Chapter-1

Java

By → (Herbert Schildt)
9th edition

Java is a platform independent / cross platform language. (program will work, on all operating systems).

Program made on one computer will also run on another (with / without same O.S)

→ Developed in 1991 by Sun microsystem.
→ was originally named OAK, in 1995 → Java.

Security feature → Applets run in the background. But applets can't access files in the computer.

Multithreading → Running a program multiple times simultaneously when another program is running.

⇒ Download jdk (java development kit) & Notepad / Notepad++

* characteristics of Java →
→ object oriented
→ Portable (can run on diff. O.S)
→ Distributive
→ Secure (no pointers, no dynamic allocation / deallocation, no memory problems).
→ Multithreaded language

Java file

→ extension → .java

Architecture of Java

filename.java (saved file / source file) —compile (Jdk)→ filename.class (Bytecode) (Intermediate file) —JVM (interpreter)→ O/P

(library)

{ here we get a bytecode (in 120 form) Now it can run on any O.S }

JVM : (Java virtual Machine)

execution : interpreting what to do from the bytecode

- In b/w we get library files.
- → It is platform dependent [logically all are same but file str. is diff.]

⇒ 'run one, run anywhere, anytime'.

→ **Advantages of Java :-**

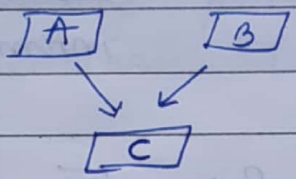
- real world programming language [anything that has some attributes and methods can be implemented using OOP.]
- reusability of code [also available in C++].
- resilience to change (as compared to C++, can change things easily)
- Information hiding [everything is in classes) [data hiding feature just like C++].

ch-2 * **Object Oriented Programming**

→ Abstraction
→ Encapsulation
→ Polymorphism
→ Inheritance.

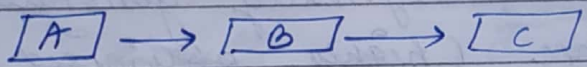
⇒ Bytecode → cross platform
↳ highly optimized code.

Multiple Inheritance implementation directly isn't allowed in Java. But with some mechanism (interfaces), we can.



X (not possible)

Multi-level Inheritance is allowed.



⇒ JVM interpreter → line by line
⇒ compiler → finally just once.

In Java, both take equal time as the intermediate class file is highly optimised.

⇒ Components of JVM

Class loader → loads all header files and the classes.

execution egine : executes the classes that have been loaded.

Just-in-time (JIT) → already has the first 2 lines of all classes. eg → class A, B, & C. Then when class A is to be executed, till the execution engine takes other lines & executes them, JIT has already executed first 2 lines)

On the fly execution

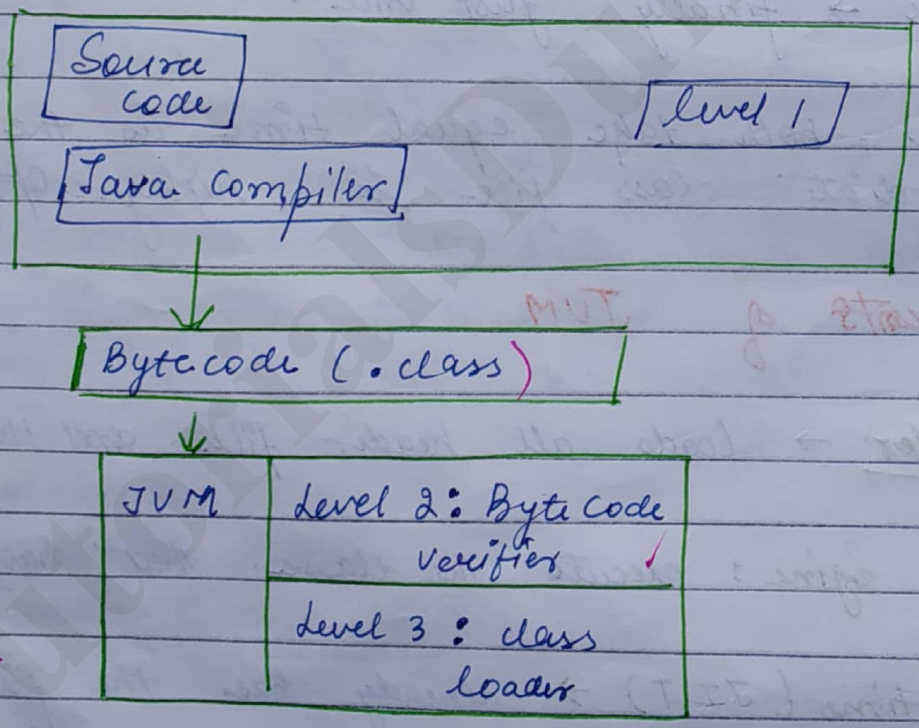
Saves execution time. { 2 lines is an example, it contains some part of class }

⇒ Java Architecture Security {3 level Security}

(not in course)
1. compile level security
2. Bytecode verifier.
3. class loader.
4. Sandbox Model.

{ Most secured language, 2nd used in internet (www) }

(1) After compiling, it checks for extra references (like empty memory spaces etc.)
(2) Byte code that is orginally generated, should not change (highly optimised).
(3) Before loading the files, it checks that no extra file should be loaded.



chapter-3

Data Types available in Java →

1) Primitive data types {already available}
2) Derived data types / Abstract data types {that can be derived from primitive DT} (String, class, enum structure)

Java is a Case Sensitive language.

⇒ Primitive datatypes →

→ Integers, (int, short, byte & long)
→ character (char)
→ floating point (float & double)
→ Boolean (1 & 0) (bool)

⇒ Sizes →

byte - 2 bytes
short - 2 bytes
int - 4 bytes
long - 8 bytes

char - 2 bytes
float - 4 bytes
double - 8 bytes
bool - 1 bit

⇒ First program & Syntax

```
class ABC
{
Public static void main (String args[])
  {
System.out.println ("Hello");
  }
}
```

Output → Hello

If we pass arguements using command line, it passes to this array

⇒ System is a class defined in Java, which has Object out & println is a method of the class

public → globally accessible
static → no need to instantiate

⇒ Class ABC
{
public static void main (String args[])
{
System.out.print ("Hello");
System.out.print ("World");
}
}

Output → Helloworld.

⇒ Class ABC
{
public static void main (String args[])
{ int a = 10;
System.out.print ("a =" + a);
}
}

used for string concatenation.

⇒ When we save the file, it has to have same name as class.

ABC.java

As, when it goes to JVM, it finds main function & if it is ABC, (runtime/execution time) then it will say, main function not found !!!

⇒ To check for Java Compilation.

command prompt

c:\> javac (if present various options occur)

If not present, path can be set in 2 ways →

① Temporary Setting of path.

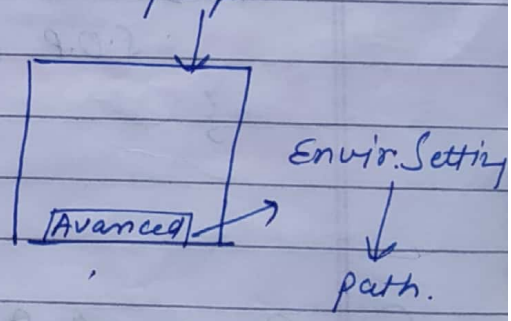
c:\> setpath = "C:\prog files\java\jdk\bin"

c:\> javac

② Permanent Setting of path.

My computer → Right click → Properties

- c:\cd _(path.)
- c:\javac _filename.java
- c:\java _filename



⇒ Types of Variables

declaration

datatype name;

int i = 10;

(Initialization).

to create reference

① class Variable

Accessible within a class & by its objects.

The class variables are declared inside the class before their use.

② Instance Variables. They are declared inside a class & are created when the class is instantiated [when ob. is created]. Objects give diff. values to instance variables as per the object definition.

3. Local Variables declared inside a method and their scope is limited within the block of a code.

```
class xyz
  {
    PSVM (String args[])
     {
  int i=10;
      if (i>5)
  {   int x=10;
      SOP (x);
       S.O.P. (i);
  {
    S.O.P (x);   // invalid as x is only valid inside if block & its scope is not defined outside the block.
  }
```

④ Static Variables

They are allocated memory only once but are globally accessible to all instances of a class i.e. when an instance of class is destroyed, the static Variable isn't destroyed & is available to all the other instances of the class.

→ Keywords in Java

abstract	instance	super	continue	private
boolean	if	assert	default	long
break	invoked	catch	goto	synchronised
byte	package	class	implements	const
case	switch	char	interface	else

Dynamic initialization
initializing at run time using new keyword.

ClassDream
Date
Page

double	int	this	while	do	next.
abs	long	enum	return	static	
final	new	extends	for	transient	
float	protected	threw	throws	try	
finally	short	volatile	void	import	

⇒ Type Casting (mowding)

(Automatic / implicit) (done by compiler itself)

(higher value to lower value)

```
int a;
byte b;
a = b;
```

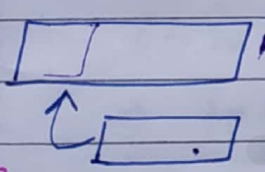
conversion of ~~cargo~~ smaller D.T to larger d.T

Explicit

done by user (convert of larger Data Type to smaller D.T)

```
b = (byte) a;
```

error → (loss of Precision)
∴ Explicit.



Chapter-4

⇒ Operators

- Bitwise operators
- Exclusive OR (XOR) → ^
- shr (>>)
- shl (<<)
- Bitwise 2 assignment (&=)

0 1 1 0
0 0 1 1
0 0 0 1

⇒ Bitwise right shift / Divide by 2 opdator (>>)

Right shift

10 → 10 10

1 0 1 0

0 1 1 0 (1st)

0 1 1 0 (n>>2)

shift no. right 2 times

Ans → 0 0 1 0 (2)

Bitwise shl (No. increases)

1000 (8)

(64) 10000 (swift all the bits and fill vacant space with 0)

⇒ Relation op →

⇒ Bitwise and (&)

⇒ Logical op :-

⇒ Control statements →

1) if else (nested)
2) switch
3) nested switch

⇒ Interaction statements →

1) while
2) for
3) do while
4) for each for loop.

NOTE → public static void main (String args [ ])

→ public keyword is an access modifier, which allows programmer to control the visibility of class members. It may be accessed by code outside the class in which it is declared.

globally accessible

→ Keyword static allows main ( ) to be called without having to instantiate a particular instance of the class. This is necessary since main ( ) is called by the JVM before any obj. is made.

→ String args[] declares a parameter named args, which is array of instances of the class String. args receive any command line arguements present when prog. is executed.

type: String

→ System.out.println(" ");

class: System — object: out — function: println

build in funct println displays the String which is passed to it.

System is a predefined class that provide access to the system, and out is the output stream that is connected to console.

⇒ Control Statements

⇒ Bitwise shift left

We are retaining (MSB)

i = 10;

⇒ 1010

(shift by 1) 10100 = 20 ← empty space

2 | 10	
2 | 5	0
2 | 2	1
	1 0

10 × 2 = 20

⇒ Right shift

We will discard LSB

1010 ⇒ 10100

100 → 0100

8 4 2 1

1 0 1 0

1 0 1 0 0 = 20

0000 0100

9-01-17

Classdream
Date
Page

⇒ Left shift (x 2)

(2) << 8

no. of times

1 0 0 0

1 0 0 0 0 0 — vacant

chapter-5

⇒ Control Statements

→ IF
→ if else, nested else if
→ switch case
→ nested switch

| while, do while

⇒ if (condition)
{
≡
}

eg → int i = 10;
if (i ≤ 10).
{ S.O.P ("Hello");
S.O.P ("World");
};

output → HelloWorld
if without braces then,
Hello World.

// independent of if condn.

Classdream
Date
Page

⇒ Nested if

```
if (condition)
{
  if (condition)
  {

  }
}
```

⇒ if else (nested)

```
if (condition)
{
  if (condition)
  {
    else ( )
  }

else ( )
{
```

⇒ if else if and switch case.

if
else if
else if, else

If we use break statement in switch case then, execution time to be taken is less than else if.

```
switch ( )
{
  case 0
  case 1
}
```

⇒ WAP to ~~find~~ check which month belongs to which season (April.)

int/char)

```
Switch (~~~~)
{
case 1:
  if(--11--11--11--)
  S.O.P. ("Spring")
~~break;~~
case 2:

break;
case 3:

~~break;~~
case 4:

break;
}
```

⇒ Nested Switch

1. Sum
2. Division

1.1.
1.2
1.3

```
Switch ( )
{
case 0
case 1
  → nested switch
```

Array → collection of similar datatypes or same values

→ collection of variables with similar datatype

→ For( ) Know how many times loop is to be run.

→ While( ) → only condⁿ is known.

→ do( )

2. ⇒ { while

⇒ when we know the no. of times, the condition is to be execute is, then, we use for() else while.

→ (For) each version (modified version of for loop) [for only array]

int a [size];

Syntax →

⇒ for (int i : a) || syntax

same as (x=0; x<5; x++)

where a is the name of the array & i is the name

[For every value of a, there is some each i.]

no need to specify upper & lower bound.

Q → Print →

(initial & last index is unknown)

Nested for

for (int i=5; i ≥ 1; i--)

{

for (int j = i; j ≥ 1; j--)

{

SOP.println("*")

}

}

Int [10];

Classdream
Date
Page

using,

For each Version

```
char a[5] = {'*','*','*','*','*'};
for (int i : a)
{
S.O.P(i);
}
```

(prog. of deletion), then write S.O.P block again.

⇒ For each (Only for array)

Syntax

```
for (index : array)
{
S.O.P(" ");
}
```

⇒ sum of elements in array.

Q→ ⇒ Print sum of array using for each.

```
for (int i : a)
{
sum += i;
}
```

Q→ for 10 array elements, print sum of 5 elements

```
for (int i : a)
{ sum += i;
if (i == 5)
break;
}
```

Chapter-6

Classdream
Date
Page

CLASSES → It is a template in which we define certain attributes / behaviour.

It is a structure in which we are defining behaviour or attributes of an object.

Object is an instance / copy of a class.

Program
→

```
class class_name
{
  PSVM ( )
  {
    variable ;
    S.O.P.
  {
}
```

⇒ // This is not Object Oriented Programming bcoz, we are defining variable in main ( ).

⇒
```
class abc
{
int x, y;
}

class def
{
  PSVM ( )
  {
    abc a = new abc ( );
```

without Inheritance,

// To use class in other class we make object of that class.

Syntax to create object

(Reference variable)

as soon as this stmt. comes, class abc get's memory.

int area;

Classdream
Date
Page

a.x = 10 ; → // initialisation of objects.
a.y = 20 ;
X area = x * y ; // variables cannot be used in other class without reference of that class.
area = a.x * a.y ;
S.O.P(area) ;
}
}

abc a ;
Here a is ref variable no memory has been allocated to it.

→ Meaning
⇒ abc a = new abc ( ) ;
class name — (reference variable, don't know where it is pointing) — with these keyword we get memory for the object created
(no, need to declare this variable)

a.x = 10 ;
a.y = 20 ;

Object is an entity which have a behaviour;

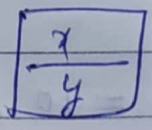
abc b = new abc ( ) ; // 2nd object of same class.
// memory

b.x // z

⇒

⇒ Thus java automatically destroys the variables Not the work of user.

ClassArcadia Date Page

⇒ abc a = new abc ( ); ⟶ // 

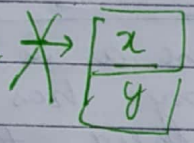
abc b; // ref. variable

b = a; Points to a.

// b is a ref. var. and point to a

In this we are not creating new memory, we are copying the reference of b into a.

a = null; // now a is not pointing to x & y, but we can refer to x & y with the reference var. b.



b = null; // now value still persists but we cannot access that. b is not pointing to that memory.

⇒ Automatic Garbage Collection.

If a var. is not in use for a long time, JVM flushes them out. In C++, we have to deallocate on our own.

Non Parameterised method

⇒

```
class abc
{
  int x, y;

  Void area ()
  { S.O.P(x*y); }
```

or

```
  int area ()
  { return x*y; }
```

```
class def
{
  P S V M ( )
  {
```

Classdream
Date
Page

```
abc a = new abc();      // abc() is a default constr.
a.x = 10;               // to initialize x & y.
a.y = 20;
a.area();      // or    int c = a.area();
                        S.O.P.(c);
                        // or
                        S.O.P(a.area());
```

⇒ Parameterised method

```
a.area(10, 20);
```

(// we do not have to initialize x & y by its object.) as we are passing the values as parameters.

⇒

Q→ WAP for a box class with characteristics height, width & depth to find the volume of the corresponding box.

```
class box
{
    int h;      int h,
    int w;      int w,
    int d;      int d;

    void vol()
    { S.O.P.( h * w * d ); }
```

```
class BOXmain
{
  PSVM( )
  {
    box b = new box( );
      b.h = 2; b.w = 3; b.d = 4;
    b. vol( );
  }
```

⇒ **Constructor** (automatic Init" of default variable)

method / function, which intializes the instance variable.
→ default values can be provided.
→ same name as that of class.

```
class abc
{
  int x, y;
  abc( ) {
      x = 10;
      y = 20;
  }
}
```

importance, with the default values program will work

```
abc a = new abc( );
```

default constructor

Passing default values in constructors as parameters

Classdream

Date
Page

Q→ WAP. to find ~~the~~ volume of a cube or cuboid that is decided at the run time.

```
class box
{
  int x, y, z;
  box ( int a, int b, int c )
  {
    a = x;
    b = y;
    c = z;
  }
  box ( int a )
  {
    a = x;
    a = y;
    a = z;
  }
  x = a;
  y = a;
  z = a;
  float volume ( )
  {
    S.O.P ( x * y * z );
  }
class boxmain
{
  PSVM ( )
  {
    box B = new box ( 10 );
    box B = new box ( 10, 20, 30 );
    B. vol ( );
  }
}
```

cd TR
Javac BoxM.ja

Classdream
Date
Page

Switch menu
Stack class → array fixed size
cube / cuboid
for loop - push
for loop → pop.

Chapter - 7

Within a class only

METHOD OVERLOADING

→ (defining same method repeatedly with same name)

Inside M-O, we can define multiple methods ~~who~~ with same name but different no. of parameter (datatypes) inside a class.

⇒

```
class abc
{
  int x, y;
  void area()          // F1
  {
    S.O.P("hello");
  }

                       // F2
  void area(int j)     // or  void area(long/char a)
  {                            (long a / 10.4
    S.O.P("b");                 no output
  }
```

will execute

Classdream
Date
Page

float → 4 byte
int → 4

```
void area (int a, int c) //F-3
{
  S.O.P("world");
}
```

here f1, f2 & f3 denote method overloading.

```
class def
{
P.S.V.M()
{
  abc z = new abc();
  z.area();
  z.area(10);
  z.area(10, 20);
}
}
```

⇒ Constructor Overloading { default initialisation of variables }

```
class Box
{
  int h, w, d;

  Box()
  {
    h = w = d = -1;
  }
  Box(int a)
  {  h = w = d = a; }
```

this Keyword

→ to use same variables


```
Box (int a, int b, int c)
{   h = a;
    w = b;
    d = c;
}
```

Q→ W.A.P, with 3 variables in which 2 variables are defined inside a constructor, find the Volume for the corresponding figure.

```
class Box
{
  int a, b, c;
  Box ()
  {  a = 2;
     b = 3;
  }
  void vol ()
  {
    S.O.P ("Volume is " + (a*b*c));
  }
}
class Main
  {
    PS.VM ()
      {
        BOX B = new Box ();
        B.d = 10;
        B.vol();
```

Classdream
Date
Page

⇒ Garbage Collection

(JVM automatically flushes memory allocated by ob. using a particular method)

method of JVM that is called at backend is finalize

Syntax →

```
protected void finalize ( )
{
}
```

(finalize → Keyword.)

(last method to be used before flushing any data.)

⇒ Arguement Passing.

call by Value

```
void swap (int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
int main ()
{
```

copies of variables are created (created)

Refrence call by

```
void swap2 (int &a, int &b)
{
}
```

address is passed, actual variables are changed.

Classdream
Date
Page

call by value

```
class Test
{
void meth (int i, int j)
{  i *= 2;
   j /= 2;
}

class def
{
  PSVM()
  {
Test t = new Test();
  int a=15; b=20;
  S.O.P.(a+" "+b);          // a=15, b=20
t.meth(a,b);
 S.O.P(a+" "+b);            // a=15 => b=20
  }
}
```

call by Reference

```
Class Test
{
  int a, b;
  Test(int i, j)                // (values get updated
    { a = i;
      b = j;
    }
void meth(Test o)               // use of obj. as obj. are passed
  { o.a *= 2;
    o.b /= 2;  }
```

Classdream
Date
Page

```
Class dy
{
  PSVM ()
  {
    Test t = new Test ();
    int a = 15; b = 20;
    S.O.P. (a + " " + b);   // a = 15, b = 20.
    Test ob = new Test (a, b);
    ob. meth (ob);
  }
}
```

value get
Object

Recursion

Static Keyword

Static methods can only be called in static methods

Static can be used as a block & variable

```
class Abc {
  Static {
    S.O.P ("Hello");   // (Execute first)
  }

  PSVM ()
  {
    S.O.P ("World"); }
  }
```

① without instantiation in main.
② as a variable;
globally accessible as variable
static int a = 10; // (no need to create Object of static variable in another class)

⇒ Final Keyword [using final with a variable name makes its value constant]

Use, ① with variable.
final int i = 10; // (value can not be changed further.)

② used with methods
// method overriding is not possible.

③ with a class. → final class abc { }
// class can not be
④ final class can not be inherited.
inherited., no overriding

Static Keyword
When a member is declared as static, it can be accessed before any objects of its class are created,
Ex → PSVM()

There are some restrictions on static.
① method declared as static can

(Static methods can be accessed in main without reference to objects)

Only call the other static methods.
(we can call any static method in Psvmain beoz main is a static method itself)

(2) Static methods can only access static data.

(3) The methods cannot be referred with this or super Keyword.
↳ Method overloading

NESTED AND INNER CLASSES

If we define a class within a class within Bracket within a scope, then this class is known as a nested class. There are two types of nested classes.

1) Static 2) Nonstatic. [inner class]

Nonstatic

```
class A{
class B
{
}
}
```
// non static class (can be accessed throughout class)

⇒
```
class Outer
{
int outer_x = 100;
void test()
{
Inner inner = new Inner();
Inner.display();
}
class Inner {
```

we can create the object of inner class in outer class but not vice versa.

Date Page

⇒ Using class within class, we can restrict scope of inner class.

⇒ Ob. of inner class can be defined in outer class but vice versa is not true.

```
void display()
  {
    S.O.P("Value of x + Outer_x);
  }
}
Class Demo
{
PSVM()
  {
  Outer outer = new Outer();
  Outer.test();
  }
}
```

⇒ **Command line Arguments**

[initialise array using cmd line arguments]

```
class abc {
PSVM(String args[])
  {
  for(int i=0; i<args.length; i++)
    {
    S.O.P(args[i]);
    }
  }
```

args.length → available in String class by default.

CLA programs

At runtime,

⇒ Java _ filename _ 0 _ 1 _ 2 (Space)

arguments passed at runtime.

→ (CLA)

⇒ Variable length arguments

```
class CArray
{
static void Test(int V[])          // static method
{
S.O.P.(V.length);
for(int x:V)
{
S.O.P(x+" ");
S.O.P();
}

PSVM()
{
int n1[] = {1,0};
int n2[] = {1,2,3};
int n3[] = { };

Test(n1);                          // 3 different arrays will be created.
Test(n2);
Test(n3);
}
}
```

because main() is static, only a static function can call a static function.

there is no need to create objects of static

```
class cArray
{
Static void test (int ... v)        // ... → Variable length Array
{
  S.O.P. (v.length);
for (int x : v)
  {
  S.O.P (x + " ");
  S.O.P. (" ");
    {
PSVM ( ) {
  int n1 [] = {10};
  int n2 [] = {1,2,3};
  int n3 [] = { };
```

⇒ No need to create this we can pass values directly.

```
Test (10);
Test (1,2,3);
Test ( );
```

Here there no. are (int)

⇒ Restrictions of variable length Arguments

① If we define more than 1 V.L. Methods (Method overloading) then the concept of variable length arguement does not hold or it creates ambiguity

Classdream
Date
Page

ambiguity

Static void test (int... v)
static void test (int x, int... v)
Static void tes (int... v, int y)

→ int (10,20);

Static void test (int... v, char... y) //

lab

Test (10, 20, A)

ch-8

⇒ INHERITANCE → Process of inheriting prop. defined in super class.

Base
↓
Parent class / child class

```
class A                          // Base class
{
int x = 10;
}

Class B extends A                // child class
{                                // extends is a Keyword which will inherit another class.
S.O.P(x);  // 10
}

Class C {
PSVM ()
{
B b = new B();
}
```

Access Specifiers → ① Public ② private ③ protected ④ Friendly (By default)

Multiple Inheritance is not allowed directly.

⇒ Class A → C (A, B) ; A, B, C → D (not directly)

{

}

Class B extends A.

Multilevel Inheritance A → B → C → D

⇒ METHOD OVERRIDING

- Same name
- Same parameters.

```
Class A
{
    int i = 10;
    void abc ()
    {
        S.O.P ("hello");
        S.O.P (i);
    }
Class B extends A
{
    void abc ()
    {
        S.O.P ("world");
        S.O.P (i);
    }
}
```

Private → [not available in sub class]

HERITANCE

Overriding

A abc() &

```
Class C
{
    PSVM ( ) {
    B b = new B ( );
    b.abc ( );
}
}
```

(Output) // world

Classdream
Date
Page

To print base class method in derived class :-

⇒
```
class A
{
  int i = 10;
  void abc () {
      S.o.P ("Hello");
      }
}

class B extends A
  {  int i = 20; // [declaring same var. in derived class, overrides base class var.]
     void abC ()
super(); {
super.abc // Super.abc(); // Base
     Super.i ;  ———  Base
     int i = 20;  → class B
     }
  }
```

Super Keyword use to call base class method, variables,

⇒
```
class A
  {
  int x, y ;
  A (int a, int b )
    {
      x = a;
      y = b;
    }

class B extends A
     {
      int a, b, c;
```

for constructors use Super();
& for methods use super.fn();

Date
Page

Lab

// or

```
B( int z , int t , int p)
{
  a = z;
  b = t;
  c = p;
}
```

```
class B extends A
{
  int c;
  B ( int z , int t , int p)
  {
    super( z , t );
    c = p;
  }
}
```

with constructors use super (

complexity decreases by using inheritance concept. no need to declare more variables. we can use that of Base class by super keyword

Lab

W.A.P to define a base class box with three parameters h, w & d. Also define a subclass boxweight with a new parameter weigh. find the corresponding volume of a box including the weight.

```
class box
{
  float h, w, d;
  float vol( int a, int b, int c )
  {
    a = h;
    b = w;
    c = d;
    S.O.P ( h * w * d );
  }
```

Super & Super

In case of Constructor → use super( ) / same para ~~constructor~~ in ~~derived~~ class

// when constructor is called in multilevel inheritance then, constructor of base class will be called first.

```
class A {
    A()
    { S.O.P("A"); }
}

class B extends A
{
    B()
    { S.O.P("B"); }
}

class C extends B
{
    C()
    { S.O.P("C"); }
}

class D {
    PSVM ()
    {
    C c = new C();
    }
}
```

Output

// A B C

Classdream
Date
Page

⇒ DYNAMIC METHOD DISPATCH OR RUNTIME POLYMORPHOSISM

Reference is decided at runtime)

Ex:

{ virtual in C++ }

```
class A
{
   void abc()
   {
      S.O.P. ("hello");
   }

class B extends A
   { void abc()
      {
         S.O.P. ("World");
      }
   }

class C {
   PSVM()
   {
      A a;          // reference to super class.
      A a = new A();
      B b = new B();
      a.abc();      // hello
      b.abc();      // World
      a = b;        // a = b;
      a.abc();      a.abc();
   }
}
```

→ this will decide dynamically which method will be called.

⇒ This is known as runtime polymorphosism.

Dynamic Method dispatch is a mechanism by which a call to overridden method is resolved at runtime. Also, it is important bcoz java implements runtime polymorphism with the help of this mechanism

Q→ Why we need overridden methods ?

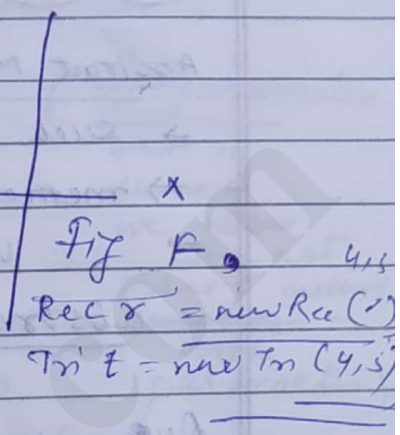
Overridden methods allow java to support runtime polymorphism. It allows a general class to specify methods that will be common to all its derivatives by allowing sub classes to define specific implementations of all those methods.

Q→ WAP to implement runtime polymorphism with the help of base class named as figure and sub classes name triangle & rectangle respectively to find the area of the corresponding classes.

```
Class figure
{ float a;
  void area ()
  {
    S.O.P ("The area is :"+a) ;
  }
}

Class Triangle extends figure
{ float b, c;
  void area int
  { a = b *
```

```
class Fig
{
int float a, b, c;
   Fig (int x, int y)                x 10
   {                                 y 5
      x = b;        b = x;
      y = c;        c = y;
   }

   void area ()
   { S.O.P ("The area is" );
     }

}

class Tri extends Fig.
{
Tri(int x,int y)      Tri (int p, int q)
{ super (x,y); } { { super. Fig (x, y); } x super (x, y);
   void area ()
   { S.O.P ( ½ * b * c ) ; } }
}

class Rec extends Fig
{
   Rec (a,b)                          } { Rec (int x, int y)
   { super. Fig(a,b); }                  { super (x, y);
                                         }
   Void area ()
   { S.O.P ( b* c); }
}
```

```
class Def
{
  Psvm ()
  {
    Fig f = new Fig (4,5);
    Rec r = new Rec ();
    Tri t = new Tri ();
    Fig R;
```




```
Fig f = new fig (10,5)
rect r = new rect (2,3);
r.area;
Fig t;
f.area();
R = t;
R.area();
R = r;
R.area();
}
```

Abstract [provides interface to other classes]

Abstract Methods

→ such methods with no implementation.
→ methods whose body has not been defined
→ void abc( ); // no body.

abstract → keyword

Due to abstract methods → the class becomes abstract class.

```
abstract class abc
{
    void abc ( ) ;  // abstract method.
    void xyz ( ){ } ;  // can have normal method
}

class def extends abc
{
    void abc ( )  // abstract method need to be defined here
    {
    }
}
```

Partial Implementation of any method bcoz ? which abstract class cannot be instantiated.

Classdream
Date
Page

⇒ abstract class abc {

These all can be used in abstract class.

{ data members ;
methods ;
abstract methods ;
constructor definition only
}

void abc ( ) ;
↓
[no. parameters
then in derived we can not create method with parameters]

→ Restrictions 2.)

- In subclass, there is no need to implement, general methods

⇒ Practical

[Overriding 2.
inheritance
Runtime polymorphism]

Classdream
Date
Page

Taking Input

USER INPUT IN JAVA

import keyword → compiler search for external file

First stt. before defining class.

→
```
import
java.util.Scanner ;  // class for taking user input.
```
[util.* ; → accessing all the classes in util.]

→
```
import java.util.Scanner ;  // first stt.
class abc
{
  PSVM ()
  {
    Scanner p= new Scanner ( );
    p.next ( ) ;
    p.nextLine ( ) ;
```

[Object of Scanner class]

System.in → input → system class

p → represents to object

// → used to read string

```
int i = p.nextInt ( ) ;
p.nextByte ( ) ;
p.nextDouble ( ) ;
```

```
String ( ) ;
int i ;
```
(declare variable)

*

[Direct method is not available in Scanner class to read a character]

Classdream
Date
Page

Practical list
→ Factorial
→ Prime no.
→ Method overloading

Multi-level hierarchy → package

imp Theoretical + output

Chapter-9

Packages

① → Collection of classes like a folder.
A package is a collection of related classes. It helps in organising classes into a folder structure & make it easy to locate and use them.

② It also increases the reusability.

Packages

Built in
(by default available in java)

java.lang
this package is imported in a program by default by java

Ex →
- java.io
- java.Exception (To Handle errors)
- java.awt

Built in packages are used in Java with the help of import Keyword.

User defined
→ If we create package that is our own folder, then.

```
package abc ;   // first statement
```
name of folder

import java.util.*; ← If we don't the exact of name of folder otherwise filename.

to access next subfolder.

→ Partially qualified name

import java.util.___;

import java.util.Scanner;

→ Fully qualified name

util package
→ contains a lot of classes.

→ To print data of System:
→ import java.util.Date; // Headerfile
→ Date d = new Date; // Object
& then Print this Refrenc.

Practical Lab ⟹ import java.util.Scanner = class abc extends java.util.Scanner

⟹ User defined package

```
package mypak;          // package pkgname;
class Book
{
  String bookname;
  String author;
  Book(String b, String c)
  {
    bookname = b;
```

// java.io.*;
This means previous name is package.

Import Package →


```
author = a;
{

class Test
{
public void show ()
{
S.O.P (bookname + " " + author);
}
}

class Test
{
PSVM ( )
{
Book d = new Book ("Java", "Herbert")
d.show();
}
}
```

To compile this program (including package)

→ javac -d . filename.java

(Space)

java packagename.filename

Note →

javac -d . Test.java } → If Program is compiled successful, mypack folder is created.
java mypack.Test

• -d → directory

-d → directory.

Oracle.com

Java API

→ if directory is not specified, then it will get saved in current directory

→ if directory is specified, then folder is created there
↳ -d c:/Bin.

→ After creation of package, now import stat. can be used in program using mypack package:

→ API → Structure in Java contains classes.
↓
{Application Programming Interface}

⇒ INTERFACE

⇒ By default
public
static
final
→ variables defined under Interface

interface → Keyword

⇒ interface name {

}

- We cannot define normal methods in interface.
- No need to use abstract keyword before methods.

* Define a constr in interface. ?

→ can define but cannot instanstiate. can access through super Keyword.

~~By~~

```
interface ~~name~~ xyz
{ int i = 10;
  void abc ();          // abstract methods.
  → void def () { }  → // error. , we cannot use normal methods.
}
```

To inherit interface class, we use Implements keyword.

```
class A implements xyz
{
  void abc ()
  {     }

  void def ();
}
```

// we are not defining def in this class.
∴ This class will become an abstract class (by default)

⇒ It is compulsory to define every method of interface class in next class to implement that class.

⇒ Not compulsory to define methods using abstract Keyword.

[Multiple Inheritance]

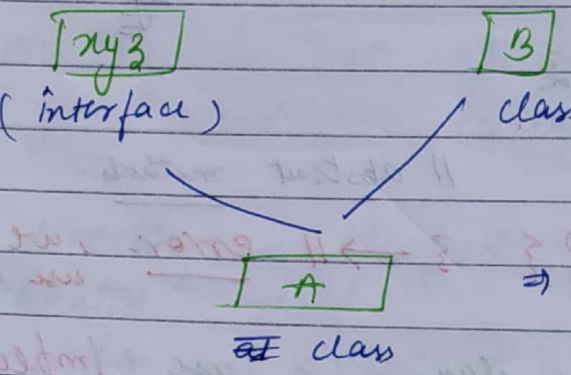
We cannot directly inherit through multiple inheritance.

But, with the help of interface, we can.

[xyz]

in

⇒ Abstract Implementation,



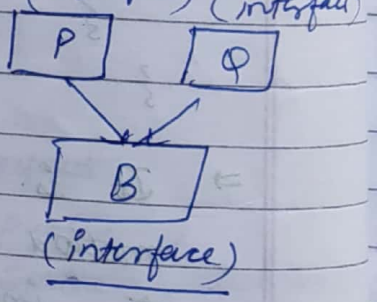
(1) ⇒

```
class A extends B
        implements xyz
{
}
```

⇒ One interface can extends more than 1 interfaces.

(2)

```
interface B extends P, Q
{
}
```



⇒ A class cannot implement other class directly, it can only extends.

class → class → extends
interface → interface → extends
class → interface → implements.

⇒ interface can never extends class.
interface can extend interfaces only.

Chapter 10 (Keywords)

⇒ Exception Handling

- we can handle Exception [some unusual behaviour of our program]

but we cannot handle Errors

two new keywords: catch, try, throw, finally, throws

Practical ⇒

⇒ Using cmd to add numbers :-

* Wrapper classes

String → integer

methods available in classes to wrap data is parse Datatype

To convert D.T. to integer.

⇒ `int x = Integer.parseInt(args[0]);`

(Integer class) — class.method-name — parameter which has to be converted.

* 1-D array

`int[] a = new int[size];`

* 2-D array

`int[][] a = new int[s][s];`

Classdream
Date
Page

Exception Handling

Throwable (superclass for exception handling)

Exception (subclass) — Error (subclass)

```
class abc
{
  PSVM()
  { int a,b,c;
    a=10; b=0;
    c=a/b;
    S.O.P(c);
  }
}
```

→ Since, there is no syntatical error, so no error will be thrown, program will compiles but at runtime it will throw an exception.

Output

ArithmeticException at java : (5) → line where we get exception

↓ classes in java

→ This exception is called Stack trace

⇒ Throwable → Exception (all exceptions of sub classes are handled by class exception not throwable class.)

→ ArithmeticException *
→ ArrayIndexOutOfBounds
→ NullPointer

Types of sub classes under exception class.

eg →

```
static void abc()
{
    C = a/b ;        → (5)
}
class abc {          → (2)
    PSVM()
    {
        int a, b, C ;
        abc() ;
    }
}
```

calling function

Exception

Arithmetic Exception()
main.abc() at java:
line 5 and line 2

Object of class with exception() is thrown.

⇒ initialize variables outside try, catch block.

⇒ multiple catch and one try.

⇒ nested try.

Access Specifiers

1. Public
2. Private
3. Protected
4. ~~Default~~

→ can be used for both classes & Variables.

⇒ Public can be used with the same class with a same package sub classes with a different package sub class & with a different package non sub classes

```
public class abc {
=
}
```

⇒ Protected can be used with a same class same package sub classes, same package non sub classes, different package subclasses but we can not use it with different package (non subclasses) (no inheritance)

⇒ ~~Private~~ default

are cannot use ~~private~~ default access specifier with a different package subclasses and with a different package non subclasses

⇒ ~~default~~ Private

we can use private access specifier with the same class only.

RAM → Primary memory

task manager
Class → all + data

Date
Page

Multi-threading

{process
Thread}

- Process is like a light weight program.
- Thread reg. less memory → it is light weight process
  eg → save & open file ; print. (any executing program running)

Process — Word [P]

Threads: [T1] [T2] [T3] →

T1 — Open file
T2 — save
T3 — Print

• Thread Priority
(Thread Priority will decide which thread to be executed first.)

- synchronisation
- Message passing → Possible through message passing. (→ Handshake)

Multithreading

Thread priority — Synch — Message passing

⇒ Thread →
To implement multithreading we use

Thread (class)
no need to create thread class object (class defined automatically like exception class)

Runnable (interface)

→ (If thread is used as a class it has to be extended)

Better as it is flexible as compared to interface.
In interface we have to give implementation of all methods in interface

Inbuilts

Methods available in thread class

- getName( )
- getPriority( )
- isAlive( )
- join( )
- run( )
- sleep( )
- start( )

(1) getName( ) ← to know which thread is getting executed. gets name of thread.

(2) getPriority( )
to display the Priority of thread.
to set & check Priority.

(3) isAlive( )
to check whether thread who has to be given value is present or not, returns a bool value.

(4) join( )
to concatenate 2 threads.

(5) sleep( )
if 10 programs are running, execute 1 prog. and keep remaining to sleep. sets time in secs for every task.

(6) run( )
to execute threads, we call run( ).
which has no delay

(7) start( )
After sleep, which func. has to start

Classdream
Date
Page

Program

```
class NewThread implements Runnable          // Interface
{
    Thread t;                                // reference
    NewThread()
    { t = new Thread("this", "Demothread"); // new object t created of type newThread
    S.O.P("Child Thread");
    t.start(); -> // inbuilt prog.
    }
    public void run()                        // method.
    {
      try {
        for(int i=5; i>0; i--)               // trying to execute the thread 5 times
        {
          S.O.P("ChildThread" + i);
          Thread.sleep(1000);                // calling object
        }                                    // delay thread in secs
      }
      Catch(InterruptedException e)
        { S.O.P("Interrupted"); }
      S.O.P("Exit");
    }
}
class Demo {
      PSVM()
      {
        new NewThread();                     // no reference var. only object creation (for new thread)
        try { for(int i=5; i>=0; i--)
          S.O.P("Main thread");
        }
```

Keyword to define t.

for part to create thread

(Calling of new Thread without creation of obj: name)

(copy try loop)

for thread t

since we are using runnable interface, 't' is a thread

Output →
child Thread 5
Main " 5
child " 4
Main " 4
(until i becomes 0)

Classdream
Date
Page

⇒ Throws (used with Method signature)

```
void abc () throws Exception 1, Exception 2
{ throw new Exception 1 ( )
}
```

Explicit declaration of Exceptions

// we can declare more that 1 Exception with throws : We just only declare the type of Exception.

Benefit → loading time is less.

⇒ **Finally** → By default, irrespective of program execution, finally block always executes.
→ to display user friendly Message.

final finalize finally

```
try {
}
catch ( )
{
}
Finally {
}
```

// (Code will display at its own position)

VIVA

⇒ Creating Own Exceptions

~~to inherit all runtime Exceptions~~

```
class abc extends Exception {
    String msg ;
    abc ( )
    { msg = new String ("Enter a no. b/w 20 & 100
    }
}
```

inbuilt class

Classdream
Date
Page

Exception
↓
Runtime Exception
↓
built-in | user defined

```
class MyException
{
  PSVM (-)
  {
    try
    {
      int x = 10;
      if((x<20) || (x>100))
        throw new abc ();
    }
    catch (abc a)
    {
      a.msg
      S.O.P (@) ;
    }
  }
}
```

Only default constr. will work.

// throwing exception in the form of object of abc class

// a is reference of abc class

i.e. we can use abc class

Error: Here message does not gets print

// (Message gets displayed)

// → abc to display message statement displayed class abc use a.msg

class name is displayed ⇒ Constructor of class abc is used as while using Stmt. throw, we create object of class so const^r has to called

⇒ java.lang
↓
Throwable
↓
Exception | Error

→ By default in java (package)

Classdream
Date
Page

Assignment

Q1 Simulate a bank account problem with a condn that a user can not withdraw a amount if the balance is below 1000

→ Menu driven method
credit / debit / balance display
withdraw → Balance current.

Q→ Differance b/w abstract clas & Interface

Abstract	Interface
→ can have abstract as well as non abstract methods	→ can have only abstract methods
→ abstract clas doesn't support multiple inheritance	→ Interface supports multiple inheritance
→ can have final, non final, static & non static variable	→ Interface can have Static & final variable only
→ can have static methods and constructors	→ Interfaces can not have Static methods or Constructors
→ The abstract keyword is used to declare an abstract class	→ Interface Keyword is used for Interfaces.

⇒ code must be defined in differences

Higher memory for Process (Chrome) new window
New tab → Date Page thread

Similarities

1. We cannot instantiate an abstract class as well as the interfaces
2. But, we can create a reference variable of both

Assignment

Q-2 Define a constr inside an interface & try to instantiate it. If the condn becomes false, try to create a reference variable for the corresponding interface.

Multi-threading

```
class Multi extends Thread
{
public void run()
  {
    S.O.P(    );
  }
}
PSVM()                    class ThreadDemo{
{ Multi t1 = New Multi();
  t1.start();
}
}
```

Note → Here we didn't require to call void run(). It is called by t1.start();

→ By default, name of 1st Thread & so on is Thread 0, 1, 2.

Classdream
Date
Page

Exception Handling (Continued)

⇒
```
class abc
{
PSVM()
{ int a, b, c;
  a=10; b=0;
  try {
    c = a/b;
    S.O.P();
  }
  catch (ArithmeticException e)
  { SOP("Div. by zero"); }
```

trying to catch an error

Immediate catch block after try.

arithmetic → object of class

NOTE →

Using try, catch block, program doesn't end abruptly, at least it displays a message as compared to provides program.

⇒ To display same exception as of java prog

```
S.O.P("Div by zero" + e);
```

output →

Div. by zero ArithmeticException at java lineno:

⇒ **Multiple Catch block →**

We can create hierarchy using multiple catch.

```
try
{
 {
catch ()
    {
     }
 Catch ()
    {
     }
 catch ()
    {
    }
```

eg→

```
class abcE
  {
```

Case I

```
  ps v M ()
  { int a, b, c;
    try {
      a=10, b=0;
        c=a/b ;
      S.O.P ();
    }
  Catch (Exception e)
    { S.O.P ("1") ; }
  Catch (Arithmetic Exception e)
    { S.O.P ("2") ; }
```

same code for case 2

Output → 1

* Since, class Exception handles all exceptions, thus it can never jump on sub classes.

NOTE → while creating hierarchy always define subclass first and then super class.

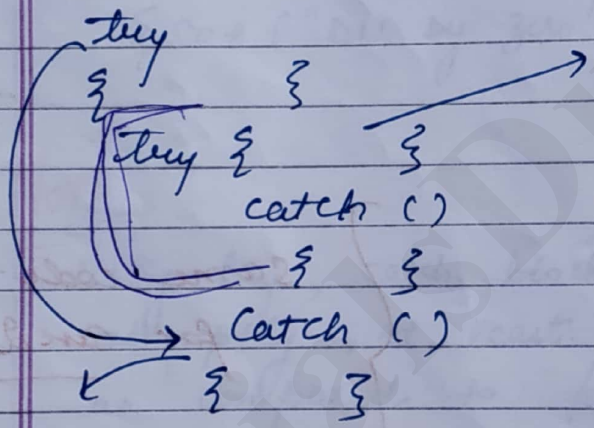
case II

(Same code as above)

```
catch ( ArithmeticException e )
    { s.o.p ("2") ; }
Catch (Exception e)
    { s.o.p ("1") ; }
```

Output → 2.

⇛ NESTED TRY



If exception isn't caught in this catch, then exception will never come out only next catch can handle it as scope of this try catch has finished.

Can't handle exceptions thrown by 2 try block as scope has finished.

Note →

```
try {
    try { }
catch ()
    { }
```

* Can't work as there is only 1 catch for both try.

Throw → explicitly tell the program which type of exception is thrown.
→ handles 1 exception
→ used in a method Signature.

Throws → handles more than 1
→ used with a method signature.

Note → All the above exceptions are implicit exception but with throw & throws we can define explicit exceptions.

```
class abc
{
static void def()
{
try {
throw new NullPointerException("hello");
} explicit creation of obj.
catch (NullPointerException e)
{ S.o.p("World"); }
```

explicitly creation of object & throwing it

Output
World ✓

⇒ In Implicit exceptions, java on its own at backend creates obj. of class & throws it.

V.Imp

↳ throw NullPointerException()

not using new means reference is being thrown and ref. are never caught in java.

* using throws →

Static void abc() throws NullPointerException, ArithmeticException

{ }

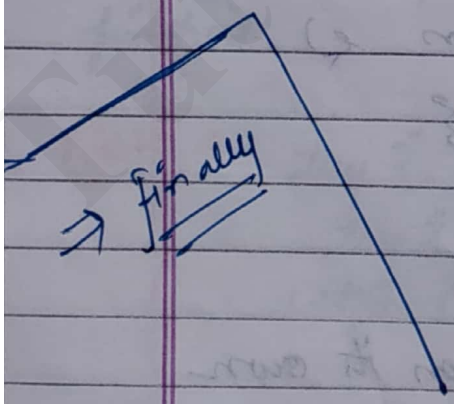
adv. is that we can define more than one exception.

Throws Keyword provide Warning to that system is ready to handle exceptions →

⇒

Q- Difference b/w throw & throws ?

Q- Difference b/w Abstract & Interface ?



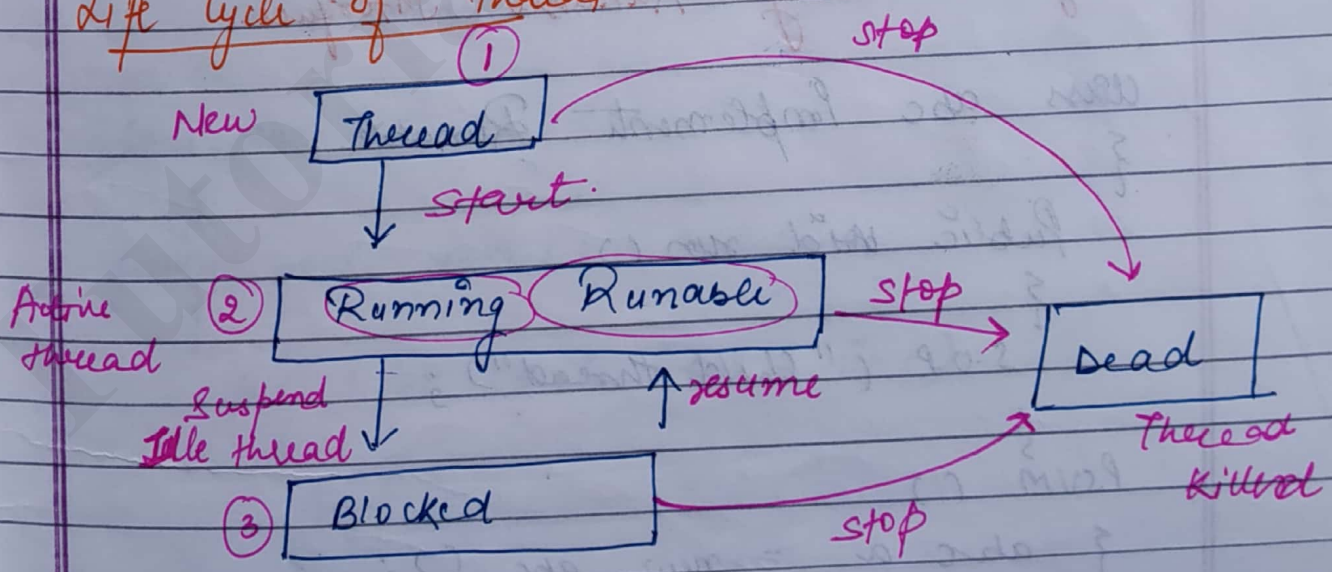
Multi-threading

It is a conceptual programming concept where a program is divided into two or more sub programs which can be implemented at the same time in parallel

Benefits

1) enables programmer to do multiple things at one time.
2) Improve performance.
3) Simultaneous access to multiple applications
4) Programmer can divide a long program into threads and executes them in parallel which eventually increase the speed of the program execution.

Life cycle of thread



(1) t.start() → new born state thread starts
(2) run()
(3) sleep

[Interface → runnable
Class → thread]

Implement run in both ~~class~~ cases but implementation is different.
(available in Java.lang.Thread)

Method 1

(1) Runnable → public void run()
{

}

Thread (compulsary Runnable ob , optional thread name)

* Program using Runable Interface.

```
class abc Implements Runable.
{
   Public void run()
   {
     S.O.P ("Child thread");
   }
PSVM ()
{ abc a = new abc();
  thread t = new Thread (a);
    t. start (); ] // no M will go to run method
  t. start (); { error at execution Illegal start expression
}
```

Classdream
Date
Page

NOTE →

We can not create start method with same object.

t1.start(); { it is alright.

extends Thread

```
class abc extends Thread
  {
   public void run()
     {
       s.o.p ("child Thread");
     }
PSVM()
   { abc a = new abc();
          a.start();
   }
}
```

→ **Sleep Method**

sleep(milliseconds);

```
class abc extends Thread
  {
public void run()
   {
     for (int i=1; i<5; i++)
       {
        try
        {
            if (i==2)
    Thread.sleep(1000);
        }
    catch (Exception e)
       { s.o.p(e); }
```

S.O.P("Exit");
}
S.O.P.("Exit");
}
PSVM()
{ abc a = new abc();
a.start();
}

⇒ directly jump to main.

* stop();
only in thread class.
* isAlive(); and join();

• isAline() is defined in Java.lang.Thread. isAlive
This method will test whether a particular thread is alive or not.

• return type
boolean isAlive();
final

→ join() waits for a thread to die.
It causes the currently thread to stop executing until the thread joins with completion of its task.

Work with isAlive();
join();
{ (first work of thread will be completed so that the program doesn't halt & do not exit) }.

⇒ Program for isAlive() & join();

Join → synchronization

⇒ class abc extends Thread
{ public void run()
{
S.O.P. ("Status" + t.isAlive());
}
}

Tanya Garg

class def {
psvm ()
{ abc a = new abc();
a.start();
a.join();
S.O.P. ("Status" + t.isAlive()); }

Tanya Garg

⇒ Thread Priority

t.setPriority(2);
object of Thread → arg. value

→ MAX-PRIORITY (10)
→ MIN-PRIORITY (1)
→ NORMAL-PRIORITY (5)

if same priority value, then random occurences will occur.

```
class abc extends Thread
{ public void run()
  {S.O.P. ("_____"); }
class def extends Thread
  {     }
class pqr
{ psvm ()
  { abc a = new abc();
    def d = new def();
    a.setpriority (10);
    d.Setpriority (1);
    a.start();
    d.start();
  }
```

or d.start();
a.start();

t.setPriority(Thread.MAX-Pr);

ClassDream
Date
Page

WRAPPER CLASSES

⇒ 4 Basic Primitive Datatypes
(1) Integer
(2) character
(3) Boolean
(4) Float.

Corresponding classes of these datatypes are known as Wrapper classes.

Wrapper classes start with Capital letter & full name

1. Integer
2. Boolean
3. Character
4. Float
5. Double
6. Byte
7. Short.

```
Integer iob = 100;  // autoboxing
int i = iob;        // unboxing
```

⇒ int i = 10;

⇓

```
Integer ob = new Integer();
ob = i;
```

{ we can directly assign value here i.e new Integer(10);

// object of Wrapper class

// assigning value to object.

⇒ • We can access method in wrapper class through ob only.

⇒ Process of wrapping of datatypes into wrapper class is known as auto-boxing

& Reverse is known as auto-unboxing.

* Practical

⇒ **Wrapper classes** → are the classes in which the primitive types are encapsulated to the corresponding classes or within the object. [6 classes]
Then there are some associated methods available with wrapper classes.

Methods

1. Char charValue ( )
2. Boolean booleanValue ( )
3. int intValue ( )
4. double doubleValue ( )
5. short shortValue ( )

All these methods returns the value of an object as the corresponding return type defined.

```
Integer ob = new Integer ( );
int i = ob.intValue ( );
```
} → Creating object & assigning value

Advantage
⇒ Typecasting becomes easy / better.

* Annotation (to be covered later)

autoboxing ⇒ It is a process of encapsulating a primitive data type to the corresponding obj. of wrapper classes.

auto-unboxing → A process of extracting a value from the object of wrapper classes & assigning it to corresponding data type

Input/Output

System (class)
- in (method)
- out (method)

⇒ java.io.* ; → package to handle input/output.
(bcoz there are many classes defined in it.)

⇒ Stream (flow)
we can access source info in two ways
(1) Byte stream (data in form of 1010)
(2) character stream (in form of characters 'A', 'B', ...)

At Backend,
⇒ without Byte stream we can not access character stream directly.

⇒ For Byte stream, classes responsible are →
- BufferedInputStream
- BufferedOutputStream
} → Implemented classes

- InputStream
- OutputStream
} → abstract classes

These are some methods in these which we have to override in our program.

for characterstream

- BufferedReader } → Implemented classes
- BufferedWriter

- Reader } → abstract classes
- Writer

To obtain a character based stream that is attached to the console.

```
Class abc
{
psvm () throws IOException
  {
    Char i ;

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

// c = br.read () ;
S.O.P (c) ;
```

(read method available in BufferedReader)

(Object to read char. value)

(ultimately we convert the input value into Byte stream)

// One character time (Use do while)

```
c = (char) br.read ();   // to read character

c = (char) br.readline();// to read string
```

ClassDream
Date
Page

(By default, type of read() is int
∴ we have to typecast into char value)

⇒ Reading String

```
import java.io.*;
class abc
{
PSVM ( ) throws IOException
  {
  String a;
  BufferedReader br = new BufferedReader ( ) ;
    a = br.readline ( ) ;
```

a.equals ( ) ;
↳ defined in String class by which we can compare any entered String

```
a.equals (" ") ;
do
⇒ while ( a.equals (" ") )
  {
```
↳ end of String

```
  }
```

⇒ for array of String

```
String [ ] a = new String[10] ;
for for (i=0;
  { a[i] = br.reader ( ) ; }
```

⇒ Writing

Method available →

To write the output, we will use print screen i.e. derived from output stream. In side this we are having a write method which can be used with System class

→ int i = 10
System.Out.print(i); // defined in System class.
System.out.write(i); // defined in Stream.
↳ import java.io.*;

~~In Print W~~
PrintWriter is a alternate version to output any data with the help of ~~Print~~ the object of PrintWriter class.
Import java.io.*;

⇒ class abc
{
PSVM()
{
PrintWriter pw = new PrintWriter(System.out, true);
(object, boolean condn)
↳
pw.println(" ");

(~~first~~ object second boolean condn)
{If True, value will flush out after display}
automatic flushout.

Classdream
Date
Page

~~Con Sole~~

Reading data from files & Writing data to files.

```
import java.io.*;
class abc
  {
  psvm ( )
    {
     int i;
try {
file InputStream fin = new FileInputStream ("");
  }
   catch (FileNotFoundException)
    {
    }
do
 {
   i = fin.read ( );
   if ( i != -1)
  s.o.p( (char) i) ;
  }}
    while ( i! = -1) ;
    }
  }
```

or we can throw with main

Specify location of file "c:/a.txt"

↳ or Exception

Pointer → file

object of file

(conan for end of the file)

Exclusively typecast

(Specify location of file)

If → ~~try~~ command prompt as parameter (args [0])

Classdream
Date
Page

• Char by char Read/Write.

⇒ To write the data into file

```
import java.io.*;

class abc
{
  PSVM()
  {
   int i;

  try
  {                    fin
  File InputStream f = new FileInputStream("  ");
  File OutStream fout = new FileOutputStream("  ");
  }
  catch(FileNotFoundException)
  {
  }

  do{
      i = fin.read();          // File is opening here

    if(i != -1)

    fout.write(i);             // File is opening
  }while(i != -1);
  {
  fin.close();
  fout.close();
  }
```

(location of file)

} ⟶ to close files

We have to close the files with a stmt.
object fin.close();

GRAPHICAL INTERFACE

⇒ Applets (Graphical User Interface) GUI

⇒ Programming in terms of Event handling (buttons)

- Applets do not have main method.

Two classes

- Graphics defined in java.awt.*; (package) (class) (helps to create image/window) — abstract window toolkit
- Applets defined in java.applet.*; (classes)

⇒ When GUI is executed, it displays a window structure.

Pixel → smallest representation unit of an image

⇒ What is Java Applet.

An applet is a Java program that runs in a web browser. It can be a fully functional Java Application because it has the entire Java API's.

Applets are executed using
(1.) help of applet viewer.
(2) using HTML

* Differences b/w an Applet & a normal Java Program

⇒ • An applet is a java class that extends java.applet.Applet ;
(package) (class)

⇒ • No main method is defined for a java applet

⇒ • A JVM is required to view an Applet. The JVM can either be a plugin of the browser or a separate runtime environment
(extra JAR)

⇒ • Applets have strict rules that are enforced by the web browser. The security of Applet is often referred as a sand box security.

• Applets can be downloaded in a single JAR files.
{Java Archives (zipping)}
Only java.lang are available.

JAR is a collection of java.class files.

⇒ Life cycle of Applets.

(1) init()
(2) start().
(3) stop()
(4) destroy()
(5) paint()

(1)–(4) → defined in Applets
(5) → defined in awt.

Sequence of Working

1. init() // initialization
2. start() // start the Program.
3. paint() // Paint the content
4. stop() // stop
5. destroy() // destroy the memory.

5 methods to Create Applets.

1. init()

This method is used for the initialization purpose of the variables or whatever content needed by an applet. This method is called only once during the life cycle.

2. start()

start() method is called automatically after the browser calls the init() method. It is also called whenever the user returns to the page containing the applet.

3. paint() method → only for display
is invoked immediately after the start method and is responsible for all the graphics involved. Paint() is called as no. of times as start() is called.

4. Stop()
is called automatically when the user moves off the page on which the applet is working

5. destroy()
This method is called only once when the browser shut downs normally. Or the applet execution is completed.

⇒ Simple Applet program:

~~public class abc extends Applet~~

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

public class abc extends Applets
{
    public void paint(Graphics g)
    {
        g.drawString("Hi", x, y);
    }
}
```

creating on browser is public

Note: all the methods under paint() can be accessed by the object of Graphics class.

drawString → displays a string message
(x, y) → coordinates (in pixels)
"Hello" → message to be displayed.

⇒

1st Method ⇒ compile using normal java
but execute using,
>> appletviewer abc
↳ (class file)
↓
(exe available in Java folder)

ENUMERATION

→ Self defined constants.
→ can't be instantiated.

enum

```
enum enum_name
{
  a, b, c, d;
constructor, method can be defined.
}
```

Ex→

```
enum Apple
{ Red, Blue, Green; }
```

} treat as a class

```
Class abc {
    PSVM() {
    Apple a;
    }
      a = Apple.Red;
        S.O.P(a);        // Red.
}
```

} → generate reference

Enumeration is a list of named constants which is created with the help of enum Keyword and can access the values defined inside the enum only.

```
If (a == Apple.Red)
{S.O.P ("Apple contains Red");* }
    switch (a)
    case Red:
    case Blue:
```

→ Inbuilt associated Values

Values and valueOf
values() // more than 1 value
valueOf()

→ Apple.values();

```
a[] = Apple.values();
for(Apple a : Apple.values())   // For each loop
    S.O.P(a);
```

Value of method returns the enum constant whose values corresponds to the string passes as a parameter.

→ value of (str)

```
a = Apple.value of ("Red");
```

→ if available returns

⇒ Java enum as class type :-

```
enum Apple
 { Red(10), Green(5), Blue(7);
   int price;
    Apple(int p)
    { price = p; }
    int getPrice()
    { return Price; }
class abc
 { PSVM()
    { Apple a;   or   a.getPrice
      S.O.P(Apple.Red.getPrice())
    }
 }
```

Classdream
Date
Page

Chapter - 16

STRING HANDLING

collection of characters (no restriction in Java to put ch in " " or ' ')

→ inbuilt class in java.

⇒ String is defined as an inbuilt class within a package java.lang. It is immutable in nature.

① `String s = new String();`
calling of default constr

② `String t = "ABC";`
`String s = new String(t);`

③ `String s = new String("ab");`
↙ parametrised call

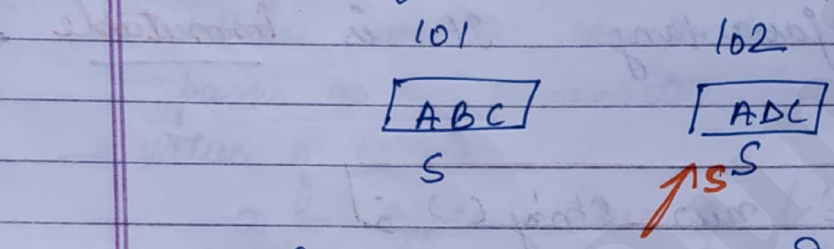
④ `char[] b = {'A', 'B', 'C'};`
`String s = new String(b);`
↙
constr defined → String(char b[])
(using the array, we instantiate string ob)

⇒ String that can't be modified / changed is immutable.

classes that can't be modified are immutable classes

```
char [] b = { 'A', 'B', 'C' };
String S= new String (b);
S = "ADC";
```

Here s is a new object and value of previous object doesn't get overwritten. can't assign new value to same object.



Since s now gets associated with 102.
∴ We can't access now 101

ex→

```
char [] b = {A,B,C};
String S = new String (b);
S.O.P.(S); // ABC
S = "ADC";
S.O.P (S); // ADC
```

- String (char chars[], int start, int num)

(const) Starting point no. of chars

eg → b = "ABC"
0 1 2

if we want to store C in array

String (b, 2, 1)
index → no. of chars to be copied

- String (byte char [ ]) { Even here we can pass ~~start~~ & num) }
  (ascii chr. array)

⇒ To assign same value to 2 different objects

```
char c[] = {'A', 'B', 'C'};
String s = new String (c);
String d = new String (s);
```

*→ Operations related to String class:-

(1) Concatenation operator.

"ABC" + "def"

ABCdef

eg →

```
S.O.P. ("ABC" + 5 + 5)
```

Output: ABC55

{ ∵ If we concatenate with String, all values get converted to String value }

```
S.O.P ("ABC" + (5+5));
```

Output : ABC10

(2) length function

length ()

↳ returns length of string.

To deal with UNICODE :-
char char [ ] , int start , int num
Code codePoints [ ] .

(3) Character Extraction

→ charAt ( ) ;
// parameter is index & is called with a defined String.
char c = S.charAt (2) ;

→ getChars ( ) ;
parameters :-
1) int source → (starting source index)
2) int target → (copy till where)
3) target array - (new array where value is to be copied.)
4) int target start → Start copying from which index.

eg → getChars (1 , 2, [ ], 1)

→ ~~get byte~~ getbyte ( ) ;

(4) String comparison :-
(case Sensitive)
→ equals ( ).
→ equalsIgnoreCase ( ) ; // (not case sensitive)
both return boolean Value.

equalsIgnoreCase
↳ has both ASCII Values, if content is same irrespective, then returns true.

```
S = "ABC";
d = "DEF";
t = "abc";
P = "ABC";
```

(Here we are not comparing obj. but are checking its content)

```
s.equals (d)              // false
s.equals (t)              // false
s.equals (P)              // True
s.equalsIgnoreCase (t)    // True.
```

(Case by case comparison)

⇒ equals → to check whether content of both object is same.

⇒ == → to check whether 2 objects are same or not.

eg →
```
char [] b = {     };
String s = new String (b);
String D = new String (s);
if (s == D)   // false
```
Here both obj. are different.

[only true when use compare S and S]

If (s.equals (D)) . // True since content is same.

⇒ regionMatches ( ) →
↓ similar function.

starts with ( )
ends with ( )
} provide a searching Mechanism

String S = "ABCDEF" ; // check whether string starts with some particular char or ends with.

S. startsWith ("AB") ; // True
S. startsWith ("B") ; // false.

S. endsWith ("EF") ; } // true.
S. endsWith ("DEF") ;

⇒ CompareTo () ;
compares acc. to length and returns an integer value. CompareTo() checks the length b/w the 2 strings and returns the result as an integer value. If invoking string is less than next string, value is < 0 (-ve). If invoking string > next string then value is > 0 (+ve). If 2 strings are equal, value will be 0.

int compareTo (String str)
↓
compareTo ("ABC") // can't work as object of String class is req.

s1.compareTo (s1) ;
invoking obj. ↳ object to be compared

(5) Searching Strings →
→ searches from left to right.
↳ indexOf () ;
→ lastIndexOf () ; // Searches from right to left.

Date
Page

⇒ ~~SubString~~

⇒ SubString ( ) ;
This method returns a sub part from a given string.

SubString (int start)
subString (int start, int last)
(index of given string)

eg → can print values in reverse order also.
S.SubString (2) ; → after this index. 2
all content would be returned as Substring.

int start →
It is a no. but for an array of chars, start is treated as index.

⇒ Concatenation
S.concat ( ) ;

⇒ To Replace ;
replace (char to be changed, char)
to change value at a particular index.

S = "ABCDEF" ; → (string length should match)
S.replace ('A', 'T') ;
returns a string.

if at string, 'A' is present at more than 1 place, then replace() causes all A's to get replaced with 'T'.

⇒ trim()
to remove whitespaces.
s.trim();

- StringBuffer class
- StringBuilder class

> not immutable
i.e. original content of the object can be changed unlike String class.

⇒ Difference

★ append()
- This func. causes changes in one of the strings.

- S.append(d);
// changes occurs in S only
d.append(s);
// changes occur in d only.

★ Concat()
- This function creates 3rd string and no change in present string

- s = "ABC";
d = "DEF";
if we concatenate both of them, they get stored in 3rd string

APPLETS (CONTINUED)

⇒ At time of execution, we will call

>> appletviewer — abc
(creates java.exe) — class where applet is created.

* HTML → used to create Static Webpages (content doesn't change)
used to run code on browser

```
<html>
<applet code = "filename.class"
    width = ____
    height = ____ >
</applet> </html>
```

width & height — compulsory to define

When we compile our code using javac, it creates a .class file & then this is used to run code on browser.

<applet → using this .class file gets embedded into HTML.

Save: abc.html

Graphics class

↓ (Subclass)

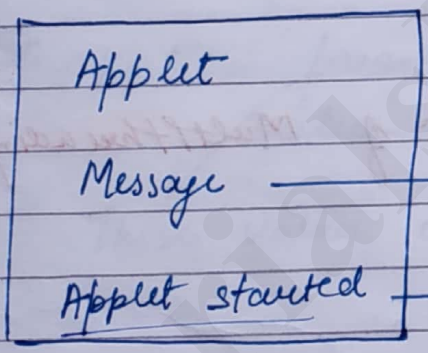
color

Methods
- setBackground
- setForeground
- color getBackground
- Color getForeground

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

class abc extends Applet
{
    public void init()
    {
        SetBackground(color.red);
    }

    public void paint(Graphics g)
    {
```

→ it is the display method of your program. Implementation not necessary if an output is needed.

Output



⇒ How to change status and display our own message.

defined method to show status →

```
showStatus();  // like a display method
```

defined in Applet class.

```
public void paint ( Graphics g )
{
    g.drawString (" ", x, y );
    ShowStatus (" ");
}
```

Here, it is not ~~necessary~~ accessed by object as ShowStatus is not defined in Graphics class but in Applet class. So methods of Applet class do not need to be accessed by object.

⇒ To rotate the message using methods defined in Applet class.

update ();
repaint ();
↳ (using concept of Multithreading)

Program

```
import java.awt.*;
import java.applet.*;
/* <applet code = "Banner"
    width = 109
    height = 120 >
</applet> */
```

To access character in Scanner class.

Sc.object.charAt ( );
↳ index.

Banner → using thread ⇒ main Program Classdream Page Banner

Note → If the user doesn't close or minimise the window, stop method won't be called.

no parameters — void repaint()
void repaint(, , , ,)
↳ with parameters
[to repaint a specify thing]

HTML Applet Tags (upto 764)

< applet code

↓
to specify class name.
various other tags can be used

< applet param = " "
value = " "

↓
These values can be passed to class file.

EVENT HANDLING

(change in state of an object)
It is an object defined in Java

→ Key Event
handling Keys of Keyboard (press or release)

Key Press
Key Release
Key Type (display of message while typing)

- Mouse Event
- Action Event

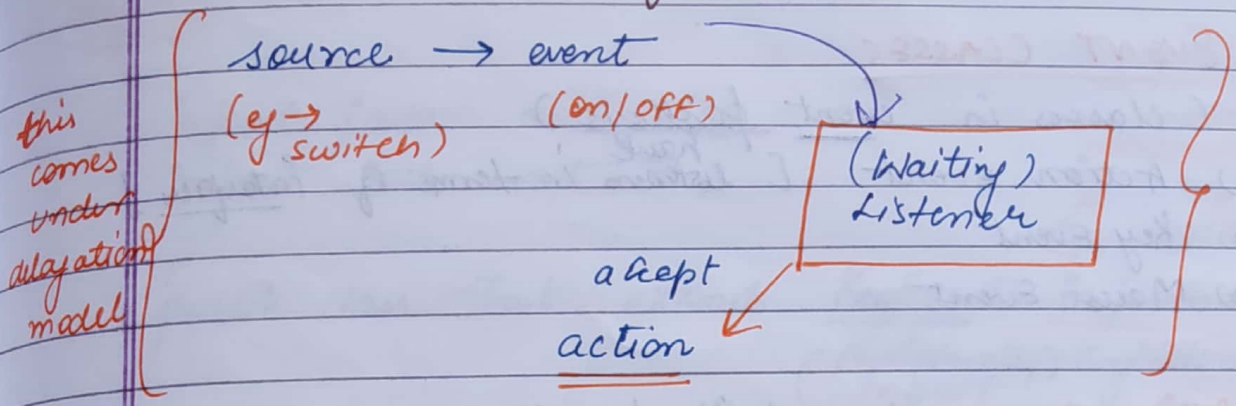
⇒ Event handling through
Delegation Model [handles change of state of object]

- source (generates an event)
obj. whose state is to be change.
(eg → mouse, screen, Keyboard)

- Listener
(Sensing model) ↙ sensor which checks state has been changed.
(available in form of interfaces)

(1) waits until event hasn't been generated
(2) Acc. to event generated, it takes some action.

Delegation Model



⇒ 2 packages responsible for handling :-

- awt → bcoz there is use of applets
- event → for event handling.

⇒ EVENT →
It is an object that describes a state change in a source.

⇒ EVENT SOURCES →
source is an object that generates an event. This occur when internal state of that object changes. Whenever a source generates an event, it must register listeners in order to receive notification.

⇒ EVENT LISTENERS :-
It is an object that is notified when an event occurs. It must have 2 things —
(1) associated with one or more sources.
(2) should have methods to receive & process those notifications.

⇒ EVENT CLASSES

(classes in event package have listeners in terms of interfaces)

(1) Action Event
(2) Key Event
(3) Mouse Event

- Action Event class will handle whenever a button is pressed, a list item is double clicked or a menu item is selected.

- Key Event class will handle whenever a key is pressed (a keyboard input occurs). This involves 3 events - pressing, releasing, typing.

- Mouse Event class will handle any changes that occur in terms of movement, pressing or clicking. → Five Methods

- Events are handled methods. i.e. (override) (because of interfaces)
  ↳ It is compulsory to write all the methods defined in these classes.

⇒ Interfaces available for

- For Action Event class, interface defined is ActionListener
- For Key Event, we have KeyListener.
- For Mouse Event, we have MouseListeners.

Classdream Date ___ Page ___

Program

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

public class Test extends KeyEvent implements KeyListener {

    String msg = "  ";
    public void init ( ) .
    {
        addKeyListener (this);
    }

    public void keyPressed (KeyEvent k)
    {
        showstatus ("KeyPressed");
    }

    public void keyReleased (KeyEvent k)
    {
        showstatus ("KeyReleased");
    }
    public void keyTyped (KeyEvent k)
    {
        msg = msg + k.getKeyChar ( );
        repaint ( );
    }
    public void paint (Graphics g)
    {
        g.drawString (msg, 20, 30);
    }
```

Annotations:
- KeyListener interface: to handle multiple events
- KeyEvent: (functionally) inbuilt Applet class
- addKeyListener: initialization of listener; // embed / register key listener with source; {to register listeners to a source}
- KeyEvent k: reference of class
- getKeyChar: to fetch character from reference
- msg in drawString: from keyTyped method i.e. while typing.

Date
Page

⇒ Methods available for Mouse Event.

The methods available in MouseListener Interface are →

(1) MouseClicked
(2) MousePressed
(3) MouseReleased
(4) mouseEntered
(5) mouseExited

} compulsory to implements all these methods

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

public class abc extends MouseEvent implements
                                    MouseListener {
    String msg = " ";
    int x, y = 0;

public void init( )
    {
        addMouseListener (this);
    }

public void MouseClicked (MouseEvent m)
    {
        x = 0; y = 10;
    msg = "MouseClicked";
        repaint( );  ⟹ // necessary to call
                          after every method
    }
```

WINDOW EVENT CLASS

(handles changes in Applet Window)

• Events

There are 10 events associated with it.

(1) Window_Activated
This event is used when the window gets activated

(2) Window_Closed
This event is used when the window has been closed.

(3) Window_Closing [when user request for closing]
Time gap b/w closing & closed state is this

(4) Window_Deactivated
This event is used when the window has been deactivated. (when destroy method is called)

(5) Window_state_changed.
This event captures the state of the window changed. [switching b/w 2 windows]

(6) Window_Open
(7) Window_Lost_focus
(8) Window_gain_focus.
(9) window_Deconified. } additional Events
(10) Window_Iconified

Adapter classes

- Adapter class provides an empty implementation of all methods in an Event listener interface.
(This class provides default implementation to those methods which are not in use)

It also simplifies creation of event handlers.
The adapter class is defined in ~~Java.awt.~~ Java.awt.event

List of adapter classes

for ~~Key~~ KeyListener interface, a class defined is ~~Key Ad~~ KeyAdapter

for MouseListener, a class defined is MouseAdapter. [for MouseMotionListener as well]

for WindowListener, we have WindowAdapter.
(for all classes starting with window)

Program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
<
/>
Public class AdapterDemo extends Applet // creating Applet
{
```

```
Public void init()
{
    addMouseListener(new MyMouseAdapter(this));
}
}
```

// compulsory stmt.

(passing object) } adapter class defined in our program

```
class MyMouseAdapter extends MouseAdapter {
```

* In init refers to current class.

```
AdapterDemo adapterDemo;
```

// defines reference of previous class i.e. AdapterDemo

```
Public MyMouseAdapter(AdapterDemo adapterDemo);
{
this.adapterDemo = adapterDemo;
}
```

```
Public void mouseClicked(MouseEvent me)
{
adapterDemo.showStatus("Mouse clicked");
}
}
```

Same code

[Provides Implementation to all methods inside MyMouseAdapter class)

⇒ INNER CLASS

↓ class within a class is called Inner class.

→ it is not req. to create Object of Inner object. We can directly refer to Inner class with object of Outer class.

⇒
```
import java.applet.*;
import java.awt.event.*;
<applet
                    >
                    </applet>

public class abc extends Applet {
    public void init()
    {
    add MouseListener (new MyMouseAdapter()); }

class MyMouseAdapter extends MouseAdapter {
    public void mousePressed (MouseEvent m);
    showstatus (" ");
    }
}
```

// class MouseAdapter is inner class of class abc
∴ not need to define object of class abc

~~Synonyms~~

Imp

Anonymous Inner class → is a one that is not assigned a name

~~It is a class~~

Anonymous Inner class is a class in which everything is defined within a same class only and has access of all the variables and methods within that particular class. We need not to define the class name for the corresponding value.

⇒ same code uptil init()

~~Public void mouseDragged~~

```
public class abc extends Applet
{
    public void init()
    {
        addMouseListener(new MouseAdapter())
        {
            public void mousePressed(MouseEvent m);
            showStatus("    ");
        }
    }
}
```

JVM this will automatically convert the class (defined at backend) as inner class. Thus it will create object for that class whose name is available at backend.

(JVM will not recognize this and gets error)

Self made classes can not be ~~forfeited~~ interpreted as anonymous inner class. All classes must be inbuilt like MouseAdapter class.

Pg →

Topics from Book

→ 2 Event Handling Mechanisms
→ Delegation Event model
→ Events / Event sources / event listeners.
→ Event classes [Table]
(Key / Action / Mouse / window)
→ EventListener Interfaces
→ MouseEvent & KeyEvent Handling.

Chapter - 25

⇒ Introduction of awt with help of Graphics and Windows (ch-25)

→ Applet is a subclass of Window class.

awt → abstract Window Tool.
This is the largest package available in Java package hierarchy.
java.awt.*

- ~~awt Event~~
- awt EventMulticaster

~~for~~

⇒ java.awt.*;

⇒ Classes available in awt (table) API

Supper class → component class
container [provides a structure but not menus]
Window
Panel
applet
frame

→ awt & Swing
↓ (deal with appearance) ↓ (deals with frame)

JavaFx
↳ (modified version of swings)

⇒ Component class
is an abstract class that encapsulates all the attributes of a visual component except of menus, and user interactive Elements / Programs. provides a structure but cannot create menus.
∴ We have container.

⇒ Container
is a component that contains other components like buttons, text fields, labels are all available in container class.
the classes that extends container are known as containers such as frame, window, dialogue & panel

⇒ Panel
it is a container that doesn't contain title bar and menu bars. Other than that it have all the components defined.

⇒ **Window**

is the container that have no borders and menu bar.

⇒ **Frame**

is the container that contains the title bar as well as the menu bar. Also, it have all the other defined components.

advanced version ⇒ **Canvas**

(Empty frame)

⇒ **Methods**

There are several methods for Window class to implement a frame

~~Setting the windows~~

(1) Setting the windows dimension.

(i) void setSize (int newWidth, int newHeight)

(ii) void setSize (Dimension newSize)

↓

a different class.

(2) Fetch the size of the window.

⇒ class.getSize ( ) → returns dimension.

Imp (3) Hiding & showing the window

⇒

↙ void setVisible (boolean flag).

(to display window screen.)

returns (true / false)

window is visible → not visible.

(4) Setting a Window Title
→ void setTitle (String newTitle)

(5) closing a frame Window. → (We can close the Window frame by setting the visibility false)

WindowClosing ()
↓
(memory gets deleted)

⇒ (memory doesn't gets flush out)

Program
⇒ Creating a frame Window using an applet.

```
import java.awt.*;
import java.awt.event.*;
import javax.applet.*;
<applet >
  * </applet>.

class SampleFrame extends Frame
{
SampleFrame ( String title)  // constructor
  {
    super(title);
  }

MyWindowAdapter adapter = new MyWindowAdapter ();
addWindowListener (adapter); } }

public void paint ( Graphics g )
  { g.drawString (" FrameWindow ", 10, 20);
    }
}
```

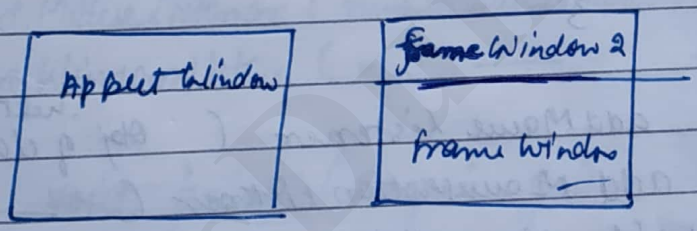
```
class MyWindowAdapter extends WindowAdapter
  {

  SampleFrame sampleframe ;
  public MyWindowAdapter ( sampleframe
                                sampleframe ) ;
    {
      this.sampleframe = sampleframe ;
    }

  public void windowClosing ( WindowEvent w) ;
    {
    sampleframe.setVisible (false)
      {
      }

  Public class AF extends Applet
      {
      frame F ;  // Reference of class frame.
      Public void init() ;
    {
f = new SampleFrame ("Frame Window 2") ;
      f.setSize (250, 250) ;
      f.setVisible (true) ; } }

  Public void start ()
      {
      F.setVisible (true) ;
      }
```

MVC (model view controller)

Classdream
Date
Page

```
public void stop ()
    {
      f.setVisible (false);
    }

public void Paint(Graphics g)
    {
      g.drawString ("Applet", 10, 20);
    }
}
```

Output



Event Handling

⇒ MOUSEMOTIONLISTENER

- MouseDragged
- MouseMoved

⇒ EVENTS INSIDE A FRAME WINDOW

SampleFrame (String title)
~~add Mou~~ {

}

~~add~~ ⇒ addMouseListener (Obj. of next class)
addMouseMotionListener (———)
public void MouseClicked (MouseEvent m)
{ int x=30; y=40; ; String msg = "Hello"; }

⇒ To fetch any defined coordinate inside an Event method can be done with the help of

⇒ getX() , getY()
→ Where this defines the original coordinate inside a constructor.

(Ans) :- Points of current class of class R

If we make any changes in other class, changes get reflected back in previous class

⇒
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class AppWindow extends Frame
{
  String Keymsg = "Test";
  String Mousemsg = "
  int mouseY = 30; mouseX = 40;
  public AppWindow()
  {
    addKeyListener(new MyKeyAdapter(this));
    addMouseListener(new MyMouseAdapter(this));
    addWindowListener(new ——— ———);

  public void paint(Graphics g)
  {
    g.drawString(Keymsg, 10, 40);
    g.drawString(Mousemsg, mouseX, mouseY);
```

AWT WITH CONTROLS

- Double clicking a button ✓
  - ↳ Action Event

components present in terms of classes:
- Buttons
- checkbox
- Radio button
- Dropdown menu

↳ layout

⇒ label is basically a reference through which we can identify the things.
- ↳ subclasses of above components

⇒ Adding & Removing Controls

1. Component add(component ref) ref of components classes
   - Component ↳ class
   - { main method that adds reference }

2. To remove
   - void remove (Component ref)

⇒ Label

is an object of type label with 3 constructors

By default left aligned
1) Label ( ) // creates blank label ✓
2) Label(String str) // string prints on label
3) Label (String str, int x) // String prints on position x

This component is also known as a passive control. (static not changeable)
Because, they do not support any interaction with the user.

// By defaults constants are defined in capital letters

Date
Page

To set the alignment of a lable
3 constants are defined.
1) LABELLEFT
2) LABELRIGHT
3) LABELCENTRE

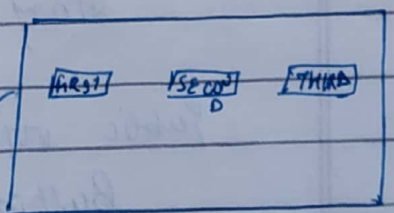
To set the alignment, methods defined is
void setAlignment (int x);

int getAlignment (); // to fetch alignment

Lable ref. setText (); // to set alignment using ref. of label class.

→ "subclass of component

class ABC extends Applet {
public void init ()
{
label a = new label ("First");
label b = new label ("Second");
label c = new label ("third");
add (a);
add (b);
add (c);
}
}



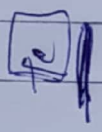
Alignment depends on calling of add

Action Event class is responsible for Implementation

Date
Page

Adding Button Controls

Button is active control. [state changes on the basis of user interaction]



// Import stmts.

[Subclass of component]

public class Demo extends Applet {

String msg = " ";

public void int ( ) {

Button b = new Button ("colour changed");

add (b);

}

}}

(Only Button will be created not colour change)

for Event Handling,

Public class Demo extends Applet implements ActionListner {

String msg = " ";

Public void init ( ) {

Button b = new Button ("color changed");

add (b);

b.addActionListner (this);

}

↗ default method

public void actionPerformed (ActionEvent a)

{ String str = a.getActionCommand ( ); // returns reference of a particular command generated.

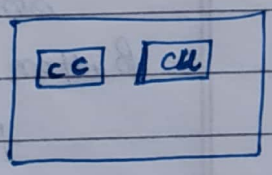
compares value

for the ActionListener Interface, ~~#~~ ActionPerformed() is defined to handle all the actions commands.
Action Event class, Refrence of Action Event class will be passed as a parameter to this method.

To fetch the corresponding labels getActionCommand() ~~is defined~~ will be called from the ActionEvent object.

Continued Program

```
If (str.equals("--:"));   // either we can use refrence of button or string used for button
{
msg = "you have pressed cc";
// repaint(); {
        } // end of actionPerformed method.
```




```
public void paint(Graphics g)
    {
    g.drawString(msg, 10, 10);
```

after message has been given value, we can call SetBackground to change color of Background and call repaint() ~~&~~

Note ⇒

Button b = new Button ("color changed");
add (b);

this can be written as,

add (new Button ("color changed"));

{but, we cannot handle event through this: we need reference variable only.}

⇒ To determine which button has been pressed, there is a method defined called getSource() which will compare the objects from a list.

⇒

```
public class Demo extends Applet implements
                                    ActionListener {
    String msg = "  ";
Button bList[] = new Button[3];
    public void init()
    {
Button a = new Button("Yes");
"      b = "      "  ("No");
"      c = "      "  ("May be");
blist[0] = (Button) add(a); // (explicitly add Button)
blist[1] = (Button) add(b);
blist[2] = " add(c);
```

(not in a loop bcoz value is different for each button)

```
for(int i = 0; i<3 ; i++)
    { blist[i].addActionlistener(this); } }
```

```
public void actionPerformed (ActionEvent ae)
{
  for (int i=0; i<3; i++)
    { if ( ae.getSource() == blist[i] )
{
  msg = "You Pressed" + blist[i].getLabel()
        repaint();
}
}
}
```

↳ object comparison

↳ fetch the name of button

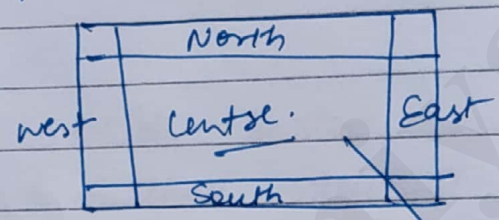
⇒ Layout

(to handle layouts we have layout Manjers)

There are five (5) different types of layout available in java.awt library.

(1) Border layout

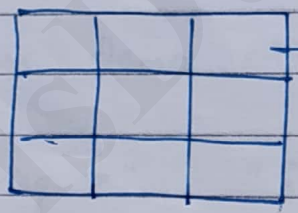
W, E, N, S & Centre



main portion is Centre

(2) Grid layout

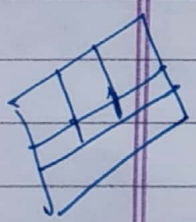
3 X 3
(Columns) (rows)



→ table structure (3x3)

divides the frame in the form of a grid containing distributive cell.
The order of placement is directly dependent on the order of in which the components are added to the frame.

(3) Gridbag

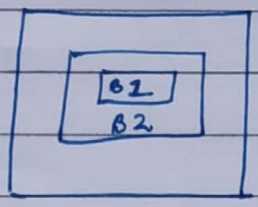


It provides the flexibility to choose the exact location of the component in a grid with the help of row span, column span, vertical gap & horizontal

Span → pointing out specific location.

(4) Card layout

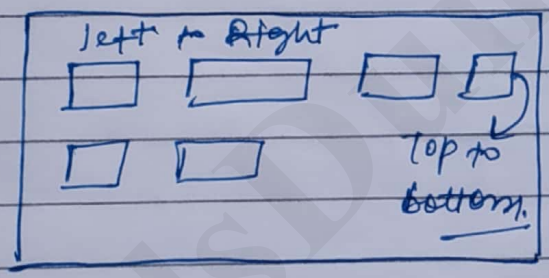
⇒ It allows the components to stay over one another and switch ~~between~~ any component to the front as per the requirement. (works as stack.)



(5)

⇒ ~~Flow~~ Flow layout

This Layout is known as default layout str. components get arranged from Left to Right & Top to bottom.



⇒ getActionCommand

name of label should be different (working on string)

getSource

Same name button can be used. (working on objects)