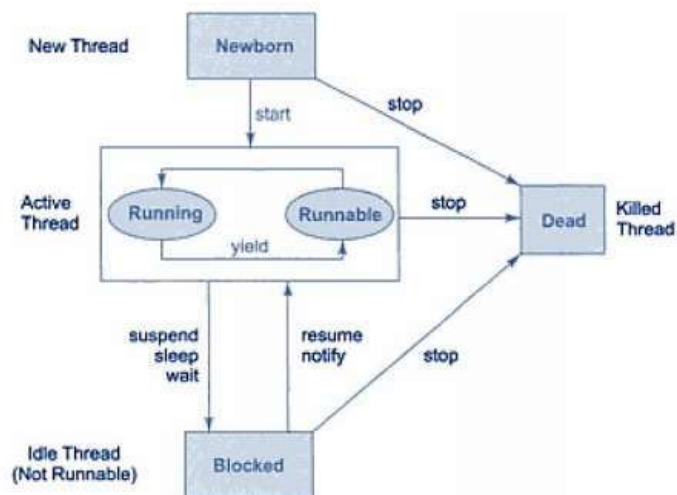
Life Cycle of Thread

A thread can be in any of the five following states

1. Newborn State: When a thread object is created a new thread is born and said to be in Newborn state.
2. Runnable State: If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion
3. Running State: It means that the processor has given its time to the thread for execution.

A thread keeps running until the following conditions occurs

- a. Thread give up its control on its own and it can happen in the following situations
 - i. A thread gets suspended using suspend() method which can only be revived with resume() method
 - ii. A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds
 - iii. A thread is made to wait for some event to occur using wait () method. In this case a thread can be scheduled to run again using notify () method.
 - b. A thread is pre-empted by a higher priority thread
4. Blocked State: If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.
 5. Dead State: A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.



Main Thread: Every time a Java program starts up, one thread begins running which is called as the main thread of the program because it is the one that is executed when your program begins.

Child threads are produced from main thread

- Often it is the last thread to finish execution as it performs various shut down operations
- Creating a Thread Java defines two ways in which this can be accomplished: You can implement the Runnable interface.
- You can extend the Thread class, itself.
- Create Thread by Implementing **Runnable**


The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run(), which is declared like this:

public void run()

You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.

Thread defines several constructors.

The one that we will use is shown here: 

Thread(Runnable threadOb, String threadName);

Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName. 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 is shown here:

void start();

Example to Create a Thread using Runnable Interface


```
class t1 implements Runnable
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

Create Thread by Extending Thread

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run() method, which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

Example to Create a Thread by Extending Thread Class

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
C:\NIEC Java>
```


SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

Q: Can we start a thread twice?

Ans: No, if a thread is started it can never be started again, if you do so, an illegal Thread State Exception is thrown. Example is shown below in which a same thread is coded to start again


```

class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
        obj1.start();
    }
}

```

As you can see two statements to start a same thread is written in the code which will not give error during compilation but when you run it you can see an Exception as shown in the Output Screenshot.

Output:

```

C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Unknown Source)
    at t2.main(t2.java:11)

```

Important Thread Methods

1. **Yield Method** - Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2. **Stop Method** - kills the thread on execution
3. **Sleep Method** - Causes the currently running thread to block for at least the specified number of milliseconds. You need to handle exception while using sleep() method.
4. **Suspend and Resume Method** - A suspended thread can be revived by using the resume() method. This approach is useful when we want to suspend a thread for some time due to certain reason but do not want to kill it.

Example

Thread Priority

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

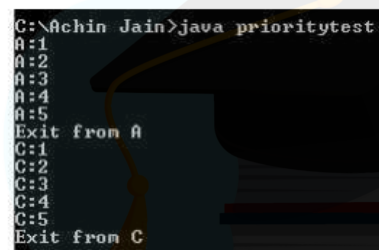
Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Example

```
class prioritytest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        a.setPriority(10);
        c.setPriority(1);
        c.start();
        a.start();
    }
}
```

In the above code, you can see Priorities of Thread is set to maximum for Thread A which lets it to run to completion ahead of C which is set to minimum priority.

Output:



```
C:\Achin Jain>java prioritytest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C
```

Use of `isAlive()` and `join()` method

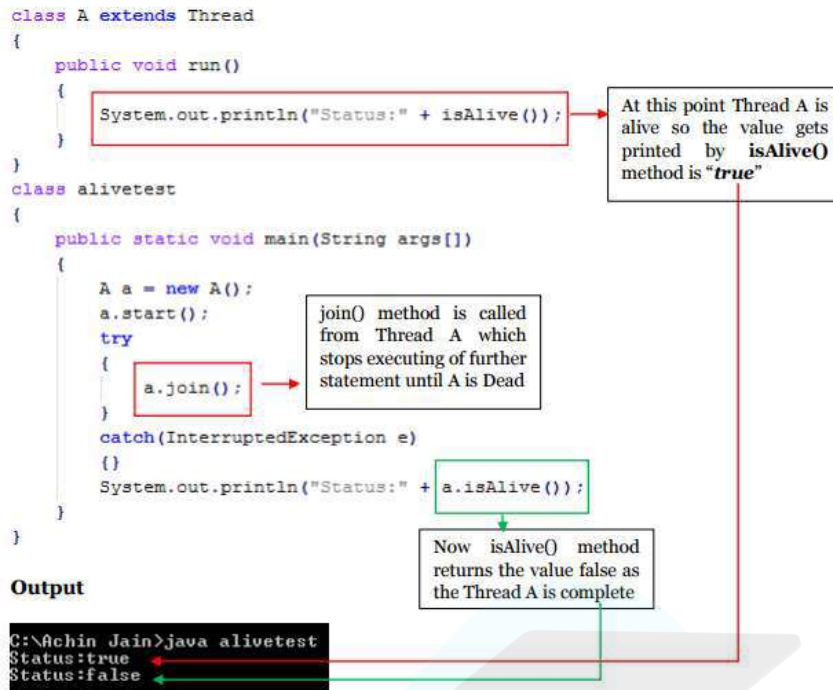
The `java.lang.Thread.isAlive()` method tests if this thread is alive. A thread is alive if it has been started and has not yet died. Following is the declaration for `java.lang.Thread.isAlive()` method

public final boolean isAlive()

This method returns true if this thread is alive, false otherwise.

join() method waits for a thread to die. It causes the currently thread to stop executing until the thread it joins with completes its task.

Example



Synchronization

When two or more threads need 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 synchronization is achieved is called thread synchronization. The `synchronized` keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

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

Problem without using Synchronization

In the following example method `updatesum()` is not synchronized and access by both the threads simultaneously which results in inconsistent output. Making a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of the code. Writing the method as synchronized will make one thread enter the method and till execution is not complete no other thread can get access to the method.


```

synchronized void updatesum(int i)
{
    Thread t = Thread.currentThread();
    for(int n=1; n<=5; n++)
    {
        System.out.println(t.getName()+" : "+(i+n));
    }
}

```



```

class update
{
    void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n));
        }
    }
}
class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}
class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}

```

Diagram illustrating the synchronization of the `updatesum` method. A red box highlights the `void updatesum(int i)` method signature in the `update` class, with an arrow pointing to another red box containing `synchronized void updatesum(int i)`. A text box explains: "Output when method is declared as synchronized".

```

C:\Achin Jain>java syntest
Thread A : 11
Thread A : 12
Thread A : 13
Thread A : 14
Thread A : 15
Thread B : 11
Thread B : 12
Thread B : 13
Thread B : 14
Thread B : 15

```

Interthread Communication

It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methods of Object Class:

wait(): This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

notify(): This method wakes up the first thread that called wait() on the same object.

notifyAll(): This method wakes up all the threads that called wait() on the same object. The highest priority thread will run first. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

Example

```
class customer
{
    int amount = 0;
    int flag = 0;
    public synchronized int withdraw(int amount)
    {
        System.out.println(Thread.currentThread().getName()+" : is going to withdraw");
        if(flag==0)
        {
            try
            {
                System.out.println("Waiting...");
                wait();
            }
            catch(Exception e)
            {
            }
        }
        this.amount-=amount;
        System.out.println("Withdraw Complete");
        return amount;
    }
    public synchronized void deposit(int amount)
    {
        System.out.println(Thread.currentThread().getName()+" : is going to Deposit");
        this.amount+=amount;
        notifyAll();
        System.out.println("Deposit Complete");
        flag=1;
    }
}

class threadcomm
{
    public static void main(String args[])
    {
        final customer c = new customer();
        Thread t1 = new Thread()
        {
            public void run()
            {
                c.withdraw(5000);
                System.out.println("After withdraw Amount is :"+ c.amount);
            }
        };
        Thread t2 = new Thread()
        {
            public void run()
            {
                c.deposit(10000);
                System.out.println("After Deposit Amount is :"+ c.amount);
            }
        };
        t1.start();
        t2.start();
    }
}
```



If both these methods are commented which means there is no communication, output will be inconsistent. See [Output 2](#)

3

83

AWT

GUI → elements of GUI include windows, dropdown list, buttons, scroll bar, icons, images & labels.

AWT is a package that provides an integrated set of classes to manage user interface component like window, dialog box, push button, list, menus, scrollbar & text-field.

- provide support for UI containers
- provides an event system for ~~provide~~ managing events.

Components of AWT

Every GUI based program consists of a screen with set of objects. In Java, these objects are called components. Components frequently used are buttons, checkboxes, radiobuttons, textfield etc.

- At the top of SWT Hierarchy, is the Component class. Component is an abstract class that encapsulates all the attributes of the components.
- All User Interface elements that are displayed on the screen and interact with user are the sub ^{classes} ~~ordinates~~ of the ~~class~~ component.

① Labels

- object of type Label
- contains a string which it displays.
- do not support any interaction with the user.

Constructors

- Label () → creates ^{Empty} label with its text alignment left
- Label (string) → creates label with given string
- Label (string, left) → creates label with given string with text alignment left

given text alignment.

available alignment

Label.Left

Label.Right

Label.Center

Ex 6

Imp

Imp

public class LabelDemo extends Applet

{

public void run()

{

~~Label one = new Label("one");~~

Label one = new Label("one");

add(one);

}

,

⑥. Buttons are simple components that trigger some action on your interface.

Button()

Button(String)

Ex 6

Applet

Applet

public class Button1Demo extends Applet

```
{
    Button yes;
    public class Button1()
    {

```

~~Button yes;~~

yes = new Button("yes");

add(yes);

}

⑦. CheckBoxes

→ Selected or deselected to provide options.

Constructors

checkboxes()

checkbox(String)

→ used as a placeholder for a group

checkbox(String, null, boolean).

To retrieve current state of checkbox
call `getState()`.

To set its state, call `setState()`.

Methods :

`Boolean getState()` → if on is true, box is checked
`void setState (boolean on)`
`String getLabel()`
`void setLabel (String str)` → otherwise false

Ex 6

~~import~~ `java.awt.event.*;`

~~import~~ `java.applet.*;`

`public class checkboxDemo extends Applet` —

{

`Checkbox winXP;`

{

`public void init()`

{

`winXP = new Checkbox("WindowsXP", null, true);`

`add(winXP);`

}

④. Radio buttons

- Radio buttons are a variation on the checkbox.
- have same appearances as checkboxes but only one in a series can be selected at a time.

```

import java.awt.*;
import java.applet.*;

public class CBGroup extends Applet
{
    CheckboxGroup cbg;
    CheckboxGroup window, windowA;

    public void init()
    {
        cbg = new CheckboxGroup();
        window = new Checkbox("window",
                               null, true);
        windowA = new Checkbox("windowA",
                               null, false);

        add(window);
        add(windowA);
    }
}

```

TOPPERWorld

⑤. Choice menu

→ pull-down menus that enable us to select an item from that menu.

Ex: Emp

```
Emp ————— choiceDemo
public class Emp extends Applet {
```

```
    choice OS, Browser;
    String msg = " ";
    public void init() {
        OS = new choice();
        OS.add("Windows XP");
        OS.add("Windows Vista");
        OS.add("Mac OS");
        OS.add("Solaris");
    }
```

```
    add(OS);
}
```


⑥. Textfield → handles a single line text.

~~Method:~~

TextArea → handles multiple lines of text.

methods:

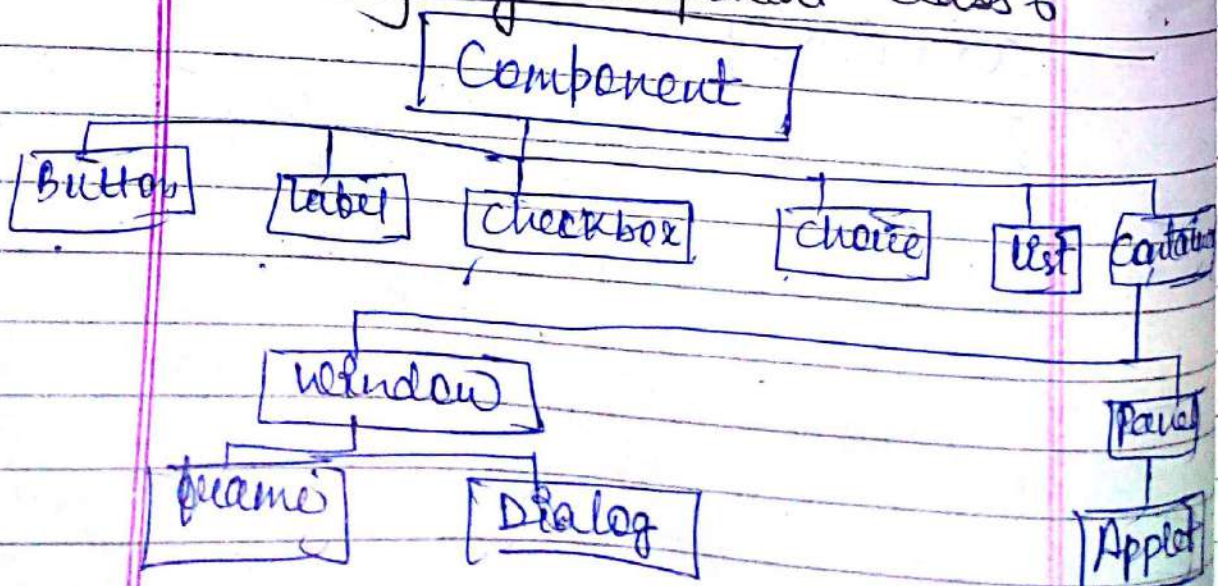
textfield tf = new Textfield();
↳ Empty constructor

Textfield tf = new Textfield(10);

Textfield tf = new Textfield("Enter name");

add(tf);

Class Hierarchy of Component class:



④. Container

- Container is a Component which contains other Components.
- Containers are themselves Components.
- Components are contained in container.

Swing Components inherit directly or indirectly from the JComponent class, which is an indirect subclass of old AWT Component class.

④. Panel

- Sub-class of container
- doesn't add any methods.
- implements container
- when screen output is directed to an applet, it is drawn on the surface of panel object.
- components can be added by add() method.
- doesn't contain title bar, menu bar or border.

Create a panel;

```
Panel p1 = new Panel();
```


add panel to a window or frames
 Frame F1 = new Frame (a Frame window)
 Panel P1 = new Panel ();
 f1.add (P1);

Layout Management

- layout management is the process that determines the size and position of components in a container.
- manages the layout of components in a container.
- Every component has a ~~perfect~~ default layout manager.
- Each time we create a container, java automatically creates a default layout manager.
- layout managers determine the placing of control in the applet's display.
- you can create different types of layout managers in order to control how your applet looks.
- set by `setLayout()` method.
- no call is made then default layout manager is used.

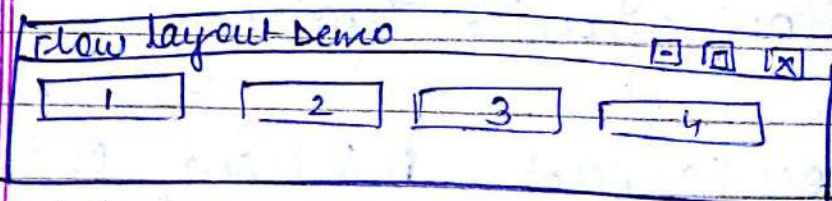
void setLayout(LayoutManager layoutObj)

AWT provides following layout managers:

- Flow layout
- Grid layout
- Border layout
- Card layout
- Grid Bag layout

Flow layout is most basic of layout.

- Components are added to the panel one at a time, row by row.
- has an alignment, which determines the alignment of each row.



- default layout manager.

Constructors for flow layout &
 flowlayout
 flowlayout (int hgap)
 flowlayout (int hgap, int vgap, int wrap)

① Flowlayout ()
 → default layout, which centres components and leaves five pixels of space between each component.

② Flowlayout (int hgap)
 → It lets us specify how each line is all aligned. Valid values for hgap are:

FlowLayout.LEFT
 FlowLayout.CENTER
 FlowLayout.RIGHT
 FlowLayout.LEADING
 FlowLayout.TRAILING

③ Flowlayout (int hgap, int vgap, int wrap)
 → specify the horizontal & vertical space left between components.

Ex 6

```

import java.applet.Applet;
import java.awt.*;

public class Flow1 extends Applet
{
    Button b1, b2, b3;
    FlowLayout flow;
    flow = new FlowLayout(FlowLayout
        LEFT, 10, 10);

```

ad
one
used

```

    setLayout(flow);
    b1 = new Button("Button 1");
    b2 = " " " (" " " 2");
    b3 = " " " (" " " 3");
    add(b1);
    add(b2);
    add(b3);
}

```

TOPPERWorld

Grid Layout

- layouts component in two dimensional grid.
- we defines number of rows & columns.

1	2	3
4	5	6
7	8	9

GridLayout () → single column grid layout formed

GridLayout (int numRows, int numColumns)

GridLayout (int numRows, int numColumns, int horiz, int vert)

for creates grid layout with the specified number of rows & columns.

Specify the horizontal & vertical space left between columns.


```
Ex6  import java.applet.* Applet;
      import java.awt.*;
      public class Geid extends Applet
      {
```

```
    public void init()
    {
```

```
        Button B1, B2, B3;
```

```
        GridLayout G1;
```

```
        G1 = new GridLayout(4 2, 2);
```

```
        setLayout(Grid G1);
```

```
        Button B1 = new Button("Button 1");
```

```
        B2 = new Button("Button 2");
```

```
        B3 = new Button("Button 3");
```

```
        add(B1);
```

```
        add(B2);
```

```
        add(B3);
```

```
    }
```

```
}
```


Border Layout

→ ~~the~~ indicate geographic directions
• north, south, east, west and
centre.

BorderLayout() → default border layout

BorderLayout(int horiz, int vert)

specify how & vertical
space between components.

Regions -

BorderLayout.Center

- 4 • East
- 4 • West
- 4 • North
- 4 • South,

add()

void add (Component compobj, object
objid)

setLayout (new BorderLayout());


```

Ex: import java.applet.Applet;
import java.awt.*;
public class Border1 extends
Applet

```

```

{
public void init()
{

```

```

    Button b1, b2, b3;
    BorderLayout bord1;
    bord1 = new BorderLayout();
    setLayout (bord1);
    b1 = new Button ("Button 1");
    b2 = new Button (" 2");
    b3 = new Button (" 3");

```

```

    add ("North", b1);
    add ("South", b2);
    add ("East", b3);
}
}

```


Card Layout

- different from other layouts. Unlike the other three layouts, when you add component to a Card layout, they are not all displayed on the screen.
- Card layout are used to produce slide shows of components, one at a time.
- only one component is visible at a time.

Constructors

CardLayout()

CardLayout(int horiz, int vert)

```

Ex 6 void import java.applet.Applet;
import java.awt.*;
public class Card1 extends Applet
{
    public void init()
    {
        Button b1, b2, b3;
        CardLayout c1;
c1 = new
        panel P1 = new Panel();
        add(P1);
    }
}
  
```



```

C1 = new CardLayout();
P1. setLayout(C1);
b1 = new JButton("Button 1");
b2 = " " " (" " " 2");
b3 = " " " (" " " 3");

P1. add("Button 1", b1);
P2. add(" " " 2", b2);
P3. add(" " " 3", b3);
}
}

```

Grid Bag Layout

- most flexible & complex.
- ~~for~~ Can resize the components by assigning weights to individual components in the grid bag layout.
- When we specify the size & position of components, you also need to specify the constraints for each component.

To specify the constraints,
we need to ~~specify~~ variables.
set the
setConstraints() method.

→ Has single constructor,
GridBagConstraints con = new

GridBagConstraints
();

* Specifying Constraints :

• anchor

GridBagConstraints.BEAST

- 4 • NORTH
- 4 • WEST
- 4 • SOUTH
- 4 • NORTH EAST
- 4 • NORTH WEST
- 4 • SOUTH EAST
- 4 • SOUTH WEST
- 4 • CENTRE

• Fill

- NONE
- HORIZONTAL
- VERTICAL
- BOTH

Example 6

Emp

Emp

```
public class gb1 extends App
{
    public void init()
    {
        Button b1, b2, b3, b4, b5, b6;
        GridLayout g1;
        g1 = new GridLayout();
        GridLayout constraints gbc;
        setLayout(g1);
```

```
gbc = new GridLayout();
```

```
b1 = new Button("Button 1");
```

```
b2 = " " (a 1 2)
```

```
b3 = " " (
```

```
b4 = " " (
```

```
b5 = " " (
```

```
b6 = " " (
```

```
gbc.fill = GridLayout.BOTH;
```

```
gbc.anchor = "CENTER";
```

```
gbc.gridwidth = 1;
```

```
gbc.weightx = 1.0;
```

```
g1.setConstraints(b1, gbc);
```


Insets If we want to leave the space between the container that holds Components ~~and~~ and the windows that contains it. To do this, override the `getInsets()` method.

Insets (int top, int left, int bottom, int right)

specify amount of space between the container & enclosing window

Ex 6

Imp
Imp

Public class InsetsDemo extends Applet

{
 in rect insets
}

}

Public Insets getInsets()

{
 return new Insets(10, 20, 10, 10);
}

Event Handling

→ An event occurs when something changes within a graphical user interface.

→ JavaEvents are the part of the Java AWT package.
An event is the way the AWT communicates to you.

Process Involved In Event Handling Includes :

1. Identifying where an event should be forwarded
2. forwarded the event
3. receive the forwarded event
4. Take some appropriate action in response.
5. Event Handler may ultimately forwarded the event to an Event Consumer.

Components of An Event :

Event Object

When the user interacts with the application by pressing a key on a clicking a mouse button, an event is generated.

The operating system traps this event & data associated with it.

This data is then passed onto the application to which the event belongs.

In Java, Events are represented

by objects that describes the events themselves.

Event Sources

→ An Event source is an object that generates an event.

Event Handler (Event Listener)

→ Method that understands the event & processes it.

→ The Event Handler method takes an event object as a parameter.

→ Listener is an object which is notified when an event occurs.

The Event Model (Delegation Event Model)

In this model, ^{we can} specify the objects ~~who~~ that are to notify when specific event occurs.

→ Advantage of this design is that the application logic that processes event is clearly separated from the user interface logic that generates those events.

This model works in following manner:

1. Source generates an event and sends it to one or more listeners.
2. The listener simply waits until it receives an event. Once it receives an event, it processes event and returns back.
3. Listener must register with a source, in order to receive event notifications.

Date :

Page No.

Event classes

Action Event

Mouse Event

Mouse Wheel Event

Key Event

Item Event

Text Event

Adjustment Event

Window Event

Component Event

Container Event

Focus Event

Listener

Interface

Action Listener

Mouse Listener
and Mouse
motion Listener

Mouse Wheel Listener

Key Listener

Item Listener

Text Listener

Adjustment
Listener

Window
Listener

Component
Listener

Container
Listener

Focus
Listener

Swing → used to create windows
based
appl's

→ build at the top of awt.
↳ ~~java.awt~~ ↳ javax.swing

Ex 6 for Event Handling

```
import java.awt.*;  
import java.awt.event.*;  
class AEvent extends Frame implements  
    ActionListener
```

```
{  
    TextField tf;  
    AEvent()  
    {
```

```
        tf = new TextField();  
        tf.setBounds(60, 50, 170, 20);
```

```
        Button b = new Button("click me");  
        b.setBounds(100, 120, 80, 30);  
        b.addActionListener(this);
```

```
        add(b);  
        add(tf);  
        setSize(300, 300);  
        setLayout(null);  
        setVisible(true);
```

```
    }  
    public void actionPerformed(ActionEvent  
                                e)  
    {  
        tf.setText("welcome");  
    }  
}
```



```

public static void main (String args[])
{
    new ABvent();
}
}

```

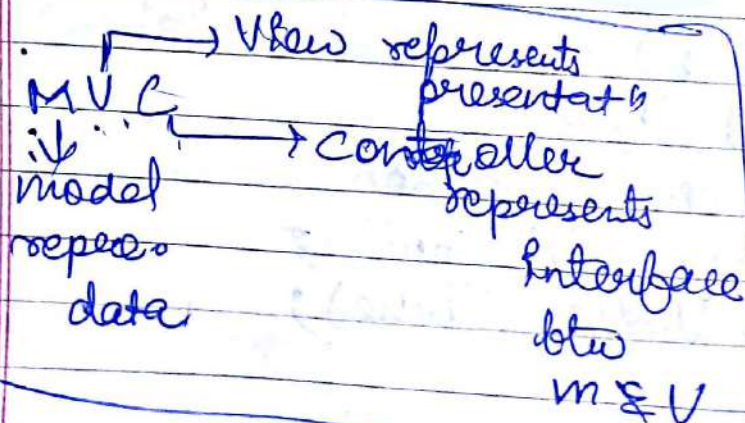
⑧ Difference btw AWT and Swing

AWT

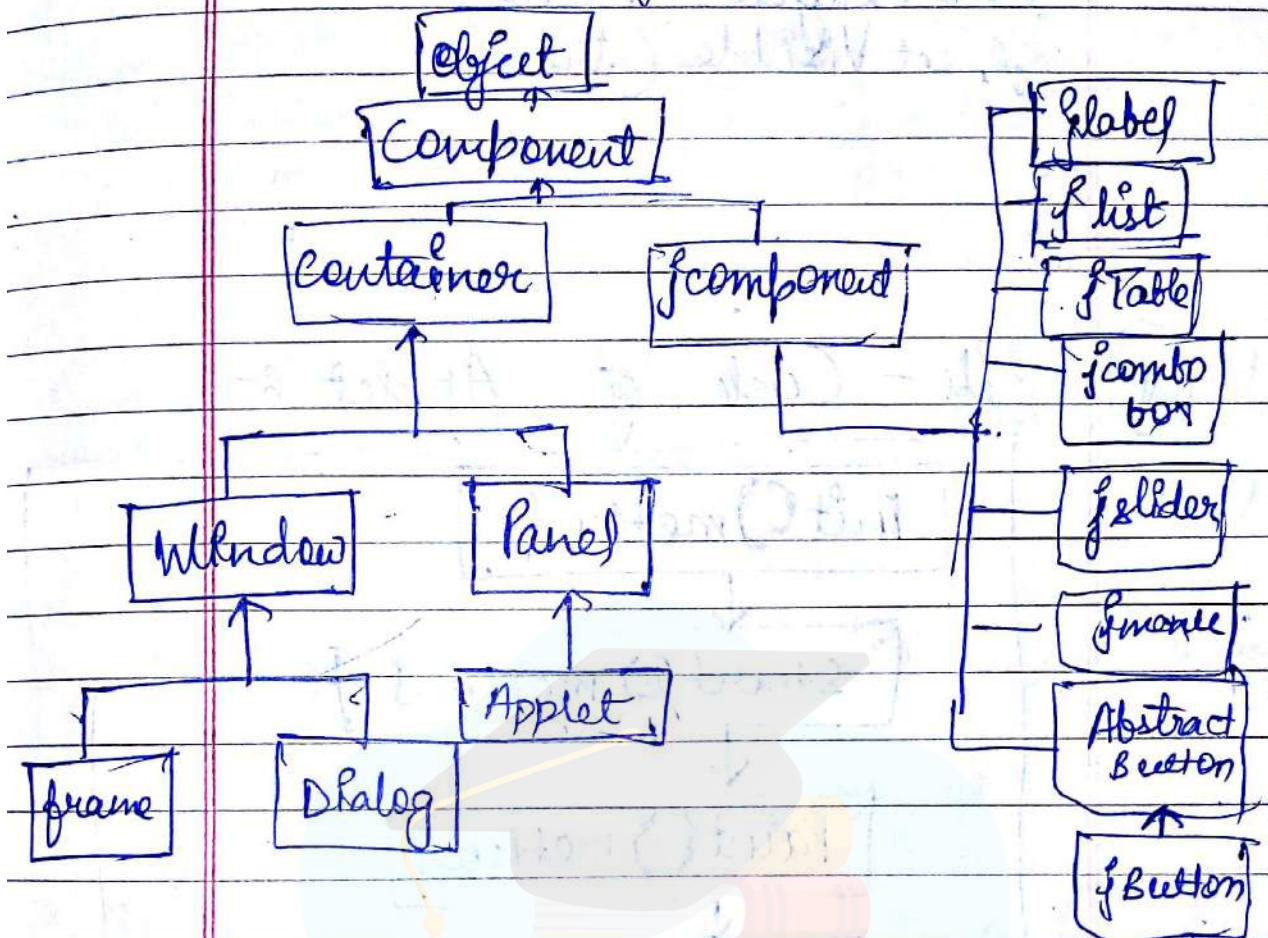
- platform dependent
- less Component
- heavyweight

Swing → Frame

- platform Independent
- more component
- lightweight



Q) ~~Q~~ Hierarchy of Java Swing



Eg:-

```

import javax.swing.*;
public class FirstSwingExample
{
    public static void main(String[] args)
    {

```

```

        JFrame f = new JFrame();

```

```

        JButton b = new JButton("click");
        b.setBounds(130, 100, 100, 40);

```

```

        f.add(b);
    }
}

```


Event Handling Ex

```
import javax.swing.*;
import java.awt.event.*;
```

```
class AEvent extends JFrame implements  
ActionListener {
```

```
    TextField tf;
    AEvent() {
    }
```

```
    tf = new TextField();
    tf.setBounds(60, 120, 80, 30);
```

```
    Button b = new Button("Click me");
    b.setBounds(120, 180, 60, 30);
    b.addActionListener(this);
    add(b);
    add(tf);
```

```
    setSize(300, 300);
    setLayout(null);
    setVisible(true);
```

```
}
```

```
public void actionPerformed(ActionEvent e)
```

```
{
    tf.setText("welcome");
}
```



```

4
public static void main (String args[])

```

```

{
    new AEvent();
}
}

```

By Outer class

```

import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame
    Implements ActionListener
    Listener
{

```

```

    TextField tf;

```

```

    AEvent2 ();

```

```

{

```

```

    tf = new TextField();

```

```

    tf.setBounds (180, 200, 60, 30);

```

```

    Button b = new Button ("click me");

```

```

    b.setBounds (80, 180, 20, 30);

```

```

    Outlet o = new Outlet (b);

```

```

    b.addActionListener (o);

```

```

    setSize (add (b));
    add (tf);

```



```
setSize (800, 300);  
setVisible (true);  
setLayout (null);  
}
```

```
public void static main (String args[])  
{  
    new ABEvent2 ();  
}
```

```
import javax.swing.event.*;  
class Outer implements ActionListener {  
    ABEvent2 obj;  
    Outer (ABEvent2 obj) {  
        this.obj = obj;  
    }  
}
```


Unit - IV

Binary I/O is more efficient than text I/O, because binary I/O does not require encoding and decoding.

Binary files are free from encoding and ~~the~~ on host machine ~~is~~ they are portable.

Java I/O \rightarrow is used to process the input and produce the o/p based on the input.

File

Data can be read ^{from} or stored in file. Data stored in file is often called persistent data.

\rightarrow File is collection of related records placed in a particular area in the disk.

\rightarrow File is a class that directly deals with file system. It does not specify how data is retrieved from or stored in file. It just deals with properties of file.

Its object is used to obtain ~~data~~ information ~~for~~ associated with file such as, date, time, directory path.

Directory is a file that contains list of other files and directories.

we call `list()` method to ~~call~~ extract the list of other files and directories.

file methods :-

`exists()`

`isdirectory()`

`isfile()`

`can read()`

`can write()`

Streams

→ Stream is an abstract demonstration of input and output devices.

→ Stream is an interface between I/O devices and users.

→ Stream helps in smooth transfer of data to the desired place.

→ Data stored in binary file are represented in the binary form.

~~Ex~~ Java ~~classes~~ source programs are stored in text editor and Java ~~classes~~ ^{text files can be read} are stored in binary file ^{is can be read by} JVM.

The Java Input/output (I/O) is a part of java.io package.

The java.io package contains a relatively large number of classes that support input and output operations.

Java defines 2 types of streams
Byte stream and character stream.

Standard Stream →

→ They read Input from keyboard and write output to the display.

Java supports three standard streams :

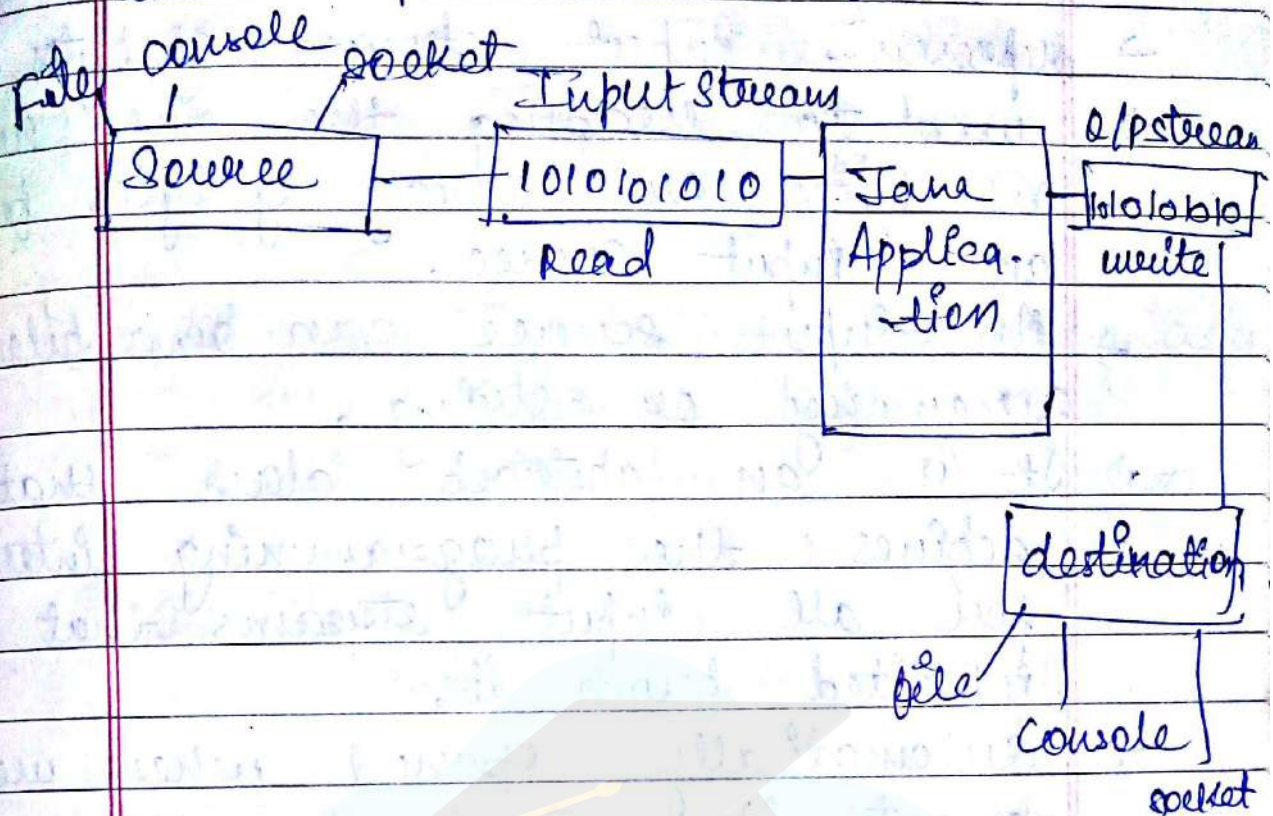
System.in → used to read i/p from keyboard
 System.out → used to write o/p to display
 System.err → used to write error output to be display.

System.in → Standard Input stream
 System.out → Standard output stream
 System.err → Standard error stream.
 → both are used to write output;
 → System.in → byte stream that has no character-stream features.

to use standard input as a character stream :-

```
InputStreamReader inp = new InputStreamReader
                               (System.in);
```


Working of Java output stream and input stream



Streams are unidirectional in nature.

There are two types of stream for file handling in Java

- Byte stream
- Character stream.

Byte stream supports 8-bit input and output operations.

There are two classes of byte stream.

- Input stream
- Output stream.

Input stream &

- Represents input stream of bytes.
- used for reading the data such as bytes or array of bytes from an input source.
- An input source can be a file, a memory or a string.
- It is an abstract class that defines the programming interface for all input streams that are inherited from it.
- automatically opened when we create it.
- Explicitly closed by close() method and implicitly closed when the object is found as a garbage.

● Sub classes inherited from input stream class can be seen in hierarchy manner

1. ByteArrayInputStream
2. File Input stream
3. Object Input stream
4. PPe Input stream
5. StringBuffer Input streams
6. Filter Input stream

7. Data Input stream
8. Buffered Input stream
9. Line number Input stream.
10. PushBack Input stream.

Byte array Input stream → It contains the internal buffer and read data from the stream

~~File~~ File Input stream → contains input bytes from a file.

Object Input stream → used to recover objects to serialize previously.

Piped Input stream → Input pipe.

String Buffer → used to buffer the string that comes through input device.

Filter Input stream → Implements Input streams

Buffered Input stream → used to create internal buffer array

Data Input Stream → This class reads the primitive data-type from Input stream.

LineNumber Input Stream → Counts the lines.

PushBack Input Stream → allows characters to be returned to Input stream.

Output Stream

This class is sibling to the Input stream class used to write byte and array of bytes to the output source.

→ O/P Source can be - a file, a string, or a memory.

- opened automatically

- closed either explicitly by call() method or implicitly when object is garbage collected.

→ Subclasses / classes inherited from an output stream in hierarchy are :

1. ByteArray Output stream

- data is written into byte array.

2. File Output stream - data is written to a file.
3. Object output stream - write the object to read by object input stream.
4. Piped Output Stream - Output pipe.
5. String Buffer Output stream - used to buffer string.
6. Filter Output stream - implements OutputStream.
7. Buffered output stream - writes 1 byte to output stream.
8. Data Output stream - writes the primitive data type.
9. Print stream - Output stream that contains print() and println() methods.

Character Streams

- Supports 16-bit unicode character input and output,
- There are two classes of character stream ~~to be~~
 - Reader
 - Writer

Reader

1. Buffered Reader
2. Line number Reader
3. CharArray Reader
4. Filter reader
5. PipedReader
6. String reader
7. Push back reader
8. File reader
9. InputStream reader

File InputStream / FileOutputStream

- File Input stream and File Output stream are used to read or write data in file.
- ~~The~~ All the methods in these classes are inherited from Input stream & Output stream. They do not introduce new methods.

→ They are used for file handling in Java.

Java FileOutputStream class

Almost all the methods in the I/O classes throw `java.io.IOException`.

therefore we have to place the code in a try-catch block as;

Syntax

```
public static void main (String[] args)
{
    try {
        // perform IO exception.
    }
    catch (Exception e)
    {
        System.out.println (e);
    }
}
```



Example of Java FileOutputStream class:-

```
// Import java.io.*;
class Test {
    public static void main (String args[]) {
        try {
            FileOutputStream fout = new FileOutputStream("abc.txt");
            String s = "Sachin is the best";
            byte b [] = s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("Success");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```


Jana File InputStream class

- obtains input bytes from a file
- used for reading streams of byte.

Example e

```

import java.io.*;
class SimpleRead {
    public static void main(String args[])
    {
        try {
            FileInputStream fin = new FileInputStream
                                   (\"file.txt\");

            int i = 0;
            while ((i = fin.read()) != -1) {
                System.out.println((char)i);
            }
            fin.close();
        }
        catch (e) { System.out.print(e); }
    }
}

```


Example 6

Reading the data of one Java file
and writing it to another file.

```
import java.io.*;
class C {
```

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

```
try {
```

```
FileInputStream fin = new FileInputStream  
("C:\a.txt")
```

```
FileOutputStream fout = new FileOutputStream  
("M.java")
```

```
int i = 0;
```

```
while ((i = fin.read()) != -1)
```

```
{ System.out.print((char) i);
```

```
}
```

```
fout
```

```
fout.write((byte) i);
```

```
;
```

```
fin.close();
```

```
}
```

```
}
```


Buffered streams

Buffered Input streams reads ~~area~~ data from a memory area called as buffer.

Similarly, Buffered Output streams ~~writes reads~~ data ~~from~~ to buffer.

There are four buffered stream classes :-

1. BufferedInputStream
2. BufferedOutputStream
3. BufferedReader
4. BufferedWriter

BufferedInputStream

→ Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

⇒ Construction of buffered Input Stream.

```
public BufferedInputStream(Input
                             stream
                             int
                             int size)
```


Parameters :

in → the underlying Input Stream
 size → the buffer size.

Example of Java BufferedInputStream

```
import java.io.*;
class SimpleRead {
    public static void main (String args[])
    {
        try {
```

```
        FileInputStream fin = new FileInputStream
            ("f.txt");
        BufferedInputStream bin = new BufferedInput
            Stream
            (fin);
```

```
        int c;
        while ((c = bin.read()) != -1)
```

```
        {
            System.out.println ("char " + c);
        }
```

```
        bin.close();
```

```
        fin.close();
```

```
    } catch (Exception e)
```

```
    {
        System.out.println (e);
    }
```


Buffered Output Stream

Ex 6 Import Java. Io.*;
Class Test {

public static void main (String args[]) {

```
FileOutputStream fout = new FileOutputStream("f1.txt");  
BufferedOutputStream bout = new  
    BufferedOutputStream(fout);
```

```
String s = "Lachin is the best";  
byte b[] = s.getBytes();  
bout.write(b);
```

```
bout.flush();  
bout bout.close();  
fout.close();  
System.out.println("success");  
}
```


JDBC

Introduction to JDBC

JDBC stands for database connectivity. JDBC is a set Java API's (Application programming interface) used for executing SQL statements.

- API contains a set of classes and interfaces to enable the programmer to develop pure Java database applications.
- Allows developers to write real client-server projects in Java.
- JDBC API defines how any connection application opens a connection, communicates with database, executes SQL statements & retrieve query results.

Role of JDBC

Java Application

JDBC Driver manager

JDBC/native
bridge

JDBC/ODBC
bridge

JDBC driver
(DBMS
specific)

JDBC
middle-
ware
(Vendor
base)

native
driver

ODBC
driver

DBMS

→ JDBC was designed to be very compact & simple interface focusing on the execution of raw SQL-statements & retrieving the results.

Characteristics of JDBC

- It is a call-level SQL interface in Java.
- JDBC mechanisms are simple to understand & use.
- It does not restrict the type of queries passed onto the underlying DBMS drivers.
- provides Java interface that stays Java consistent to the rest of the Java system.

Java and JDBC

- The combination of Java with JDBC is very useful for the programs to run developer his/her program on any platform.
- Java programs are secure, robust and Java is a good language to create database application.

Advantages of using Java with JDBC :

- Easy and Economical
- continued usage of already installed database.

- Development time is short
- Installation is simplified.

How JDBC Works :

→ JDBC defines set of objects & methods to interact with underlying database.

- A Java program first opens a connection, makes a statement object, passes the SQL statement using these statement object to the underlying DBMS and retrieves the result as well as information about result set.

There are two type of interfaces :

- low level interface. (not user friendly)
- high level interface. (user-friendly)

JDBC is a low level API interface.

JDBC v/s ODBC

~~The most used~~

→ used to access relational database.

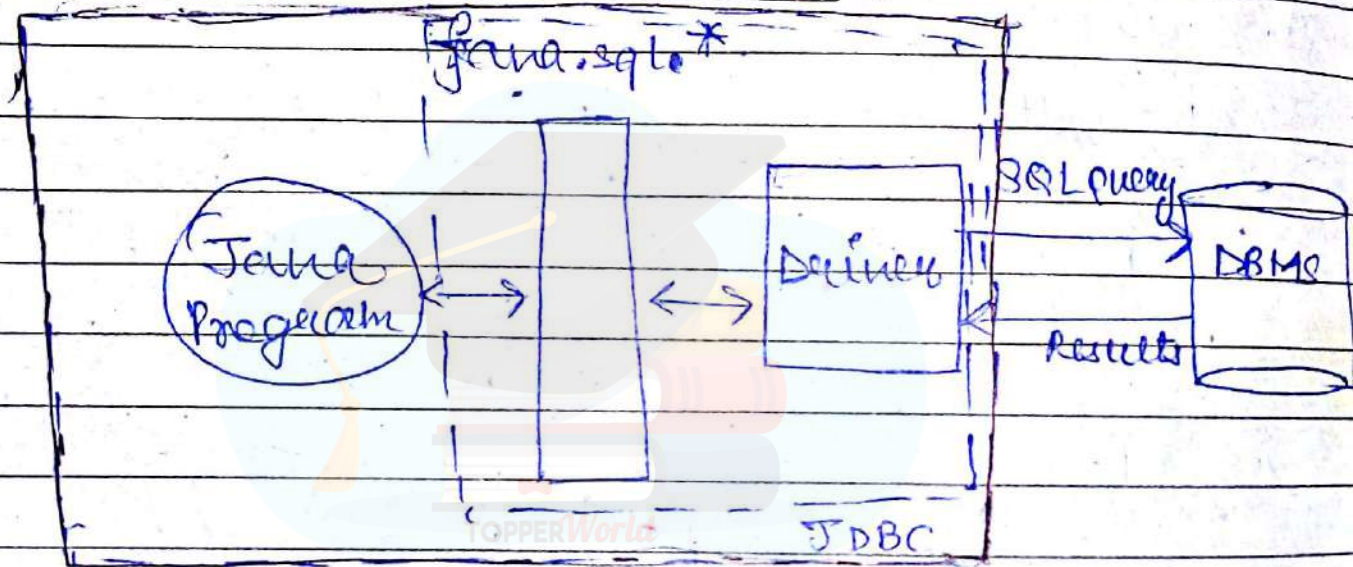
JDBC is preferred of the two :

1. ODBC cannot be directly used with Java because it uses a C interface. - calls from Java to native C code have a number of drawbacks in the security, implementation etc.
2. ODBC makes use of pointers which have been totally removed from Java.
3. ODBC mixes simple and advanced features together and has complex options for simple queries. But JDBC is designed to keep simple.
4. JDBC is to Java programs and ODBC is to programs written in other languages than Java.
5. ODBC is used between applications and JDBC is used by Java programmers to connect to database.

Details about JDBC

- JDBC API is in the package `java.sql`.
- 8 interfaces
- 6 classes
- 3 exceptions, in JDK 1.0.1.

JDBC driver model



JDBC Driver Types

There are 4 types of JDBC drivers.

- JDBC Type 1 Driver : They are JDBC-ODBC bridge drivers i.e. ~~these~~ Translate JDBC into ODBC ~~and~~ and they delegate the work of data access to ODBC API.

JDBC-native Bridge

- JDBC Type-2 Drivers They mainly use native API for data access i.e.; connects JDBC to data base vendors native SQL calls and provides ~~java wrapper classes~~

JDBC-net Bridge

- JDBC Type-3 Drivers Translates JDBC to a DBMS independent network protocol.

→ Uses vendor independent Net-protocol to access a vendor independent call listener. The listener then maps the vendor independent calls to the vendor dependent ones. This extra step adds complexity and decreases the data access efficiency.

Direct JDBC Drivers

- JDBC type-4 Drivers Most efficient among all driver types.

→ directly translates JDBC to native API used by RDBMS.

→ doesn't require anything to be installed on client's machine.

JDBC classes (6 classes)

Ex: ① DriverManager

→ ~~open~~ used to obtain connections to the database.

② Types

→ Defines constants which identify SQL types.

③ Date

→ used to map between java.util.Date to SQL DATE type

④ Time

→ used to map between java.util.

⑤

Time Stamp

to SQL ^{Time} TIME type

→ used to map between

java.util.Timestamp to

SQL TIMESTAMP type

JDBC Interfaces (8 Interfaces)

①. Driver

→ All JDBC drivers must implement the driver interface.

→ used to obtain the connections to the specific database type.

②. Connection

→ represents the connection to a specific database.

→ used for creating statements

→ used for calling stored procedures

→ used to create callable statements

③. Statement

→ used to execute SQL statements against the database.

④. ResultSet

→ represents the result of an SQL statement.

→ provides methods for navigating through the resulting data

⑤. prepared statement
→ similar to stored procedures
→ An SQL statement is compiled & stored in a database.

⑥. Callable statements
→ used for executing stored procedures.

⑦. Database metadata
→ provide access to database system catalogue

⑧. Resultset meta data
→ provide information about the data contained in the result set.

Advantage

- Better performance than all other drivers.
- NO software is required at client side or server side.

Basic steps to use a database in Java

1. Establish a connection.

- load the driver class with
~~load~~ ^{register} `forName()` method.

eg: `Class.forName("oracle.jdbc.driver.OracleDriver");`
→ class name

- ~~Establish~~ Establish connection with `getConnection()` method.

eg: `Connection con = DriverManager.getConnection(`
`"jdbc:oracle:thin:@localhost:1521:xe",`
`"system", "Password");`
↓ ↓
string name string password
→ string URL

②. Create the ~~object~~ statement object -

→ create statement () method is used,
→ responsible to execute queries with the database.

eg; Statement stmt = con.createStatement();

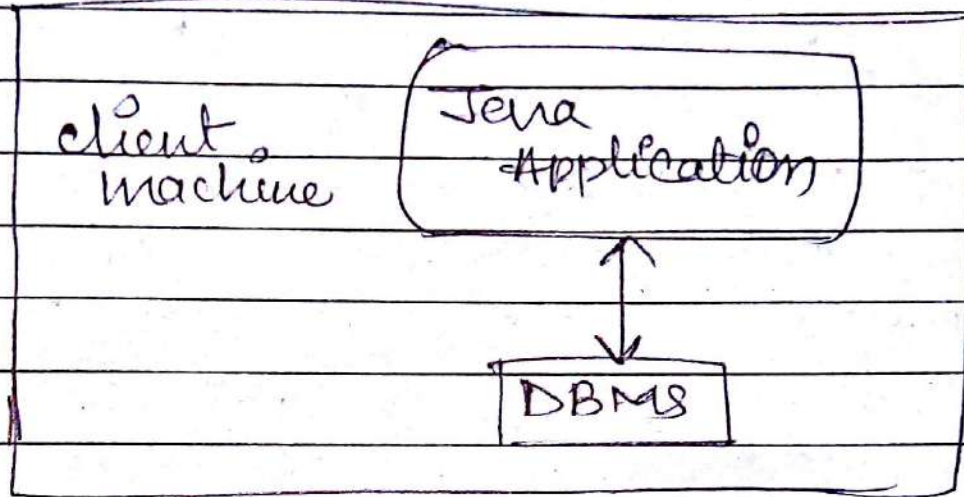
③. Execute the query & get result set
→ executeQuery () method is used.

eg;

ResultSet rs = stmt.executeQuery("select * from emp");

④. Close connection
stmt.close();
con.close();

11. Two-tier Architecture for Data Access



→ In a two-tier model, a Java application talks directly ~~with~~ to the data source.

→ This requires a JDBC driver that can communicate with the particular data source being accessed.

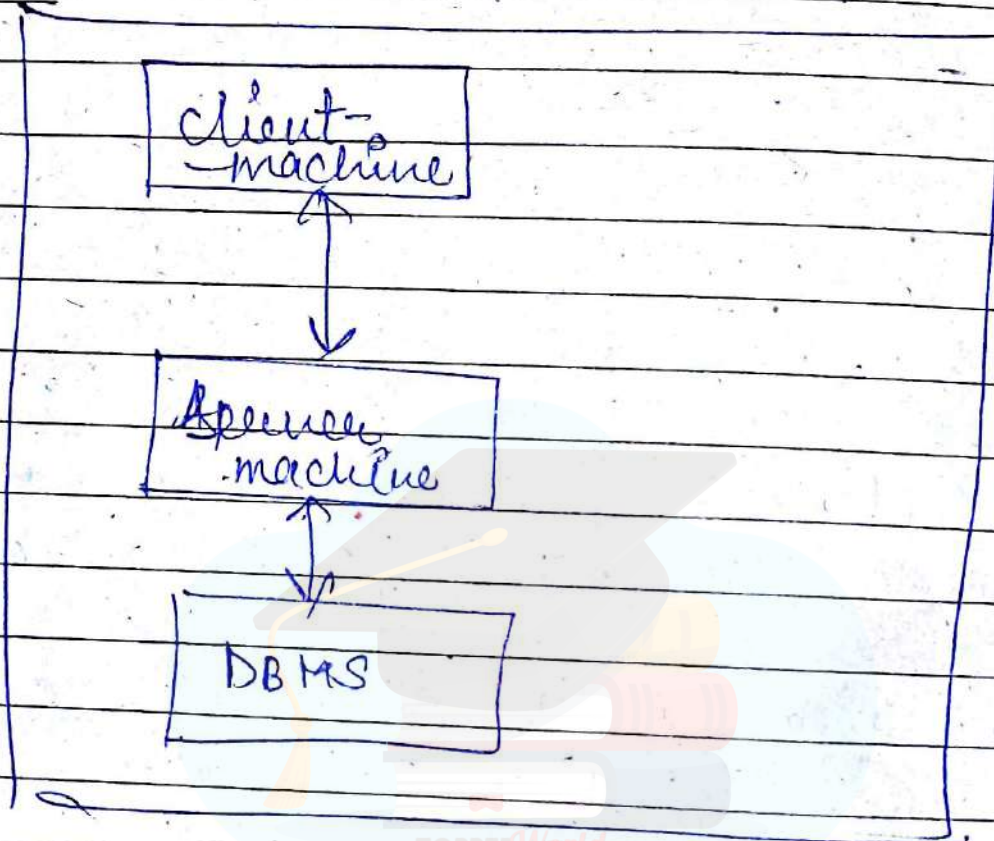
→ User's commands are sent to the database & then those results are sent ^{back} to the user.

→ The data source may be located ~~at~~ on another machine to which the user is connected via a network.

This is referred to as a client/server configuration, with the user's machine as a client & the machine

the data source is server.

Three-tier architecture for Data Access.



→ In three-tier model, Commands are send to the middle-tier which then sends the commands to data source. The data source processes the command, these commands are then sent to the middle-tier & then it sends them to the User.

→ This model is very attractive because the middle tier makes it possible to y control ~~source~~ access maintain

→ provides performance advantages.

Database Connectivity with ~~Oracle Access~~ Oracle

For connecting Java application with the Oracle database, you need to follow the steps for database connectivity;

①. Driver class:

→ 'Oracle.jdbc.Driver, OracleDriver;

②. Connection URL:

~~jdbc:thin:~~

jdbc:oracle:thin@localhost:1521:xe

where,

jdbc → API

Oracle → database

thin → driver

localhost → servername

1521 → Port number

xe → Oracle service name

③. Username & System

④. password → given by user at the time of installation,

Example 6

```
import java.sql.*;
class OracleCon {
```

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

```
            Class.forName("oracle.jdbc.OracleDriver");
            Connection con = DriverManager.getConnection(
                "jdbc:oracle:thin@localhost:1521:xe",
                "system", "password");
```

```
            Statement stmt = con.createStatement();
```

```
            ResultSet res = stmt.executeQuery("select * from emp");
```

```
            while (res.next())
                System.out.println(res.getInt(1) + " " +
                                    res.getString(2) + " " +
                                    res.getString(3));
```

```
            stmt.close();
```

```
            con.close();
```

```
        } catch (Exception e) { System.out.println(e); }
    }
}
```


Database Connectivity with MySQL

→ Some steps are to be performed followed to perform MySQL connectivity.

1. Driver class :- The driver class for MySQL database is `com.mysql.jdbc.Driver`.

2. Connection URL :-

`jdbc:mysql://localhost:3306/sbny`

where,

`jdbc` → API

`mysql` → database

`localhost` → username on which MySQL is running.

`3306` → port number.

`sbny` → database name.

3. Username → default user name for MySQL is "root".

4. password → given by the user at the time of installation.

5) Create a table in mysql database, firstly create database

```
Create database sunny;
```

```
Use sunny;
```

```
Create table emp (id int(10), name varchar  
(15), age int(3));
```

Example 6

```
import java.sql.*;  
class MySQLCon {  
    public static void main (String args[])  
    {  
        Class.forName("com.mysql.jdbc.Driver");  
        Connection con = DriverManager.getConnection  
        ("jdbc:mysql://localhost:3306/sunny",  
         "root", "root");  
        Statement stmt = con.createStatement();
```


ResultSet rs = stmt.executeQuery("select * from emp")

```
while (rs.next()) {
    System.out.println(rs.getInt(1) + " " +
        rs.getString(2) + " " +
        rs.getString(3));
}
```

stmt.close();

con.close();

} catch (Exception e)

{ System.out.println(e);

}

}

Database Connectivity with Access

There are two ways to connect Java application with access database

- ① without DSN
- ② with DSN.

① without DSN.

→ We have created login table in the access database.

Ex 6

```

import java.sql.*;
class Test {
    public static void main(String args[])
    {
        try {
            String database = "student.mdb";
            String url = "jdbc:odbc:Driver={Microsoft
                Access Driver (*.mdb)};
                DBO = " + database + "; DriverID=22; READING
                = "Text";
        }
    }
}
    
```



```

class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection
    (URL);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("select * from
    emp");

```

```

while(rs.next())
{
    System.out.println("rs.getInt(1)");
}
catch (Exception e)
{
    System.out.println(e);
}
}
}
}

```

② with DSN

→ assume your dsn name is mydsn.

```

import java.sql.*;
class Test {
    public static void main(String args[])
    {
        try
        {

```


string url = "jdbc:odbc:mydsn";

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Connection con = DriverManager.

getConnection(url);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("select
*
person login");

while (rs.next())

System.out.println(rs.getString(1));

}

catch (Exception e)

System.out.println(e);

}

}

}

Networking

→ The term network programming refers to writing programs that execute across multiple computers (systems), in which all devices are connected to each other using a network.

The java.net package provides support for two common network protocols

①. TCP : TCP stands for Transmission Control protocol, which allows for reliable communication between two applications.

TCP is typically used over the Internet protocol, which is referred to as TCP/IP.

②. UDP : stands for User datagram protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket Programming

- Java socket programming is used for communication between the applications running on different JRE.
- Java socket programming can be connection-oriented or connection-less.
- Socket classes are used to represent the connection between client program and a server program.

Socket and serverSocket ⇒ classes are used for connection-oriented socket programming.

DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

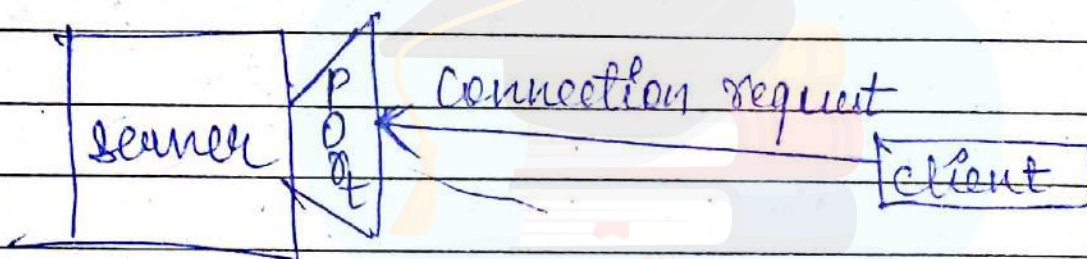
Socket → at client side of the connection
 ServerSocket → at server side of the connection.

Socket is simply an end-point for communications between the machines.

Socket Communication :

A server runs on a specific computer and has a socket that is bound to a specific port.

The server waits and listens to the socket for a client to make a connection request.



If everything goes well, the server accepts the connection.

Upon acceptance, the server gets a new socket bound to a different port with different port number.

So that it can be continue to listen to the original socket for connection requests while serving the connected client.



Important methods

① socket class

public InputStream getInputStream()
→ returns the InputStream attached with this socket.

public OutputStream getOutputStream()
→ returns the OutputStream attached with the socket.

public synchronized void close()
→ closes this socket.

② ServerSocket class

public Socket accept()
→ establishes connection between client & server.

Ex. of Java socket programming Read from both side

```
import java.net.*;  
import java.io.*;  
class MyServer {
```

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

```
ServerSocket ss = ss.accept() new ServerSocket  
(3333);
```

```
Socket s = ss.accept();
```

```
DataInputStream dis = new DataInputStream  
(s.getInputStream());
```

```
DataOutputStream dos = new DataOutputStream  
(s.getOutputStream());
```

```
BufferedReader br = new BufferedReader  
(new InputStreamReader  
(System.in));
```