17/01/25

Page No.
Date: / /

OBJECT ORIENTED PROGRAMMING

What is a Computer Program?

It is a set of inst^{n}s which is used to perform certain task.

Characteristics of a Good Program

1) Readability
2) Maintainability (Easy to update)
3) Efficiency
4) Reliability (Every time output is correct
5) Portability
6) Reusability

Programming Paradigm (Ways)

- Imperative (How)
  - Eg. C, C++, JAVA
    - Procedural: Algorithm + Data; Control, Sequence, Iteration structures
    - Object oriented: Object + Data
- Declarative (What)
  - Eg. SQL
    - Functional: func^{n} com-position func^{n}; Eg. LISP, Haskal
    - Logical: Facts + Rules; PROLOG

Page No.
Date: / /

Roots of OOP

1) Procedural (control, sequence, iteration)
2) Modular Programming (func^n's / Modularity)
3) Object Based (supports Encapsulation, Abstraction)
4) Object Oriented (Inheritance, Polymorphism)

~~22/01/25~~

1) **Class**

=> Collection of characteristics & behaviour.

=> Collection of similar properties with diff. behaviour.

=> Collection of data member (properties of a class) & member methods (operations)

2) **Object**

Instance of a class / Real world entity.

3) **Modularity**

Various func^n's used in a program.

Page No.
Date: / /

4) Abstraction

Hiding implementing details & only showing essential features to the user.

5) Encapsulation

Bundling data & methods into a single unit. called a class.

6) Inheritance

One class is allowed to inherit the features of another class.

Creating new classes based on existing classes.

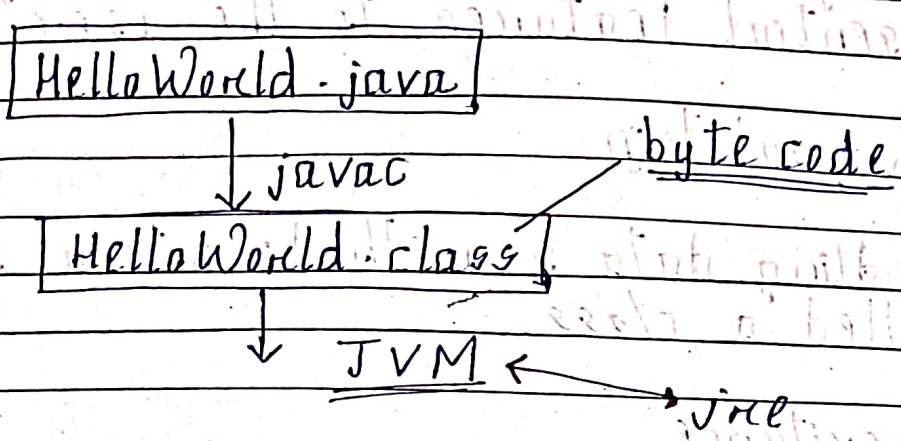
7) Polymorphism

Poly ⟶ many Morphism ⟶ form

Page No.
Date: / /

INTRODUCTION TO JAVA

1) JAVA is portable / Architecture Neutral



JVM → JAVA Virtual Machine
jre → JAVA Runtime Environment

When you compile JAVA code, it is translated into an intermediate language called bytecode, which is interpreted by JVM. Since bytecode is platform-independent, the same compiled JAVA program can be run in any platform with a compatible JVM installed

2) JAVA is interpreted (both interpreter & compiler)

3) JAVA is Robust (Follows 7 OOP concepts & is efficient)

Can the main have int ?(Lab)

Page No.
Date: / /

First JAVA Program

```
class HelloWorld
{
  public static void main (string a[])
  {
    System.out.println ("Hello World");
  }
}
```

Run :-

```
>javac HelloWorld.java
>java HelloWorld
```

public :- It is the access modifier responsible for mentioning who can access the method & what is the limit. It is responsible for making the main func available It is made public so that JVM can access it outside the class as it is not present in the current class.

static :- It is a keyword used so that we can use the element without initiating the class so to avoid the unnecessary allocation of the memory.

void :- It is a keyword used to specify that a method doesn't return anything. As the main func doesn't return anything we use void.

Page No.
Date: / /

main : It represents that the func" declared is the main func". It helps JVM to identify that the declared func" is the main func".

string arg[] : It is a command line argument.

println() : It displays the output on the screen

System.in → keyboard
System.out → screen

Out : Object from printstream class.

24/01/25

Data Types & Variables

Data Types

- Integers
  - Byte 8 bits
  - Short 16 bits
  - int 32 bits
  - long 64 bits
- Float
  - Float 32 bits
  - Double 64 bits
- String
  - Char 16 bits
- Boolean
  - True
  - False

* ASSIGNMENT (To Check in LAB)

```
int x = 0;
if (x)
    system.out.println ("Hello");
else
    system.out.println ("Bye");
```

Page No.
Date: / /

Q- Write a JAVA program which finds out the total dist. travel by a light in 1000 days.

Speed of light = $3 \times 10^8 \, ms^{-1}$

1 day → 24 hr → $24 \times 3600 = 86400$ s

For 1000 days → 8.64×10^7 s

Dist. = S × T

$= 25.92 \times 10^5$ km

```
class CircleApp
{
public static void main (String arg [])
{
  int days = 1000;
  long speed = 3000000,00;
  long sec = 24*60*60*days;
  long dist = speed * sec;
  System.out.println(" Dist. travelled : "+dist);
}
}
```

Variables

Syntax : datatype variable-name;

eg. int x;
char a;
float z;

```
{
  int x = 10;
  printf ("%d", x);
  {
    int y = 20;
    printf ("%d %d", x, y);
  }
  printf ("%d %d", x, y);
}
```

scope

Error since the scope of the variable is not defined.

Rules for Variable Names

→ Must be started with alphabet or underscore.
→ No special characters are allowed except underscore.
→ No keyword can be used as variable.
→ Should be meaningful.

27/01/25

Page No.
Date: / /

Type Conversion & Casting

byte → short → int → long → double
char → int

```
byte b = 15
byte c = b * 2      Error  → typecasting
byte c = (byte)(b * 2)
```

Control Structures

1) if

Syntax

```
if (logical expression)
{
    statements ;
    ...
    ...
}
```

if-else

Syntax

```
if (expression)
{
    statement ;
}
else
{
    statement ;
}
```

Page No.
Date: / /

Operators

1) Arithmetic Operators :-
   +, -, %, *, /, ++, --, +=, -=

2) Logical Operators :-
   &&, ||, !, &, |, ==, ?:

3) Relational Operators :-
   <, >, <=, >=, ==, !=

4) Bitwise Operators :-
   <<, >>, ~, ^, V

5) Assignment Operators :-
   =

```
int x, y, z
x = y = z = 100;
```

Nested if

Syntax

```
if (expression)
{
    if (expression)
    {
        :
    }
}
```

Page No.
Date: / /

if-else ladder

Syntax

```
if (expression)
{
    statement ;
}
else if (expression) ;
{
    :
}
else
{
    statement ;
}
```

Switch

Syntax

```
switch (expression)
{
  case
  {
     statement ;
     break ;
  }
     :
  default :
     statement ;
}
```

Page No.
Date: / /

Iterative Statements

1) For Loop

Syntax

```
for (initialisation ; cond^n ; incre/decre)
{
    ____
    ____
    ____
    ____
}
```

NOTE When in for loop, we use double semicolon. Then the output will be infinite.

2) While loop

Syntax

```
initialisation ;
while (cond^n)
{
    ____
    ____
    ____
    incre/decre ;
}
```

Difference bet^n For & While Loop

A "for" loop is used when you know exactly how many times you want to repeat a block of code while a "while" loop continues to execute as long as a specific cond^n remains true.

29/01/25

Page No.
Date: / /

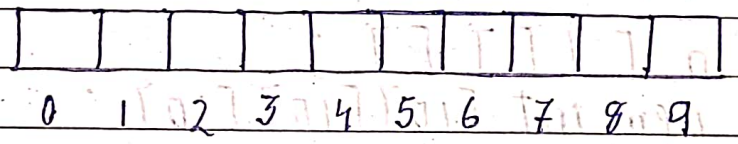
Arrays

It is a collection of similar data types having index from 0 to size - 1.

It is of 2 types :
i) 1 - D
ii) 2 - D
iii) Multi - dimension

Syntax datatype variable [ ] = new datatype [size]

Eg. int a [ ] = new int [10] ;



1D

Q= Write a program which accepts 5 integers with command line argument & store it in an array & find the ~~array~~ avg.

```
Class Array
{
  public static void main (String arg [])
  {
    int i, sum = 0;
    int a[];
    a = new int [5];
```

```
for (i = 0 ; i < 5 ; i++)
{
  a[i] = Integer.parseInt (arg[i]);
   sum = sum + a[i] ;
}

System.out.println ("Avg : "+(sum/5));
}
}
```

Multi Dimensional Array

```
int a[][] ;
a = new int [2][2] ;

int a[][][] ;
a = new int [10][10][10] ;
```

2-D

Q= Write a program to add ~~2×2 matrix~~ two 2×2 matrix

```
Class MatAdd
{
public static void main (String arg[])
{
  int i, j ;
int a[] = {{10, 20}, {30, 40}};
int b[] = {{40, 50}, {60, 70}};
int c[][] ;
 c = new c[2][2] ;
```

```
for (i = 0 ; i < 2 ; i++)
{
  for (j = 0 ; j < 2 ; j++)
  {
    c[ ][ ] = a[i][j] + b[i][j] ;
  }
}

System.out.println ("Sum :") ;
for (i = 0 ; i < 2 ; i++)
{
  for (j = 0 ; j < 2 ; j++)
  {
    System.out.println (c[i][j] + "  ") ;
  }
}
}
```

Length of Array

For 1 D ⟶ a.length

For 2 D ⟶ a.length (for rows)
a[0].length (for columns)

31/01/25

Page No.
Date: / /

INTRODUCTION TO CLASSES

Syntax class Class-Name
{
datatype variablename ;
:
}

```
class Box
{
 int len, br, ht ;
}

class BoxDemo
{
 public static void main (String arg[])
 {
  Box b1 = new Box() ;
```

To allocate memory we declare object of box class.

```
  b1. length = 10 ;
 System.out.println (b1.length + b1.breadth +
                     b1.height) ;
 }
}
```

05/02/25

Page No.
Date: / /

Constructors

=> It is a member method of the class.

=> It is a method which is named same as a class name.

=> It has no datatype.

=> Always execute automatically when the corresponding object of the class is created.

=> It is of 2 types :-
i) Default (Without parameter)
ii) Parameterised (With parameter)

```
class Box
{
  int l, b, h;
    Box ()
    {
       l = 0;
       b = 0;
       h = 0;
    }

    Box (int l1, int b1, int h1)
    {
       l = l1;
       b = b1;
       h = h1;
    }
}
```

Page No.
Date: / /

```
class BoxDemo
{
  public static void main (String arg[])
  {
    Box b1 = new Box();
    Box b2 = new Box(10,20,30);
    System.out.println(b1.l + b1.h + b1.b);
    System.out.println(b2.l + b2.b + b2.h);
  }
}
```

Output :- 0 0 0
10 20 30

- **this keyword**

=> It refers to the current object in a method or constructor.

=> "This" keyword is used to distinguish bet" data member & parameter of method

```
class Box
{
  int l, b, h;
    Box (int l, int b, int h)
    {
  This.l = 10;
  This.b = 20;
  This.h = 30;
    }
```

Page No.
Date: / /

Q- WAP to implement class stack and test the class.

```
class Stack
{
  int top, data[];
   stack() // default constructor
   {
     top = -1;
     data = new int[5]; // data.length = 5
   }
   boolean isEmpty()
   {
     if (top == -1)
        return true;
     else
        return false;
   }
   boolean isFull()
   {
     if (top == data.length - 1)
        return true;
     else
        return false;
   }
   void push (int element)
   {
     if (! isFull())
     {
        data[++top] = element;
     }
   }
```

```
~~void~~int pop()
{
    if (isEmpty()== false)
    {
        return (data[--top]);
    }
}
}

class StackDemo
{
 public static void main(String.arg[])
 {
    Stack s1 = new Stack();
    s1.push(10);
    s1.push(20);
    s1.push(30);
    System.out.println("Popped element:"+s1.
                                        pop()
 }
}

void peek()
{
  System.out.println("Top element:"+data[top]);
}
```

07/02/25

Page No.
Date: / /

- **Garbage Collection**

=> JAVA garbage collection is an automatic process that manages memory in the heap.

=> It identifies which objects are referenced & which are not referenced.

=> Unreferenced objects can be deleted to free up memory.

=> It makes JAVA memory efficient.

- **Func^n Overloading (Polymorphism)**

=> When more than one func^{n}s are defined on the same name with diff. no. of parameters. and diff. type of ~~parameters~~. return type.

```
class Number
{
  int a, b;
  float c, d;
  Number ()
  {
     a = 0 ;
     b = 0 ;
     c = 0 ;
     d = 0 ;
  }
```

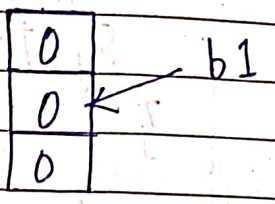
```
void add (int a, int b)
                  └→ formal parameters
{
  This.a = a;
  This.b = b;
  System.out.println("Sum :" + (This.a + This.b));
}

void add (float c, float d)
                  └→ formal parameters
{
  This.c = c;
  This.d = d;
  System.out.println("Sum :" + (This.c + This.d));
}
}

class NumberDemo
{
public static void main (String arg[])
{
  Number n = new Number();
  n.add (10, 20);
  n.add (10.5, 20.5);
                  └→ actual parameters
}
}
```

Constructor Overloading

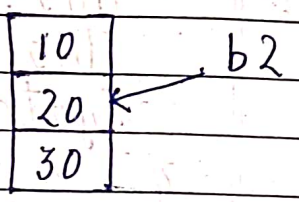
In a class, we can have no. of constructors having diff. type & diff. no. of parameters.

Page No.
Date: / /

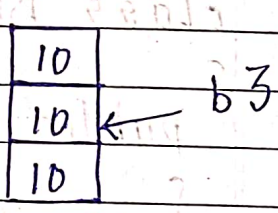
```
class Box
{
  int l, b, h;
  Box ()
  {
    l = 0 , b = 0, h = 0;
  }
```




```
  Box (int l, int b, int h)
  {
    This.l = l;
    This.b = b;
    This.h = h;
  }
```




```
  Box (int s)
  {
    l = b = h = s;
  }
}
```




```
class BoxDemo
{
  public static void main (String arg [])
  {
    Box b1 = new Box ();
    Box b2 = new Box (10, 20, 30);
    Box b3 = new Box (10);
  }
}
```

10/02/25

Page No.
Date: / /

Object as Parameter

```
class Box
{
  int l, b, h;
  int volume (int l, int b, int h)
  {
    This.l = l;
    This.b = b;
    This.h = h;
    return (l*b*h);
  }
}

class BoxDemo
{
  public static void main (String arg[])
  {
    Box b = new Box();
    int vol = b.volume (10, 20, 30);
  }
}
```

call by value

OR

By using object as parameter

Page No.
Date: / /

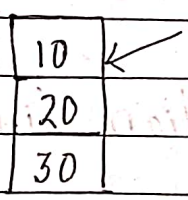
```
class Box
{
  int l, b, h;
  Box ()
  {
    l = 10;
    b = 20;
    h = 30;
  }
  int volume (Box b2)
  {
    l = b1.l;
    b = b1.b;
    h = b1.h;
    return (l*b*h);
  }
}

class BoxDemo
{
  public static void main (String arg[])
  {
    Box b1 = new Box ();
    Box b2 = new Box ();
    int vol = b1.volume (b2);
  }
}
```

call by reference

Page No.
Date: / /

By passing object in constructor

```
class Box
{
  int l, b, h;
  Box (Box b2) ← Copy Constructor
  {
     l = b2.l;
     b = b2.b;
     h = b2.h;
  }

  Box ()  ← Default Constructor
  {
     l = 10;
     b = 20;
     h = 30;
  }

  int volume ()
  {
     return (l*b*h);
  }
}

class BoxDemo
{
  public static void main (String arg[])
  {
     Box b1 = new Box ();
     Box b2 = new Box (b1);
     b1.volume ();
  }
}
```



Page No.
Date: / /

Q: WAP to implement the class Fraction which has data members numerator & denominator. Implement the following operations.
i) Addition
ii) Subtraction
iii) Multiplication

```
class Fraction
{
  int num, denum;
  Fraction ()
  {
    num = 10;
    denum = 10;
  }

  Fraction (int n, int d)
  {
    num = n;
    denum = d;
  }

  Fraction Add (Fraction f) //Returning an object
  {
    Fraction f3 = new Fraction ();
    f3.num = (this.num * f.denum) +
             (f.num * this.denum);
    f3.denum = this.denum * f.denum;
    return (f3);
  }
```

Page No.
Date: / /

```
Fraction subtract(Fraction f)
{
  Fraction f4 = new Fraction();
  f4.num = (this.num * f.denum) - (f.num *
           this.denum);
  f4.denum = this.denum * f.denum;
  return (f4);
}
Fraction multiply(Fraction f)
{
  Fraction f5 = new Fraction();
  f5.num = (this.num * f.num);
  f5.denum = this.denum * f.denum;
  return (f5);
}
}
class fraction
{
  public static void main(String arg[])
  {
    Fraction f1 = new Fraction(2,3);
    Fraction f2 = new Fraction(3,4);
    fraction f3, f4, f5;
    f3 = f1.add(f2);
    f4 = f1.subtract(f2);
    f5 = f1.multiply(f2);
  }
}
```

19/02/25

Page No.
Date: / /

Recursion

=> It is the process of defining a "func" in terms of itself.

=> A method "func" that calls itself is called recursion.

// Factorial of a number

```
class Factorial
{
  int fact (n)
  {
    if (n == 1)
      return 1;
    else
      return (n * fact (n-1));
  }
}

class FactorialDemo
{
  public static void main (String arg[])
  {
    Factorial f = new Factorial ();
    System.out.println ("Factorial of 3 : "f.fact
                                          (3))
  }
}
```

Page No.
Date: / /

NOTE Programs in Recursion
1) Fibonacci
2) Binary Search
3) Quick sort
4) Tower of Hanoi
5) GCD of 2 no.s

Access Control

Access control helps to restrict the scope of a class.

It is of 4 types :- 1) public
2) private
3) protected (Inheritance)
4) default (Packages)

1) public access control

=> Specified using the keyword public.

=> Classes, members or data members that are declared as public can be accessible from everywhere in the program.

=> There is no restriction on the scope of public data members.

Page No.
Date: / /

```
class AccessDemo
{
 int a = 10;
 public int b = 20;
 private int c = 30;

 void show()
 {
  System.out.println("A = "+a, "B = ",+b, "C = "+c);
 }
}

class Driver
{
 public static void main (String arg[])
 {
  AccessDemo obj = new AccessDemo();
  obj.show();
  System.out.println("A = "+obj.A, "B = "+obj.B,
                     "C = "+obj.C);
 }
}
```

OUTPUT : Compile Time Error
since 'c' is declared private & outside the class it cannot be accessed.

Page No.
Date: / /

2) private access control

=> Specified using the keyword private.

=> Accessible only within the class in which they are declared.

- Static keyword

Static keyword is a non-access modifier that can be accessed without creating an object of a class.

It is applicable for the following:
i) data members
ii) Member func^n's
iii) Block
iv) Class.

21/02/25

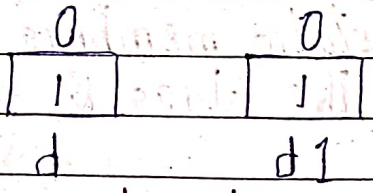
i) Data Member

static members are those which belongs to the class & one can access them without

Page No.
Date: / /

```
class Demo
{
static int count = 0;
 Demo ()
 {
  count ++;
 }
 void show ()
 {
  System.out.println (count);
 }
}

class Test
{
public static void main (String arg[])
{
  Demo d = new Demo ();
  d.show ();
  Demo d1 = new Demo ();
  d1.show ();
}
}
```

We can directly access a data member by its class name without creating any object.



OUTPUT
1
1

Here, the object create memory address as many times it is being called.

If we make the 'count' as static then it will print (1, 2) as there will be only one memory address & the value is stored there.

ii) Member Func"

Methods declared as 'static' have following restrictions:

=> They can only call other static methods.

=> They must only access static data.

=> They cannot refer to 'this' OR 'super' keyword in any way.

```
class Demo
{
  static int a = 10;
         int b = 20;
  static void show()
  {
    System.out.println("A = "+a);
    System.out.println("B = "+b);
  }
}
class Test
{
  public static void main(String arg[])
  {
    Demo.show();
    Demo d1 = new Demo();
    d1.show();
  }
}
```

OUTPUT FROM

24/02/25

Page No.
Date: / /

iii) Static Block :

=> A static block is also known as a static initialization block as it is used for static initialization of classes & variables.

=> Executed before the main method.

```
class X
{
  static int a = 10;
  static int b;
  static
  {
     b = a * 10;
  }
  static void show()
  {
    System.out.println("A = " + a + "B = " + b);
  }
}
class Y
{
  public static void main(String arg[])
  {
     X.show();
  }
}
```

OUTPUT a = 10
b = 100

iv) static class

Nested class

Static | Non-static / Inner class

Nested Class

=> The scope of a nested class is bounded by the scope of its enclosing class.

=> A nested class has the access to the members including private of outer class.

=> A nested class is also a member of its enclosing class.

=> As a member of its enclosing class, a nested class can be declared as public, private, protected & default.

Static Nested Class

It is a type of nested class which cannot directly access non-static member

```
class Outer
{
  static int a = 10;
  static class Inner
  {
    void show()
    {
      System.out.println("A = "+a);
    }
  }
}
class Test
{
  public static void main(String arg[])
  {
    Outer.Inner obj = new Outer.Inner();
    obj.show();
  }
}
```

Since the inner class is inside the outer class hence we required the outer class to call inner class.

Since show() is a non-static method, thus, an object is required to access it

Page No.
Date: / /

Non-static Class (Inner Class)

```
class Outer
{
  static int a = 10;
  class Inner
  {
    void show()
    {
      System.out.println("A = "+a);
    }
  }
}

class Test
{
  public static void main(String arg[])
  {
    Outer o = new Outer();
    Outer.Inner i = o.new Inner();
                  OR
    Outer.Inner i = new Outer().new Inner();
```

Q-

```
class Outer
{
  staticint a = 10;
  static b = 20;
  class Inner1
  {
    void show()
    {
      System.out.println("A = "+a+"B = "+b);
    }
```

```
static class Inner2
{
  void display()
  {
    System.out.println("A = "+a+"B = "+b);
  }
}
```

A- Output
A = 10
B = 20
A = 10
B = 20

Q-
```
class Outer
{
  int a = 10;
  class Inner
  {
    int a = 20;
    void show(int a)
    {
      System.out.println("Outer A = "+a);
      System.out.println("Inner A = "+a);
      System.out.println("Func A = "+a);
    }
  }
}
```

Shadowing
Shadows all other values & focus only on the value given in main()

outer.this.a

this.a

Page No.
Date: / /

```
class Test
{
public static void main (String arg[])
{
  Outer.Inner i = new Outer().new Inner();
   i.show (23);
}
}
```

A- Output
Outer A = 23
Inner A = 23
Func° A = 23

Local Inner Class

=> It is an inner class which should not be static.

```
class Outer
{
 void show()
 {
  System.out.println ("Hello");
 class Inner
 {
  void display()
  {
   System.out.println ("Hii");
  }
 }
 Inner i = new Inner(); } We cannot write this in main as
  i.display();           the class is local to
}                        show() & the obj. must be
                         called in show() only
```

```
class Test
{
  public static void main (String arg[])
  {
    Outer o = new Outer();
    o.show();
  }
}
```

OUTPUT Hello
Hii

- **Final keyword**

The final keyword is used to store const. value.

Eg.

```
final final pi = 3.14;
```

```
class Test
{
  public static void main (String arg[])
  {
    Outer o = new Outer();
    o.show();
  }
}
```

OUTPUT Hello
Hii

- **Final keyword**

The final keyword is used to store const. value.

Eg.

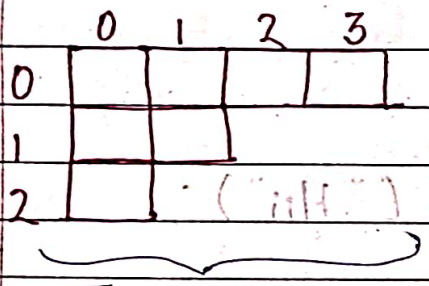
final final pi = 3.14;

12/03/25

- **Jagged Array**

=> It is an extension of 2D array.

=> In Jagged array, each row has different no. of columns.



JAGGED ARRAY

Page No.
Date: / /

Syntax

datatype variable [][] = new datatype [rows][];

variable [row index] = new datatype [cols];

Eg. int x[][] = new int [3][];
x[0] = new int [4];
x[1] = new int [2];
x[2] = new int [1];

Various Ways to Declare Jagged Array

1) int a[][] = {{10,20,30},
{40,50},
{60}};

2) int a[][] = {new int [] {10,20,30},
new int [] {40,50},
new int [] {60}};

3) int a[][] = new int [][] {
new int [] {10,20,30},
new int [] {40,50},
new int [] {60}
};

All these gives same O/P & create Jagged arrays.

17/03/25

Page No.
Date: / /

Strings

=> Collection of characters. (sequence)

String str = "KIIT"
↓
object of type String

```
class Demo
{
  public static void main (String arg[])
  {
    String str = new String ("Hello");
    System.out.println ("Char at 0 : "+str[0]);
  }
}
```

OUTPUT :- ERROR since we cannot do indexing in string as string is not character array.

```
System.out.println ("String is "+str);
String str1 = "Hi"
System.out.println ("Char at 0 : "+str1[0]);

String str1 = new String ("Hi");
```

String pool

OUTPUT :- ERROR COMPILE TIME

String, StringBuffer, StringBuilder

↓

Java.Lang

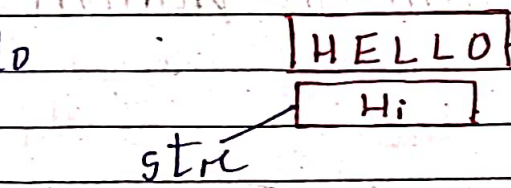
Page No.

Date: / /

=> Strings are immutable because once a string object is created, it cannot be changed. but we can change the reference of object

```
class Demo
{
public static void main (String arg[])
{
  String str = new String ("Hello");
  System.out.println ("String :" +str);
  String str = "Hi";
  System.out.println ("String :" +str);
}
}
```

OUTPUT Hello
Hi



Here, str is first assigned to HELLO and print it.
Then str is reassigned to HI & HELLO becomes eligible to garbage collection as no one is referencing it.

String Constructors

1) String s = new String (); // Default

2) String (char chars[]); // Parameterized.

Page No.
Date: / /

3) String (char chars[], int startIndex, int numChar)

Eg. char c[] = {'a', 'b', 'c', 'd'};
String s = new String (c, 2, 3);

4) String (String ob); // Copy

Eg. String s = new String ("Hello")
↳ This is string literal which is a string object hence it is a copy constructor.

5) String (byte chars[]) (byte to character conversion)

byte ascii[] = {65, 66, 67, 68, 69, 70};
String s1 = new String (ascii);

Output : A B C D E F

```
class Demo
{
public static void main (String arg[])
{
    int [] a = new int [3];
    String b = "Hello";
  System.out.println ("" + a.length);
  System.out.println ("" + b.length);
}
}
```

OUTPUT ERROR

Page No.
Date: / /

① String length (return the length including space)

Syntax
```
st = "computer";
st.length();
```

19/03/25

=> 3 types of String class (pre-defined)
a) String (immutable)
b) StringBuffer } mutable
c) StringBuilder }

```
class Demo
{
  public static void main (String arg[])
  {
    String a = new String ("Hello");
    System.out.println ("length :" +a.length());
    System.out.println ("length :" + "Hi".length());
  }
}
```

OUTPUT : 5
2

Using string literal we can directly call any string func^n without creating any object.

② Concatenation of String

'+' operator is used to concatenate the string.

Page No.
Date: / /

```
class Demo
{
public static void main (String arg[])
{
System.out.println("Sum of 2*2: " + 2*2);
}
}
```

To solve this, bracket is used (2+2)

OUTPUT Sum of 2+2 = 22

The compiler will convert an operand to its string equivalent whenever the other instance is String.

③ toString() String to String()

=> It is a method of object class.

=> It returns a String object that contains the human readable string that appropriately describes an object of the class.

```
String a = new String();
System.out.println(a);
```

It will print the object reference in human readable form.

```
class Box
{
  double w;
  double h;
  double d;

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

  public String toString()
  {
    return "Dimensions are" + width + "by"
        + height + "by" + depth + ".";
  }
}

class BoxDemo
{
  public static void main (String arg[])
  {
    Box b = new Box(10, 14, 12);
    System.out.println(b);
    String s = "Box b; " + b;
    System.out.println(s);
  }
}
```

Page No.
Date: / /

④ a) charAt() (returns the character at given index)

Syntax: char charAt(int idx);

ch1 = "abc".charAt(1);
It will print 'b'.

b) getChars()

Syntax: void getChars(int start, int end, char target[], int targetSta

start → idx of beginning
end → idx of end
char target[] → recieve the characters
Target start → idx within the target where string will be copied.

```
class getCharDemo
{
  public static void main(String arg[])
  {
    String s = "This is the demo of getChar method";
    int start = 10;
    int end = 14;
    char buf[] = new char[end-start];
    s.getChars(start, end, buf, 0);
    System.out.println("Char at 0: " + buf[0]);
  }
}
```

OUTPUT: Char at 0 : d
buff[] = ['d', 'e', 'm', 'o'].

c) toCharArray()

Syntax: char[] toCharArray()

Directly converts string into an character array.

20/03/25

(5) Equals() (returns true if both strings are same else false)

Syntax: boolean equals(object str)

⇒ It is case sensitive

```
class Demo
{
  public static void main(String arg[])
  {
    String s1 = new String("Hello");
    String s2 = new String(s1);
    System.out.println(s1==s2);
    System.out.println("s1" + "equals" + "s2" + ":" + s1.equals(s2));
  }
}
```

→ object reference of string. (invokes ToString)

OUTPUT :- False
True

→ compares the references of the string objects.

→ compares the contents of the string objects.

Page No.
Date: / /

```
class StringEqual
{
public static void main (String arg[])
{
  String s1 = "Hello";
  String s2 = s1;
  String s3 = "HELLO";
  Integer s4 = new Integer (10);
  System.out.println (" " + s1.equals (s2);
  System.out.println (" " + s1.equals (s3);
  System.out.println (" " + s1.equalsIgnore (s3))
  System:out.println(" " + s1.equals (s4));
}
}
```

OUTPUT :- True
False
True
False

b) equalsIgnoreCase ()

=> It is a func" of string class which is the updated version of equals () method.

Syntax : equalsIgnoreCase (String str)

⑥ compareTo ()

=> compares the string lexicographically.

Page No.
Date: / /

=> Compares which string is less than or greater than or equal.

Syntax: int compareTo (String s1)

```
class Demo
{
public static void main (String arg[])
{
static String a[] = {"Now", "we", "are",
    "implementing", "compare", "strings"};
for (int j = 0; j < a.length; j++)
{
for (int i = j+1; i < a.length; i++)
{
if (a[i].compareTo(a[j]) < 0)
{
    String t = a[j];
    a[j] = a[i];
    a[i] = t;
}
}
System.out.println(a[j]);
}
}
```

Output: Now
are
compare
implementing
strings
we

21/03/25

Page No.
Date: / /

⑦ Searching Strings

Mainly 2 func"s :- char
i) IndexOf() < string
ii) lastIndexOf() < char
string

IndexOf(char) L → R scan

search the first occurence of a character

Syntax :- int indexOf(int ch)

NOTE :- Similarly, in string also it will search for the first occurence of the string.

lastIndexOf() R → L scan

search the last occurence of a character & string.

Syntax :- int lastIndexOf(int ch)

Extensions

Syntax :- int indexOf(int ch, int startIndex)
↳ from start index to end

Syntax :- int lastIndexOf(int ch, int startIndex)
↳ from start index to zero

Page No.
Date: / /

```
class
{
  public
  {
    String s = "New" is "The time for all
               good men" + "to come to the
               aid of their country";
    System.out.println(s);
    System.out.println(s.indexOf('t'));
    System.out.println(s.lastIndexOf('t'));
    System.out.println(s.indexOf('The'));
    System.out.println(s.lastIndexOf('The'));
    System.out.println(s.indexOf('t', 10));
    System.out.println(s.lastIndexOf('t', 60));
  }
}
```

OUTPUT :
7
55
7
65
11
65

(8) subString()

Syntax : String subString (int startIndex)

Eg. String s = "ABCDE"

s.subString(1)

Output BCDE

Page No.
Date: / /

Syntax : String subString (int startIndex, int endIndex)

Eg: s.subString (1, 3)

Output BC

```
class Demo
{
public static void main (String arg [])
{
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
       res = "";
int i;
do
{
  System.out.println (org);
  i = org.indexOf (search);
  IF (i != -1)
  {
    res = org.substring (0, i);
    res = res + sub; // Thwas
    res = res + org.substring (i + search,
                               length ());
    org = res;
  }
} while (i != -1);
}
}
```

Page No.
Date: / /

OUTPUT This is a test. This is, too.
Thwas is a test This is, too.
Thwas was a test. This is, two.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.

24/03/25

⑨ concat()

Syntax : String concat (String str)

=> Creates a new object that contains the invoking string with the contents of the str.

=> Uses "+" operator to concat the strings

Eg. String s1 = "one";
String s2 = s1.concat("two");

⑩ replace()

Syntax : String.replace (char original, char replacement)

=> Replaces all occurrence of one character in the invoking string with another character.

String s1 = "Hello".replace ('l', 'w')
OUTPUT :- Hewwo

Page No.
Date: / /

(11) trim()

=> Removes spaces (" "), tabs (\t) & newline (\n) characters from beginning to end of the string.

=> The whitespaces range is till U to 2000.

Syntax :- String trim()

Eg. s = "\t Hello \n World \n \t ".trim()

(12) strip()

=> Does the same work like trim().

=> The only difference is strip() can remove all whitespaces but trim() can remove whitespaces till certain range.

Syntax :- String strip()

(13) Value Of()

=> converts data from its internal format into human readable form.

String valueOf (int num)
String valueOf (double num)
String valueOf (char chars[])
String valueOf (boolean true)
String valueOf (object ob)

Page No.
Date: / /

(14) toLowerCase()

=> converts all characters from uppercase to lowercase.

Syntax: String.toLowerCase();

(15) toUpperCase()

=> converts all characters from lowercase to uppercase.

Syntax: String.toUpperCase()

(16) Join Strings

Syntax: String join (char sequence delim; char sequence ... strs)

Delimiator is used to separate the character sequence specified by string.

Eg. String.join(" ", "Alpha", "Beta", "Gamma")

OUTPUT: Alpha Beta Gamma

String.join(", ", "John", "ID#569",);

OUTPUT: John, ID#569

Page No.
Date: / /

StringBuffer

=> Supports a modifiable string.

=> Used to create mutable string object.

=> It is well synchronized.

StringBuffer Constructors

StringBuffer () // Default
StringBuffer (int size)
StringBuffer (String str)
StringBuffer (CharSequence chars)

NOTE: In StringBuffer () it assigns 16 characters by default to it without reallocation.

length () & capacity ()

int length ()
int capacity ()

Current length of StringBuffer

Total capacity allocat

Page No.
Date: / /

```
class
{
  public
  {
    StringBuffer sb = "Hello";
    System.out.println(sb);
    System.out.println(" " + sb.length);
    System.out.println(" " + sb.capacity);
  }
}
```

OUTPUT : Hello
5
21

** ensureCapacity()

Syntax void ensureCapacity(int minCapacity)

=> If currentCapacity < minCapacity, then a new internal array is allocated with greater capacity.

=> If minCapacity > twice the old capacity +2 then the new capacity = minCapacity else new capacity = twice the old capacity + 2.

Page No.
Date : / /

```
if minCapacity > 2 x oldCapacity + 2
{
    newCapacity = minCapacity
}
else
{
    newCapacity = 2 x oldCapacity + 2.
}
```

```
class
{
 public
 {
    StringBuffer str = new StringBuffer("
                        I LOVE JAVA");
    System.out.println("  " + str.Capacity());
    str.ensureCapacity(42);
    System.out.println("  " + str.capacity());
 }
}
```

OUTPUT 27
56

→ 42 > 2 x 27 + 2
42 > 56
⇒ newCapacity = 56

Page No.
Date: / /

setLength()

Syntax :- ~~String~~ void setLength(int len)

=> Increases the string but if space is not available then it will terminate.

Eg. String s = "Hello".setLength(2);

OUTPUT Hel

charAt() and setCharAt()

Syntax :- char charAt(int idx)
void setCharAt(int idx, char ch)

```
class Demo
{
public static void main(String arg[])
{
StringBuffer sb = new StringBuffer("Hello");
sb.setCharAt(2,'x');
System.out.println(sb);
}
}
```

OUTPUT HeXlo

append()

Page No.
Date: / /

Syntax : StringBuffer append (String str)
StringBuffer append (int num)
StringBuffer append (object obj)

```
class Demo
{
public static void main (String arg[])
{
StringBuffer sb = new StringBuffer (40);
sb.append ("a = ").append (42).append ("! ");
System.out.println (sb);
}
}
```

OUTPUT

insert()

Syntax : StringBuffer insert (int idx, String str)
StringBuffer insert (int idx, char ch)
StringBuffer insert (int idx, object obj)

```
class Demo
{
public static void main (String arg[])
{
StringBuffer sb = new StringBuffer ("I JAVA!");
sb.insert (2, "LIKE");
System.out.println (sb);
}
}
```

OUTPUT I LIKE JAVA !

Page No.
Date: / /

reverse()

```
class Demo
{
  public static void main (String arg[])
  {
    StringBuffer sb = "abcdef";
    sb.reverse();
    System.out.println(sb);
  }
}
```

OUTPUT fedcba

delete() and deleteCharAt()

Syntax :- StringBuffer delete (int startidx, int endidx)
StringBuffer deleteCharAt (int char)

replace().

StringBuffer replace (int startidx, int endidx, String str)

String Builder

=> These are non-synchronized.

=> More efficient than string Buffer.

02/04/25

Page No.:
Date: / /

Inheritance (OOP)

=> Inheritance allows a class to inherit properties & behaviours from other class.

=> It is a mechanism of deriving a new class from existing class.

=> Uses keyword extends.

Types : 1) Simple
2) Multilevel
3) Hierarchical
4) Multiple
5) Hybrid

1) Simple

NOTE A class member that has been declared as private will remain private to its class. It is not accessed by any subclass.

Page No.
Date: / /

```
class A
{
  private int i;  X
  public int j;
}
class B extends A
{
  int k;
```

Here class B can inherit the public specifier but not the private specifier.

```
class A
{
  int i, j;
  void show()
  {
    System.out.println("In A");
  }
}
class B extends A // subclass
{
  int k;
  void sum()
  {
    System.out.println("Sum " + (i+j+k));
  }
}
class Demo
{
  public static void main(String arg[])
  {
```

Class B can access both i & j.

obj 2. show ();

Page No.
Date: / /

```
A obj1 = new A ();
B obj2 = new B ();
obj1.show ();
obj2.i = 10;
obj2.j = 20;
obj2.k = 30;
obj2.sum ();
    }
}
```

OUTPUT In A
Sum: 60

Superclass Variable can Reference a subclass Object

```
class A
{
  int a, b = 10, b = 20;
  void sum ()
  {
    System.out.println ("Sum : " + (a+b));
  }
}

class B extends A
{
    int c;
}

public static void main (String arg[])
{
    B o1 = new B ();
```

```
o1.c = 20;
A o2;
o2 = o1;
o2.sum();
o2.c = 30; // ERROR
System.out.println(c);
}
}
```

When a reference of object class variable store the reference of ~~super~~ base class, it cannot access the properties of subclass, it can only access the properties of base class.

Super keyword

It refers to the immediate parent class of a class.

It is used to eliminate the confusion bet^n super class & sub class that have methods of same name.

①
```
class A
{
  int i;
}
class B extends A
{
```

Page No.
Date : / /

```
    int i;
    B()
    {
        i = 10;   super.i = 20;
    }
    void show()
    {
        System.out.println("Base : " + super.i);
        System.out.println("Derived : " + i);
    }
}
class Demo
{
    public static void main(String arg[])
    {
        B o1 = new B();
        o1.show();
    }
}
```

OUTPUT

10	10
10	20

To call constructor of super class

```
class A
{
    int a;
    A(int n)
    {
        a = n;
    }
}
```

```
class B extends A
{
    int b;
    B(int n1, int n2)
    {
        super(n1); // immediate call the constructor of base class.
        b = n2;
    }
}

class Demo
{
    public static void main(String arg[])
    {
        B obj = new B(10, 20);
    }
}
```

04/04/25

2) Multilevel Inheritance & Constructor calling

A class inherits properties from a class which again has inherits properties from other class.

```
class A
{
    A()
    {
        System.out.println("Hello");
    }
}
```

Base class

~~Base~~
~~Intermediate~~
~~Bac~~

Page No.
Date: / /

```
class B extends A
{
  B()
  {
    System.out.println("Hi");
  }
}
```

Inter-mediate class

```
class C extends B
{
  C()
  {
    System.out.println("Bye");
  }
}
```

Derived class

```
class Demo
{
  public static void main(String arg[])
  {
    C obj = new C();
  }
}
```

OUTPUT
Hello
Hi
Bye

NOTE In default constructor case in child class, "super" ~~class~~ is optional there. Without using super keyword, we can call the super class.

Page No.
Date: / /

Method Overriding

If a sub class has the same method as declared in the parent class, it is known as method overriding.

It has :- Same name
Same no. of parameters
Same type of parameters
Same returntype
Same prototype

```
class A    // Base class
{
  int a = 20;
  void show ()
  {
    System.out.println ("A = "+a);
  }
}

class B extends A //Derived class
{
  int b = 10;
  void show ()
  {
    System.out.println ("B = "+b);
  }
}
```

Page No.
Date : / /

```
class Demo
{
  public static void main (String arg[])
  {
    B obj = new B();
    obj.show();
  }
}
```

OUTPUT b = 10

1)
```
Class Employee
{
  String name = "Emp";
  void printName()
  {
    System.out.println(name);
  }
}

class Programmer extends Employee
{
  String name = "Prog";
  void printName()
  {
    System.out.println(name);
  }
}

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

Variables are bound at compile time
Methods are bound at run time.

Page No.
Date: / /

```
Employee emp = new Employee();
Employee prog = new Programmer();
System.out.println(emp.name);
System.out.println(prog.name);
emp.printName();
prog.printName();
}
}
```

** Dynamic Dispatching (Late binding)

OUTPUT : emp
emp (prog is a emp variable of prog employee)

Dynamic Dispatch

=> It is the mechanism through which the correct version of an overridden method is called at runtime.

=> When a subclass overrides a method from its superclass, the overriden method in the subclass is executed when called on an instance of the subclass, even if the reference to the object is of the superclass type.

Page No.
Date: / /

Abstract Class & Methods

=> An abstract class cannot be directly instantiated with the new operator.

=> You cannot declare abstract constructors, or abstract static methods.

=> Any class that contains one or more abstract methods must also be declared abstract.

=> Any subclass of an abstract class must either implement all the abstract methods in the superclass, or be declared abstract itself.

=> It can also contain concrete methods (definitio

```
abstract class Shape
{
  abstract void show();
  abstract int area();
}
class Square extends Shape
{
   int side = 10;
   void show()
   {
     System.out.println("Area: " + area());
   }
```

```
    int area()
    {
        return (side * side);
    }
}
class Demo
{
    public static void main (String arg[])
    {
        Square obj = new Square();
        obj.show();
    }
}
```

OUTPUT Area : 100

Final with Inheritance

Following are the uses of final keyword :-

1) It can declare any variable as const.

2) Methods declared as final cannot be overriden.

3) Declaring a class as final implicitly declares all of its methods as final, too & prevent a class from being inherited.

Page No.
Date: / /

```
class A
{
  int a;
  final void initT()
  {
    a = 10;
  }
  void show()
  {
    System.out.println(a);
  }
}

class B extends A
{
  int b;
  void initT()
  {
    b = 10;
  }
  void show()
  {
    System.out.println(b);
  }
}
```

EARLY BINDING

OUTPUT ERROR

Page No.
Date: / /

Early Binding

=> Early binding occurs at compile time means the compiler determines which method will execute.

=> It is typically associated with methods that are not subject to method overriding in inheritance.

=> Final methods are declared to be not overridden in subclasses. Because their implementation cannot be changed in derived classes, the compiler can directly associate the method call with the class during compilation. leading to early binding.

Interface

=> Using a keyword interface, you can fully abstract a class' interface from its implementation.

=> There methods are declared without any body.

=> An interface cannot be directly instantiated with the new operator.

Page No.
Date: / /

=> To implement an interface, a subclass must provide the complete set of methods required by the interface.

→ Variables can be declared inside a interface declaration.

→ They are implicitly final & static, meaning they cannot change by the implementing class. They must also be initialized.

09/05/25 => If my implementing class fails to implement all func's of interface then the implementing class ~~fails~~ needs to make as abstract.

=> Objects cannot be created in interface.

=> Whenever implementing func" of an interface, it must be declared public in implementing class.

```
interface Call
{
    int Ok = 1 ; //static & final
    void show() ; // by default abstract
}
                          → To call interface
class A implements Call
{
    public void show()
    {
```

```
        Ok = 2;
        System.out.println ("Hello");
    }
}
class Demo
{
    public static void main (String arg[])
    {
        A obj = new A();
        obj.show();
    }
}
```

OUTPUT Hello if Ok = 2 is commented.
otherwise ERROR

Interfaces Can Be Extended in a Class.

=> An interface can extend another interface but a class cannot extend an interface

```
interface A
{
    void F1();
}
interface B extends A
{
    void F2();
}
```

Page No.
Date: / /

```
interface C extends B
{
  public void f3();
}
class D implements C
{
  public void f3()
  {
    System.out.println("Hello");
  }
}
class Demo
{
  public static void main(String arg[])
  {
    D obj = new D();
    obj.f3();
  }
}
```

OUTPUT ERROR
since we are not overriding the method f2() & f1() in class D.

Nested Interface

=> We can declare an interface in another interface or class.

=> It follows the same syntax as a regular

Page No.
Date: / /

interface but is enclosed with an outer class OR interface.

```
class A
{
  interface B
  {
    void show();
  }
}

class C implements A.B
{
  public void show()
  {
    System.out.println("Hello");
  }
}

class Demo
{
  public static void main(String arg[])
  {
    C obj = new C();
    obj.show();
  }
}
```

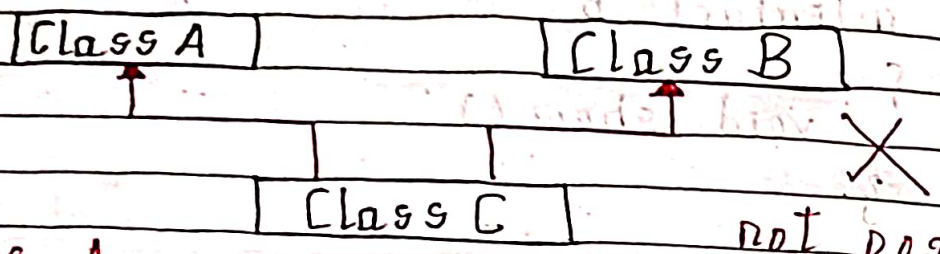
Page No.
Date: / /

```
interface A
{
   interface B
   {
      void show();
   }
}
class XYZ implements A.B
{
  public void show()
  {
    System.out.println("Hello");
  }
}

class Demo
{
  public static void main(String arg[])
  {
    A.B obj = new XYZ();
    XYZ.show();
  }
}
```

Multiple Inheritance

Java does not support multiple inheritance of classes because it introduces complexity & ambiguity, particularly

The 'diamond problem'. Instead it provides interfaces.



not possible in JAVA

```
interface A
{
  void show()
}

interface B
{
  void show();
}

class C implements A, B
{
  public void show()
  {
    System.out.println("Hello");
  }
}
```

Although, two show() methods were inherited from interface A & B, class C overrides the method to avoid ambiguity & ensures only one copy is used. Thus preserving the concept of multiple inheritance through interfaces.

11/04/25

-d . MyCal.java

Page No.
Date: / /

Packages

⇒ A java Package is a group of similar types of classes, interfaces & sub-packages.

Why use Packages?

1) Organising
2) Easy access
3) Scope / resolve the environment

```
import java.util.Scanner;
       ||        ||
    Package    Class
```

```
package MyPack;

public class Calculator
{
public int add (int i, int j)
   {
      return (i+j);
   }
}
```

MyPack
└ Calculator.class
User defined package

```
import MyPack.Calculator; // The above package is imported
class Demo
{
  public static void main (String arg [])
  {
```

```
Calculator c = new Calculator();
int res = c.add(10,20);
System.out.println("Res : "+res);
}
}
```

Member Access in Packages

	Private	Default	Protected	Public
① Same Class	✓	✓	✓	✓
② Same package sub class	✗	✓	✓	✓
③ Same package non-sub class.	✗	✓	✓	✓
④ Different package sub class	✗	✗	✓	✓
⑤ Different package non-sub class	✗	✗	✗	✓

4/25

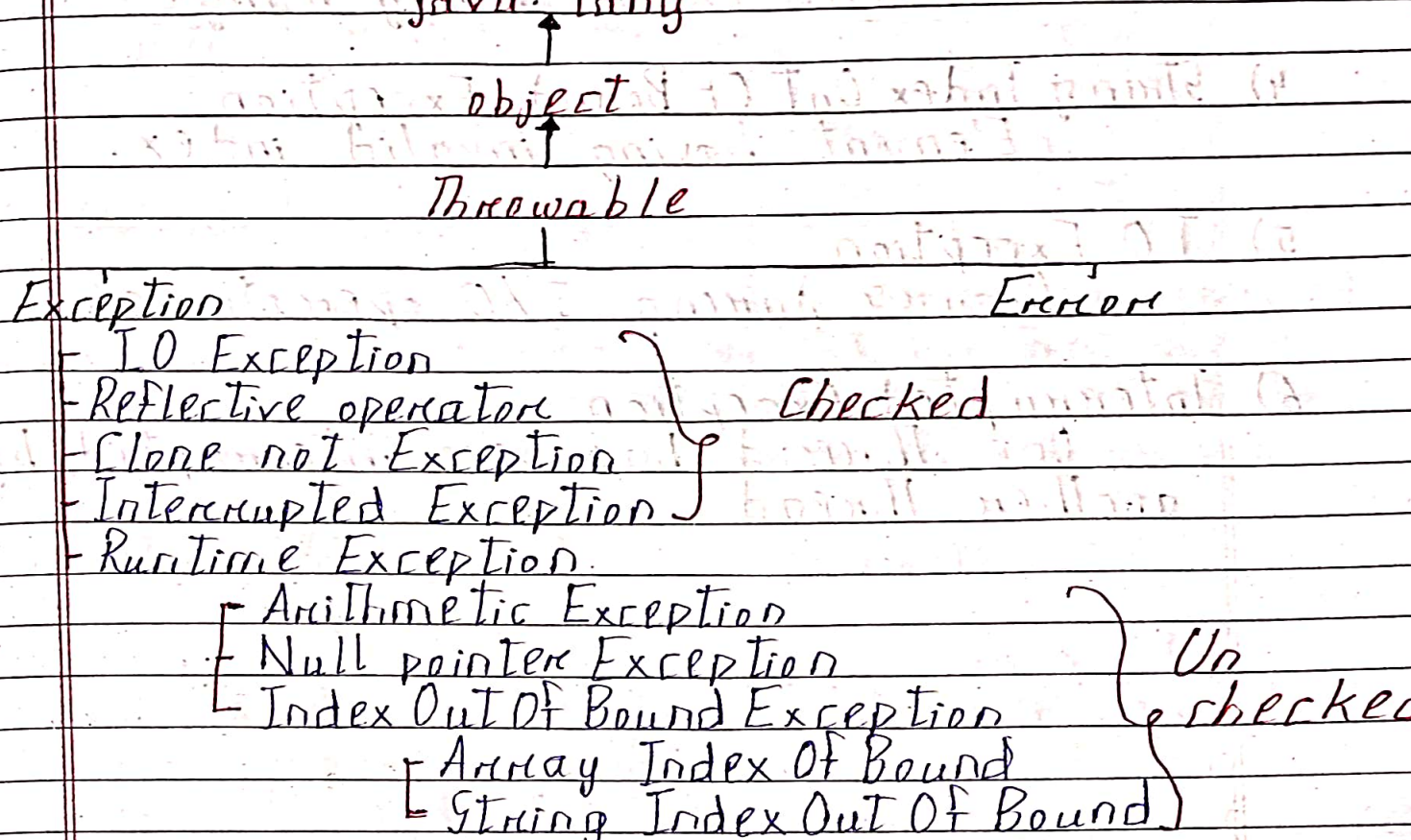
Page No.
Date: / /

Exception Handling

=> Exception is an event that disrupts the normal flow.

=> Error is a type of exception.

=> Exception handling in JAVA is a mechanism to handle the runtime errors so that the normal flow of an application is maintained



Page No.
Date: / /

Types of Exception

→ Mandatory to handle

1) Checked (Checked at compile time.)
2) Unchecked (Run time exception)

1) Arithmetic Exception
   Arithmetic error such as divide-by-zero

2) Null Pointer Exception
   Invalid null reference

3) Array Index Out OF Bound Exception
   Element exceed array range

4) String Index Out OF Bound Exception
   Element having invalid index.

5) IO Exception
   Occurs during I/O operations.

6) Interrupted Exception
   One Thread has been interrupted by another Thread

Page No.
Date: / /

Uncaught Exceptions

=> Handled by JVM

Exception keywords

1) try
2) catch
3) finally
4) Throw
5) Throws

1) try block

Used to specify a block where we should place an exception code. It means we can't use try block alone. It must be followed by either catch OR finally.

2) catch block

Used to handle exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

Page No.
Date: / /

Syntax:
```
try
{
    // Code that throw an exception
}
catch (ExceptionType e)
{
    // Exception handling code
}
```

```
class Main
{
public static void main (String arg[])
{
  try
  {
    int data = 100/0;
  }
  catch (ArithmeticException e)
  {
    System.out.println(e);
  }
  System.out.println ("rest of the code..");
}
}
```

OUTPUT
```
java.lang.ArithmeticException:
    / by zero
rest of the code...
```

Page No.
Date : / /

3) finally block

=> Used to put "cleanup" code such as closing a file etc.

=> The important statements that are to be printed can be placed in this block.

Rule :- For each try block there can be zero or more catch block but only one finally block

=> It is always executed whether an exception is handled or not.

=> It will not be executed if the program exits.

Case (1) When an exception occurs but not handled by the catch block

```
class Final
{
  public static void main (String arg[])
  {
    try
    {
      int data = 25/0;
    }
```

```
finally
{
    System.out.println("final block is executed");
}
System.out.println("rest of the code...");
}
}
```

OUTPUT
final block is executed
rest of the code
Exception handled by JVM

finalize() method

⇒ It is a method called by the garbage collector before an object is destroyed.

⇒ It is now deprecated & not recommended in modern Java.

```
class Main
{ protected void finalize {
  public static void main(String arg[])
  {
      Main obj = new Main();
      obj = null; //Eligible for garbage collection
      system.gc(); JVM to run G.C.
      System.out.println("Garbage collected");
  }
}
}
```

Page No.
Date: / /

Difference bet^n final, finally & finalize

Final keyword

⇒ keyword, access modifier that apply restrictions on a class, method or variable.

⇒ For variable, final variable becomes const. & cannot be modified
For method, final method cannot be overridden by sub class.
Final class cannot be inherited.

⇒ Gives compile-time control.

Finally block

⇒ Executed as soon as the try-catch block is executed. It is not dependent on the exception.

⇒ Cleans up all the resources used in a try block.
Runs the important code even if an exception occurs or not.

⇒ Provides run time control to handle normal flow.

NOTE | Throw clause is mandatory for checked exceptions.

Page No.
Date: / /

finalize() method

=> Used to perform cleanup processing just before an object is garbage collected.

=> Performs the cleaning activities w.r.t object before destruction.

=> Invoked by JVM.

4) **Throw keyword**

=> Used to throw an exception explicitly.

Syntax : Throw new exception_class ("user defined message");

=> We can also define our own set of cond"s & throw an exception explicity using throw keyword

```
class Main
{
  public static void main (String arg[])
  {
    int age = 23;
    if (age < 18)
    {
      throw new ArithmeticException ("Not
                eligible to vote");
    }
```

Page No.
Date: / /

```
else
    System.out.println("Eligible");
}
}
```

OUTPUT

5) Throws keyword

=> Used to declare an exception.

=> Gives information to the programmer that there may occur an exception

```
import java.io.*;
class M
{
  void method() throws IOException
  {
    throw new IOException("device error");
  }
}
class Main {
public static void main(String arg[])
                    throws IOException
{
  M m = new M();
    m.method();
  System.out.println("normal flow...");
}
}
```

24/04/25

Page No.
Date: / /

**

Input Output Handling

Streams :- Represents a sequence of data.
There are 2 types of streams :
i) Input
ii) Output

NOTE Java.io package contain large no. of stream classes.
Byte Stream → read & write a single byte
Character stream → read & write a single character

I/O Streams

Byte Stream		Character stream	
Input stream	Output stream	Reader stream	Writer stream

Input Stream

⇒ Abstract superclass that represent an input stream of bytes.

Page No.
Date: / /

Hierarchy

java.io.InputStream
- ↳ ByteArrayInputStream
- ↳ FileInputStream
- ↳ ObjectInputStream
- ↳ BufferedInputStream
- ↳ DataInputStream

Methods : int read()
int read(byte b[])
int read(byte b[], int off, int len)
int available()
void close()

Output Stream

=> Abstract superclass that represents an output stream of bytes.

Hierarchy

java.io: OutputStream
- ↳ ByteArrayOutputStream
- ↳ FileOutputStream
- ↳ ObjectOutputStream
- ↳ BufferedOutputStream
- ↳ DataOutputStream

Page No.
Date: / /

Methods :- void write (int b)
void write (byte b[])
void write (byte b[], int off, int len)
void flush ()
void close ()

Reader Stream

Abstract superclass that represents a stream of characters to read data.

Hierarchy:

java.io.Reader
↳ InputStreamReader
↳ FileReader
↳ BufferedReader
↳ StringReader
↳ CharArrayReader

Methods :- int read ()
int read (char cbuf[])
int read (char cbuf, int off, int len)
void close ()

Writer Stream

Abstract superclass that represents a stream of characters to write data.

Page No.
Date: / /

Hierarchy

java.io.writer
↳ OutputStreamWriter
↳ FileWriter
↳ BufferedWriter
↳ StringWriter
↳ CharArrayWriter
↳ PrintWriter

Methods: write(int c)
write(char cbuf[])
write(int string)
flush()
close()

Reading Console Input

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

To execute this we need to use try-catch block to avoid exceptions.

Q- WAP to accept your name from console & print it.

A- import java.io.*
class Main
{
public static void main(String arg[])

```
{
  try
  {
    System.out.println("Enter your name: ");
    BufferedReader br = new BufferedReader
                        (new InputStreamReader(System
                                               .in));
    String name = br.readline();
    System.out.println("Name: "+name);
  }

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

Writing Console Output

```
BufferedWriter bw = new BufferedWriter(new Output
                                StreamWriter(System.out));
bw.write(name);
bw.flush();
```

Page No.
Date: / /

File Handling

=> File handling means various operation such as read, write, append, delete are performed on a file.

=> Following are the classes used for file handling:

1) File
2) ~~Input~~File Input Stream
3) File Output Stream
4) Random Access File

Opening a File

Following are the constructors used to open a file:-

```
File myfile;
myfile = new File (filename);
myfile = new File (pathname, filename);
myfile = new File (Directory, pathname);
```

Testing a File

Following are the methods

i) exists()
ii) canwrite()
iii) canread()
iv) isFile()
v) isDirectory()

Page No.
Date: / /

Reading a File

```
import java.io.*;
class Main
{
 public static void main(String arg[])
 {
  int b;
  FileInputStream inFile = null;
  try
  {
   while (b = inFile.read() != 1)
   {
    System.out.println ((char)b);
   }
   inFile.close();
  }
 catch (IOException ioe)
 {
  System.out.println (ioe);
 }
}
}
```

Writing a File

```
import java.io.*;
class File
{
 public static void main (String arg[])
```

```
{
FileOutputStream outfile = null;
char cities[] = {'c', 'u', 't', 't', 'a', 'c', 'k'};
try
{
  outfile = new FileOutputStream ("city.txt");
  outfile.write (cities);
}
catch (IOException ioe)
{
  System.out.println (ioe);
}
finally
{
   try
      {
         if (outfile != null)
              outfile.close()
      }
   catch (IOException ioe)
   {
     System.out.println (ioe);
   }
}
}
}
```

Page No.
Date: / /

RandomAccessFile

=> It is a class in java.io package that allows reading from & writing to a file at any pos?

=> It allows the file pointer to move & access any part of the file randomly.

=> Allows read or write or read-write simultaneously.

Common Methods :- read()
write(int b);
seek(long pos)
length();

```
import java.io.*;
import java.io.RandomAccessFile;

class Demo
{
public static void main(String arg[]) throws
                                    exception
{
RandomAccessFile raf = new RandomAccess
                    File("FileDemo.txt", "r");
raf.seek(4); //moving file pointer to 4th
byte b[] = new byte[12];                byte
raf.read(b);
raf.close();
System.out.println(new String(b));
}
}
```

Page No.
Date: / /

Serialization

=> It is a mechanism of converting the state of an object into a byte stream.

=> The class implements serializable interface which has no member methods.

```
FileOutputStream fos = new FileOutputStream
                        ("object.dat");
ObjectOutputStream oos = new ObjectOutputStream
                        (fos);
oos.writeObject(pin);
```

Deserialization

=> It is the reverse process where the byte stream is used to recreate the actual Java object in memory.

```
FileInputStream fis = new FileInputStream
                        ("object.dat");
ObjectInputStream ois = new ObjectInputStream
                        (fis);
Person (pout) = (Person) ois.readObject();
System.out.println(pout);
```

Page No.
Date: / /

Multi Threading

⇒ It is a process of executing multiple Threads simultaneously.

⇒ A Thread is a lightweight sub process, the smallest unit of processing.

Main Thread

⇒ When a JAVA program starts, one Thread begins running by default i.e. the main Thread.

⇒ It is important because it creates other child Threads & is often last to finish execution.

Creating a Thread

To JAVA, we can create Thread by using 2 ways :
a) By implementing runnable interface
b) By extending the Thread class

a) Implementing Runnable Interface

```
class NewThread implements Runnable
{
  Thread t;
  NewThread()
```

```
{
    t = new Thread (this, "Demo Thread");
    System.out.println ("Child Thread : "+t);
    t.start ();          calls the run() auto-
}                        matically
public void run()
{
try
{
for (int i=5; i>0; i--)
    System.out.println ("Child Thread : "+i);
    Thread.sleep (500);
}
catch (InterruptedException e)
{
    System.out.println ("Child interrupted...");
}
System.out.println ("Exiting Child Thread.");
}
}
```

OUTPUT:

```
5
4
3
2
1
Exiting Child Thread.
```

Whenever we are implementing runnable, we need to pass the reference of the object of the current class.

Page No.
Date: / /

b) Extending Thread Class

```
class NewThread extends Thread
{
  NewThread
  {
    super ("Demo Thread");
    System.out.println("Child Thread :" + this);
    start();
  }
  public void run()
  {
    try
    {
      for (int i=5; i>0; i--)
        System.out.println("Child Thread :" +i);
      Thread.sleep(500);
    }
    catch (InterruptedException e)
    {
      System.out.println("Child interrupted...");
    }
    System.out.println("Exiting Child Thread");
  }
}
```

OUTPUT

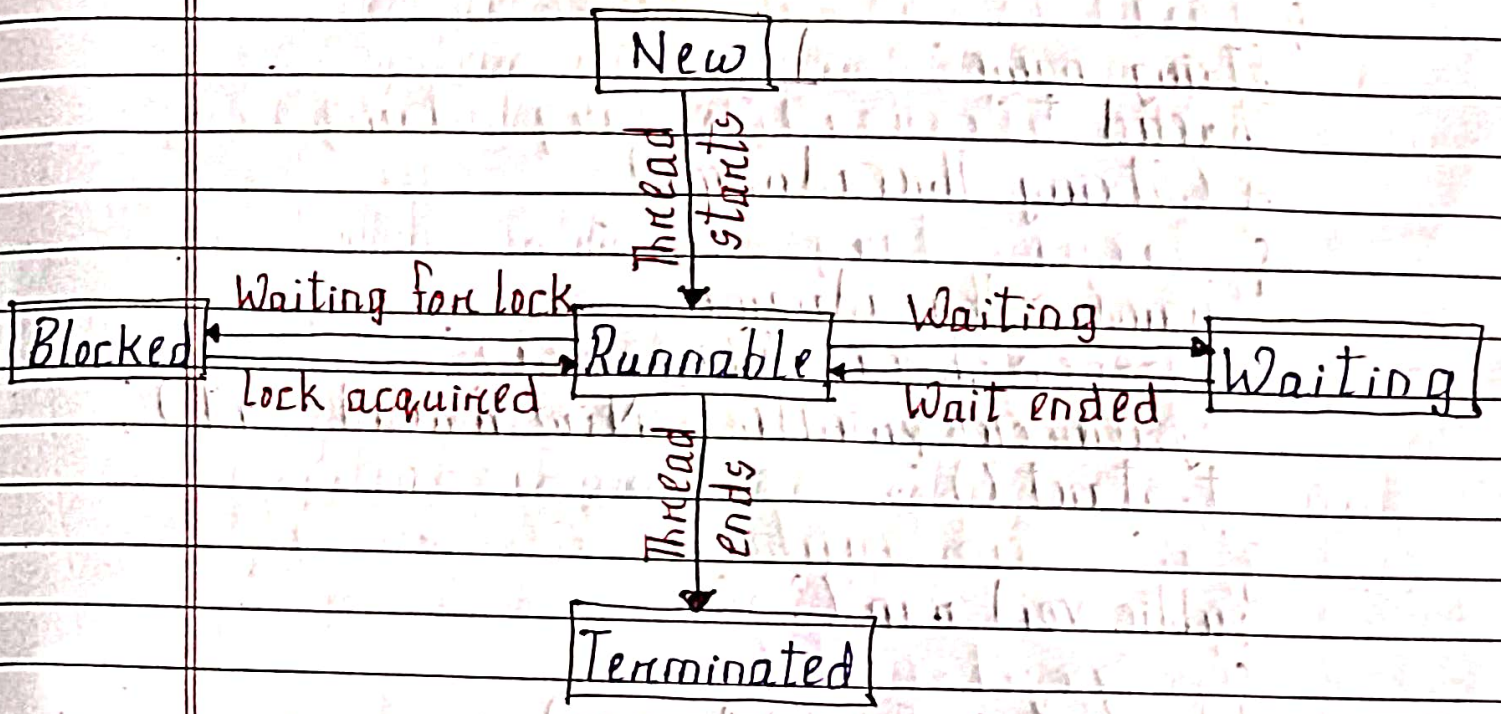
5
4
3
2
1
Exiting Child Thread

Page No.
Date: / /

Which approach is more preferred?

Runnable approach is more preferred because JAVA allows only one class to be extended, so using Runnable lets you extend another class too.

Thread life Cycle



1) New : A Thread has not begun execution.

2) Runnable : Either currently executing or will execute when it gains access to the CPU.

3) Waiting : Suspended execution because it is waiting for some action to occur. For eg. call to a non-timeout version of wait() or join()

Page No.
Date: / /

4) Blocked : Suspended execution because it is waiting to acquire a lock.

5) Terminated : Completed Execution

Multiple Threading

```
class A implements Runnable
{
String name;
Thread t;
A (String Threadname)
{
  name = Threadname;
  t = new Thread (this, name);
  System.out.println ("New Thread : " + t);
  t.start ();
}
public void run ()
{
  for (int i = 5; i > 0; i--)
    System.out.println (name + ": " + i);
    Thread.sleep (1000);
  System.out.println (name + "exiting.");
}
}
class Demo
{
public static void main (String arg [])
{
  new A ("One");
```

new A ("Two");
new A ("Three");
}
}

OUTPUT All the 3 threads run concurrently but not in synchronous way.

isAlive(): returns true if thread is still running.

join(): waits for thread to terminate.

Synchronization

It is a mechanism that controls the access of multiple threads to shared resources, ensuring that only one thread can access a specific resource at a time.

It has 2 ways: a) Synchronization method
b) Synchronization block

a)
```
class CallMe
{
Synchronized void call (String msg)
    {
    System.out.print("[" + msg);
    try
    {
      Thread.sleep (1000);
    }
    catch (InterruptedException e)
```

```
{
System.out.println("Interrupted");
}
System.out.print("]");
}
class Caller implements Runnable
{
String msg;
CallMe target;
Thread t;
public Caller(CallMe targ, String s)
{
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run()
{
target.call(msg);
}
```

We have to make the method as synchronized

b)
```
synchronized (objRef)
{
// statements to be
    synchronized.
}
```

Page No.
Date / /

Inter Thread Communication

1) wait() :- Give up the monitor & go to sleep until some other thread enters the same monitor.

2) notify() :- Wakes up a thread that called wait() on the same object.

3) notifyAll() :- Wakes up all the threads that called wait() on the same obj.

⇒ It is a way for threads to coordinate with each other mainly to avoid busy waiting.

Producer Consumer Problem using wait() & notify()

```
class Q
{
  int n;
  boolean valueSet = false;

synchronized int get()
{
  while (!valueSet)
    try
    {
      wait();
    }
    catch (InterruptedException e)
```

```
{
System.out.println("Exception Caught");
}
System.out.println("Get:" +n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n)
{
while(caught)
  try
  {
   wait();
  }
  catch (InterruptedException e)
  {
   System.out.println("Exception Caught");
  }
this.n = n;
valueSet = true;
System.out.println("Put:" +n);
notify();
}
}
```

Page No.
Date: / /

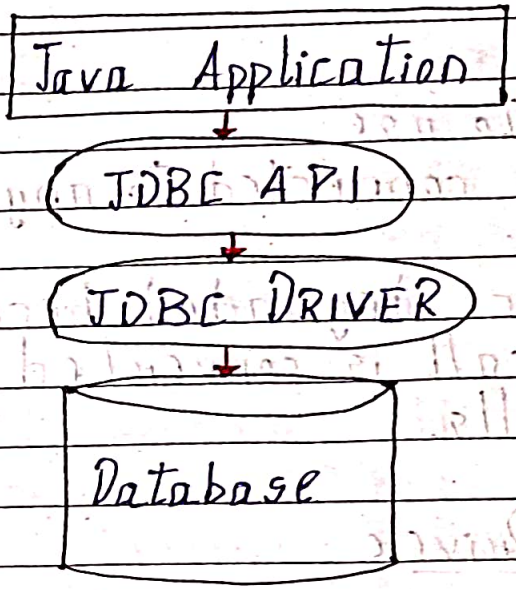
JDBC Connectivity

=> Stands for Java Database Connectivity.

=> It is a JAVA API to connect & execute the query with the database & processing the results.

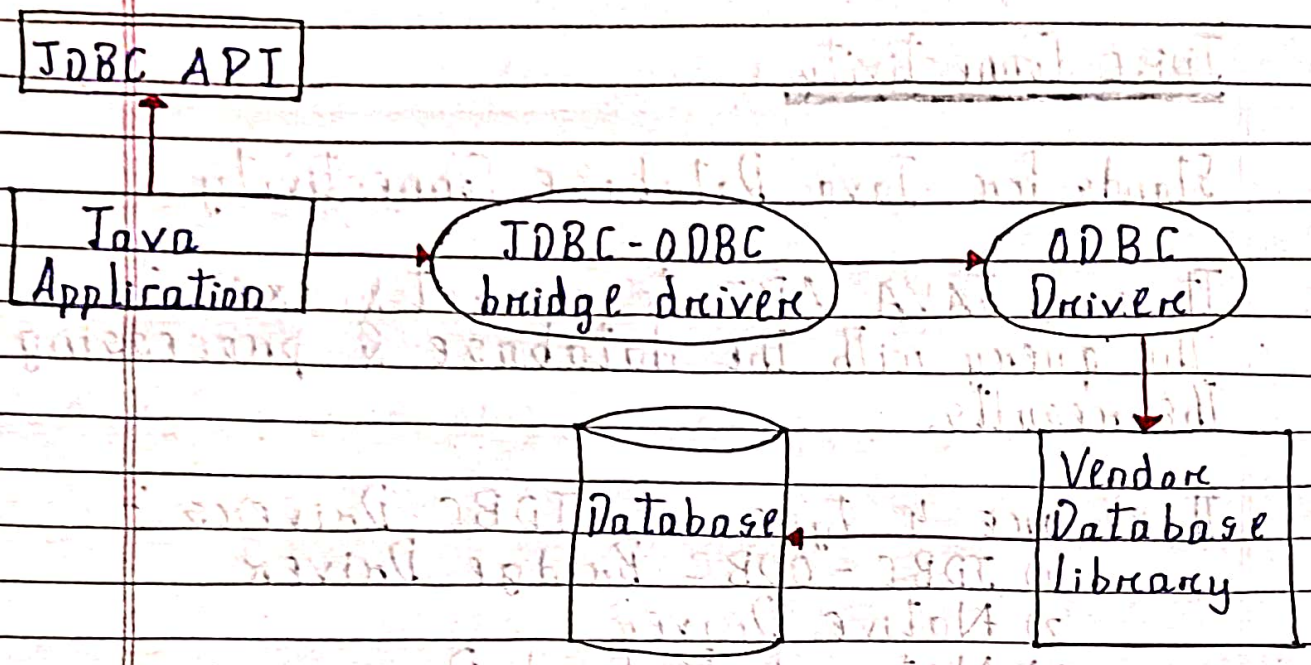
=> There are 4 types of JDBC Drivers :-

1) JDBC - ODBC Bridge Driver
2) Native Driver
3) Network Protocol Driver
4) Thin Driver



JDBC Drivers

1) JDBC - ODBC Bridge Driver



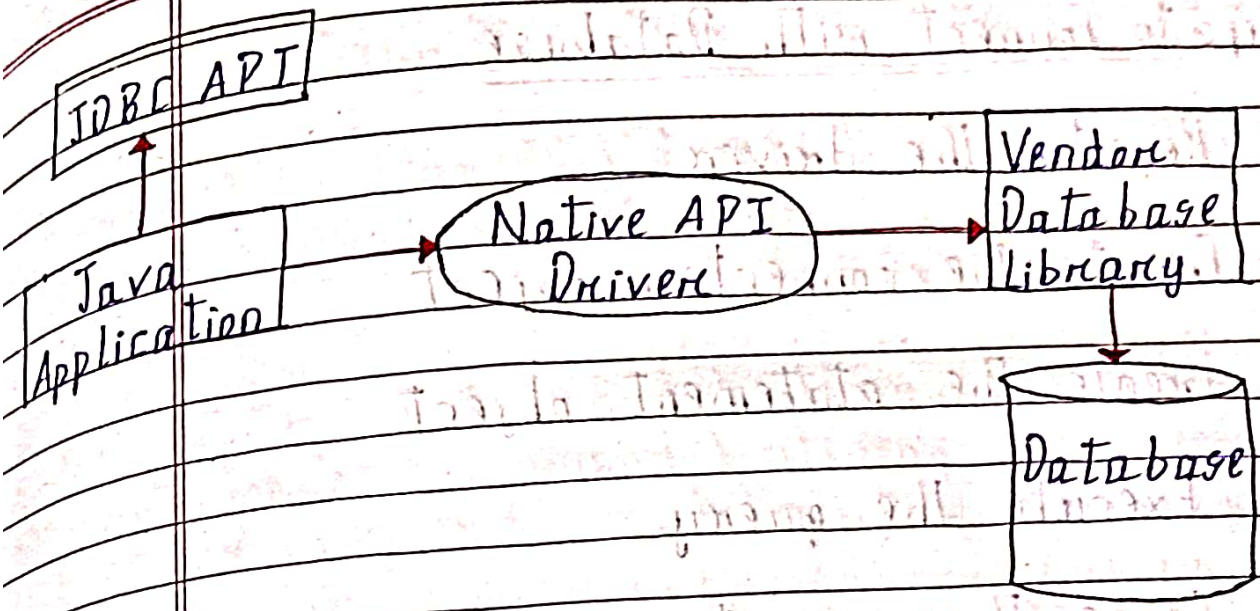
Now, the JDBC-ODBC bridge ~~driv~~ has been removed.

Pros : i) Easy to use
ii) Easily connected to any database

Cons : Performance degraded because JDBC method call is converted into ODBC func° calls.

2) Native API Driver

The driver converts JDBC method calls into native calls of the database API.



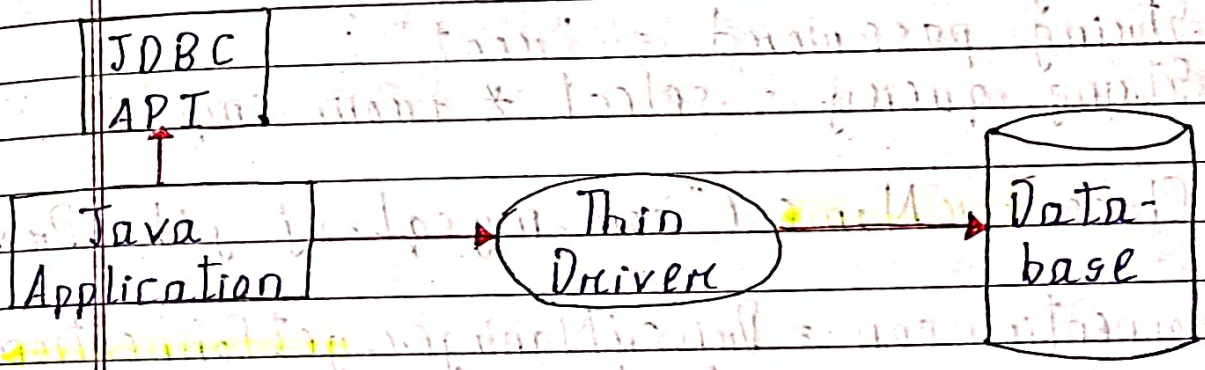
Pros :- Performance upgraded than JDBC-ODBC bridge driver

Cons :- Not entirely written in JAVA.

3) Thin Driver

=> JDBC calls directly into the vendor-specific database protocol.

=> Fully written JAVA language.



Page No.
Date: / /

Steps to Connect with Database

STEP I Register the driver

STEP II Create the connection object

STEP III Create the statement object

STEP IV Execute the query

STEP V Close the connection

JDBC Connection with MySQL

```
import java.sql.*
class Db
{
public static void main(String arg[]) throws
                                    Exception
{
  String url = "jdbc:mysql://local host:
               3306/xyz";
  String username = "root";
  String password = "root";
  String query = "select * from emp";

  Class.forName("com.mysql.cj.jdbc.Driver");

  Connection con = DriverManager.getConnection
                   (url, username, password);
```

```
System.out.println("Connection Established
                    Successfully");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(query);
while(rs.next())
{ String name = rs.getString("name");
  System.out.println(name);
}
st.close();
con.close();

System.out.println("Connection Closed...");
}
}
```

JDBC Connection with Oracle

```
import java.sql.*
class Db2
{
public static void main(String arg[]) throws
                    exception
{
Class.forName("oracle.jdbc.driver.OracleDri-
ver");
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:
xe","system","system");
```

Page No.
Date: / /

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select * from emp");
while(rs.next())
{
System.out.println("Eid: "+rs.getInt(1));
System.out.println("Ename: "+rs.getString(2));
System.out.println("Esal: "+rs.getFloat(3));
}
st.close();
con.close();
}
}
```

Statement

=> This interface is used for executing simple SQL statements without parameters.

=> Compiled & executed each time it's run.

=> To create the statement object we have to use the createStatement() method of statement interface.

Prepared Statement

=> It represents a precompiled SQL statement that can be executed multiple times.

Page No.
Date: / /

=> It accepts parameterized SQL queries, with ? as placeholders for parameters, which can be set dynamically

```
PreparedStatement ps = con.prepareStatement
                        (query);
String query = "Insert into people(name,
                age) Values (?,?)";
ps.setString(1, "Ayan");
ps.setInt(2,25);
```

Callable Statement

=> It is used to execute stored procedures in the database.

=> Useful for executing complex operations that involve multiple SQL statements.

```
CallableStatement cs = con.prepareCall(
                "{call ProcedureName(?,?)}");
```

Page No.
Date: / /

Event Handling

=> An event is a change in the state of an object triggered by some action such as clicking a button, moving cursor etc.

=> java.awt.event package provides various event classes to handle these actions.

=> Event Handling is a mechanism that allows programs to control events & define what would happen when an event occurs.

=> Mainly consists of 2 components :

Source : It is an object on which event occurs. It is responsible for providing information of the occurred event to it's handler.

Listener : It is used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

S. NO	Event Class	Listener Interface	Methods
1	ActionEvent	ActionListener	actionPerformed()
2	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()

Page No.
Date: / /

3	ComponentEvent	ComponentListener	componentMoved() componentShown() componentResized() componentHidden()
4	ContainerEvent	ContainerListener	componentAdded() componentRemoved()
5	FocusEvent	FocusListener	focusGained() focusLost()
6	ItemEvent	ItemListener	itemStateChanged()
7	keyEvent	keyListener	keyTyped() keyPressed() keyReleased()
8	TextEvent	TextListener	textChanged()
9	MouseEvent	MouseListener	mousePressed() mouseClicked() mouseEntered() mouseReleased()
		MouseMotionListener	mouseMoved() mouseDragged()
10	MouseWheelEvent	MouseWheelListener	mouseWheelMoved()

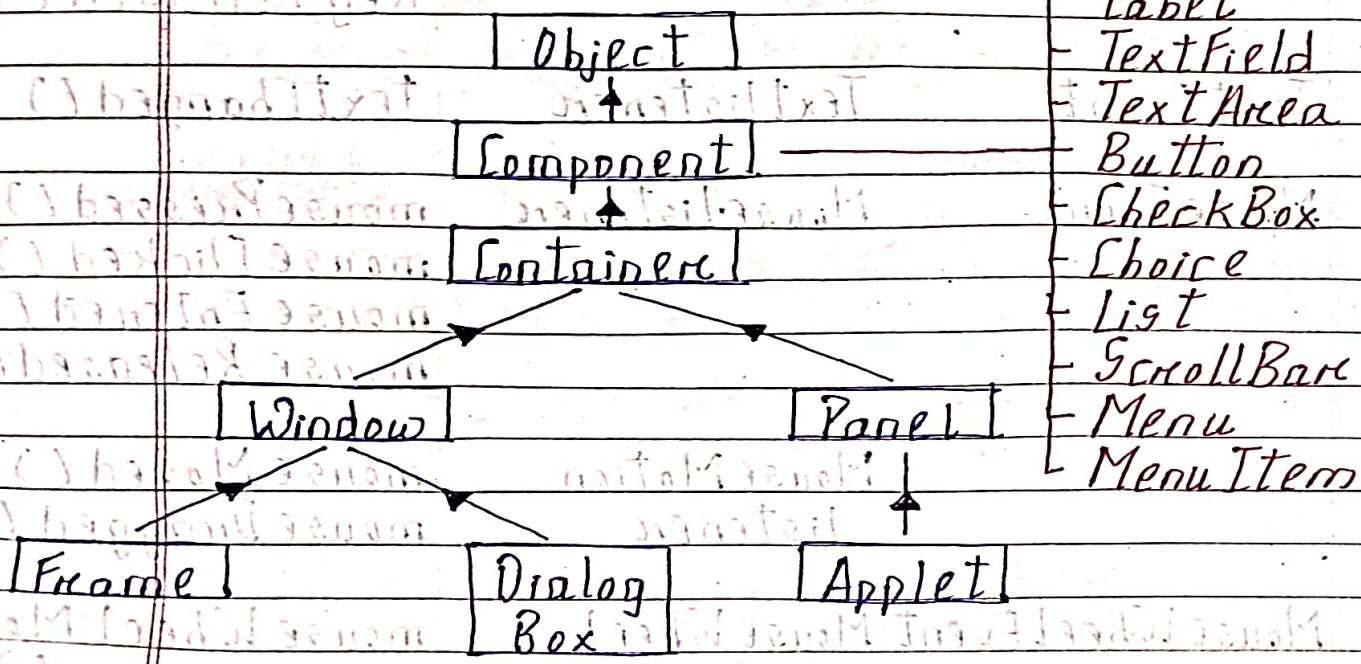
Page No.
Date: / /

11	WindowEvent	WindowListener	windowActivated() windowDeactivated() windowOpened() windowClosed() windowClosing() windowIconified() windowDeiconified()

AWT (Abstract Window Toolkit)

=> It is a collection of pre-defined classes to develop GUI type of application.

=> java.awt package handles all these operations.



Page No.
Date: / /

Type of Containers

1) **Window** : Represents a graphical or dialog box. Extends the container class which contains other components such as buttons, labels & textfields.

2) **Panel** : Lightweight container that can be used for grouping other components together within a frame.

3) **Frame** : Contains the title bar & borders & can have menu bars.

4) **Dialog Box** : Temporary window an application creates to retrieve user input.

1) **Label**

Constructors : Label()
Label(String)
Label(String, int Alignment)

Alignment : Label.LEFT
Label.RIGHT
Label.CENTER

Methods : getText()
setText(String)
setBackground(Color)
setForeground(Color)
setFont(Font)

2) TextField

Constructors :- TextField ()
TextField (String)
TextField (int cols)
TextField (String, int cols)

Methods :- getText ()
setText (String)
getEchoChar ()
setBackground (Color)
setForeground (Color)
setFont (Font)

3) Button

Constructors :- Button ()
Button (String)

Methods :- getText ()
setText (String)
setFont (Font)
setBackground (Color)
setForeground (Color)

Page No.
Date: / /

Layout Manager

=> To arrange the components in the screen following are the layouts :-

1) Flow layout
2) Grid layout
3) Border layout
4) Grid Bag layout
5) Card layout

1) Flow Layout

=> Used to create this layout in left to right sequence.

Constructors :- FlowLayout()
FlowLayout(int hgap, int vgap)

=> Default layout of applet container.

2) GridLayout

=> Screen is divided equally in specific rows & columns.

Constructors :- GridLayout()
GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap, int vgap)

Page No.
Date : / /

3) Border Layout :

=> Dividing the screen into 5 regions (East, West, North, South, Centre)

=> Constructors : BorderLayout ()
BorderLayout (int hgap, int vgap)

4) Grid Bag Layout

=> Flexible layout

=> Divide containers into specific rows & columns & can adjust its size.

=> To specify rows & columns, we can use another pre-defined class i.e. GridBagConstraints.

=> Datafields of GridBagConstraints :
a) gridx (col index)
b) gridy (row index)
c) gridwidth
d) gridheight
e) weightx = 1
f) weighty = 1
g) fill