LectureNo.1 Overview of oop

Topic(s) to becovered	oop - Advantages - object oriented programming paradigms - Features of object oriented Programming - Java buzzwords.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
LO1	To understand object and classes	understand
LO2	To know the advantages of oop	understand
LO3	To describe oop paradigms	Understand
LO4	To list out the features of oop.	Remember

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Overview of oop:

* oop is a programming paradigm.
* It is based upon objects
* It aims to incorporate modularity and reusability.
* object oriented programming is about creating objects, but the olden procedural programming

is about writing procedures (or) methods that performs operations on data.

* objects contain both data and methods.

* A class is a template for objects and an object is an instance of a class.

---

object oriented programming (oop) features:

1. programming design follows bottom-up approach.
2. programs are organized around objects and grouped in classes.
3. Reusability of design [creating new classes by adding features to existing classes].
4. Focuses on data and methods to operate upon object's data.
5. Interaction b/w objects through functions.
6. Data is hidden from external functions.
7. objects may communicate with each other through a function called message passing.

OOP advantages:

* Fastes and easies to execute.
* provides clear structure for the programs
* Makes the code easies to maintain, modify and debug.
* Helps to create reusable applications with less code and shorter development time.

Object oriented programming paradigm:

1. Objects
2. Classes
3. Encapsulation
4. Abstraction
5. Inheritance
6. polymorphism
7. Messaging.

Objects:

objects are real or abstract items that contain data to define object and methods to manipulate that information. Objects are the combination of data ~~of~~ and methods.

Classes:

* Class is a group of objects that has the same properties and, behavior and, the same kind of relation-ship and semantics.

* Objects are the variables of the class. Each object is associated with the data type of the class.

* Classes are the collection of objects of a similar type.

Encapsulation:

It is the process of wrapping up data and functions into a single unit. It makes the data not accessible to the outside world. The functions which are wrapped in the class can access the data. It provides the interface b/w data objects & program.

Abstraction:

It represents essential features and doesn't represent the background details & explanations. (ie) It hides unnecessary information from the users. It defines set of abstract attributes (or) methods ~~to access~~ (or) class as abstract.

Inheritance:

one class extends another class's properties, including additional methods and variables. Original class is called as super class and the class that extends another class is called as sub class.

polymorphism

It is the ability to take more than one form. An operation may exhibit different behaviour in a different instance. Behaviour depends on the types of data used for the operation.

Messaging:

object oriented system consists of sets of objects that communicate with each other. objects communicate one another by sending/receiving data.

It invokes a method of receiving object that ~~gene~~ generates desired result.

Java buzzwords: (Characteristics of Java)

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multi-threaded
7. Architecture-neutral
8. Interpreted
9. High performance
10. Distributed
11. Dynamic.

Simple:

Java is designed to be easy for the programmer to learn and use effectively. Java inherits the syntax of C/C++ and many of the object-oriented programming features of C++. so the learning becomes will require only little effort.

Object oriented:

* Java is object-oriented programming language.
* Like C++, It provides most of the O-O-features.
* The object model in Java is simple and easy to extend. But the primitives types like Integers are non-objects.

Robust:

Java encourages error-free programming by being strictly typed and performing run-time checks.

Multithreaded:

Java provides integrated support for multithreaded programming. more than one process (threads) run simultaneouly in multithreading

Architecture Neutral:

* Java is not tied to a specific machine (or) operating system structure.
* Thus Java is Machine Independant.

Interpreted:

Java supports of cross-platform code through

the use of Java bytecode.

Bytecode can be interpreted on any platform by JVM (Java Virtual Machine)

* Thus Java is platform independent.

Distributed:

* Java is designed with the distributed environment.
* Java can be transmitted over Internet.

Dynamic:

* Java programs carry substantial amounts of 'run time type' information.
* Using that information, verification and resolving accesses of object at runtime, is possible.

Suggested Questions / Assignments / Home works / any other
1. Explain the features and characteristics of Java. 2. Give the differences b/w oops and procedural programming. 3. Narrate the object oriented programming paradigm. 4. list out the advantages of Java.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1.	Java: The Complete Reference	Herbert schildt	Mcgraw Hill. 2019.

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Overview of Java.

Topic(s) to becovered	Datatypes - Integers, floating point, characters - Boolean - variables - types.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
LO1	to know all the primitive and non-primitive types	Understand.
LO2	to write programs for datatypes.	apply.
LO3	To know the types of variables, declarations.	understand
LO4	to use variable declarations in programs	Apply

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate / Discuss / Peer to Peer Learning / Quiz / Role Play / Any other

Lecture Notes

Datatypes:

* Java is a strongly typed language to provide safety and robustness. Java Compiler checks all expressions and parameters to ensure that the types are compatible.

Primitive data types (or) Simple types:

* Java defines 8 primitive types of data

1) Byte
2) Short
3) Int
4) long
→ Integers

5) char → Characters

6) float
7) double
→ floating point numbers

8) Boolean.

Integers:

* This group represents whole-valued signed numbers [+ve, -ve].
* This group includes byte, short, int and long datatypes.
* width and range are displayed as follows.

Name	width	Range (2^{width})
long	64	-9223372036854 775808 to 9223372036854775807
int	32	-2147483648 to 2147483647
short	16	-32,768 to 32767.
byte	8	-128 to 127.

---

byte:

* smallest integer type
* signed 8 bit number type, ranges from -128 to 127.

long:

* 64 bit type
* used, when int is not large enough to hold the desired value.
* used to store big, whole numbers.
* declaration:

```
long b;
```

Example program:

```
// compute distance travelled by light.
class light
{
public static void main (String args[])
{
    int lightspeed = 186000; // assume.
    long days      = 1000;   // assume.
    long seconds, distance;
    seconds = days * 24 * 60 * 60;
    distance = seconds * lightspeed;
    System.out.println("Distance travelled by
                light in " + days + ": " + distance);
}
}
```

* It is very useful when we use a stream of data transfer in a network or file.
* Also it is useful to work with raw binary data.
* Byte variables declaration:

```
ex: byte b, c;
```

keyword — variable names.

Short:

* Short is a 16 bit type.
* It's range -32768 to 32767.
* Least used Java type.
* variable declaration:

```
ex: short t;
```

keyword. variable name

---

int:

* mostly used integer type is int.
* 32 bit type
* Range -2,147,483,648 to 2,147,483,647.
* used in controlling loops and indexing arrays.

declaration ex:

```
int a;
```

Floating point types: (real numbers)

* This group includes float and double, which represents number with fractional precision.
* They are useful when evaluating expressions
* Example: calculations such as square root(or) transcendentals such as sine and cosine needs floating point type.
* Java implements a standard (IEEE-754) to set of floating point types.
* Two kinds of floating point types: 1. float 2. double

Name	Width in bits	App. Range
double	64	$4.9\ e^{-324}$ to $1.8\ e^{+308}$
float	32	$1.4\ e^{-045}$ to $3.4\ e^{+038}$

float:
* Specifies a single-precision value.
* 32 bits for storage / width.
* It is faster than double precision on some computers.

* It takes half space as double precision.
* It becomes imprecise when the values are either very large (or) very small.
* float is useful when large degree of precision is not important.

declaration:

```
float highTemp;
```

double:

* It denotes double precision.
* 64 bits width.
* fastes than single-precision on some modern processors.
* All transcendental math functions (cos, sin, sqrt...) return double values.
* To maintain accuracy in calculations, double is used.
* declaration:

```
float a,b;
```

Many iterative calculations (or) manipulation of large numbers needs double.

// Program to compute area of circle:

```
class area {
    public static void main (String args[]) {
    double pi, rad, area;
    pi = 3.1416;
    rad = 10.5; // radius.
    area = pi * rad * rad; // Compute area
    System.out.println ("Area of circle is" + area);
}
}
```

Characters:

* Char datatype is used to store characters.
* char in java is different from C/C++.

In C/C++, char is 8 bits wide. But in java, char is represented by unicode.

* width 16 bits.
* range 0 to 65,536. (no negative characters)
* The standard set of characters known as ASCII ranges from 0 to 255.
* Eventhough unicode contains some inefficiency, it is used for global portability.

* declaration: `char ch1, ch2;`

1) // Program for demonstrating characters.

```
class charDemo {
    public static void main(String args[])
    {
        char ch1, ch2;
        ch1 = 88;      // decimal code 'X' (Refer ASCII set)
        ch2 = 'Y';
        System.out.println("ch1:" + ch1);
        System.out.println("ch2:" + ch2);
    }
}
```

```
// Answer: (output)

ch1 : X
ch2 : Y
```

2) // Program to show char variable behave like integers.

```
class charDemo2 {
    public static void main(String args[]) {
```

```
char ch1 = 'x'; // decimal value 88
System.out.println("ch1 is" + ch1);
ch1++; // decimal value 89. ← 88+1
System.out.println("ch1 now becomes" + ch1);
}
}
```

O/P:

```
ch1 is x
ch1 now becomes y.
```

Booleans:

* This datatype is for logical values.
* It can have only true (or) false values.
* It is the return type of all relational operators and required by conditional statements (expressions)
* declaration: `boolean ans;`

```
// Program to demonstrate boolean values.
class BoolTest {
    public static void main(String args[])
    {
        boolean b;
```

```
    b= false;
System.out.println(" value of b"+ b);
    b = (10>9);
System.out.println(" value of b"+ b);
}
}
```

O/P:

```
value of b is false.
value of b is true.
```

Variables:

* Basic unit of storage in java program.
* definition: A variable is a combination of an identifier, a type and an optional initializer.
* All variables have scope to define its visibility and a life time.
* declaration of a variable:

Syntax:

```
type identifier;
type identifier = value;
type identifier1 = value1, identifier2 = value2;
```

Example:

```
int a, b, c; // declare 3 ints. a, b, c.
int k=3, e, f=5;   // declare 3 ints k, e, f and
                   // initializing k and f.
byte z=22; // declare and initialize z. as byte.
double avg=98.2; // declare avg as double and initialize
char x = 'x'; // variable x has 'x'
```

Dynamic initialization:

Java allows variables canbe initialized dynamically using any expressions, at the time the variable declared.

Ex: `double c = Math.sqrt(a*a + b*b);`

The scope and lifetime of variables:

* Java allows variables to be declared within any block.
* A block is begin with ({) open curly brace and ended with (}) closed curly brace.
* A block defines scope.
* Scope definition: A scope determines what objects are visible to other parts of ~~your~~ the program. It also determines the lifetime of those objects.

2 types of scope: (1) global (2) local. [other languages]

* Types of scopes in java:

(1) defined by class (or) class scope

(2) defined by a method (or) method scope.

* The scope defined by a method begins with its opening curly brace.

* When a variable is ~~defined~~ declared within in the scope, it is said to be localised and protected from unauthorised access (or) modification. This is need of scope.

* variables declared inside a scope are not visible to code that is defined outside of the class.

* Nested scope: outer scope encloses, inner scope. The objects declared in the outer scope will be visible to inner scope. But the reverse is not true.

Example:

```
// Demonstration of block scope
class scope {
  public static void main(String args[])
   {
  int x=10;
    if(x==10)
    {
     int y=20;
     System.out.println("x and y:"+x+" "+y);
    }
```

```
        System.out.println("x and y is" + x + "  "+y);
                                // Error! y is not known here.
    }
}.
```

Explanation:

x is declared at main()'s scope. so it is visible till the end of the main().

But, y is declared with in the if()'s scope and not accessible to main().

Type Conversion and Casting.

Type Conversion: Conversion of one data type to another datatype.

Automatic type conversion:

* when one type of data is assigned to another type of variable, an automatic type conversion will take place.

* Conditions:
  1* The two types should be compatible
  2* The destination type is laeger than source type.

* Type casting:

* It performs an explicit type conversion between incompatible types.

* syntax: (target type) value.

* Example:

(1)
```
int a;
byte b = 120;
a = (byte) b;
```

(2)
```
int a = 140;
byte b;
b = (byte) a;
```

---

* Truncation: when an floating point value is assigned to integer, truncation will happen.

---

* Example for type conversion and casting:

```
class conversion {
  public static void main(String args[])
  {
      byte b;
      int i = 257;
      double d = 323.142;

b = (byte) i; // Conversion from int to byte
System.out.println(" i and b" + i + " " + b);
```

```
i = (int) d ;  // Conversion of double to int.
System.out.println("d and i" + d + "   " + i);
b = (byte) d;  // conversion of double to byte.
System.out.println(" b and d" + b + "  " +d);
}
}
```

Answer / output:

i and	b	257	1
d and	i	323.142	323
d and	b	323.142	67.

Note:1 Converting to byte: divide by 256, and use remindes.

257 % 256 → 1.

323 % 256 → 67.

Type promotion:

Note 2: * Java automatically promotes byte, short or char operand to int, when evaluating expressions.

* If one operand is long, whole expression will be promoted to long.

* same for double, float.

Suggested Questions / Assignments / Home works / any other
* list out the primitive datatypes. * define byte. * what is the difference between type casting and type conversion? * write a Java program to find area of circle. * what is boolean datatype? * Define variable. * Explain about scope and types of scope with example.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1.	Data Structures and Algorithm Analysis in C, 2nd Edition, 2005	Mark Allen Weiss	Pearson Education

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Arrays.

Topic(s) to becovered	one dimensional assays – Multi dimensional assays – string datatype – Examples.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
LO1	describe assay and its types	Understand.
LO2	differentiate 1-D and multi-D assays.	Understand
LO3	understand string datatype	Understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Assays: (Collection)

* An assay is a group of same-type variables, that are refered by a common name.

* Types:
  1. One-dimensional assay.
  2. Multi-dimensional assay.

* Elements of an assay can be accessed by index.

One dimensional Array: (Or) 1-D array:

* A list of same-typed variables
* Syntax: type varname[];
* Example: int masks[];
* Allocation of memory to arrays:

  Syntax: arrayname = new type[size];

  example: masks = new int[5];

* Example:

1)
```
class ArrayExample {
    public static void main (String args[])
    {
        int month_days[] = {31, 28, 31, 30,
                            31, 30, 31, 31, 30, 31, 30, 31};
        System.out.println("Apsil has" + month_days[3]
                           + "days");
    }
}
```

2.
```
class Average {
    public static void main (String args[])
    {
```

```
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
int i;
for (int i=0; i<5; i++)
    result= result + num[i];
System.out.println("Average is " + result/5);
}
}
```

Multi-dimensional assay: Assays of assay.

* when an assay has more than one dimension, it is multi-dimensional assay.

Ex: 2-D assay (2-dimensional assay)

Ex: `int matrix[][] = new int[3][2];`

This will create a matrix with 3 rows and 2 columns.

Example program:

```
Class Matrix {
    public static void main (String args[])
    {
        int matrix[][] = new int[3][2];
        int k = 0;
```

```
for (int i=0; i<3; i++)        // storing value into 2-D
                               // matrix
{
  for (int j=0; j<2; j++)
  {
     matrix[i][j] = k++;
  }
}

// To display 2-D matrix
  for (int i=0; i<3; i++)
  {
     for (int j=0; j<2; j++)
     {
       System.out.print(matrix[i][j] + " ");
     }
     System.out.println();
  }
}
```

O/P:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Alternative array declaration syntax:

```
type [] varname;
```

Ex: `int a1 [] = new int [3];`

⇓

`int [] a1 = new int [3];`

This alternative form is useful when specifying an array as returntype.

Strings:

* String datatype is used to declare string variables.
* A double quoted string constant can be assigned to string variable.
* array of strings also possible.
* syntax: `String var = "value";`
* Example. `String name = "Java";`

```
String subjects [] = new String [4];
String subjects [] = {"Java", "DS", "DM", "Python"};
```

Example program:

Program to sort (or) arrange the names in ~~Ascending~~ Alphabetical order.

```
import java.io.*;

class Test {
    public static void main(String args[])
    {
        int n = 4;
        String names[] = { "Rahul", "Rita", "Gousia", "Riya" };
        String temp;
        for(int i=0; i<n; i++)
        {
            for(int j=i+1; j<n; j++)
        if(names[i].compareTo(names[j]) > 0) {
            // swap
            temp = names[i];
            names[i] = names[j];
            names[j] = temp;
        }

// to print the output assay.
System.out.println(" Alphabetical orders");
```

→

```
for( int i=0; i<n; i++)
 {
    System.out.println( names[i]);
  }
}
}
```

Output:

Alphabetical order
Gourav
Rahul
Rita
Riya.

Suggested Questions / Assignments / Home works / any other

1. write a java program to display minimum element in an array.
2. write a java program to display student grade. 5 marks stored in array.

Questions:

* Define array. what are the types of array.
* explain one-D and 2-D arrays with example.
* Explain the usage of String datatype.

Text Books / Reference Books / Any other suggested Materials

S.No	Title	Author	Publisher
1.	Data Structures and Algorithm Analysis in C, 2nd Edition, 2005	Mark Allen Weiss	Pearson Education

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Operators.

Topic(s) to be covered	Types - Arithmetic - Bitwise - Relational and logical operators - examples - operator precedence & priority.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	Explain the types of operators	understand
Lo2	apply the operators to write expressions	Apply
Lo3	describe the operator precedence	understand

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Operators:

They are used to perform operations on variables and values.

Ex: `int x = 100 + 10`

Here '+' is used to add two values.

`int y = x + 5;`

Here + is used to add variable & value.

Expression:

* It has 2 parts < operator / operands.

* The variables (or) constants that operators can act upon are called as operands.

* An operator is a java symbol (or) a keyword that tells the compiler to perform a specific mathematical (or) logical operations.

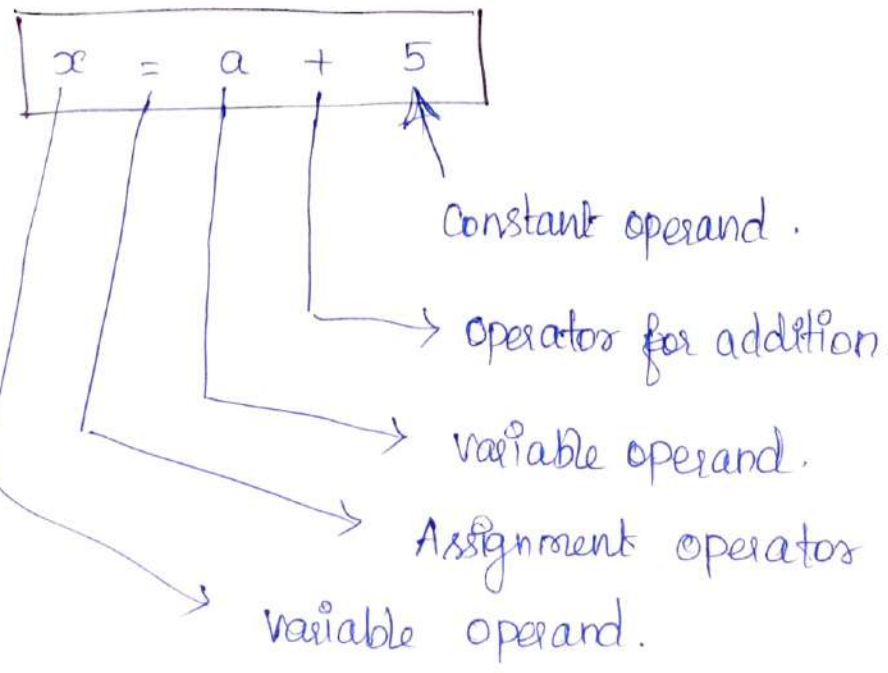
* operators used to manipulate the data and variables in program.

* They generally form mathematical (or) logical expressions.

* An expression is a combination of variables, constants and operators.

Example:

- Expression.



Operators:

Four groups: (or) types

1) Arithmetic operators
2) Bit-wise operators
3) Relational operators
4) Logical operators.

1. Arithmetic Operators: used in mathematical expressions. operands must be of numeric type.

Operator	Result	
+	Addition	
-	Subtraction (or) Unary minus	
*	Multiplication	
/	Division	
%	Modulus	
++	Increment	
+=	Addition assignment	Compound assignment
-=	Subtraction assignment	
*=	Multiplication assignment	
/=	Division assignment	
%=	Modulus assignment	
--	Decrement	

The Basic arithmetic operators: Addition, subtraction, multiplication and division. Unary minus has single operand.

Example:

```
class BasicMath
{
public static void main(String args[])
{
 // Integer arithmetic
 System.out.println("Integer Arithmetic");
     int a = 1+1;
     int b = a*3;
     int c = b/4;
     int d = c-a;
     int e = -d;
System.out.println("a "+a+" b "+b+"c "+c+
                   "d "+d+" e "+e);
//Double arithmetic.
 System.out.println("Floating point arithmetic");
     double da = 1+1;
     double db = da*3;
     double dc = db*/4;
     double dd = dc-da;
     double de = -dd;
System.out.println("da "+da+"db "+db+"dc "+dc+"dd "+
                   dd+"de "+de);
 }
}
```

Op:

Integer Arithmetic

a 2 b 6 c 1 d -1 e 1

Floating point arithmetic

da 2.0 db 6.0 dc 1.5 dd -0.5 de 0.5

Modulus operator (or) Modulo %

* This operator returns the remainder of a division operation.

Example:

```
class Modulus
{
public static void main (String args[])
{
        int x = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
}
}
```

Output:

x mod 10 = 2

y mod 10 = 2.25

Arithmetic Compound assignment operators.

Java has special operators to combine arithmetic and assignment operations.

Old Syntax: var = var op expression

↓

Syntax for Compound assignment: var op = expression.

Example: a = a + 4 → a += 4

a = a % 2 → a %= 2.

Increment and Decrement:

Increment operator (++): Increases its operand by one.

Decrement operator (--): Decreases its operand by one.

Example:

operand

	Postfix		Prefix.
x = x + 1	x++	(or)	++x
x = x - 1	x--	(or)	--x

Prefix form:

operand is incremented (or) decremented before, the value is used in expression.

Postfix expression form:

* Here the operand's value is used in expression, then it is incremented (or) decremented.

Example:

```
x = 42;      x=42      a=6;      a=6;
y = x++;     y=++x;    b=a--;    b=--a;
```

Ans:

```
x = 43;      x=43      a=5;      a=5
y = 42;      y=43      b=6       b=5;
```

Bitwise operators:

* These operators can be applied to integer types (long, short, int, byte) and char.
* These operators act upon the individual bits of operand.

operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bit wise exclusive OR
>>	shift right
>>>	shift right zerofill.
<<	shift left

&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	shift right assignment
>>>=	Shift right zero fill assignment
<<=	shift left assignment

The Bitwise logical operators : &, |, ^ and ~

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The bitwise NOT: ~ (or) Bitwise complement

* It inverts all of the bits of its operand.

Ex:

42 → 00101010

↓ NOT

11010101

The bitwise AND : &

* It produces 1, if both operands are 1.

It produces 0, in all other cases.

Ex! 42 → 0 0 1 0 1 0 1 0
15 → 0 0 0 0 1 1 1 1 &

Ans: 10 ← 0 0 0 0 1 0 1 0

Bitwise OR : |

* It combines the bits such as if either of the bits is 1, then result will be 1.

Ex!

42 → 0 0 1 0 1 0 1 0
15 → 0 0 0 0 1 1 1 1 | ← OR operand.

Ans: 47 ← 0 0 1 0 1 1 1 1

Bitwise XOR: ^

* It combines the bits such that, if exactly one bit is 1, then result will be 1.

Ex!

42 → 0 0 1 0 1 0 1 0
15 → 0 0 0 0 1 1 1 1 ^

37 ← 0 0 1 0 0 1 0 1

left shift: << equal to multiply by 2.

* It shifts all of the bits in a value to the left, for a specified numbers of times.

* **syntax:** value <<num.

↳ no. of positions to left shifts.

* Higher order bit is shifted out (or) lost, and a '0' is attached to right.

Ex: int i = 64 << 2;

O/P: i = 256.

execution:

64 → 01000000

1st shift (left) ← → 10000000

2nd shift left ← → 100000000

↓

256

Right shift: >> is equal to divided by 2.

* It shifts all of the bits to right, for specified numbers of times.

syntax: value >> num;

ex:

int a = 32;

a = a >> 2;

O/P: a = 8

execution:

32 → 00100000

→

1st shift right → 00010000

2nd shift right → 00001000

↓

8

Relational operators:

- used to determine the relationship, equality or ordering that one operand with other.

Operator	Result.
==	Equal to
!=	Not equal to
>	greater than
<	less than.
>=	greater than or equal to
<=	less than or equal to.

(result)
* out come of relational operation is boolean.
* used to control loops and conditional statements.

Ex:
```
int a = 4;
int b = 2;
boolean c = a < b;
```

o/p: c is false.

Boolean logical operators: they combine two boolean values into one boolean resultant value.

Operator	Result
&	logical AND
|	logical OR
^	Logical XOR (exclusive OR)
||	Short circuit OR
&&	Short circuit AND
!	logical unary NOT
&=	AND Assignment
|=	OR assignment
^=	XOR assignment
==	Equals to
!=	Not equal to
?:	Ternary if-then-else.

Assignment operator: '=' sign. (ie) equal

Syntax: var = expression.

ex: int a = 10;

int b = a + 5;

int c = d = e = b;

?: operator:

* Ternary operator: (three - way).
* equivalent to if - then - else.

Syntax:

```
expression1 ? expression2 : expression3
```

expression1: Expression to be evaluated. Result will be boolean.

Expression2: If result of expression 1 is true, expression 2 will be evaluated. (or) taken.

Expression 3: If result of expression 1 is false, expression 3 is evaluated (or) taken.

example:

```
int i=10, k;

k = i<0 ? -1 : 1;
```

o/p:

k = 1

because i<0, false
So 1 is assigned to k.

Operator precedence: It determines the order in which the operations in an expression is done.

Highest precedence

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

Lowest precedence

Associativity:
In an expression of 2 or more operators with same precedence, then the expression is evaluated based on associativity.

Ex:
Assignment operators: Right to left
Prefix, ~~post~~ ++, -- and unary: Right to left.
All other operators: left to right

[ ] → Indexing
. → dereference objects.

Parenthesis: ( ) → used to alter the precedence of an operation.

ex: $(a+b) * c \neq a + (b * c)$

Suggested Questions / Assignments / Home works / any other
* what is operator ? what are the types of operators? * write a java program to check the given number is odd or even. * write a java program to calculate the sum of 3 digit numbers. * what is operator precedence? How ambiguity solved? * write the difference b/w the post increment and pre-increment operator.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1.	Data Structures and Algorithm Analysis in C, 2nd Edition, 2005	Mark Allen Weiss	Pearson Education

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Control Statements.

Topic(s) to be covered	selection statements - Nested if - else-if ladder, - switch - Iteration statements (loop) - while - do while - for - foreach - Nested loop Jump statements - break, exit, continue.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	List out the types of control statements	Under-stand
LO2	apply the control structure to write programs	apply.
LO3	Explain about the jump statements	apply.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Decision making statements: (Selection statements).

1. if → Simple if, if-else, if-then-else, nested if..
2. switch-case

if: Conditional branch statement

Syntax:

```
if (Condition) statement1;
```

Here,

Condition → any expression that returns a boolean value.

Statement(s) → set of statements to be executed.

if-else:

syntax:

```
if (condition) statement1;
else statement2;
```

Working: If condition is true, statement1 will be executed. If the condition is false, else part will be working and statement2 will be executed.

Nested-ifs:

syntax:

```
if ( Condition)
   if (Condition)
   . statements1;
   else
      statements2;
else
      statement 3;
```

if-else-if ladder: sequence of nested ifs.

Syntax:

```
if (Condition1)
  statement1;
else if (Condition2)
  statement2;
else if (Condition3)
  statement3;
  :
else
  statementn;
```

Working: The if statements are executed from top to down. If any one of the Condition is true, the corresponding statement will be executed. If no conditions is true, then else part will be executed. This acts like default condition.

Examples:

1) Simple if:

```
if ( 10 > 5)
    System.out.println("10 is greater");
```

→

2. if-else:

```
if (10>5)
System.out.println("10 is greater");
else
System.out.println("5 is greater");
```

3. if-else-if ladder:

```
int month = 4;  // April
String season;
if (month == 12 || month == 1 || month == 2)
    season = "Winter";
else if (month == 3 || month == 4 || month == 5)
    season = "Spring";
else if (month == 6 || month == 7 || month == 8)
    season = "Summer";
else season = "not a valid month";

System.out.println("Season is" + season);
```

4) nested-if:

```
int a = 5, b = 10, c = 2;

if (a>b) {
   if (a>c) {
      System.out.println("a is greater");
   }
```

```
else {
    System.out.println(" c is greates");
    }
}
 else {
     if (b>c)
     {
   System.out.println(" B is greates");
     }
     else
     System.out.println(" C is greates");
   }
  }
```

Switch - case: - multiway branch statement.

- easy way to dispatch execution to different parts of your code, based on the condition.
- alternative to large else-if series.

Syntax:

```
switch (expression)
{
case value1:
        statements; break;
```

```
case value2:
        statements;
        break;
        .
        .
default:
        default statements;
}
```

Working: * The value of expression is compared with each of the literal values in case statements.

* If match found, the code sequence (statements) following the case statement is executed.

* If no case matches, default statement will be executed.

Example:

```
class Season {
    public static void main(String args[])
    {
        int month = 4;
        String season;
        switch (month)
        {
```

```
        case 12:
        case 1:
        case 2:  season = "Winter"; break;

        case 3:
        case 4:
        case 5:  season = "spring"; break;

        case 6:
        case 7:
        case 8:  season = "Summer"; break;

        case 9:
        case 10:
        case 11: season = "Autumn"; break;

        default:
                season = "Not a valid month";
    }
    System.out.println(season);
  }
}
```

```
C:\> spring
```

Iteration Statements (or) Looping statements:

Loop will execute the same set of statements repeatedly until a termination condition is met.

* types:

1) for
2) while
3) do-while.

while:

- fundamental loop statement
- It repeats a statement or block while its expression (or) condition is true.
- Syntax:

```
while ( condition)
{
  // statements;
}
```

- when the condition is false, control passes to the next line of code after the loop.

Example:

```
class Example{
public static void main( String args[])
{
   int n=5;
   while (n>0)
      System.out.println( "tick " + n--);
}}
```

o/p:

```
tick 5
tick 4
tick 3
tick 2
tick 1
```

Loops

do-while:

* used to execute the body of the loop atleast once, even if conditional expression is false to begin.

* Syntax:

```
do {
    // body of the loop
    // statements;
    } while (condition);
```

* Example:

```
class Example {
public static void main( String args[])
{
    int n=5;
    do{
      System.out.println(" tick " + n--);
    } while (n>0);
} }.
```

---

for loop:

- for
- for-each.

for: Syntax:

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

Working:

- Initialization portion will work, when loop first starts.
- Condition has a loop control variable, acts as a Counter to control the loop.
- If condition is true, body of the loop will be executed.
- Then Iteration part will be executed.
- once condition is false, ~~Condition~~ control will come out of loop.

Example:

```
Class Example {
public static void main ( String args[] )
{
    int n;
    for (n=5; n>0; n--)
        System.out.println ( "tick " + n);
}
}
```

O/p:

```
tick 5
tick 4
tick 3
tick 2
tick 1
```

for-each: Enhanced forloop

* Syntax:

```
for (type variable : collection)
{
    statements;
}
```

* Working: Here, the type → datatype of iteration variable. This variable will receive the elements from a collection (Example: Array).

* Example:

```
Class Example {

public static void main (String args[])
{
    int nums[] = {1, 2, 3, 4, 5};
    for (int n : nums)
    {
        System.out.println(n);
    }
}
```

* output:

```
1
2
3
4
5
```

Nested loops: one loop within another loop.

Example:

```
class Example
{
public static void main ( String args[])
{
int matrix [][] = new int [2][2];
int k=1;
for (int i=0; i<2; i++)
 {
    for (int j=0; j<2; j++)
      {
        matrix [i][j] = k++;
      }
 }
}
```

Jump Statements: - to transfer the control to another part of the same program.

Types: (1) Continue
(2) break
(3) return.

Continue:

* It is used to continue the execution of the loop ~~for~~
* It is loop control structure.
* It helps to continue the current flow of the program and

Skips the remaining code at specific condition.

* It can control for, while and do-while loop.

Example:

```
Class Demo
{
public static void main (String args[])
{
    for(int i=0; i<10; i++) {
      System.out.println( i+ "  ");
    if (i%2 ==0)
        continue;
     System.out.println(" "); // prints newline
    }
}
}
```

Working: This code uses the % (modulo) operator to check i is even. If i is even number, then the loop continues without printing a new line.

O/P:

```
0 1
2 3
4 5
6 7
8 9
```

break:

break has 3 uses

(i) It terminates a statement sequence in switch-case.

(ii) It is used to exit a loop.

(iii) It is the new version of 'goto' statement.

Example:

Break to exit a loop: * It forces the immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

* when break is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
class Demo{
public static void main(String args[])
{
  for(int i=0; i<100; i++)
  {
    if(i==10)
      break; // terminates the loop, if i is 10.
      System.out.println("i:"+i);
  }
  System.out.println("loop complete");
}
}
```

return:

* The return statement is used to explicitly return the control from a method.

* It causes program control to transfer back to the caller of the method.

* It immediately terminates the method in which it is executed.

Example:

```
class Demo
{
public static void main(String args[])
{
    boolean t = true();
    if(t) return; // return to caller main().
    System.out.println("Example"); // will not execute.
}
}
```

unreachable code

Control statements:

- selection statements (or) Decision making statements.
- Iteration (or) loop statements.
- Jump statements.

Suggested Questions / Assignments / Home works / any other
* list out the types of control structures. * what is nested if? * what is switch-case statement.. * write a program to find the sum of n natural numbers. * write a program to print the series of even numbers. * Explain about jump statements.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1.	Data Structures and Algorithm Analysis in C, 2nd Edition, 2005	Mark Allen Weiss	Pearson Education

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Defining classes in Java.

Topic(s) to be covered	class – declaring objects – Methods – Return value – Constructors – this – Garbage Collection – Stack –

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	explain about classes and constructors.	Under Stand
LO2	describe about objects and 'this' keyword	Under – Stand
LO3	write programs using classes & objects.	Apply.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Class:

* A class is a group of objects which have common properties.
* It is a template (or) blueprint from which objects are created.
* It is a logical entity. (or) real world entity.
* It represents the set of properties (data) and (or) set of methods common to all objects. (that is)
* It represents state and behaviour of the object.

General format or syntax for class:

```
class classname
{ // member variables
  type variable1;
  type variable2;
    :
  type variablen;

  // member functions (or) methods
  type methodName1( parameter list)
  { .... }.
  type methodName2( parameter list)
  { ... }.
    :
  type methodNamen( parameter list)
  { ... }.
}
```

Keyword.

Instance variables (or) state (or) data (or) properties (or) member variable

functions (or) methods (or) behaviour (or) member methods.

Instance variables: Variables defined within a class are called instance variables. because each instance of the class contains own copy of these variables. Thus data for one object is seperate and unique from other object. It gets memory @ runtime when the object is created.

Member Methods: These are defined in the class to do a particular work on the member variables. It exposes the behaviour of the methods.

Class Advantages: → Code reusability
Code optimization

A Simple class:

create a class called

Example: Box { width, height and depth }, no methods.

```
/* Program used to create the Box class */
class Box
{
  double width;
  double height;
  double depth;
}

// Execution & object creation.

class demo
{
  public static void main (String args[])
  {
      Box  b = new Box(); // object creation.
      b.width = 10;          // assign values to instance variables.
      b.height = 12.5;
      b.depth = 7.2;

     // calculate and display volume.
      double vol = b.width * b.height * b.depth;
      System.out.println("volume of the Box:" + vol);
  }
}
```

o/p: Volume of the Box : 900.

Declaring objects:

* An object is an instance of a class.
* It is ~~both~~ logical ~~and~~ or physical entity.
* An entity that has state and behaviour is called objects. Ex: Car, table, fan, student, Animal etc...
* object is the real world entity. also runtime entity.

Declaring objects: Creating objects for classes is a 2-step process.

(i) declare a variable of the class type.

(ii) Create actual (or) physical copy of object and assign to variable.

Syntax:

```
Classname objname;
```

Ex: (i) `Box b = new Box();` (ii)

Classname — objectname.

'new' keyword: In java new is a operator. This operator dynamically allocates (allocates @ runtime) memory for an object and returns a reference to it.

i.e/ address of the object is created/alloted by 'new'.

Ex 2:

```
~~Box bb = new Box();~~
Box bb; //declare reference to object.
bb = new BB(); //allocate a Box to object.
```

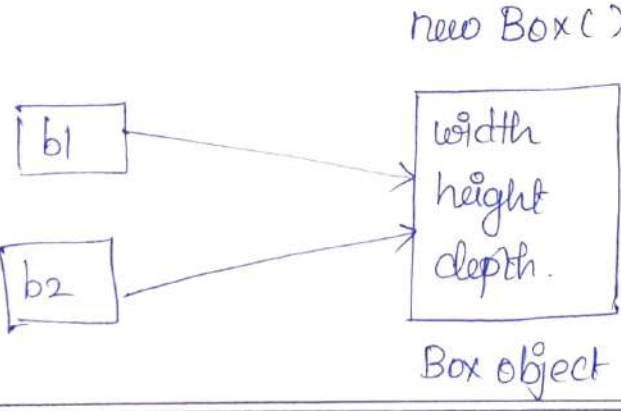
Box b; ⟶ null (b)

b = new Box(); ⟶ memory allocation by new. ⟶ width, height, depth. (box object)

Assigning object reference variables:

```
Box b1 = new Box();
Box b2 = b1; // b1 and b2 refers to the same object.
```

Execution:



Methods:

general form:

```
typename methodname (Parameter list)
{
  : statements;
}
```

type → return type.

parameter list → sequence of arguments passed to the method during function call.

methodname → name of the method.

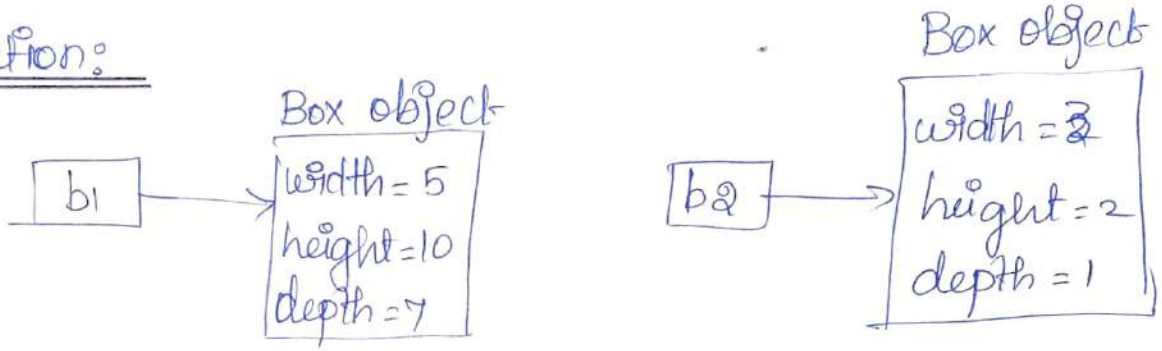
If the method has no parameters, then put empty parenthesis

Adding methods to Box class:

Example:

```
class Box {
    double width;
    double height;
    double depth;

    // display the volume of the box.
    void volume()
    {
     System.out.println("volume is", width * height * depth)
    }
}
class Demo
{
 public static void main(String args[])
 {   // object creation b1
     Box b1 = new Box();
     b1.width = 5;
     b1.height = 10;
     b1.depth = 7;          // assigning values to member variables of b1
     b1.Volume();           // function call.

     Box b2 = new Box();    // object creation b2
     b2.height = 2;
     b2.width = 3;
     b2.depth = 1;          // assign values to member variables of b2
     b2.volume();
}}
```

Execution:



O/P:

Volume is 350

Volume is 6.

Returning value from method:

```
Class Box
{
  int width, height, depth;
  double volume()
  {
    return width * height * depth;
  }
}
Class demo
  {
      Box b1 = new Box();
          b1.width = 3;
          b1.height = 5;
          b1.depth = 2;
        double volume = b1.volume();
        System.out.println("volume is "+volume);
  } }
```

Constructors: & types < default constructor / parameterized constructor

* Initializes an object immediately after creation.

(i.e) Initialize all of the variables in the class each time an object (or) instance is created.

* name of the constructor is same as the class.
* no explicit return type.

Example:

```
Class Box {
    double width;
    double height;
    double depth;

// Constructor (or) default constructor:
    Box ()
    {
        width = 10;
        depth = 10;
        height = 10;
    }

    double volume ()
    { return width * height * depth ; }
}

class demo
{
    public static void main ( String args[] )
    {
        Box b1 = new Box(); System.out.println ( b1.volume);
    }
}
```

Note:

A Java constructor cannot be abstract, static, final (or) synchronized.

default constructor:
-> no parameters.
-> to provide default values to the object.

o/p:

Volume is 1000.00

Note:

When we do not explicitly define a constructor for a class, java creates a default constructor. It automatically initializes all instance variables to zero.

Parameterized constructors: - accepts parameters to initialize the instance variables with distinct values.

Example:

```
class Box
{
    double width;
    double height;
    double depth;
  // Parameterized constructor
    Box (double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }

    double volume()
    {
        return width * height * depth;
    }
}
```

```
Class demo {
public static void main (String args[])
{
    Box b1 = new Box ( 5, 10, 15);
    Box b2 = new Box ( 2, 3, 4);
    System.out.println ( b1.volume);
    System.out.println ( b2.volume);
}
}
```

o/p:

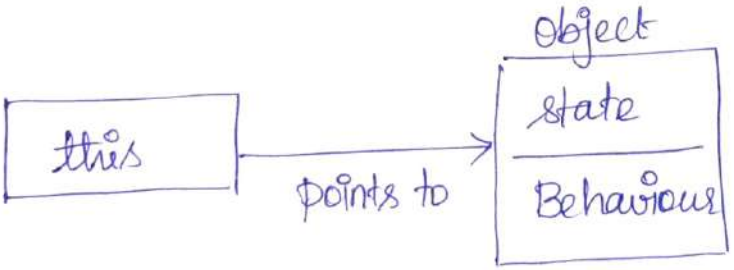
~~volume~~ 750

24.

---

this keyword:

* 'this' - can be used ~~as a keyword~~ inside any method to refer to the current object.



Uses of this keyword:

(1) used to refer current class object (or) Instance variable.

(2) used to invoke current class method.

(3) 'this' can be used to invoke current class constructor.

(4) 'this' can be passed as an argument in the method call.

(5) 'this' can be used to return the current class object instance from method.

Example: Box class constructor

```
Box ( double width, double height, double depth)
{
this . width = width;
this. height = height;
this. depth = depth;
}.
```

Garbage collection:

* A process by which the java program performs automatic memory management.

* The memory occupied (or) allocated to an object is reclaimed (or) released, when no reference to that object exists. automatically.

* In c++, garbage collection is done by manual code.

finalize() method:

* It specifies those actions that must be performed before an object is destroyed.
* Java run time calls this method, whenever to recycle the object of that class.

Syntax:

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

protected → key word : It makes the finalize method accessed by outside of the class.

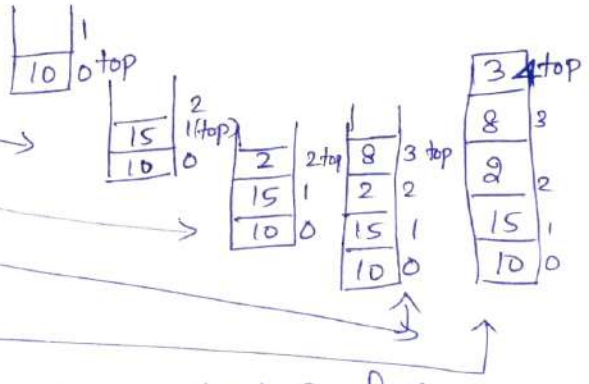
Stack class:

* Stack follows last-in first out (LIFO) strategy.
* 2 operations: 1) push → to put element in stack.
  2) pop → to remove an elements from stack.
* 2 conditions: 1) stack overflow (or) stack full
  2) stack underflow (or) stack empty.
* pointer : top ← element will be inserted (or) deleted from top position.

Program:

```
class stack {
    int stk[] = new int[5]; // array to hold elements.
                            (size of the array)
    int top;
    // to initialize the top as -1 by constructor
    stack()
    {
      top = -1;
    }

    // push (or) insert element into stack.
    void push(int k)
    {                        (array size - 1)
                             (5-1)
        if (top >= 4)
            System.out.println("stack is full");
        else
            stk[++top] = k;
    }

    // To pop (or) delete element from stack.
    int ~~void~~ pop()
    {
        if(top < 0)
        {
            System.out.println("stack (empty) underflow");
```

```
    }
    else
        return stk[top--];
  }
}

class Demo  // execution.
{
 public static void main (String args[])
 {
    Stack s = new Stack();
    s.push(10);
    s.push(15);
    s.push(2);
    s.push(8);
    s.push(3);
    s.push(18); // displays stack is full message.

  // remove elements
    for(int i=0;i<5;i++)
      System.out.println(s.pop());  } removes 5 elements from stack.

    s.pop(); → displays stack empty message
  }
}.
```



Constructor overloading:

* more than one constructor is defined in a class, called as constructor overloading.
* Each overloaded constructor should have different ~~constructor~~ signatures. (ie) different list of parameters.
* constructors have same name as class, but each overloaded constructor will do different work.

Example:

```
class Box
{
    int width;
    int depth;
    int height;
Box ( int width, int depth, int height)
{                                    // parameterized constructor
    this.width = width;
    this.depth = depth;
    this.height = height;
}

Box ()          // default constructor
{   width = 5;
    height = 7;
    depth = 2;
}}
```

```
// Method to display volume
void Volume()
{
S.O.P( width * height * depth);
```

```
Class Demo{
 public static void main ( String args[])
 {
    Box b1 = new Box(); // calling default constructor
       b1. volume();
    Box b2 = new Box( 3, 4, 9); // calling parameterized
       b2. volume;                  // constructor.
 }
}
```

(This topic discussed in second unit in detail).

Suggested Questions / Assignments / Home works / any other
* Define (i) class (ii) object (iii) constructor. * What is the use of 'this' keyword? * Explain class and object concept with an example. * What are the types of constructor. * Constructor overloading – Explain with example.

	Text Books / Reference Books / Any other suggested Materials		
S.No	Title	Author	Publisher
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Access specifiers.

Topic(s) to be covered	Access control - static - final - ~~stack~~ class ~~example~~ - Nested class - String class - Command line arguments - ~~varargs~~ - ambiguity.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	describe the access specifiers in Java	Under Stand.
LO2	Implement the static nested class.	Apply.
LO3	know the usage of Commandline arguments	Under-stand

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Access Control:

* Encapsulation wraps the data and code that manipulates. also it provides access control.

* Access control means to control what parts of the program can access the members of a class.

* We can avoid the misuse of members using access specifiers.

Two types of modifier: 1. Access modifiers 2. Non-access modifiers.

Java access specifiers:

Accessed inside (or) outside of the class/Package/Other

1) public ⇒ members can accessible by any code.

2) private → members can accessible only by the other members of class. ← only within class

3) protected.
   - → It is applied during inheritance.
   - → member accessible only within package and outside of the package only by childclass

4) default: only within package.

Example for Private and public & default:

```
class demo Test {
    int a;  // default access.
public int b; // public access
private int c; // private access.
// method to set 'c', becoz it is protected
void setC (int d)
  {
    c=d;
  }
int getC ()  // method to get/return value of c.
                becoz c is protected.
{
  return c;
}
}
```

```
Class Demo {
    Test Test t = new Test();
        t.a = 10; // no essor, because a is default
        t.b = 20; // no essor, because b is default.
        t.c = 30; // Creates essor. because protected
                  // variable not accessible outside of its
                  // class.
        t.setC(100); // Protected variable access thru method
                     is ok.
        System.out.println("a, b and c:" + t.a +
                           t.b + t.getC());
    }
}
```

o/p:

a, b and c is : 10 20 30.

Non-Access modifiers:

(1) Static ✓
(2) abstract ✓
(3) Synchronized ✓
} Important

(4) native
(5) volatile
(6) transient etc

"static" modifier:

* "when a member is declared as static, it can be accessed without any objects.", called static member.
* static members can be used by itself, without reference to specific instance.
* It is common to all objects (ie) direct control of class.
* They are global variables.
* when object creation is made, no copy of static member cannot be put in that.

Ex:

main() → This method is declared as static. Because it must be called before any object exist.

Rules for static Methods:

(1) A static method can call only other static method

(2) They must access only static data.

(3) They cannot be referred by ~~sta~~ super(or) this.

(4) All instances (or) objects of a class can ~~access~~ share the static variable.

(5) Static block can be executed only once, when the classes first loaded.

// program for static variable, method (or) block;

```
class staticclass
{
    static int a=3;  } // static variable.
    static int b;
    static void meth (int x)  // static method
    {
      System.out.println("a" +a + "b" +b + "x" +x);
    }
    static {   // static block
      S.o.p("This is static block");
       b= a*4;
    }
    public static void main (String args[])
                // main function is inside the class.
    {
      meth(42); // static method is called without
                // using object here.
    }
}
```

o/p:

x 42 a 3 b 12

---

outside the class: To access the static method out side the class use class name and . (dot) operator

Syntax: classname . StaticMethodName ();

example: StaticClass . meth (42);

final: → creates constants.

Can be

* A variable is declared as final will become constant.

Use * this prevents its contents from being modified.

* final variable should be ~~declared~~ initialized when it is declared.

Example:

```
final int Arr_len = 5; // final is used to declare constants
```

String class: - collection of characters.

- String object is immutable. (i.e) once a object created, its content can't be altered.

- Java defines a peer class string, StringBuffer which allows strings to be altered and all string manipulations are possible.

Example:

```
class StringDemo
{
public static void main(String args[])
{
String ob1 = "Welcome";
String ob2 = "To Java";
```

```
String ob3 = ob2;
System.out.println("Length of Steing object1" + ob1.length());
System.out.println("Char at index 3 in ob1" + ob1.charAt(3));
if (ob1.equals(ob2)) // Compare.
    System.out.println("equal string (or) Same");
else
    System.out.println("Not equal");
System.out.println(ob1 + ob2); // Concatenation
}
}
```

O/P:

```
Length of Steing object1: 7
Char at index 3 in ob1: c
Not equal.
Welcome To Java.
```

Inner classes:

* non-static and nested classes called Inner classes.
* We can't create an object (or) Instance of Inner class without creating object of outer class.
* But Inner class can access the members of outer class without Instance.
* It makes program simple and consise.

Outer ~~Nested~~ classes: The class in which nested class is defined is called outer class.

Nested class:

A class defined within another class means it is called nested class. It may be static (or) non-static.

Static class: * It is the way of grouping classes together.
* It is also used to access the private primitive member of the outer class through the object reference.
* These classes are loaded by class loader, at the time of first usage of only, not when its outer class gets loaded.

→

Command line arguments:

* Passing information or i/p to the program while running the program is accomplished by Command line arguments to main() function.

* Command line arguments directly follows the program's name on the command line, when it is executed.

* These are stored as string arguments in an array.

arg1 → args[0]
arg2 → args[1]
:

* Syntax:

D:\> Javac ProgramName arg1, arg2, arg3 .....

Example:

```
class Example
{
public static void main ( String args[] )
 {
    for( int i=0; i< args.length ; i++)
    {
       System.out.println( args[i]);
    }
 }
```

Command line arguments stored here

→

Running in command prompt:

> javac Example 10 20 30

> java Example.

O/P:

10

20

30

Suggested Questions / Assignments / Home works / any other

* what are the access specifiers available in Java?
* what is the use of static and final keyword?
* what is nested class?
* Explain the concept of static nested class with an example.
* what is command line argument?

Text Books / Reference Books / Any other suggested Materials

S.No	Title	Author	Publisher
1.	Data Structures and Algorithm Analysis in C, 2nd Edition, 2005	Mark Allen Weiss	Pearson Education

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Java Doc Comments.

Topic(s) to be covered	The Javadoc tags - General format of Java document comment - ~~out put~~ - Example.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
Lo_1	Understanding the tags used to create documentation	understand
Lo_2	know the general format of JavaDoc	understand
Lo_3	~~Able~~ write Java documentation in a program	apply

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Comments:

* It is used to provide information (or) explanation about variable, method, class or any statement in a program.

* It makes the code more readable.

* These comment statements are not executed by the compiler and interpreter.

* It makes easy to maintain the code and to find error easily.
* It prevents the ~~error~~ execution of program code while testing alternative code.

Types of Java Comments: three types

1. Single line Comment (//) → Comments only one line
2. Multi line comment → /*.... */ Comments multiple lines
3. Documentation Comment → It is used to create documentation API. This is done by Javadoc tool.
   → /** .... */
   → It allows to embed information about program into the program itself.
   → Javadoc utility used to extract the information and put it into an Html file
   ↳ webpage

The Java Doc Tags:

1. @ author → identifies the author of the class
2. @ version → specifies version of the class
3. @ param → Documents a method's parameters (or) arguments
4. @ return → Documents a method's return value

5. @docRoot → to depict relative path to root directory of generated document from any page.

6. @code → to show the text in code part without interpreting it as html markup (or) nested javadoc tag.

Example:

```
import java.io.*;

/**
 * <h2> Calculation of numbers </h2>
 * This program implements an application to perform
 * addition of numbers and prints the result.
 * <p>
 * @author Herbert Schildt
 * @version 16.0
 * @since 2021-07-06.
 */

public class calculate
{
/**
 * sum -> This method calculates the addition of
 * two integers.
 * @param a This is parameter 1
 * @param b This is parameter 2.
```

```
* @return int This returns ~~od~~ result of addition of a & b.
*/
    public int sum ( int a , int b)
    {
       return a +b ;
    }

  /**
   * This is the main method for executions.
   */
    public static void main (String args[])
    {
     /**
      * object creation done for class 'calculate'
      * obj is the objectname
      */
      Calculate obj = new Calculate ();

     /**
      * result is the variable which receives the result
      * of addition of sum function
      * pass 2 values in sum function.
     */
        int result = sum( 10 , 5);
     // display the result
        System.out.println (" Result" + result);
    } }
```

Other details about JavaDoc:

* JavaDoc is a document generator which was developed by Sun micro systems
* It is used to generate API documentation in HTML format from Java source code.
* HTML format is used to hyperlink the related documents. Html documents → webpages.
* All Javadoc comments are removed at compile time. So it doesnot affect the performance.
* Writing comments and Java doc (documentation) is for better understanding the code and provides better maintainance.
* The JDk javadoc tool uses doc comments during the preparation of program documentation automatically

other tags in javadoc comments:

1. @ docRoot → It represents relative path to the generated document's root directory from any generated page.

2. @ deprecated → adds a comment indicating that this API should no longer been used.

3. @ exception → Adds a throws subheading to the generated documentation with the class name and description text.

4. @ Inherit Doc : → Inherits a comment from the nearest inheritable class (or) implementable interface

5. @ link → Insests an in-line link with the visible text label that points to the documentation for specified package, class, or member name of a referenced class.

6. @ linkplain → Identical to @link, except the link's label is displayed in plain text than the code font.

7. @ see → Adds a "See Also" heading with a link (or) text entry that points to reference.

8. @ serial → Used in the doc Comment for a default serializable field.

* ~~Uses defined method of~~

Steps in creating Javadoc:

1. Compile the java program (or) source file.
2. write

   Javadoc filename (or)

   Javadoc packagename

This command creates no. of html files and open the file named index to see all information about classes Thus Java documentation Api is created

---

Java doc in Eclipse:

1. Project Menu → Generate JavaDoc → option — It will open a wizard.
2. In that wizard
   → give location for JavaDoc file.
   → default : 'c' drive.

i) @ serial field :

Documents an ObjectStream field component.

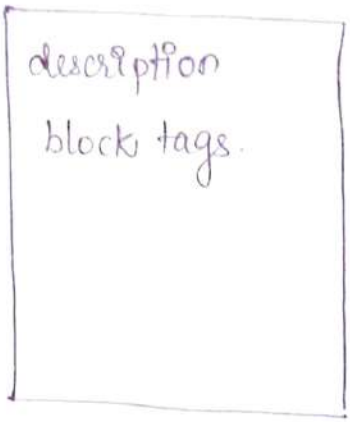
ii) @ since : Adds a "since" heading with the specified since text to the generated documentation.

iii) @ throws :

the throws keyword is similar to exception keyword.

iv) @ value : when @ value is used in the doc comment of a static field, it displays the value of that constant.

Java Doc format types has 2 parts.



Generation of Java Doc :

Some IDE (Integrated development environment) like NetBeans, Eclipse and IntelliJ IDEA are automatically generate the Java Doc file.

3. Select the projects and then the packages for which you want to create JavaDoc file.

4. Select the classes for which you want to generate JavaDoc. By default all the classes will be selected.

5. Select the necessary visibility

6. Select the destination location where the generated JavaDoc will be placed.

7. Finish.

8. If you select Next, then select the Document title and other basic options.

Example 2:

```
package exa;
import java.util.Scanner;
/**
 * author Hasbest
 */
public class Example2
{
```

```
/**
 * This the program for finding power of 2 numbers.
 */
public static void main (String args[])
{
  /**
     Declared 2 variables x and y and
   * taking input from scanner class *
   */
    int x, y;
    Scanner ss = new Scanner (System.in);
  // Reading the values of x and y.
      x = ss.nextInt();
      y = ss.nextInt();
  /** store the result in variable ~~sum~~ result
   * which is of type integer *
   */
  int result = Math.pow (x, y);
    // Printing the result
      System.out.println( result ); } }
```

Suggested Questions / Assignments / Home works / any other
* Define comment. * List out the various types of comments in Java. * Explain in detail the tags usage in creation of Java documentation.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Static, nested & inner classes.

Topic(s) to be covered	Nested class - Inner class - outer class - Static class - examples.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	To define nested class and outer class	Remember
Lo2	To explain the nested class with example	understand.
Lo3	discuss about static classes	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Nested classes:

* Defining one class within another class, called as nested class. It may be static (or) non-static.

* They enable us to logically group classes that are used in one place. Thus it increases encapsulation and creates more readable and maintainable code.

Properties:

* The scope of nested class is bounded by

the scope of its enclosing classes (or) outer classes.

* It doesn't exist independantly without outerclass.

* Syntax:

```
class Outerclass // enclosing class
{
    :
    class Inner class
    {
    :
    }
}
```

nested class.

* A nested class can access to the members (including private members) of the ~~class~~ ~~in~~ outer class (or) enclosing class. The reverse is also true.

* A nested class is also a member of its outer class

* A nested class can be declared private, public (or) protected or default.

* Nested classes are divided into 2 categories:

1. Static nested class: declared with static keyword.
2. Inner class: A inner class is a non-static nested class.

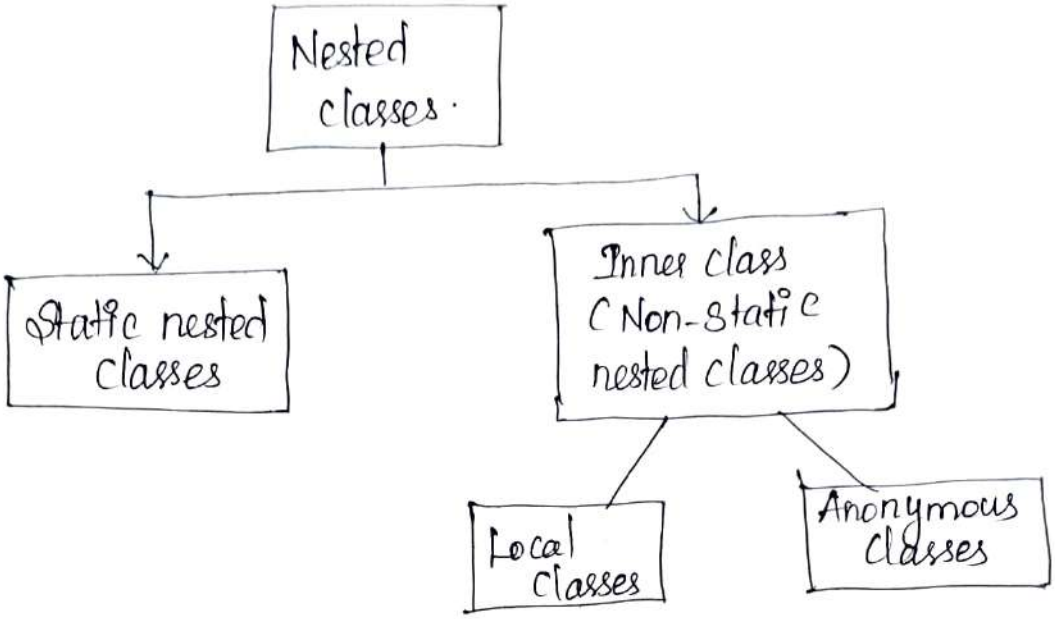


fig: classification of nested classes.

Static nested classes:

* It is not strongly associated with the outer class object.
* without an outer class object existence, there may be a static nested class object can alive.
* A static nested class cannot refer directly to instance variables or methods in its outer classes. It can use them through object reference.

(ie) `Outerclass.StaticNestedClass.variablename`

Example:

```
class Outerclass
{
```

```
static int outer_x = 10;
int outer_y = 20;
private static int outer_private = 10;

static class staticNestedClass
{
    void display()
    {
    System.out.println("outer_x" + outer_x);
    System.out.println("outer_private" + outer_private);
    System.out.println("outer_y" + outer_y);
                        // error. can't access.
    }
}
}
public class demo
{
  public static void main (String args[])
  {
    Outerclass.staticNestedClass obj = new
                Outerclass.staticNestedClass();
                obj.display();
  }
}
```

Use of static class (or) Need of static class:

* when we don't need to create object (or) instance of a class and only want to utilize the peopesties and functions of a class, then create it as static class.

* Example: (1) Utility classes.
(2) Built in console classes.

Characteristics of static class:

* A static class cannot be instantiated. That is, object is not created.

* Inner classes can be static and it is enclosed by non-static outer class.

* Inner class (ie) static nested class access the members of outer class, without any reference (or) object.

Note: Inner class access all members of outer class. But static inner class can access only the static members of outer class.

* All static classes are nested classes. but reverse is false.

* we can create static blocks, variables and methods inside a static class.

* A class can have multiple static classes.

* A static class can contain multiple static classes.

Example:

```
public class demo
{
  private static String s = " Java Programming";
  // static and nested class.
   static class Nestclass
   {
      public void show()
       {
          System.out.println (s);
       }
   }
   public static void main( String args[])
   {
      demo.Nestclass obj = new demo.Nestclass();
      obj.show();
    }
}
```

Inner classes:

* To Instantiate an Inner class, you must first Instantiate the outer class.

* Create the Inner object within the outer object using Syntax:

```
Outerclass . Innerclass  innerobj = outer object . new
                                    Inner class ();
```

* Two types of Inner classes:

1. Local Inner class:
2. Anonymous Inner class.

Example for Inner class:

```
class  Outer class
 {
    static int outer_x = 10;  // static member
    int outer_y = 20; // non-static member.
    private int outer_private = 30;  // private member

    class  Inner Class
    {
       void display()
       {
          System.out.println ("outer_x" + outer_x);
          System.out.println("outer_y" + outer_y);
```

```
System.out.println(" outer_private "+ outer_private);
}
}
}

public class demo
{
public static void main(String args[])
{
    OuterClass outerObject = new OuterClass();
    OuterClass.InnerClass innerobject = outerobject.new
                                        InnerClass();
innerObject.display();
}
}
```

Comparision between normal (or) regular nested class and static nested class.

SNo	Normal nested class or normal or Inner class	Static nested class (or)
1.	without an outer object existence, there can't be Inner class object. (i.e) Inner class object is associated with outer class object.	without an outer class object existing, there may be a static nested class object. They need not be associated with outer class object.

2.	Inside normal / regular inner class, static members can't be declared.	Inside the static nested class, static members can be declared.
3.	As main method can't be declared, regular inner class can't be invoked directly from the command prompt.	As main method can be declared, they can be invoked directly from the command prompt.
4.	Both static and non-static members of outer class can be accessed directly.	only a static member of outer class can be accessed directly.

Inner class:

* The classes that are non-static and nested are called inner classes.

* we can't create object for inner class, without creating object for outer class.

Outer class:

* The class in which a nested class is defined is called as outer class.

Suggested Questions / Assignments / Home works / any other
1. What is nested class? How it can be accessed? 2. Define Inner class, outer class, static class. 3. Explain normal nested class and static nested class with suitable examples.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

UNIT-II **Lecture No.** Inheritance.

Topic(s) to be covered	Basics – member access – Example – super – types of Inheritance – Constructor calling –

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	define Inheritance	Remember
Lo2	Explain the various types of Inheritance with examples	Understand
Lo3	define the keyword supes and its uses	Remember

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Inheritance:

* It is a mechanism by which one object acquires all the properties and behaviours of a parent object. It is represented by IS-A (parent-child) relationship.
* It is an important property of oops.
* Creation of new classes from the existing classes, so that we can reuse the methods & fields of parent class. we can also add new methods to newly created classes.

Need (or) use of Inheritance:

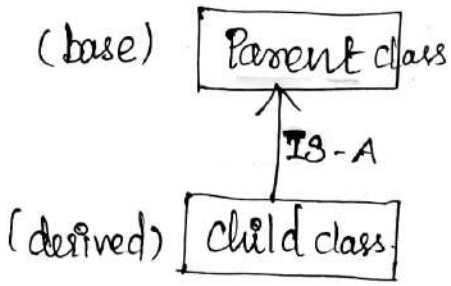
* code reusability.
* to achieve method overriding (or) runtime polymorphism.

Terms:

1) class: class is a group of objects which have common properties. It is a template (or) blueprint from which objects are created.

2) Subclass: (or) child class: It is a class which inherits the other class. It is also called a derived class (or) extended class.

3) Super class (or) parent class: super class is a class from which a subclass inherits the features. It is also called as base class.



4) Reusability: It is a mechanism of using the fields and methods of the existing class, when you create a new class.

Inheritance basics:

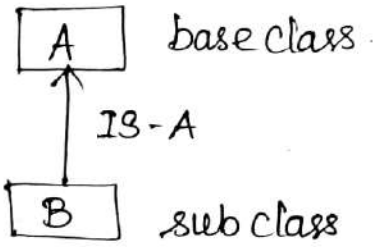
* To inherit a class, the definition of one class is used into another by using **extends** keyword.

* Syntax:

```
class subclass_Name extends superclass_Name
{
    :
    // methods and fields.
}
```

* **extends:** It indicates that making a new class that derives from an existing class.

Example:




```
class A        // parent (or) base class
{ int i,j;
  void display()
  { System.out.println("value of i & j" + i + "  " +j);
  }
}

class B extends A          // sub class (or) child class.
{
  int k;
```

→

```
void show()
{
    System.out.println("value of k" + k);
}

void sum()
{
    System.out.println("sum:" + (i+j+k));
}
}

Pu class SimpleInheritance
{
    Public static void main(String args[])
    {
        A parent = new A();
        B child = new B();

        parent.i = 10;   // store value to i
        parent.j = 20;   // store value in j
        parent.display(); // display value of i & j from parent

        child.i = 5;   // store value to i
        child.j = 7;   // store value to j
        child.k = 9;   // store value to k
        child.display(); // display value of i & j
        child.show();  // display value of k
        child.sum();   // display total of i, j and k
    }
}
```

// Access from parent

// Access from child class.

Explanation:

class A contains i, j as members and display() function to display value of i and j.

class B inherits A, so i, j and display() function are inherited to B. Additionally, B has member k and functions show() to display the value of k and sum() to display the sum of i+j+k.

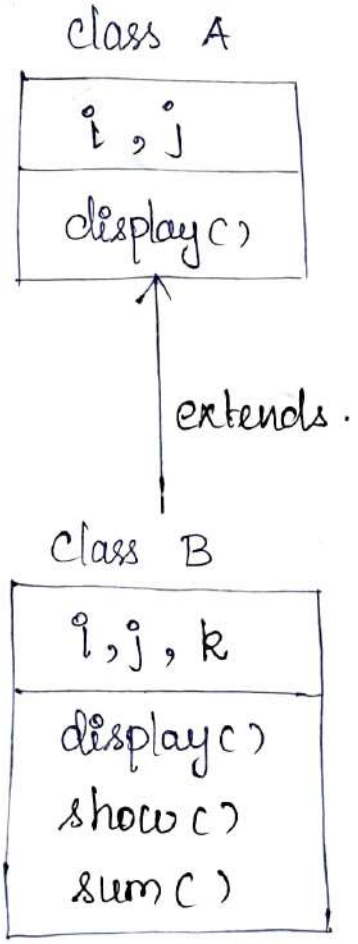


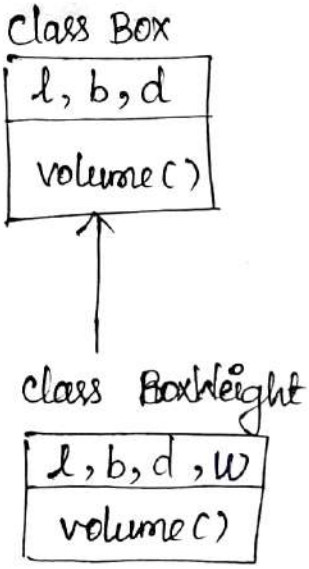
Fig: Simple Inheritance.

Note: Member access:

A private variable cannot be inherited to subclass. ~~only~~ But that private variable is accessible in its own class (where it is declared).

public (or) default variables ~~can~~ are inherited to subclass.

Example: Simple Inheritance (or) Single Inheritance.




```
Class Box
{
    double length; // l
    double breadth; // b
    double depth; // d
```

→

// constructors (overloaded).

```
Box( double w, double h, double d)
{
    length = h;
    breadth = w;
    depth = d;
}

Box ()
{
    length = -1;
    breadth = -1;
    depth = -1;
}

Box ( double len)
{
    length = len;
    width = len;
    depth = len;
}

double volume()
{
    return length * width * depth;
}
```

→

```
    Box ( Box obj)  // creating copy (or) clone
    {
        length = obj.length;
        breadth = obj.breadth;
        depth = obj.depth;
    }
}

class BoxWeight extends Box
{
    double weight;

    BoxWeight ( double w, double h, double d, double wt)
    {
        length = h;
        breadth = w;
        depth = d;
        weight = wt;
    }
}

Class demo
{
  public static void main( String args[])
  {

      BoxWeight Box1 = new BoxWeight(10,20,15, 34.3);
      double vol = Box1.volume();
```

```
System.out.println(" Volume of Box1 " + vol);

BoxWeight Box2 = new BoxWeight(2, 3, 4, 0.07);
    vol = Box2.volume();
System.out.println(" Volume of Box2 " + vol);
}
}
```

Superclass variable reference a subclass variable:

consider the above example's demo class.

It can be rewrite as

```
class demo {
public static void main(String args[])
{
    BoxWeight Box1 = new BoxWeight(10, 20, 15, 34.3);
    Box parent = new Box();
    double vol;
    parent = Box1; // assigning child class obj to parent class obj
    vol = parent.volume();
// (or)  vol = Box1.Volume();

    System.out.println(" parent value of length " +
                                    parent.length);
    System.out.println("value of weight " + parent.weight);
                // this is error. ∵ weight is not member of parent
}}
```

Both produce same answer.

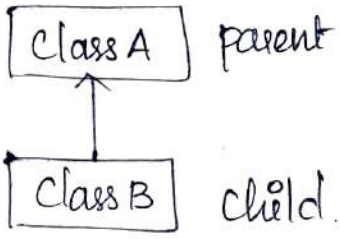
Types of Inheritance in Java:

1. Single (or) Simple.
2. Multilevel
3. Hierarchical
4. Multiple } These two are not directly supported by Java.
5. Hybrid } But they achieved through "Interfaces".

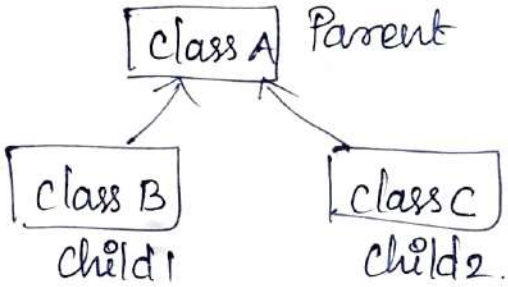
Single, multilevel, Hierarchical ⇒ } Has one parent (or) single parent.

multiple, Hybrid ⇒ Has ~~one~~ 2 or more parents.

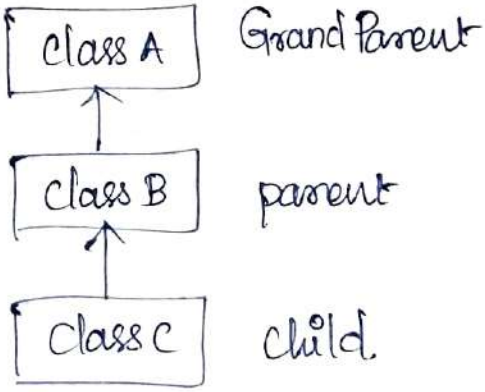
Single:



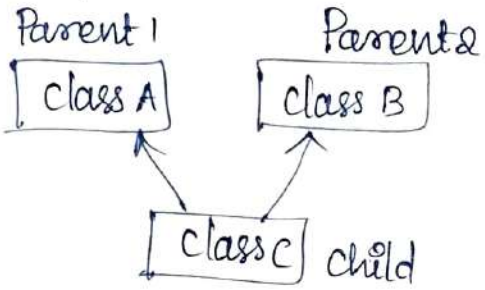
Hierarchical:



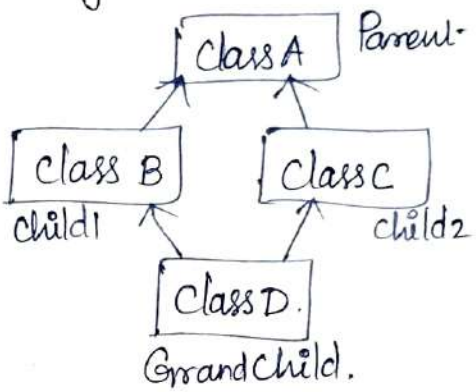
Multi level:



Multiple:



Hybrid:



Why multiple Inheritance is not supported in Java?

* Because multiple Inheritance results in ambiguity.

* Consider situation, when there exists methods with same name and signatures in both super (parent) classes, the compiler cannot determine which class method to be executed on calling the method.

* To prevent this ambiguity and complexity, Java doesn't directly supports multiple Inheritance.

Super:

* when a subclass needs to refer to its immediate parent class (or) superclass, 'super' keyword is used.

* 'super' used to call superclass constructor from child class constructor.

* 'super' also used to access a member of parent

class, which is hidden by subclass member.

'super' to call superclass constructors:

A subclass can call a constructor defined by its superclass by the following syntax:

```
super ( parameters list);
```

↳ arguments needed by parent class constructor

Example:

We are rewriting the constructor of BoxWeight class:

```
Class BoxWeight extends Box    // subclass.
{
    double weight; // weight of the box.
    // constructor.
    BoxWeight ( double w, double h, double d, double wt)
    {
        super (w, h, d); // calls superclass 'Box' constructor.
        weight = wt;
    }
}.
```

Here, BoxWeight is the subclass of Box (Superclass). The ~~keyword~~ super (w, h, d) calls the constructor in Box class and send the values w, h, d to it.

Second use of 'super':

Syntax: `super. member`
→ method / variable.

Example:

```
class A
{
  int i;
}

class B extends A
{
  int i; // this 'i' hides 'i' in A

  B ( int v1, int v2)
  {
    super();
    super.i = v1; // calls i in A.
    i = v2;  // 'i' in class B
  }

  void show()
  {
    System.out.println(" i in superclass " +
                       super.i);
    System.out.println(" i in subclass "+ i);
  }
}

class demo
{
  public static void main ( String args[])
  {
    B obj1 = new B (1,2);
    obj1.show();
  }
}
```

this: * It is a keyword refers to the current object in a method (or) constructor.

* It eliminates the confusion between class member attributes and the parameters of the constructors. If attributes and parameters have same name, this keyword acts there.

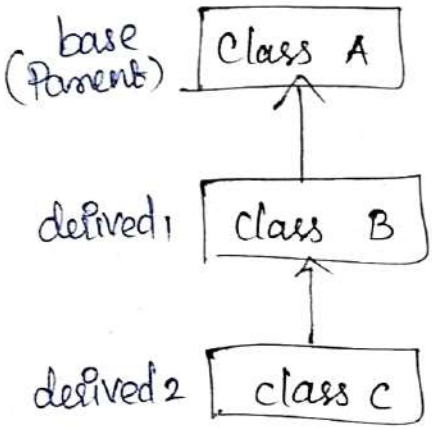
Example:

```
Class Box
{
  double length;
  double width;
  double height;

  // constructor

  Box( double length, double width, double height)
  {
     this.length = length;
     this.width = width;
     this.height = height;
  }
}
```

// this → refers to current object

Multilevel Inheritance:

A class extends a class that extends another class.



B extends A.
C extends B.

Class A: base class for class B.
Grand parent for class C.

Class B: child class of class A and
Parent for class C.

class C: child class of Class B.
Grand child of ~~parent~~ A.
Class

Example:

```
Class   A
{
   int  V1;
    A( int V1)
    {
      this.V1 = V1;
    }
}
```

```
class B extends A
{
    int v2;
    B ( int v1, int v2)
    {
      super ( v1);
    this. v2 = v2;
    }
}
```


```
Class C extends B
{
    int v3;
    C ( int v1, int v2, int v3)
    {
        super ( v1, v2);
        this. v3 = v3;
    }
    void display ( )
    {
        System. out. println ( "sum " + (v1 + v2 + v3));
    }
}
```

→

```
class demo
{
public static void main ( String args[])
{
    C  obj = new C (10, 20, 30);
    obj.display();
}
}.
```

o/p:

Sum 60

Suggested Questions / Assignments / Home works / any other
1. Define: Inheritance, super, this, extends. 2. What are the types of Inheritance? 3. Why multiple Inheritance not supported in Java? 4. Explain single & multiple Inheritance with example. 5. Explain Hybrid Inheritance with an example.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Method oversiding.

Topic(s) to becovered	Method oversiding- Example- Dynamic method dispatch – Example – need of oversidden methods- Apply oversiding.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
Lo1	explain about method oversiding	Understand.
Lo2	discuss about dynamic method dispatch	Understand.
Lo3	narrate the needs of oversiding	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Method oversiding:

* when a method in a sub class has same name and type signature as a method in its parent class, then method of subclass overrides the method in parent class.

* child class and parent class both have a method with same name & type signature means oversiding occurs.

* when a overridden method is called from the subclass, the method on the superclass is hidden and the method on subclass only accessed.

Example:

```
class A          // Parent class.
{
  int i,j;
  A( int a, int b) // Parent constructor
   {
      i=a;
      j=b;
   }
  void show() // method to display value of i & j
  {
   System.out.println("i and j" + i + " " +j);
  }
}

class B extends A          // child class
{
   int k;
   B( int a, int b, int c) // Child Constructor
   { super(a,b);
     k=c; }
```

```
void show ()    // to display value of k
{
  System.out.println(" value of k:" +  k);
 }
}

 class demo
 {
  public static void main ( String args[])
  {
     B obj = new B( 3 , 4 , 2);
      obj.show() ; // this calls the method in
                   // classB , not in A.
   }
 }
```

output:

value of k:2

Explanation:

Class A contains method show()

Class B also defines " show()

Both has same name and same type signature. so overriding happens. child class method only accessible during runtime

Why overridden methods? (or) Need (or) use of overridden methods:

* overridden methods allow java to support run-time polymorphism.
* It allows a class to specify methods that will be common to all its derived classes. subclasses are allowed to use (or) define some (or) all of those methods.
* overriding provides "one interface, multiple methods" aspect polymorphism.
* Using combination of inheritance & overriding methods, a super class can define the general form of the methods that will be used by all its subclasses.
* Through dynamic run time polymorphism, code reuse and robustness are achieved.

Applying overriding: (Example)

```
Class Figure
{
  double dim1, dim2;
    Figure ( double dim1, double dim2)
    { this.dim1 = dim1;
      this.dim2 = dim2; }
```

```
double area()
{
  System.out.println(" Not defined ");
  return 0;
 }
}

class Rectangle extends Figure
{
    Rectangle ( double dim1, double dim2)
    {
       super( dim1, dim2);
     }

// overridden method
    double area()
   {
     System.out.println(" Inside Rectangle);
     return  dim1 * dim2;
    }
 }

  class  Triangle  extends  Figure
  {
       Rect  Triangle ( double dim1, dim2)
             {  super( dim1,  dim2)
              }
```

```
// overridden method
  double area()
  {
   System.out.println("Inside Triangle");
   return dim1 * dim2 / 2;
  }
}

class demo
{
public static void main(String args[])
{
      Figure f = new Figure(10,10);
      Rectangle r = new Rectangle(9,5);
      Triangle t = new Triangle(10,8);

      Figure obj; → Parent class object / Base class obj
        obj = f; // assigning baseclass object
          System.out.println("Area is" + obj.area());

        obj = r; // assigning Rectangle's object
          System.out.println("Area is" + obj.area());

        obj = t; // assigning Triangle class's object
          System.out.println("Area is" + obj.area());
  }
}
```

output:

Area is not defined.
Area is 0.

Inside Rectangle
Area is 45

Inside Triangle
Area is 40.

Dynamic method dispatch:

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

Dynamic method dispatch is the way of implementing runtime polymorphism in java.

Principle: Superclass object refers to subclass object

This principle is used to solve the overridden methods at runtime.

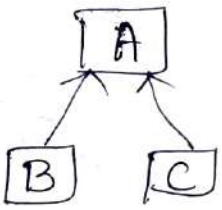
When an overridden method is called through superclass object / reference variable, Java determines

which version of that method to execute, based on the type of object being referred.

Example 1: // use the example for overriding here.

Example 2:

```
class A
{
  void callme()
  {
    System.out.println("Inside A");
  }
}
```




```
class B extends A
{
  void callme()
  {
    System.out.println("Inside B");
  }
}

class C Extends A
{
  void callme()
  {
    System.out.println("Inside C");
  }
}
```

```
class demo
{
  public static void main (String args[])
  {
      A a = new A(); // object of A
      B b = new B(); // object of B
      C c = new C(); // object of C.

      A ref; // reference obj of type A.
             // (i.e) superclass reference variable.
      ref = a; // now 'ref' refers to A's object
      ref.callme(); // calls the method in A.

      ref = b; // now "ref" refers to 'B' object
      ref.callme(); // calls to method in B.

      ref = c; // now "ref" refers to C's object.
      ref.callme(); // calls to method in C.
  }
}
```

Output:

```
Inside A
Inside B
Inside C.
```

Suggested Questions / Assignments / Home works / any other
1. Explain about method overridding with an example. 2. Discuss about dynamic method dispatch with suitable example. 3. Why overriding is needed in java?

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Abstract class

Topic(s) to be covered	using abstract class – abstract method – using final with inheritance –

	Lecture Outcome (LO)	
	At the end of this lecture, students will be able to	**Bloom's Level**
LO1	define abstract class and abstract method	Remember.
LO2.	use the abstract class concept to ~~create~~ program	Apply.
LO3	needs of final keyword in java	Understand

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Abstraction:

It is a process of hiding the implementation details and showing only functionality to the user. (i.e.) showing only essential things and hiding the internal details.

2 ways to achieve abstraction:

1. Abstract class (0 to 100%)
2. Interfaces (100%)

Abstract class:

* A class which needs to be extended and its methods to be implemented but cannot be instantiated, is called abstract class.

* Abstract class is defined with 'abstract' keyword.

* A class for which object cannot be created (instantiated) is called abstract class.

* Syntax:

```
abstract class classname
{
    abstract methodname();
    :
}
```

* Abstract class can have constructors, methods and static methods along with abstract methods.

(i.e) It can have both abstract and non-Abstract methods

* Abstract Method: A method without any implementation and defined with 'abstract' keyword is called abstract methods.

example: abstract void area(); // method without body.

These abstract methods are implemented in subclasses.

→

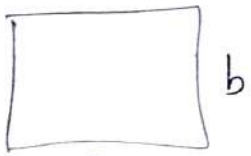
* An abstract class also can have 'final' methods. It will force the subclass not to change the body of method (or) implementation.

Example for Abstract class & methods:

```
abstract class shape
{
    double a;
    double b;   // variables.

    Shape ( double a, double b) // constructor
    {
        this.a = a;
        this.b = b;
    }

    abstract double area( ); // abstract method
}

class Rectangle extends Shape {
    Rectangle ( double a, double b)
    {
        super ( a, b);
    }
```

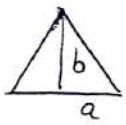
// override (or) define method area()

```
double area()
{
    return a*b;
}
}
```



Area = l * b
= a * b.

```
class Triangle extends Shape
{
    Triangle (double a, double b)
    {
        this.a = a;
        this.b = b;
    }

    double area()
    {
        return a * b*/2;
    }
}

class Circle extends Shape
{
    Circle ( double a)
    {
        super (a, a);
    }
```



area = ½ bh.
= ½ ab.
= a*b/2.

```
    double area() {
}           return 3.14 * a * a; }
```

area = πr^2
= 3.14 * a * a

```
class demo
{
 public static void main(String args[])
{
    Rectangle  R = new Rectangle (10,12);
    Triangle   T = new Triangle (9,5);
    Circle     C = new Circle (10);

S.O.p(R.area());
S.O.p(T.area());      (or) use overriding concept
S.O.p(C.area());
   }
}
```

Output:

120

22.5

314.

Using final with Inheritance:

final: It is a keyword in Java and has 3 uses.

(i) It can be used to create the constants.

```
ex: final int a = 10;   // a becomes constant
                           value can't be altered.
```

(ii) It is used to prevent overriding of a method.

```
ex: final void show()
    {
    :
    }
```

This method can't be overridden in subclass.

(iii) It is used to prevent Inheritance.

```
ex: final class A
    {
    :
    }
```

This class can't be Inherited (or) extended.

Using final to prevent overriding:

To disallow a method from being overridden (or) to prevent overriding of a method, it should be declared as 'final' method.

Example program:

```
class A                    // base class
{
   final void show()
   {
      System.out.println("Inside A");
   }
}

class B extends A          // child class.
   void show()  // Error! can't be override.
   {
      System.out.println("Inside B");
   }
```

Use of final methods:

* provides performance enhancement
* compiler is free to inline call access them, because it knows they are final methods and can't be overridden.
* when a 'small final method' call arises, Java Compiler copy the bytecode for the subroutine directly inline with the compiled code of the calling method. and eliminates costly overhead.
* final methods can't be overridden and a call to it can be resolved in compile time, is called as early binding. (dynamic (or) runtime means late binding).

Using final to prevent inheritance:

* When we want to prevent a class being inherited, then declare the class as 'final'.

* When a class is declared as final then all of the members and methods become final.

* It can't be inherited (or) extended.

Example:

```
final class A
{
  :
}

Class B extends A  // Error! Can't create subclass of A.
{
  :
}
```

Suggested Questions / Assignments / Home works / any other
* what is the uses of final keyword?. * what is abstract class & abstract method. * Explain abstract class concept with suitable example.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Unit - III.

Lecture No. Exception Handling Basics.

Topic(s) to be covered	Exception handling fundamentals - Exception types - uncaught exceptions - using try & catch.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
LO1	know the keywords used in exception handling.	Under -Stand
LO2.	Explain the types of exceptions.	Under stand.
LO3.	discuss the need of try-catch block.	Under stand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate / Discuss / Peer to Peer Learning / Quiz / Role Play / Any other

Lecture Notes

Exception Handling Fundamentals:

* **Exception:** A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.

* Exceptions are created (or) generated by Java runtime system. (or) by manually generated by the code (program).

* The exceptions used to report some error condition to the caller of the method.

* when exception arises, an object is generated for that exception and thrown in the method.

* This method handles that exception by own (or) pass that exception object.

* An exception should be caught and processed.

* Exception handling is managed via five keywords:

(1) try
(2) catch
(3) throw
(4) throws
(5) finally.

* try: Program statements that we want to monitor for exceptions are contained with in try block.

* * if an exception occurs in try block, it is thrown.

Catch block: (or) catch:

* The exception thrown by the try block, should be caught by our code (using catch).

* The exception is handled by catch block.

Throw: To manually throw an exception, throw keyword is used.

Throws: Any exception that is thrown out of a method must be specified by throws (keyword) clause.

finally: * Any code that absolutely must be executed after the completion of try block, put in to finally block.

* This block is always executed. (ie) The code inside this block is executed irrespective of exception.

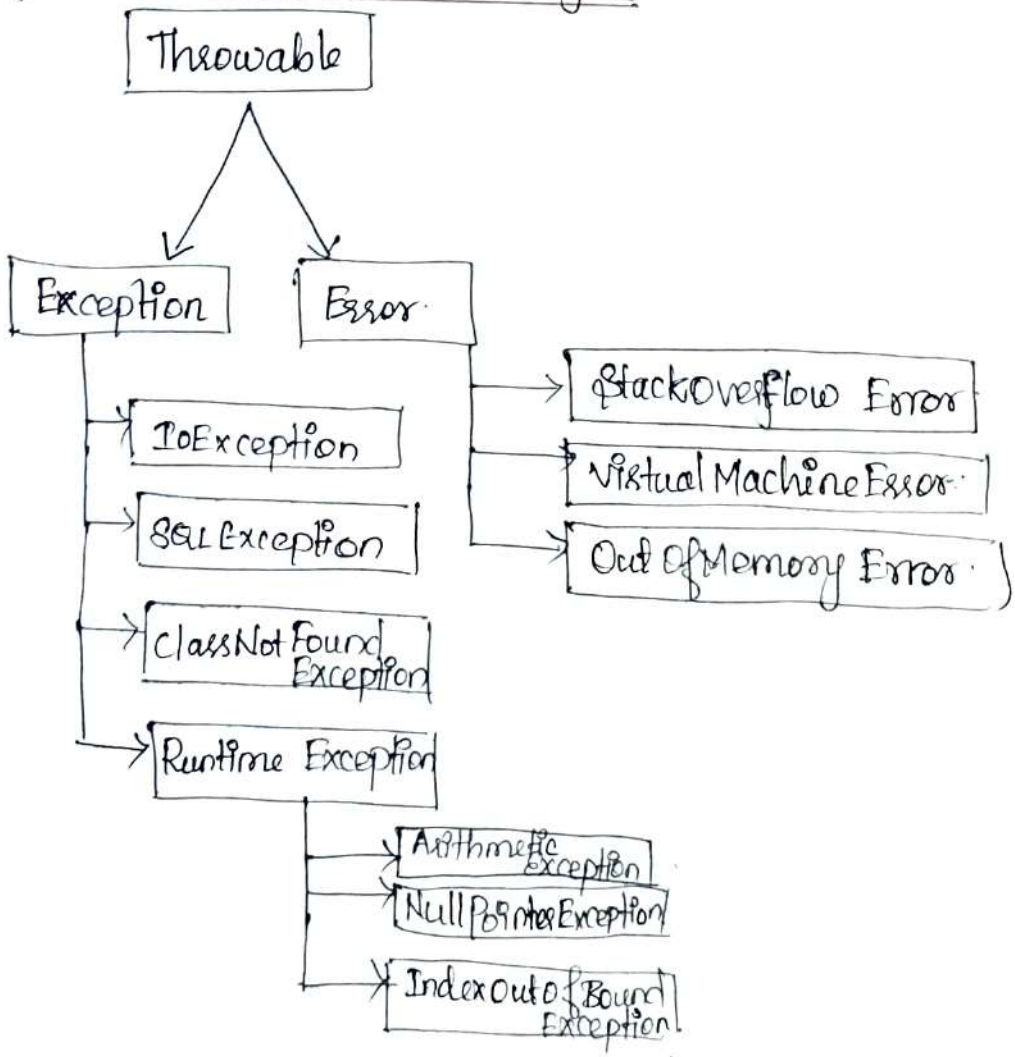
General format of Exceptional handling block:

```
try
{
  // block of code to monitor errors.
}

catch ( ExceptionType1  ob1)
{
  // exception handler for exception type1
}
```

→

```
catch ( ExceptionType2  ob2)
{
 // exception handler for exceptionType2
}
//.....
....
finally
 {
  // block of code, always executed after tryblock completion
 }.
```

~~Exception Types~~ (or) Exception Hierarchy:



* All exceptions are subclasses of the built-in class Throwable. So it is in the top of the hierarchy.

* Throwable has 2 subclasses: Exception and Error.

* Exception class: This is useful for exceptional conditions that user programs should catch. This is the baseclass for all Exceptions.

* one type of Exception is RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as
ex: Division by zero and invalid array index.

* Error Class:

(*) These are the exceptions that cannot be caught under normal circumstances of the program.

(*) It is used ~~to~~ by the Java runtime system to indicate errors ~~having to do~~ to itself.

* Ex: StackOverflow error.

Un caught Exceptions:

* If an exception is not caught by the exception handler, it leads to stop the execution of the program at that point

Example:

```
class Sample
{
public static void main ( String args[])
{
    int d=0;
    int a = 42/d; // attempt to divide by 0
}
}
```

Here an attempt to divide by zero occurs. So a new exception object is created and thrown. But no handler to process the exception. It leads to stop the execution of the program at that point.

o/p :

```
java.lang.ArithmeticException : / by zero
    at Sample.main ( Sample.java : 5)
```

Progsam name ↳ 5th line exception.

Using try and catch: Java has built-in-exceptionhandler for helping debugging.

* manual exception handler has 2 benefits:
  1. Fixing the error.
  2. preventing the program from termination.

* using printStackTrace, to know what happened in the program whenever error occured, is a tedious process.

* We can use try-catch block to overcome this.

try: Enclose the code that you want to monitor inside a try block.

Catch: Immediately after the try block, enclose a catch block that specifies the exception type to catch.

Program with try-catch block to process ArithmeticException generated by division by Zero error.

```
Class Ex2
{
  public static void main ( String args[ ])
  {   try{
      int d=0;
      int a = 42/d;
      } catch (ArithmeticException e)
       { System.out.println (e+" Division by Zero error");}
```

o/p: Java.lang.ArithmeticException : Division by Zero error

Suggested Questions / Assignments / Home works / any other
1. What is exception? What are the types? 2. Explain the keywords associated with exception handling mechanisms. 3. What is the need for try-catch block? 4. What is divide by zero error?

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Multiple catch & Nested try.

Topic(s) to becovered	Nested try statements :- Example - throw -throws - finally - Multiple catch clauses- example.

	Lecture Outcome (LO)	
	At the end of this lecture, students will be able to	**Bloom's Level**
LO1	Explain nested try statements.	Understand
LO2	differentiate throw and throws.	understand.
LO3	define finally	Remember.
LO4.	discuss about multiple catch clause	Understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Nested try statements:

* try statement can be nested. One try statement can be inside the block of another try, is called nested try.

* Syntax:

```
try {                    → outer try block
    try                  → inner try block.
    {
    }
    catch ( Exception e)
    { }
} catch(Exception e
    { }.
```

* If the inner try block contains no catch clause, the next try statement's catch clause are inspected for match.
* If no catch matches, Java run time system will handle the exception.

Example for Nested try statements:

```
class Example
{
  public static void main (String args[])
  {
    try // outer try block
    {
      int a = args.length; // if no command line arguments, this will be 0, a=0
      int b = 42/a;  // if a=0, then / by zero error
      System.out.println("a" + a);
      try { // nested try block (or) inner try block
        if (a==1)
          a = a/(a-a);
        if (a==2)
        {  int c[] = {1,2};
           c[10] = 15;
        }
      } catch (ArrayOutofBoundsException e)
      { s.o.p(e); }.
    } catch (ArithmeticException e) { s.o.p(e)};
  }
}
```

To run the program with command line argument:

(i) C:\> java Example

Here no command line arguments after program name.
So args.length = 0 , a = 0.
So java.lang.ArithmeticException occurs

(ii) C:\> java Example 10 20.

Here 2 command line arguments available.
So args.length = 2 ⇒ a = 2.
So java.lang.ArrayIndexOutOfBoundsException occurs.

(iii) C:\> java Example 10

Here only one command line argument.
a = 1 , args.length = 1.
So java.lang.ArithmeticException occurs.

throw:

* our program can throw an exception explicitly using throw statement.

Syntax: throw ThrowableInstance;

* The flow of execution stopped immediately after the try statement. any subsequent statements are not executed. control is transfessed to matching catch block.
* If no matching catch block, the default exception handler halts the program and prints the stack trace.

Example:

```
class Example
{
    static void calculate( int a, int b)
    {   try {
            if (a == 0)
                throw new ArithmeticException("Divide by zero");
            else
                System.out.println( b/a);
            }
        catch ( ArithmeticException e)
            { System.out.println(e);
            }
    }
}
```

* ~~throws~~ ~~keyword~~ ~~cannot~~ ~~be used to~~

Note:

* ThrowableInstance must be the subtype of Throwable class
* For Example, ArithmeticException is the subclass of Throwable and user defined exceptions also can be used.
* Unlike C++, datatypes such as int, char, floats and non-throwable classes cannot be used as Exceptions here.

throws:

* It is used in the method's signature to indicate that, it might throw any one of the listed type exceptions

Syntax:

```
datatype methodname(parameters) throws Exception
```

ex1:

```
void calculate(int a, int b) throws ArithmeticException
{
    ...
}
```

ex2:

```
void checkEligible(int age) throws InvalidAgeException
{
    if(age>18)
        System.out.println("Eligible for voting");
    else
        throw new InvalidAgeException("Invalid Age");
}
```

* The caller of the method, has to handle the exception using try catch block.

* To prevent compile time error, we can handle the exception in 2 ways

1. By using try catch.
2. By using throws keyword.

* We can use throws keyword to delegate the responsibility of exception handling to the caller and the caller ~~met~~ of the method handles the exception.

Example:

```
class test
{
    public static void main (String args[]) throws
                                        InterruptedException
    {
        Thread.sleep(10000);
        System.out.println("welcome to Java");
    }
}
```

O/P: Welcome to Java.

Explanation:

Thread.sleep throws InterruptedException and it is mentioned near the signature of main method. So no compile time error here. after 10000 ms, Program prints "Welcome to Java".

Example 2:

```
class NegativeAmountException extends Exception
{
    NegativeAmountException(String msg
                            ~~int Amount~~)
    { super(msg); } }
```

→

```
class Example
{
static void Validate(int amount) throws NegativeAmountExcepti
    {
        if (amount>0)
            System.out.println("Amount deposited");
        else
            throw new NegativeAmountException("Invalid
                amount");
    }
}
class demo
{
    public static void main (String args[])
    {
        try {
            int amount = 5000;
        Example.Validate (amount);
            amount = -1000;
        Example.validate (amount);
        }
        catch (NegativeAmountException e)
        {
            System.out.println(e);
        }
    }
}.
```

Note:

* throws keyword is only for Checked Exceptions.
* It will not prevent abnormal termination of the program.
* It can provide the information to the caller of the method about the Exception.

Difference between throw and throws:

throw	throws
1. This is used to throw an Exception explicitly. It is used inside a method.	1. This is used in method signature to indicate that exception will be thrown.
2. Checked Exception cannot be propagated using throw.	2. Unchecked Exception cannot be propagated using throws.
3. throw keyword is followed by an instance of Exception to be thrown.	3. throws keyword is followed by the class names of exception to be thrown.
4. throw is allowed to throw only one exception at a time.	4. throws keyword can declare multiple Exceptions thrown by the method

```
    System.out.println(e + "Division by zero error");
  }
}
```

O/p: java.lang.ArithmeticException : Division by zero error.

Multiple catch clauses:

* when more than one exception raised by the program, to handle the situation, 2 or more catch clauses should be used.

* different catch blocks will catch different type of exception

* when an exception is thrown, each catch block is inspected in order and find the catch block whose type matches with the exception.

Example:

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

```
int a = 0;
int b = 42 / a;
int c[] = {1, 2, 3};
c[42] = 99;
String s = null;
System.out.println(s.length());
} catch (ArithmeticException e)
  {
    System.out.println(e + "Divide by zero");
  }
  catch (ArrayOutOfBoundsException e)
  {
    System.out.println(e);
  }
  catch (StringOutOfBoundsException e)
  {
    System.out.println(e);
  }
}
}
```

o/p: Divide by zero & ArithmeticException.

java.lang.ArrayOutOfBoundsException

java.lang.StringOutOfBoundsException.

Note:

If a program contains catch blocks for Exception class and ArithmeticException, then all the exceptions will be routed to catch block with Exception class.

Because 'Exception' class is the base for all Exceptions. So the other catch blocks will not be executed.

~~Nested try statements:~~

LectureNo. Java Built in Exceptions.

Topic(s) to becovered	Exception - types of exception - checked - unchecked exception - built in exception - user defined exception - Examples.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	know about exception & types	understand.
LO2	differentiate b/w checked & unchecked exception	understand.
LO3.	discuss about types of examples.	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Exception :

* It is an event occurs during the execution of a program and distrupts the normal flow of the program's instructions.

* The bugs (or) errors, we don't want and restrict our program's normal execution of the code, are refered to exception.

→

Types of exceptions:

1. Built-in-Exceptions
   - checked Exception
   - unchecked Exception.
2. User-defined Exception.

List of built-in-Exceptions:

checked	unchecked.
1) ClassNotFoundException	1) ArithmeticException
2) InterruptedException	2) ClassCastException
3) IOException	3) NullPointerException
4) InstallationException	4) ArrayIndexOutOfBoundsException.
5) SQLException	5) ArrayStoreException.
6) FileNotFoundException	6) IllegalThreadStateException.

Built-in Exceptions: * Exceptions that are already available in Java libraries are refessed to as built-in-exception.

* These exceptions able to define the error situation, so that we can understand the reason of the error.

Checked Exception:

* They are called compile-time exceptions
* They are checked at compile time by the compiler.
* programmer has to handle the exceptions. ~~are~~ otherwise system display it as "compilation error."

Unchecked Exception:

* The compiler will not check these exceptions at compile time.
* RuntimeException class is able to resolve all the unchecked exceptions.
* Eventhough we not handle the unchecked exception, no compiler error will be shown.

User defined Exception:

* Creation of our own exception class by extending Exception class and if we can throw our own exception on a particular condition using 'throw' keyword.

~~Example for userdefined exception.~~

Example for Built in Exceptions:

1) ArrayIndexOutOfBoundsException:

```
int arr[] = new int[] {91, 10, 21};
        (or)
int arr[] = {91, 10, 21};
try {
        System.out.println(num[3]);
        System.out.println(num[-1]);
        System.out.println(num[7]);
    }
catch (ArrayIndexOutOfBoundsException e)
  {
    System.out.println(e);
  }
```

Explanation:

* Array contains 3 elements and index positions are 0, 1, 2.
* Exception arises (i) if we access element at index 3.
* (ii) if we access element at -ve index. (-1)
* (iii) If we access element at greater index (7).

2. Arithmetic Exception:

(write example for divide by zero).

3. ClassCastException:

* When we try to convert an Integer class object onto String object, ClassCastException arises.

ex:
```
try{
Object obj = new Integer(100);
System.out.println(String(obj));
}catch(ClassCastException e)
{
System.out.println(e);
}.
```

4. ClassNotFoundException:

* When we try to load/execute a class, by passing its name. But the class is not found anywhere in our system means, it is classNotFoundException.

Ex:
```
try
{
Class.forName("ext1.demo");
ClassLoader.getSystemClassLoader().LoadClass("ext1.demo")
}
catch(ClassNotFoundException e)
{ System.out.println(e);}.
```

5) FileNotFoundException:

(Refer file program).

6) IllegalStateException: It signals that a method has been invoked at an illegal (or) inappropriate time.

7) ArrayStoreException: It handles the situations like when an attempt has been made to store the wrong type of object into an array of objects.

8) NullPointerException: It is thrown when program attempts to use an object reference that has a null value.

Ex: A person object is null and we are accessing its fields.

```
Class Person
{
    String PersonName;
    String getPersonName() { return PersonName; }
    String setPersonName(String n) { PersonName = n; }.
}

Person obj = null;
try {
    String name = obj.getPersonName();
} Catch ( NullPointerException e)
    { System.out.println(e); } }
```

9. NumberFormatException:

Invalid conversion of a String to a number format.

```
Ex:  String Str1 = "88";
     String Str2 = "100ABCD";
     try {
         int x = Integer.parseInt(Str1); ✓
         int y = Integer.parseInt(Str2); X
     } catch (NumberFormatException e)
       {
           System.out.println(e);
       }
```

10. StringOutOfBoundsException:

When the refered index is not present in the given String, this exception occurs.

```
Ex:  String str = "Hello World";
     try
       { System.out.println("str.charAt(-1));
         System.out.println(str.charAt(15));
       }
     catch (StringOutOfBoundsException e)
      { System.out.println(e); }.
```

LectureNo. User defined Exceptions.

Topic(s) to becovered	User defined exception – Steps to Create user defined exception – Examples.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
LO1	about User defined exception	understand.
LO2	use user defined exception	Apply.
~~LO3.~~		

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

USER DEFINED EXCEPTION: (CUSTOM EXCEPTION)

* we can create our own exception class and throw that exception using throw keyword. These exceptions are called users defined exceptions.
* other name is custom exception.
* These exceptions are created by the user as per the requirements of the application.

Example:

1) Voting age of India: If a person's age entered is less than 18 years, the program throws "Invalid age" as custom / user defined exception.

2) A banking application, when a customer types a negative amount, then the program throws "Invalid amount" as user defined exception.

Creation of user defined ~~class~~ Exception. (Steps)

Step1: User defined exceptions are created simply by extending Exception class.

Ex:

```
Class NegativeAmountException extends Exception
```

NegativeAmountException: user defined exception

Exception: base class for all exceptions

Step2: If you want to store the exception details, define a parameterized constructor with String as a parameter.

Call the superclass (Exception) constructor and store the variable str in that.

→

Ex: ← Constructor

```
NegativeAmountException(String str)
{
    super(str); // call to superclass constructor
}
```

Step 3: we need to create an object of user-defined exception class and throw it using throw clause.

Ex:

```
throw new NegativeAmountException("Invalid Amount");
```

Example Program 1:

```
public class NegativeAmountException extends
                                    Exception.
{
    NegativeAmountException(String str)
    {
        super(str);
    }
}

public class Example
{
    public static void validate(int amount)
    {
        if(amount<0)
```

```
throw new NegativeAmountException("Invalid amount");
else
    System.out.println("Valid amount");
}
}
public class demo
{
public static void main(String args[])
{
  Example ex1 = new Example(5000);
          ex1.validate(5000)
  Example ex2 = new Exception(-10000);
          ex2.validate(-10000);
}
}
```

O/P:

```
Valid amount        // 5000
Invalid amount      // -10000.
```

Example program 2:

```
public class InvalidAgeException extends Exception
{
    InvalidAgeException(String str)
    {
        super(str);
}   }
```

```
public class Example
{
    public static void checkEligible (int age)
    {
        if (age < 18)
            {throw new InvalidAgeException ("You are
                not eligible for voting"); }
        else
            {
                System.out.println ("You are eligible
                    for voting");
            }
    }
}
public class demo
{
  public static void main (String args[])
  {
      Example ex1 = new Example ();
          ex1.CheckEligible (28);
          ex1.CheckEligible (16);
  }
}
```

→

Output:

~~You~~ You are eligible for voting // 28.

You are not eligible for voting. // 16

LectureNo. Java thread model.

Topic(s) to becovered	Thread – Thread model (life cycle model) – ways of creating thread – Runnable Interface – Thread class – Thread priority – methods.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	know about Thread	understand.
LO2.	Explain about life cycle of thread	understand
LO3	Explain the two ways of creating thread.	understand.
LO3.	know about thread priority & methods	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Thread:

(*) It is a light weight process.

(*) Smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

* It is executed without affecting main program.

* Each thread has its own path for execution. So it is independant.

* If one thread gets error (or) exception, it will not affect the execution of other thread.

* Each thread share a

1) common memory
2) own stack
3) local variables
4) program counter.

Multithreading: when multiple threads are executed in parallel at the same time, it is called as multithreading.

Thread model:

Thread has several states.

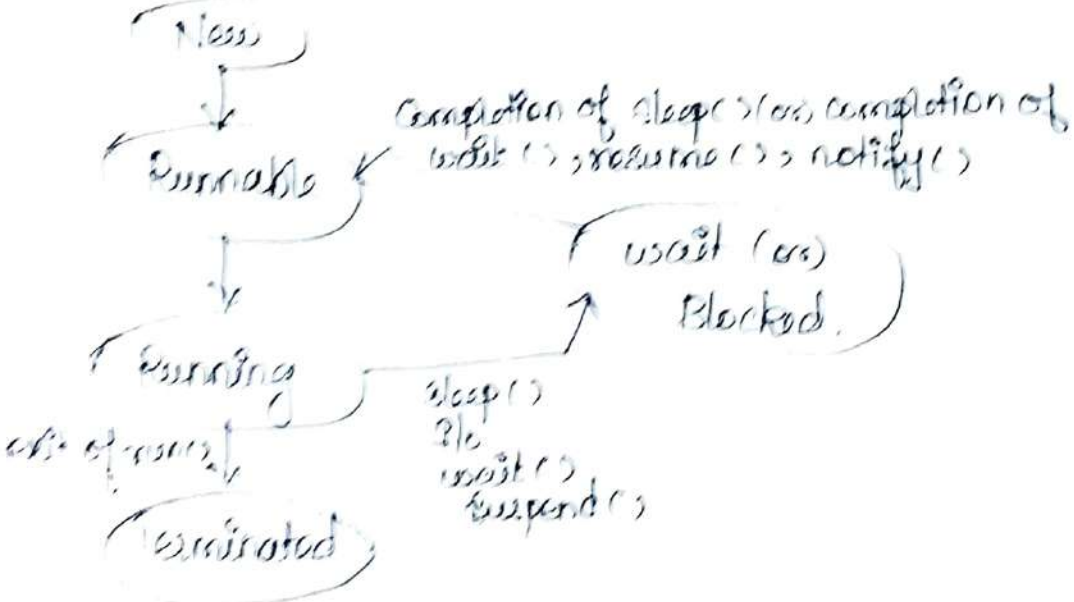


Fig: Thread model

1) New (Ready to run): when a thread is created it is said to be in new state (or) born state.

2) Runnable / Ready: when a thread calls start() method, then the thread is said to be in the Runnable state (or) Ready state.

3) Running: when the thread is under execution, it is said to be running. The thread gets allocated in CPU. This is done by calling run() method of the thread, which is automatically called after start() method.

4) Blocked / waiting: when the thread is waiting for resources to be allocated, it is called as blocked state.
The thread in the running state, may move to the blocked state due to various reasons like sleep() method, wait(), suspend(), join() etc.

The thread from waiting state (or) blocked, it can move to Runnable state by calling notify(), notifyAll(), resume() method.

Ex:

```
Thread.sleep(1000);
wait(1000);
wait();
suspended()
(Running to waiting)
```

```
notify();
notifyAll();
resume();
(waiting to running)
```

Dead / Terminated state:

* Thread in the running state may move to dead state due to either its execution completed normally (or) by forced termination by calling stop() method.

Creating a thread:

* the thread is created by

1) creating (or) implementing Runnable Interface.
2) extending Thread class.

1. Creating Thread by implementing Runnable:

* The easiest way to create a thread is to create a class that implements the Runnable interface. The class should implement a method run().

Example:

```
public class Sample implements Runnable
{
    Public void run()
    {
        try {
            for( int i=0; i<5; i++)
            {
```

```
System.out.println(" child Thread "+i);
Thread.sleep(1000);
}
catch (InterruptedException e)
{
  System.out.println(" child Interrupted");
}
}
}

public class demo
{
  public static void main(String args[])
  {
    Sample s = new Sample();
    Thread t = new Thread(s);
    t.start();
  }
}
```

2. Extending thread class.

* second way of creating thread is to create a new class that extends Thread, and then creates an instance of the class.

* The extending class must override run() method, which is the entry point of the new thread.

* To begin the execution of the new thread, call start() method.

Example:

```
public sample extends Thread
{
    public void run()
    {
        try {
            for(int i=0; i<5; i++)
            {
                System.out.println(i); Thread.sleep(1000);
            }
            } catch (InterruptedException e)
            {
                System.out.println(e);
            }
    } }

public class demo
{
    public static void main (String[] arg)
    {
        sample s = new sample();
        s.start();
    }
}
```

Thread priority:

* Every thread has a priority. They are represented as numbers b/w 0 to 10. The thread scheduler schedules the thread according to their priority (Preemptive scheduling).

3 constants (or) 3 types of priority:

(i) MIN_PRIORITY ← 1
(ii) NORM_PRIORITY ← 5 (default priority)
(iii) MAX_PRIORITY ← 10.

Example:

```
Public class Sample extends Thread
{
    Sample(String s)
    {
        super(s);
        start();
    }

    public void run()
    {
        for(int i=0;i<5;i++)
        {
            Thread current = Thread.currentThread();
            current.setPriority(Thred.MAX_PRIORITY);
            int p= current.getPriority();
```

→

```
System.out.println("Threadname" + Thread.currentThread()
                                    .getName());
System.out.println("Thread priority" + current);
  }
  }

public class demo
{
  public static void main (String args[])
  {
    Sample S = new Sample("My thread1");
  }       S.start();
}
```

Thread methods:

1) start() → It starts the thread.
2) getState() → It returns the state of the thread.
3) getName() → It returns the name of the thread.
4) getPriority() → It returns the priority of the thread.
5) sleep() → stops the thread for specified amount of time
6) ~~join~~ join() → stops the current thread until, the called thread gets terminated.
7) isAlive(): Check if the current Thread is alive (or) not.

LectureNo. Multithreading.

Topic(s) to becovered	Multithreading - difference b/w multithreading and multi-tasking - Example - main Thread - Suspending, resuming and stopping threads.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	define multithreading	Remember.
Lo2	differentiate b/w multithreading and multitasking	understand.
Lo3	understand controlling threads.	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Multithreading:

* It is a programming concept in which a program(process) is divided into 2 (or) more subprograms(process), and can be implemented ~~as~~ at the same time in parallel.

* Multithreaded program contains 2 (or) more parts that can run concurrently, to maximize the cpu utilization.

* Each part is called as a thread and each thread defines a seperate path of execution.

Multithreading	Multitasking.
1) executes multiple threads simultaneously.	1) multitasking is to run multiple processes on a computer concurrently
2) The cpu switches between multiple threads in the same process, during execution	2) In multitasking, the cpu switches b/w multiple processes to complete execution.
3) Resources are shared among multiple threads in a process	3) resources are shared among multiple processes.
4) light weight and easy to create	4) heavy-weight and tough to create.

Example of multi-threading:

Program Question: Create 2 threads xSquare and xCube to find out x^2 and x^3. Create another thread, which implements random numbers generator to generate 5 random numbers. If the random numbers is even, call the xSquare thread else call xCube thread.

```
class Sample extends Thread
{
    public void run()
    {
        int num;
```

→

// Random number generator

```
Random r = new Random();
```

→ available in util package

```
try {
    for(i=0; i<5; i++)
    {
        num = r.nextInt(100);
```

↙ It means (0 to 100) range.

```
        System.out.println("Main thread:" + num);
        if (num % 2 == 0)   // check even
        {
            Thread t1 = new Thread( new XSquare(num));
            t1.start();
        }
        else   // odd.
        {
            Thread t2 = new Thread( new XCube(num));
            t2.start();
        }
        Thread.sleep(1000);
        S.O.P(".....");
    }
    } catch (Exception e)
        { S.O.P(e);
        }
}
}
```

→

```
public

class XSquare implements Runnable          //x^2
{   public int x;
    public XSquare(int x)
    {  this.x = x;
    }
    public void run()
    {  System.out.println("child thread: xSquare of "+
                          x + "=" + (x*x));
    }}

class XCube implements Runnable            //x^3.
{   public int x;
    public XCube (int x)
    {  this.x = x;
    }
    public void run()
    {  System.out.println("child thread : X Cube of "+
                          x + " = " + (x*x*x));
    }}
```

→

```
Class demo
{
  public static void main (String args[])
  {
      Sample S = new Sample();
      S.start();
  }
}
```

Main Thread:

* when program is started to run, main program (or) main thread is running.

* It must be the last thread, waits to finish all the other threads joined to it. Because it performs various shutdown actions.

Suspending, Resuming and Stopping threads: (or) Controlling threads:

1. suspend() → to pause the Execution of thread
2. resume() → to restart ,,
3. stop() → to stop the execution of thread.

→

Program:

```
// Suspending & resuming a thread
class NewThread implements Runnable
{   String name;
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New Thread" + t);
        suspendFlag = false;
    }

    public void run()
    {   try{
            for(int i=0; i<5; i++)
            {
                S.o.p(name + ":" + i);
                Thread.sleep(200);
                synchronized(this)
                {
                    while(suspendFlag) { wait(); }
                }
            }
        } catch(InterruptedException e)
          { S.o.p(e);
          }
    }
```

```
synchronized void mysuspend()
  {
    suspendFlag = true;
  }
  synchronized void myresume()
   {  suspendFlag = false; notify();
   }
}

class demo
{
   public static void main (String args[])
   {
      NewThread t1 = new NewThread("one");
      NewThread t2 = new NewThread("two");

            t1.t.start();
            t2.t.start();   } // start threads

      try {
         Thread.sleep(1000);
         t1.mysuspend();
         S.o.p("suspending thread one");
         Thread.sleep(1000);
         t1.myresume();
         S.o.p("resuming thread one");
```

→

```
ob.mysuspend();
S.o.p("Suspending thread two");
Thread.sleep(1000);
ob.myresume();
S.o.p("Resuming thread two");
}
catch(InterruptedException e)
{
S.o.p(e);
}
// wait for threads to finish.
try{
ob1.t.join();
ob2.t.join();
}catch(InterruptedException e)
{
S.o.p(e);
}
}
}
```

LectureNo. Synchronization

Topic(s) to becovered	

	**Lecture Outcome (LO)**	**Bloom's Level**
	At the end of this lecture, students will be able to	

Teaching Learning Material	**Student Activity**
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Synchronization:

* It is the capability to control the access of multiple threads to any shared resource. So that only one thread is allowed to access the shared resource at a time.

* It is used to prevent
  (i) thread interference
  (ii) consistency problem.

Types of synchronization:

1) process synchronization
2) thread synchronization.

Thread synchronization:

2 types: (1) mutual exclusive
(2) Inter-thread communication. (or) cooperation

Mutual exclusive can be achieved by

(1) Synchronized method
(2) Synchronized block
(3) Static synchronization.

Synchronized method:

```
package Thread;

public class SyncThread
{
    public static void main(String args[])
    {
        Share s = new Share();
        MyThread T1 = new MyThread(s, "Thread1");
        MyThread T2 = new MyThread(s, "Thread2");
        MyThread T3 = new MyThread(s, "Thread3");
    }
}
```

```
class MyThread extends Thread
{
    Share s;
    MyThread (Share s, String str)
        {
            super (str);
            this.s = s;
            start();
        }
    public void run()
        {
            s.myMethod (Thread.currentThread().getName());
        }
}
class share
{
    public synchronized void myMethod (String str)
    {
        for(int i=0; i<5; i++)
        {
            try {
            }       Thread.sleep(1000);
            catch (Exception e)
            {
                S.o.p(e);
            }
    } }
}
```

Synchronized Block:

* synchronized block can be used to perform synchronization on any specific resource of the method.

* Suppose we have 50 lines of code in a method, but we want to synchronize only 5 lines, we can use synchronized block.

* If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Example:

```
class Table
{
    void printTable(int n)
    {
      synchronized (this)   // synchronized block
      {
        for(int i=1; i<=5; i++)
        {
          System.out.println(n*i);
          try {
            Thread.sleep(400);
          } catch (InterruptedException e)
            { e.o.p(e);
            }
        }
      }
}

class MyThread1 extends Thread
{
    Table t;
```

```
thi
MyThread1( Table t)
{
    this.t = t;
}
public void run()
 {
   t.printTable(5);
  }
 }
}

class MyThread2 extends Thread
{
   Table t;
   MyThread2 ( Table t)
   { this.t = t;
   }
  public void run()
  { t.printTable (100);
   }
}

public class demo
 { public static void main( String args[])
    {
        Table table = new Table ();
```

→

```
MyThread1 t1 = new MyThread1 ( table);
MyThread2 t2 = new MyThread2 (table);
t1.start();
t2.start();
}
}
```

Static Synchronization:

* If we make any static method as synchronized, the lock will be on the class, not on object.

* If static keyword used in synchronized method, it is static synchronization.

Example:

```
Class Table
{
synchronized static void printTable(int n)
{
    for(int i=0; i<=10; i++)
    {
      System.out.println(n*i);
      try {
          Thread.sleep(400);
        } catch(Exception e)
          { s.o.p(e);
      } } }
```

```
class MyThread1 extends Thread
{
    public void run()
    {
        Table.printTable(1);
    }
}

class MyThread2 extends Thread
{
    public void run()
    {
        Table.printTable(10);
    }
}

class MyThread3 extends Thread
{
    public void run()
    {
        Table.printTable(100);
    }
}

public class demo
{
    public static void main(String args[])
    {
        MyThread1 t1 = new MyThread1();
        MyThread2 t2 = new MyThread2();
        MyThread3 t3 = new MyThread3();
        t1.start(); t2.start(); t3.start();
    }
}
```

LectureNo. Inter Thread Communication

Topic(s) to be covered	

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Inter Thread communications

* other name is co-operation.

* It is a method of allowing synchronized threads to communicate with each other.

* It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or) lock in the same critical section to be executed.

→

* Exchanging of Information between threads is called Interthread communication.

* For example, consider 2 threads A & B.

Thread A → produces data.

Thread B → It uses the data produced by A and perform its task.

* If thread B waits for thread to produce data, it will waste many cpu cycles.

* If these threads communicate with each other when they have completed their tasks, they do not have to wait and check each other's states everytime.

* Thus, the cpu cycles will not waste.

Way of achieving Interthread communication:

* 3 methods provided by object class of java.lang. package.

1. wait()
2. notify()
3. notifyAll()

* These methods can be called only within a synchsonized method (or) synchsonized block of code otherwise, exception "IllegalMonitorStateException" is thrown.

→

* These methods are declared as final
* They will throw checked exception. so they must be covered with try catch block

wait():

* wait() method notifies the current thread to give up the monitor (lock) and to go into sleep state, until another thread wakes it up by calling notify() method.
* This method will throw "InterruptedException"

Different forms of wait() method (or) syntax:

1) public final void wait()
2) public final void wait(long millisecond) throws InterruptedException
3) public final void wait(long millisecond, long nanosecond) throws InterruptedException.

long millisecond -> It indicates the amount of time the thread should wait
(or)
-> It specifies the waiting time of the thread

Note:

1. A monitor is an object which acts as a lock. It is applied to a thread only when it is inside a synchronized method.

2. Only one thread can use monitor at a time. When a thread acquires a lock, it enters the monitor.

3. When a thread enters into the monitor, other threads will wait until first thread exits monitor.

4. A lock can have any number of associated conditions.

notify() method:

* It wakes up a single thread, that called wait() method previously on the same thread.
* If more than one thread is waiting, this method will awake any one of them.
* Syntax: public final void notify()

notifyAll() method:

* It is used to wakes up all threads that called wait() method on the same object.

* The thread having highest priority will run first.

* Syntax: public final void notifyAll()

* Example Program 1: (without interthread communication)

This program contains a thread, which uses the data delivered by another thread without using wait() and notify().

```
pubic class A
{
  int i;
  Synchsonized void send(int i)
  {
      this.i=i;
      System.out.println("Data delivered"+i);
  }
  Synchonized int receive()
  {
     System.out.println("Data received"+ i);
     return i;
  }
  }
```

→

```
public class Thread1 extends Thread
{
    A obj;
    Thread1 (A obj)    // constructor
    {
        this.obj = obj;
    }
    public void run()
    {
        for(int j=1; j<=5; j++)
        {
            obj.send(j);
        }
    }
}
public class Thread2 extends Thread
{
    A obj;
    Thread2 ( A obj) // Constructor
    {
        this.obj= obj;
    }
    public void run()
    {
```

→

```
for ( int K=0; K<=5; K++)
 {
   obj. receive();
  }
}
}

public class demo
{
 public static void main ( String args[])
 {
      A obj= new A();
      Thread t1= new Thread1 (obj);
      Thread t2= new Thread2 (obj);
       t1. start();
       t2. start();
   }
   }
```

output:

```
Data delivered : 1
Data delivered : 2
Data delivered : 3
Data delivered : 4
Data delivered : 5
Data received : 5
      :
```

→

Data received : 5
Data received : 5
Data received : 5
Data received : 5

not correct due to no communication.

Explanation:

* data send by Thread1 is used by thread 2.
* Due to no communication between these two threads, thread2 used the data '5' into five times in a row

Example 2: Interthread communication (wait(), notify().

```
public class A
{
    int i;
    boolean flag = false; // flag will be true when data Production over.
    synchronized void send ( int i)    // produce
    {
        if (flag)
        {
            try
            {
                wait(); // waits till a notification from Thread 2
            }
            catch (InterruptedException e)
            {
                System.out.println(e)
            }
```

```
this.i = i;
flag = true;
System.out.println("Data delivered" + i);
notify();   // data is ready to consume : notify to Thread2.
}

synchronized int receive()   // Consume.
{
   if(!flag) {
   try {
      wait();
      }
    catch(InterruptedException e)
    {
    System.out.println("Data received"+i);
    notify();   // when the data is received, it notifies Thread1, to send next data.

    return i;
   }

public class Thread1 extends Thread.   // Produces
{  A obj;
   Thread1(A obj)
   { this.obj = obj;

   }
```

→

```
public void run()
{
    for(int j=1; j<=5; j++)
    {
        obj.send(j);
    }
}}

public class Thread2 extends Thread.        // Consumer
{
    A obj;
    Thread2(A obj)
    {
        this.obj=obj;
    }
    public void run()
    {
        for(int k=0; k<=5; k++)
        {
            obj.receive();
        }
    }}

    public class demo {

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

```
A obj = new A();
Thread1 t1 = new Thread1(obj);
Thread2 t2 = new Thread2(obj)
    t1.start();
    t2.start();
}}
```

O/P:

```
Data delivered: 1
Data Received : 1
Data delivered : 2
Data Received : 2.
    :
Data delivered : 5
Data Received : 5.
```

Explanation:

* wait() and notify() methods are called inside ~~deliver~~ Send() and receive() method. Both methods enable Thread1 to notify Thread2 after producing data and will until Thread2 complete usage.

* Thread2 after using the data, notifies Thread1 and wait until Thread1 produces and delivers the next data. Thus the output comes in a Synchronized form.

LectureNo. Wrappers & Autoboxing.

Topic(s) to be covered	Type wrappers - Char - boolean - Number type - Auto boxing - methods - Auto boxings unboxing in expressions - bool & char - errors

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	list out the types of wrappers	Understand
Lo2	define Auto boxing & unboxing	Remember
Lo3	describe the autoboxing in expressions	Understand

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Type Wrappers: (or) Wrapper classes:

* The Wrapper classes in Java provides the mechanism to convert primitive types into object and objects to primitive types. * It encapsulates primitive types within an object.

Need (or) use of Wrappers classes in Java:

Java is a oop language deals with Collections, Serialization, and Synchronization, etc.

→

1 (*) Serialization:

- we need to convert the objects into streams to perform the serialization. So if we have a primitive value, it should be converted into objects through wrappes classes.

2 (*) Synchronization:

- Java synchronization works with objects in multithreading.

3 (*) Utility classes:

- Util package provides utility classes to deals with objects.

4 (*) Collection framework:

- Java Collection framework works with the objects only. All classes of collection frame work (ArrayList, LinkedList, Vector, HashSet, TreeSet, PriorityQueue, etc) deals with objects only.

5 (*) (Pass by value) To change the value in method:

- Java supports call by value only. But if we pass a primitive value to a method, the changes will not reflect in the original value.

- So primitive value should be converted in to object before passing to method. Thus it changes the original value.

Type wrappers classes in Java: | Wrapper class has methods to fully integrate the primitive type into object.

* The eight classes of the java.lang.package are known as wrapper classes in Java. They are

1. Boolean → Boolean type wrapper classes.
2. Character → character type wrapper class.
3. Byte
4. Short
5. Integer
6. Long
7. Float
8. Double.

} Number type wrapper classes.

Primitive types and Wrapper classes.

	Primitive type	Wrapper class.
1.	boolean	Boolean.
2.	char	Characters
3.	byte	Byte
4.	short	Short
5.	int	Integer
6.	long	Long
7.	float	Float
8.	double	Double.

Characters:

Wrapping of characters:

* Character is a wrapper around char.
* Constructor of Character is

Syntax: `Character ( char ch)`

ch -> character to be wrapped to object

* Example:

```
char ch = 'a';
Character charobj = new Character(ch);
System.out.println( charobj);
```

↳ character object

* o/p: `a`

Unwrapping characters:

* To obtain the character value from character object; charValue() method is used. It returns encapsulated char.

Syntax: `char charValue()`

* Example:

```
char ch = charobj.charValue();
System.out.println("character value:" + ch);
```

* o/p:

```
character value: a
```

Boolean:

Wrapping: Boolean is a wrapper around boolean values.

Syntax:

```
Boolean ( boolean  boolValue)   -> true/false.
Boolean ( String   boolString)  -> "true"/"false"
```

Unwrapping: to obtain boolean value from boolean object.

Syntax:

```
boolean  booleanValue( )
```

The Numeric type Wrappers: ~~Integer &~~

* These are wrappers to numeric values.
* They are Byte, Short, Integer, Float, Double.
* All numeric types wrappers inherits the abstract class Numbers.
* Number declares methods that return the value of an object in each of the different number formats.

Wrapping:

Syntax:

```
Byte ( byte  byteValue)
Short ( short  shortValue)
```

→

Long (long longValue)

Integer (int intValue)

Float (float floatValue)

Double (double doubleValue).

unWrapping:

Syntax:

byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( ).

Example Program:

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

        byte b = 10;

        short s = 20;

        int i = 30;
```

```
long l = 40;
float f = 50.0F;
double d = 60.0D;
char c = 'a';
boolean bb = true;
```

//Wrapping. primitive value to object.

```
Byte byteObj = new Byte(b);
Short shortObj = new Short(s);
Integer IntObj = new Integer(i);
Long longObj = new Long(l);
Float floatObj = new Float(f);
Double doubleObj = new Double(d);
Character charObj = new Character(c);
Boolean boolObj = new Boolean(bb);
```

//Un Wrapping & display.

```
System.out.println("byte value:" + byteObj.byteValue();
System.out.println("Short value:" + ShortObj.shortValue();
System.out.println("double value:" + doubleObj.doubleValue();
System.out.println("float value:" + floatObj.floatValue());
```

```
System.out.println("Integer value:" + intObj.intValue());
System.out.println("Long value:" + longObj.longValue());
System.out.println("Character value:" +
                   charObj.charValue());
System.out.println("Boolean value:" + boolObj.
                   boolValue());
}
}
```

O/P:

byte value: 10
Short value: 20
Integer value: 30
Long value: 40
float value: 50.0
double value: 60.0
Character value: a
Boolean value: true.

Exception: If any value mismatch happens, the eight wrapper classes throw NumberFormatException.

Autoboxing:

* The automatic conversion of primitive data type into its equivalent wrapper type (object) is called as autoboxing.

Ex:
```
int a=10;
Integer intObj = a;
```
Autoboxing.

Boxing: Conversion of primitive datatype to its equivalent wrapper type (object) is called as boxing.

Ex:
```
int a = 10; // Primitive type 'int' data.
Integer intobj = new Integer(a);
```
- Integer → wrapper class for int.
- intobj → Integer object
- new Integer(a); → Boxing manually (or) explicitly:

Unboxing: Conversion of wrapper type (object) into its primitive type equivalent (or) extracting the primitive type value from its equivalent wrapper object is called Unboxing.

Ex:
```
int ans = intObj.intValue();
```
↓ 10.

Unboxing manually (or) explicitly.

Auto Unboxing: Automatic conversion of wrapper type (object) into its primitive type equivalent.

Ex:
```
int ans = intObj;
```

Example:

```
// auto boxing and auto unboxing

class Example
{
  public static void main (String args[])
  {
      Integer iOb = 100; // autoboxing of int
      int i = iOb; // auto unboxing.
      System.out.println (i + "  " + iOb);
  }
}
```

o/p: 100 100

Autoboxing in methods:

autoboxing occcurs whenever a primitive type is assigned to an object. auto unboxing occcurs whenever an object is assigned to primitive type.

* Autoboxing & auto unboxing occcurs when an argument is passed to a method and when a value is returned by a method.

Example:

```
Class Example
{
    static int add ( Integer A, Integer B)
    {
        return ( A+B);
    }

    public static void main ( String args[])
    {
        int a = 10;
        int b = 20;

        Integer C = add ( a, b);

        System.out.println ("answer:" + C); //auto unboxing
    }
}
```

Explanation:

add (a,b) ⟶ add (Integer A, Integer B)

10,20 Auto boxing.

return (A+B) ⟶ int

auto unboxing.

add (a,b) ⟶ Integer C.

auto boxing.

⟶

Auto boxing and unboxing in expressions:

```
Class Example
{
  public static void main ( String args[ ])
  {
      Integer ob1, ob2;
      int i;
      ob1 = 100;  // Autoboxing.
      System.out.println ("value of ob1" + ob1); // auto
                                                  unboxing
      ++ob1;  // unboxing & reboxing automatic.
      System.out.println (" value of ob1 after increment "
                          +ob1); // auto unboxing.
      ob2 = ob1 + (ob1/3); // automatic unboxing &
                             reboxing.
      System.out.println ("value of ob2" + ob2); //
                                                  auto unboxing
  }
}
```

auto boxing & unboxing boolean & character values:

Ex:

```
Boolean B = true;  // auto boxing.
if (B)  // auto unboxing
{
  System.out.println (" true");
}
else
  { System.out.println ("false");}
```

Auto boxing / unboxing helps to prevent errors.

Ex:

```
Class Example
{
    public static void main (String args[])
    {
        Integer ob = 1000; // auto boxing.
        int i = ob.byteValue(); // manual unboxing to byte.
        System.out.println(i);
    }
}
```

wrong manual conversion.

O/P: 24. → wrong output.

Advantages of Autoboxing & auto unboxing:

1. No need of manual conversion b/w primitive type to wrapper object.
2. less coding.
3. Helps to prevent errors / avoid errors in manual conversion.

Disadvantages of autoboxing and unboxing:

1. adds overhead to the compiles.
2. affects the performance of the program.

Suggested Questions / Assignments / Home works / any other
1. what is type wrapper? 2. what are the types of type wrappers? 3. what is Autoboxing & unboxing? 4. Explain the autoboxing & unboxing with program 5. List out the advantages of type wrappers.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Synchronization

Topic(s) to becovered	Synchronization – types – thread synchroniza tion – Synchronization block – static Synchronization –

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	define synchronization	Remember. ~~Understan~~
LO2.	discuss about thread synchronization	understand.
LO3.	discuss about static synchsonization	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Synchronization:

* It is the capability to control the access of multiple threads to any shared resource. So that only one thread is allowed to access the shared resource at a time.

* It is used to prevent
  (i) thread interference
  (ii) consistency problem.

LectureNo. Inter thread communication.

Topic(s) to becovered	Inter thread communication – methods – wait() – notify() – notifyAll() –

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	define inter thread communication	Remember
LO2.	explain the methods for ITC	Understand
LO3	implement the concept of ITC	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen /Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Inter Thread communication:

* other name is co-operation.

* It is a method of allowing synchronized threads to communicate with each other.

* It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or) lock in the same critical section to be executed.

→

LectureNo. Wrappers & Auto boxing.

Topic(s) to becovered	Type wrappers - Char - boolean - Number type - Auto boxing - methods - Auto boxings unboxing in expressions - bool & char - essors

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
L01	list out the types of wrappers	Understand
L02	define Auto boxing & unboxing	Remember
L03	describe the autoboxing in expressions	Understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Type Wrappers: (or) Wrapper classes:

* The Wrappes classes in Java provides the mechanism to convert primitive types into object and objects to primitive types. * It encapsulates primitive types within an object.

Need (or) Use of Wrappes classes in Java:

Java is a oop language deals with Collections, Serializations and synchronization, etc.

1 (*) Serialization:

- we need to convert the objects into streams to perform the serialization. So if we have a primitive value, it should be converted into objects through wrapper classes.

2 (*) Synchronization:

- Java synchronization works with objects in multithreading.

3 (*) Utility classes:

- Util package provides utility classes to deals with objects.

4 (*) Collection framework:

- Java Collection framework works with the objects only. All classes of collection frame work (ArrayList, LinkedList, Vector, HashSet, TreeSet, PriorityQueue, etc) deals with objects only.

5 (*) (Pass by value) To change the value in method:

- Java supports call by value only. But if we pass a primitive value to a method, the changes will not reflect in the original value.

- So primitive value should be converted in to object before passing to method. Thus it changes the original value.

Type wrapper classes in Java:

Wrapper class has methods to fully integrate the primitive type into object.

* The eight classes of the java.lang.package are known as wrapper classes in Java. They are

1. Boolean → Boolean type wrapper classes.
2. Character → character type wrapper class.
3. Byte
4. Short
5. Integer
6. Long
7. Float
8. Double.

} Number type wrapper classes.

Primitive types and Wrapper classes.

	Primitive type	Wrapper class.
1.	boolean	Boolean.
2.	char	Character
3.	byte	Byte
4.	short	Short
5.	int	Integer
6.	long	Long
7.	float	Float
8.	double	Double.

Characters:

Wrapping of characters:

* Character is a wrapper around char.

* Constructor of character is

Syntax: `Character ( char ch)`

ch -> character to be wrapped to object

* Example:

```
char ch = 'a';
Character charobj = new Character(ch);
System.out.println( charobj);
```

→ character object

* o/p: `a`

Unwrapping characters:

* To obtain the character value from character object; charValue() method is used. It returns encapsulated char.

Syntax: `char charValue()`

* Example:

```
char ch = charobj.charValue();
System.out.println("character value:" + ch);
```

* o/p:

`character value: a`

Boolean:

Wrapping: Boolean is a wrapper around boolean values.

Syntax:

```
Boolean ( boolean  boolValue)   -> true/false.
Boolean ( String   boolString)  -> "true"/"false"
```

UnWrapping: to obtain boolean value from boolean object.

Syntax:

```
boolean  booleanValue()
```

The Numeric type Wrappers: ~~Integer &~~

* These are wrappers to numeric values.
* They are Byte, Short, Integer, Float, Double.
* All numeric types wrappers inherits the abstract class Numbers.
* Number declares methods that return the value of an object in each of the different number formats.

Wrapping:

Syntax:

```
Byte ( byte  byteValue)
Short ( short  shortValue)
```

→

Long (long longValue)

Integer (int intValue)

Float (float floatValue)

Double (double doubleValue).

unWrapping:

Syntax:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( ).
```

Example Program:

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

        byte b = 10;

        short s = 20;

        int i = 30;
```

```
long l = 40;

float f = 50.0F;

double d = 60.0D;

char c = 'a';

boolean bb = true;
```

// Wrapping. primitive value to object.

```
Byte byteObj = new Byte(b);
Short shortObj = new Short(s);
Integer IntObj = new Integer(i);
Long longObj = new Long(l);
Float floatObj = new Float(f);
Double doubleObj = new Double(d);
Character charObj = new Character(c);
Boolean boolObj = new Boolean(bb);
```

// On Wrapping & display.

```
System.out.println("byte value:" + byteObj.byteValue());
System.out.println("Short value:" + ShortObj.shortValue());
System.out.println("double value:" + doubleObj.doubleValue());
System.out.println("float value:" + floatObj.floatValue());
```

→

```
System.out.println("Integer value:" + intObj.intValue());
System.out.println("long value:" + longObj.longValue());
System.out.println("character value:" +
                    charObj.charValue());
System.out.println("Boolean value:" + boolObj.
                                    booleanValue());
}
}
```

o/p:

byte value: 10
Short value: 20
Integer value: 30
Long value: 40
float value: 50.0
double value: 60.0
character value: a
Boolean value: true.

Exception: If any value mismatch happens, the eight wrapper classes throw NumberFormatException.

Autoboxing:

* The automatic conversion of primitive data type into its equivalent wrapper type (object) is called as autoboxing.

Ex:
```
int a=10;
Integer intObj = a;
```
Autoboxing.

Boxing: Conversion of primitive datatype to its equivalent wrapper type (object) is called as boxing.

Ex:
```
int a = 10; // Primitive type 'int' data.
Integer intObj = new Integer(a);
```
- Integer → wrapper class for int.
- intObj → Integer object
- new Integer(a); → Boxing manually (or) explicitly:

Unboxing: Conversion of wrapper type (object) into its primitive type equivalent (or) extracting the primitive type value from its equivalent wrapper object is called Unboxing.

Ex:
```
int ans = intObj.intValue();
```
↓ 10.

intObj.intValue(); → Unboxing manually (or) explicitly.

Auto Unboxing: Automatic conversion of wrapper type (object) into its primitive type equivalent.

Ex:
```
int ans = intObj;
```

Example:

```
// auto boxing and auto unboxing
class Example
{
  public static void main (String args[])
  {
      Integer iOb = 100; // autoboxing of int
      int i = iOb; // auto unboxing.
      System.out.println (i + "  " + iOb);
  }
}
```

o/p: 100 100

Autoboxing in methods:

autoboxing occurs whenever a primitive type is assigned to an object. auto unboxing occurs whenever an object is assigned to primitive type.

* Autoboxing & auto unboxing occurs when an argument is passed to a method and when a value is returned by a method.

Example:

LectureNo. I/o basics.

Topic(s) to becovered	I/o basics - Streams & types - character Stream Classes - Byte Stream Classes - Predefined Streams.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	define Streams	Remember
Lo2.	Explain about Character stream classes.	Undesstand
Lo3	Explain about byte stream classes.	Undesstand.
Lo4	discuss about predefined Streams	Undesstand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

I/o basics:

* Java I/o (Input and Output) is used to process the input and produce the output based on the input.

* Java uses the concept of stream to make the I/o operations fast.

* The package which contains all the classes required for input and output operation: Java.io package.

Streams:

* Java program performs I/o through streams.
* A stream that either produces information (or) consumes information.
* A stream is linked to a physical device by Java (program) I/o System.
* Input stream abstracts diff types of input:
    1. disk file
    2. keyboard
    3. network socket.
* output stream abstracts diff types of output:
    1. Console
    2. Monitor
    3. disk file
    4. network connection.

Types of streams:

1. Byte Stream
2. Character Stream.

Byte Stream: It provides convenient means for handling input and output of bytes. It is very useful when reading or writing binary data. Lowest level I/o is byte oriented

Character Stream: It provides convenient means for handling input and output of characters. They use unicode and more efficient and convenient.

Byte stream classes:

* Byte stream is defined by 2 abstract classes
  1) Input stream
  2) output stream. (o/p)

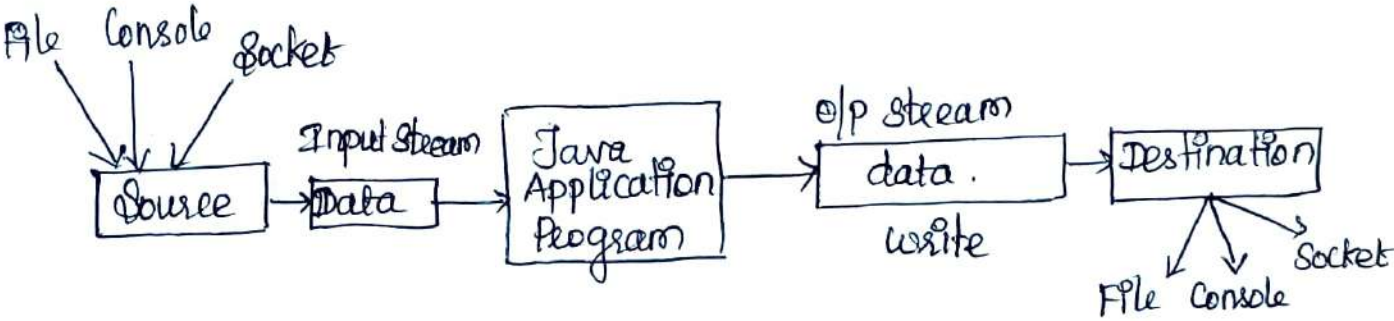


Fig: Working of Java I/o.

(1) Input Stream: It is used to read data from a source. Source may be a file (or) console (or) N/w socket (or) device

(2) output Stream: It is used to write data into destination. Destination may be a file (or) console (or) device (or) N/w Socket.

Byte Stream classes with meaning:

Byte stream class	Meaning
1) BufferedInputStream:	Buffered Input Stream.
2) BufferedOutputStream:	Buffered output Stream
3) ByteArrayInputStream:	Input stream to read a byte array
4) ByteArrayOutputStream:	output stream to write to a byte array
5) DataInputStream:	Input stream that has methods to read standard java types.

6. DataOutputStream: An output stream that contains methods for writing the java standard datatypes.

7. FileInputStream: InputStream that reads from a file.

8. FileOutputStream: OutputStream that writes to a file.

9. FilterInputStream: Implements InputStream.

10. FilterOutputStream: Implements OutputStream.

11. InputStream: Abstract class that describes stream i/p.

12. OutputStream: Abstract class that describes stream o/p.

13. PrintStream: OutputStream that contains print() , println()

14. ObjectInputStream: InputStream of objects.

15. ObjectOutputStream: output stream of objects.

The abstract classes OutputStream and InputStream define key methods.

Ex: read() → InputStream. method to read bytes of data.

write() → OutputStream. method to write bytes of data.

Character Stream Classes:

It is defined by 2 class hierarchies.

1. Reader
2. Writer

} These abstract classes can handle unicode character streams.

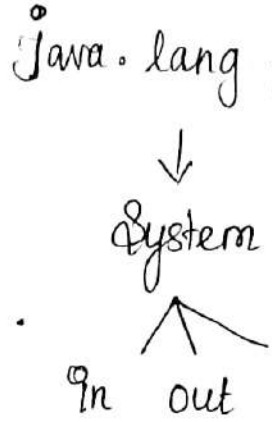
read() → charaa Reader() method to read characters of data.

write() → Writer method to write characters of data.

These methods are overridden by other classes.

Character Stream classes (I/O).	Meaning
1. BufferedReader :	Buffered Input characters stream.
2. BufferedWriter :	Buffered output character stream
3. CharArrayReader :	Input stream to read char Array.
4. CharArrayWriter :	Output Stream to write char Array.
5. FileReader :	Input Stream to read from a file
6. FileWriter :	Output Stream to write from a file.
7. FilterReader :	Filtered Reader
8. FilterWriter :	Filtered writes.
9. InputStreamReader :	Input stream translates byte to characters
10. LineNumberReader :	Input stream to count lines.
11. OutputStreamReader :	Output Stream translates Characters to bytes
12. PrintWriter :	Output Stream that has print(), println()
13. StringReader :	Input Stream that reads from String.
14. StringWriter :	Output Stream that writes from String.
15. Reader :	Abstract class that describes characters Stream I/P
16. Writer :	Abstract class that describes character Stream o/p

Predefined Streams:



* All java programs automatically imports java.lang package
* It contains a ~~class~~ System class, which encapsulates several aspects of Runtime environment.
* Ex: It has methods to obtain current time and system settings.
* System class also contains 3 predefined stream variables:

1. in:	System.in	standard input stream (Console)	(ie) keyboard.
2. out:	System.out	Standard output stream (Console)	(ie) Monitor
3. err:	System.err	Standard error stream (Console)	

System.in ← object of type InputStream.
System.out } ← object of type PrintStream.
System.err }

Suggested Questions / Assignments / Home works / any other
1. What is stream and its types. 2. Explain about character streamsclasses. 3. Explain about Byte stream classes. 4. Explain predefined streams.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Reading console Input.

Topic(s) to becovered	ways of reading console Input – Character based Stream – BufferedReader – read(), readLine() – Scanner class – Console class.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
LO1	know the ways of reading from console	Understand.
LO2	explain about the character based Streams.	Understand.
LO3.	discuss about BufferedReader and its functions.	Understand.
LO4	Use scanner class and console class.	Apply.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Reading Console Input:

* The preffered method to read console Input is to use a character oriented stream. than byteoriented Stream.
* Character oriented Stream is easy to use and maintain
* Console Input is done by reading from System.in

3 ways to read console I/o:

1. Use BufferedReader class
2. Use Scanner class
3. Use Console class

To obtain character based stream for Console: (System.in)

* Use **BufferedReader** class.
* This supports bufferedInputStream.
* Its constructor is BufferedReader(Reader inputReader)
* to get Reader, attached with System.in:

  use InputStreamReader class.: Converts bytes to characters.

  Syntax: InputStreamReader(InputStream inputStream

* InputStream has the object System.in.

Syntax: to obtain System.in (or) keyboard:

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

Here System.in object is wrapped using BufferedReader object.

Reading characters: Use read().

read(): - reads a character from input stream and returns it as integer value.

- It returns -1, when end of stream is reached
- It can throw IOException.

Program: To read characters from keyboard till 'q' is typed.

```
import java.io.*;

class BRead
{
  public static void main (String args[]) throws IOException
  {
    char c;
    BufferedReader br = new BufferedReader ( new
                        InputStreamReader ( System.in));
    System.out.println(" Enter characters, 'q' to quit.");
    // read characters one by one.
    do
    {
      c = (char) br.read();  // reads single character
      System.out.println( c);
    } while ( c != 'q');
  }
}
```

Sample output (or) Run:

```
Enter characters, 'q' to quit.
123 abcq
1
2
3
a
b
c
```

Reading Strings:

* readLine() method from BufferedReader. to read a line of text (or) String.
* Syntax: String readLine() throws IOException.

Program1: To read one string.

```
import java.io.*;
class BBReadLines
{
  public static void main (String args[])
                    throws IOException.
  {
    BufferedReader br = new BufferedReader( new
                InputStreamReader( System.in);
    String str;
    System.out.println("Enter your name: ");
    str = br.readLine();
    System.out.println("Hai" + str);
  }
}
```

Sample output:

Enter your name : Rajini kanth.

Hai Rajini Kanth.

Program 2 : To read many lines of text, till you type "stop"

```
import java.io.*;

Class BBReadLines
{
  public static void main (String args[]) throws IOException
  {
    BufferedReader br = new BufferedReader (new
                        InputStreamReader (System.in));
    String str;
    System.out.println("Enter lines of text, 'stop' to quit");
    do {
        str = br.readLine(); // reads single line of text
        System.out.println(str);
        } while ( ! str.equals ("stop"));
  }
}
```

Sample Output:

```
Enter lines of text, 'stop' to quit
java is object oriented language.
java is Object oriented language.
java supports Inheritance and polymorphism.
java supports Inheritance and polymorphism.
stop.
```

Using Scanner class (Parser) to read input from console:

* Scanner class is the mostly used method to read i/p from console.
* It is also used to parse primitive types and Strings using regular expressions.
* It can also used to read input from the user in command line.

methods of Scanners class:

1. next() → to read one word of String.
2. nextLine() → to read a line of text (or) String.
3. nextInt() → to read integer.
4. nextFloat() → to read floating point data.

⋮

etc.

Example:

```
import java.util.Scanner;
class GetInput
{
    public static void main(String args[])
    {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter your name:");
        String name = in.nextLine();
        System.out.println("Enter your Last 2 digits of RNo:");
```

```
Syst
    int Rno = in.readInt();
System.out.println(" Enter your Average mark");
    double avg = in.nextDouble();
System.out.println(" Your details");
System.out.println("Name :" + name);
System.out.println(" RNO:" + Rno);
System.out.println(" Average:" + avg);
  }
}.
```

Output:

```
Enter your Name: Sangeetha J
Enter your last 2 digits of RNo: 34
Enter your average mark: 93.5

your details
        Name: Sangeetha J
        RNO: 34
    Average: 93.5.
```

→

Reading using Console class:

* It is used for reading user's ~~input~~ password like input, without echoing the characters.
* format string also canbe used, System.out.printf()

Example:

```
public class Sample
{
  public static void main (String args[])
  { System.out.println("Enter your name:");
    String name = System.Console().readLine();
    System.out.println("your name is"+name);
  (or)
    System.out.printf("%s your name is %s",
                      name);
}}
```

Suggested Questions / Assignments / Home works / any other
* what are the ways to read Console Input? * Explain about BufferedReader and its functions with suitable example. * Explain the Scanner class in reading console I/P. * what is the use of Console class?

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Writing Console Output.

Topic(s) to becovered	Writing console output – print Stream class – print(), println() – PrintWriter class – write().

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	List out the ways of writing console output	understand.
LO2	Explain about print Stream class.	understand
LO3	Explain about Printwriter class	understand

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Writing Console Output:

The methods used to display output / write the output in the console are

1) print()
2) println() } PrintStream class
3) write()

4) printf() → PrintWriter class.

PrintStream class:

* It is Built-in class that provides two methods print() and println() to write console output.
* Both methods can be used with System.out.
* System.out.print() → writes console o/p in same line.
* System.out.println() → writes console o/p in newline (or) seperate line. This method can be used with console and also with other output resources. But print() can be used with console o/p only.

Example: // writing console o/p using print() & println()

```
int [] arr = new int[5];
for(int i=0; i<5; i++)
    arr[i]= i*10;
// Printing in sameline.
for(int i : arr)
   System.out.print(arr[i]);
// Printing in newline each time.
for (int i : arr)
    System.out.println( arr[i]));
```

O/P:

0 10 20 30.

O/P:

0
10
20
30
40.

PrintStream also provides write() method for console output.

write() method take integer as argument. It writes the ASCII equivalent character onto the console. It also accept escape sequences.

Example:

```
int [] ass = new int [5];
for (int i=0; i<5; i++)
    ass[i] = i + 65;
```
} => 65, 66, 67, 68, 69.

```
for ( int i : ass)
  {
    System. out. write (i);
    System. out. write ("\n"); --> new line.
  }.
```

O/p:

A
B
C.
D
E.

PrintWriter Class:

* It is the implementation of writer class.
* It is used to print, formatted representation of objects to the console output.

methods of printStream:

(1) println() → prints integer / array / boolean etc.

(2) append() → used to append the specified character/ set of characters to the writer.

(3) checkError() → flushes the stream and check error state

4) setError() → it is used to indicate an error occurs.

5) clearError() → clears the error state of a stream.

6) flush() → flushes the stream.

7) close() → closes the stream.

8) format() → to write formatted string to the writer using specified arguments & Strings.

Example:

```
// write some data.
PrintWriter writer = new PrintWriter(System.out);
writer.write("Java tutorials preparation");
writer.flush();
writer.close();

// write to
```

O/P:

Java tutorials preparation.

Suggested Questions / Assignments / Home works / any other
* list out the ways of writing console o/p * Explain about PrintStream class methods. How they can be used to write Console o/p. * Explain about PrintWriter class with suitable examples.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Reading and writing files.

Topic(s) to becovered	Supporting classes – Display a file – Copy a text file. – ways to reading from file – ways to write to file.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	understand various ways of reading a text file.	understand
Lo2.	use Buffered Reader to read text file	Apply
Lo3.	use File Input Stream for reading	Apply
Lo3	use File Output Stream for writing	Apply

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Reading and writing files:

* All files in java are byte oriented.
* Java provides methods to read and write bytes from and to a file.

Different ways of Reading a text file:

* File Reader
* File Stream classes.
* Buffered Reader
* Scanner.

BufferedReader provides buffering of data for fast reading. Scanner provides parsing ability.

Methods of reading file:

* 1. Using BufferedReader class
2. Using Scanner class
3. Using FileReader class
4. Reading the whole file in a list.
5. Reading a text file as string.
* 6. FileInputStreamReader

Line by line Reading:

Both BufferedReader and Scanner can be used to read a text file line by line in java.

Using BufferedReader class:

- Reads text from a characters Input Stream.
- It does buffer for efficient reading of characters, arrays and lines.
- Buffer size can be specified by user (or) default size can be used. But should be large enough
- byte stream should be wrapped in to Characted Stream.
- The read() operations of FileReader ~~is~~ or InputStreamReader is Costly. So wrapping is done

Syntax:

```
BufferedReader br = new BufferedReader (new
                    FileReader (String filename/path),
                                int size));
```

Example Program:

```
// Reading from file using BufferedReader and display
// its contents in monitor/console.

Class Example
  {
    public static void main( String args[])
    {
      // File path of the input file.

      File file = new File ("c:\\users\\san\\input.txt");

      // creating bufferedReader to read file.
BufferedReader br= new BufferedReader (new FileReader(
                    file));

      String str;

     // Reading file line by line.
    while ( (str = br.readLine())!=null)
    {
      System.out.println(str); // display line in
                                  Console.
    }
  }
}
```

I/p: Input.txt

Rno	name	Avg.
1	Akilu	78
2	Amisth	90
3	Bhuvana	82.

O/p: Console.

Rno	name	Avg
1	Akilu	78
2	Amisth	90
3	Bhuvana	82.

---

2. Reading a file and write o/p into console. using File Input Stream class.

* FileInputStream creates byte streams linked to files.
* To open a file, create object of FileInputStream class and specify the name of the file as argument in the constructor.

Syntax:

Constructor:

(name of the file to open)

FileInputStream (String filename) throws FileNotFoundException

To open a file:

* FileInputStream fin = new FileInputStream (String filename);

* when we create FileInputStream, if the file cannot be opened / created means, FileNotFoundException will be thrown.

close() : when the work is done (reading is completed), file should be closed using close() method.

Syntax: void close() throws IOException.

read(): * To read the file byte by byte and returns it as integer value.

* it returns -1, if EOF (or) End of the file is reached.

* It can throw IOException, if any error occurs.

Syntax:

int read() throws IOException.

Program for reading a file and display its contents in console using FileInputStream class:

Program:

```
Class Example
{
public static void main(String args[])
{
int i;
```

→

```
try {        // open the input file for reading.
    FileInputStream fin = new FileInputStream (
                          "C:\\Users\\San\\input.txt");
} catch ( FileNotFoundException e)
  {  // file cannot be opened for reading.
     System.out.println(e); return;
  }

// read characters, until EOF is encountered.
    do {
       i = fin.read();  // read a byte.
       if(i != -1)                    // EOF = -1.
       System.out.println((char) i); //display in console
    } while ( i != -1); // repeat till EOF occurs.
    fin.close();  // close the file after reading.
  }
}.
```

I/P: input.txt

```
List of fruits
Jack fruit
pine apple
melon.
```

o/p screen: / console:

```
List of fruits
Jack fruit
pine apple
melon.
```

File Output Stream:

* This class is used to open a file in writing mode.

* If the file cannot be opened/created, it will throw (file not found) FileNotFoundException.

* If the file is ready/opened/created, it uses write() method to write byte by byte in o/p file.

* Syntax: void write(int byteVal) throws IOException

If any error occurs while writing a file, it will throw IOException.

* close() method is used to close ~~of~~ the file after writing.

(*) Write a Program to copy one file into another file.

(or)

(*). write a program to read from one file and write the contents to another file using FileOutputStream

→

Program:

```
import java.io.*;

class example
{
public static void main ( String args[] ) throws
                                    IOException
{
    int i;
    try {
            // file to be read.
FileInputStream fin = new FileInputStream (
                        "C:\\Users\\san\\input.txt");
            // file to write.
FileOutputStream fout = new FileOutputStream(
                        "C:\\users\\san\\output.txt");
  }
catch ( FileNotFoundException e)
{       // File cannot be opened
   System.out.println(e);
   return;
}
```

→

Reading a file using Scanner class:

Ex:
```
// open input file
File file = new File("C:\\users\\san\\input.txt")

Scanner sc = new Scanner(file); // open input file

while (sc.hasNextLine()) // check file has content next line.
{System.out.println(sc.nextLine());}
                          ↳ display line in consol
```

Reading a file using FileReader class:

It is used to read characters files.
FileReader has 3 constructors:

1. FileReader(File file)
2. FileReader(FileDescriptor fd)
3. FileReader(String fileName)

Example:

```
// input file.
File file = new File("c:\\users\\san\\input.txt");

FileReader fr = new FileReader(file); // FileReader.
int i;
while ((i = fr.read()) != -1) // read byte till EOF.
{ System.out.println((char) i); // display in console
}.
```

```
try
  {
  do
  {
    i = fin.read();  // reading a byte from input file
    if(i!=-1) // if not EOF reached.
      fout.write(i); // writing a byte into o/p file
  } while(i!=-1);
 } catch ( IOException e)
   {
     System.out.println(" File error"+e);
   }
 }

 fin.close(); // closing input file
 fout.close(); // closing output file
} }.
```

(*)

Input file : input.txt

```
List of animals
  Lion
  Tiger
  Elephant
```

O/p file :
output.txt

```
List of animals
Lion
Tiger
Elephant
```

Writing to a file.

4 ways:

1. File Writer
2. Buffered Writer
3. File Output Stream.
4. Files.

FileWriter:

- Simplest way of writing to a file.
- write() method used to write array, byte, Strings into file.
- It writes directly into files, without use of buffers.
- It should be used when no. of writes is less. (ie) less content in file (small file).

Example:

```
File file = new File("C:\\Users\\San\\Output.txt");
```

if big file:

```
FileWriter fw = new FileWriter(file);
BufferedWriter br = new BufferedWriter(fw);
```

(or)

if big file:

```
{FileWriter fw = new FileWriter(new BufferedWriter(file))
```

→

```
File file1 = new File("C:\\Users\\san\\input.txt");
FileReader fr = new FileReader(file1);
String str;
while ( (str = fr.readLine()) != null)
{
    fw.write(str);
}
```

Note:

If the content is less (or) for small files, use FileWriter only.

Syntax:

```
FileWriter fw = new FileWriter(String file);
```

---

2. BufferedWriter:

* It provides overloaded methods to write int, byte array and String to a file.
* It is similar to FileWriter but it uses internal buffer to write data into file.
* If the number of write operations are more, the actual I/O operations are less and performance is better.
* When the writeOperations is more, we can use BufferedWriter.

3. FileOutputStream :

* FileWriter and BufferedWriter are meant to write text to the file but when we need raw stream data to be written into file, you should use FileOutputStream.

4. Files:

* Java 7 introduces Files utility class and we can write file using its write function. Internally it uses OutputStream to write byte array into file.

Example: // four ways to write into a file.

```
String data = "Java is an object oriented programming
               language";
int numLines = 10000;

// File output stream
    outputStream os = new FileOutputStream(new File(
                      ("/users/java/output.txt"));
    os.write(data.getBytes(), 0, data.length);
    os.close();

// Files
    Files.write(Paths.get("/users/java/output1.txt
                ));
```

→

// using FileWriter.

```
File file = new File("/users/java/Output3.txt");
FileWriter fr = new FileWriter(file);
fr.write(data);
fr.close();
```

// using BufferedWriter.

```
File file = new File("/users/java/Output4.txt"));
FileWriter fr = new FileWriter(file);
BufferedWriter br = new BufferedWriter(fr);
String datas = data + System.getProperty("line.separator");
    try{
        for (int i=0; i<numLines; i++)
            br.write(datas);
br.close();
fr.close();
```

— x —

Suggested Questions / Assignments / Home works / any other
1. What are ways to read a text file? 2. Explain about BufferedReader class with an example. 3. Explain about FileInputStream class with an example. 4. Write a Java program for copy one file into other. 5. Explain about reading from file.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Generics

Topic(s) to becovered	Generics – Generic programming – Simple generics – Generics with a type parameters – Bounded type – Generic method – Restrictions on generics.

	Lecture Outcome (LO)	
	At the end of this lecture, students will be able to	**Bloom's Level**
LO1	define generics and generic programming.	Remember.
LO2.	write programs of for generics	Apply.
LO3.	explain generic method and bounded type	Understand.
LO4	list out the restrictions on generics	Understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Generics:

* Generics means parameterized types.
* Using parameterized types, it is possible to create classes, interfaces and methods work with different types of data.
* A class works with ~~ge~~ parameterized types are called as generic class. (or) A class that can refer to any data type is known as generic class.

* A method works with parameterized type are called as generic method.

Generic programming:

It enables the programmer to create classes, interfaces and methods that automatically works with all types of data (Integer, String, Float etc). It is expanded the ability to reuse the code safely and easily.

Advantages of java generics:

1. Type-safety: We can hold only single type of objects, and doesn't support mixed datatype.
2. Type casting is not required:
3. Compile time checking: So no problem will occur at run time.

Example for Simple Generics:

```
Class example <T>
{
    T obj;
    example (T obj)
    { this.obj = obj; }
```

```
// function for
// Returning object

 T   getObject ()
 {
    return obj;
 }

// display the type of object

   void showType()
   {
      System.out.println("Type of T is" +
                        obj.getClass().getName());
    }
}

Class demo
{
  public static void main (String args[])
  {
       Example < Integer > obj1 = new example <Integer>
                                          (88);

       obj1.showType();

       int v = obj1.getObject();
       System.out.println(v);

       example < String > obj2 = new example <String>("Hai");
        obj2.showType();
  }}  System.out.println( obj2.getObject());
```

Output:

Type of T is java.lang.Integer.

~~value~~: 88.

Type of T is java.lang.String.

Hai.

Explanation:

* T is type parameter. It is contained in < >
* example is generic class, because it is associated with <T>
* T is used to declare object: T obj;
* T is used in constructor and return type of method
* Inside the demo class
  - create Integer version of the class using

    ~~Example~~ <

    example <Integer> obj1;
  - This Integer passed as parameter to 'example' class. So all of its 'T' will be converted into Integer.
  - String version of the class using

    example < String > obj2;

    String is passed as parameter to 'example' class. So all of its T ~~as~~ becomes String.

Limitations of the generics : (disadvantages)

* Generics work only with objects.

example < Integer> obj1; ✓

example < int> obj1 ; X (wrong)

* Generics type differ based on their type argument.

example < Integer> obj1;
example < String> obj2;

obj1 = obj2 ; X (wrong)

one version of generic type is not compatible with another version.

General form of the generic class:

```
class  class_name < type-param-list >
{
  //....
}
```

General form of reference to generic class from demo class:

class_name ~~ob~~ <type- param - list> objname =

new class- name <type-arg-list> (cons-arg-list);

A generic class with two type parameters:

* It is possible to declare more than one type parameter in a generic type.
* use comma seperated list format for that.

Example:

```
Class  Example < T, V >
{
   T ob1;
   V ob2;

   Example ( T ob1 , V ob2)
   {
     this. ob1 = ob1;
     this. ob2 = ob2;
   }

   void showType( )
    {
    System. out. println( "Type of T is" +
        ob1. getclass Name ( ). getName ( ));
    System. out. println ( " Type of V is" +
      ob2. getClass ( ). getName ( ));
    }

    T   getObject1 ( )
    {
      return ob1;
    }
```

```
V getobject2()
{
  return obj;
}
}

class demo
{
 public static void main(String args[])
 {

 Example < Integer, String> obj = new Example
          < Integer, String) ( 88, "Hai");
  obj. showType();
  System. out. println( obj. getObject1());
  System. out. println( obj. getObject2());

 }
 }
```

Generic method Example:

```
public class Example
{
   public static <T> void printArray(T[] elements)
   {
       for (T element : elements)
           System.out.println(element);
   }
```

```
public class demo
{
 Public static void main ( String args[])
 {
    Integer[] intArray = { 10, 20, 30, 40, 50};
    Character[] charArray = { 'J', 'A', 'V', 'A' };
    System. out.println(" Integer array");
    printArray (intArray);
    System. out. println ("Character Array");
    printArray ( charArray);
 }
}
```

Bounded Type:

The type parameter could be replaced by ~~any~~ any class type. This is used to limit the 'types' that can be passed to a type argument.

Syntax: < T extends superclass >

Example:

```
Class A <T extends Number>
{
    T nums;
```

```
A ( T[] obj)
{
   nums = obj;
 }

 double average ()
 {
   double sum= 0.0;
   for (int i=0; i<nums.length ; i++)
      sum += nums[i].doubleValue();

   return sum/nums.length;
 }

 public class demo
 {
   public static void main ( String args[])
    {
        Integer[] IntArray = {1,2,3,4,5};
        A < Integer> obj1 = new A<Integer> (IntArray);
        System.out.println("average"+obj1.average());

        Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5};
        A <Integer> obj2 = new A<Double> (doubleArray
        System.out.println(" obj2.average());
    } }
```

Restrictions on generics:

* Cannot Instantiate generic types with primitive types.
* Cannot create Instances of type parameters.
* Cannot declare static fields whose types are type parameters
* Cannot use casts (or) Instanceof with parameterized types.
* Cannot create arrays of parameterized types.
* cannot create, catch or Throw object of parameterized types.
* Cannot overload method where the formal parameter types of each overload erase to the same raw type.

— x —

Suggested Questions / Assignments / Home works / any other
1. Define generic programming and generics. 2. Explain generic with simple type (or) single type and two type parameters. 3. Explain about generic method with example. 4. List out the restrictions on generics.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Strings

Topic(s) to becovered	Basic String class – methods for Character extraction, String comparison, Searching strings, modify the string.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	understand String class & uses.	Understand
Lo2.	explain various methods of String class.	understand
Lo3	use the methods in programming.	Apply.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

String:

String is an object that represents sequence of character values. An array of characters also works as a string. The java.lang.String class is used.

Ex:

```
Char [] ch = { 'J', 'A', 'V', 'A' };
String S = new String (ch);
```

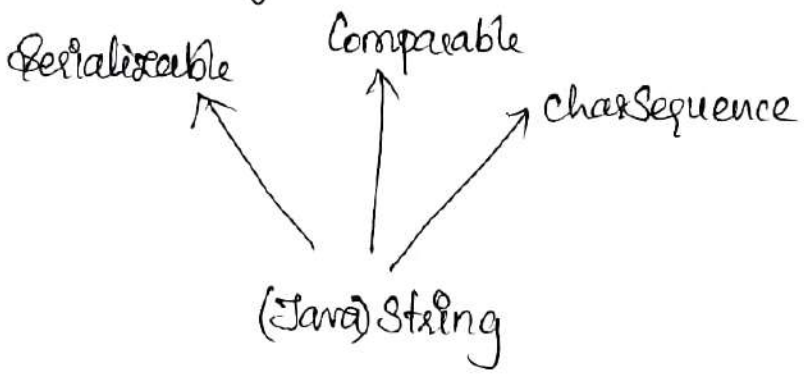
(or)

```
String S = "Java";
```

List of operations provided by Java String class:

1. compare()
2. concat()
3. equals()
4. split()
5. length()
6. replace()
7. compareTo()
8. intern()
9. substring() etc...

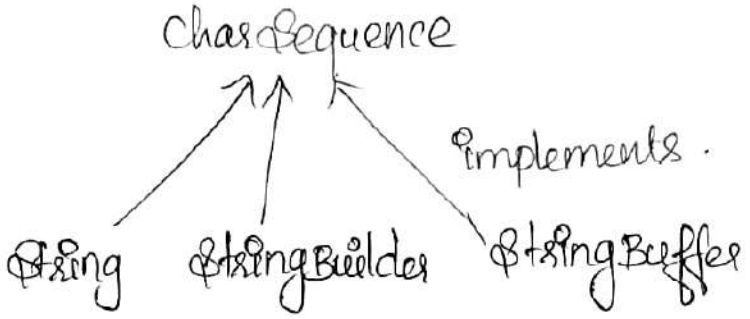
Java String class implements Serializable, Comparable, charSequence interfaces.



charSequence Interface:

This interface is used to represent the sequence of characters. We can create Strings by using three classes:

(1) String
(2) String Buffer
(3) String Builder



* The String is immutable in java. It means the value cannot be changed. whenever we change any String a new instance is created.

* To create mutable / changeable Strings, use StringBuilder (or) StringBuffer.

Creating String object: (or) creating String:

1. By String literal
2. By 'new' keyword.

String Literal:

* Java String is created by using double quotes.

Ex: String str = "Welcome";

* need:

To make Java more memory efficient (because, no new objects are created if it exists already in the String constant collection).

Using new keyword:

String Str = new String("welcome");

Explanation:

(i) new String object is created in heap.

(ii) literal "Welcome" is placed in String constant collection.

(iii) The variable Str refers to the String in heap.

Java String example 1:

Constructing (or) Creating one String from another String.

```
class example {
    public static void main (String args[])
    {
        char c[] = {'J', 'A', 'V', 'A'};
        String S1 = new String(c); // creating string from char array.
        String S2 = new String(S1); // string copy. S2 ← S1.
        System.out.println(S1);
        System.out.println(S2);
    }
}
```

O/P:

```
JAVA
JAVA
```

String Example 2:

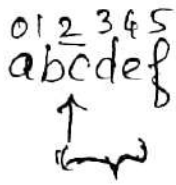
Creating a String using char array substring.

```
int startIndex = 2, numchars = 3;
char chars[] = {'a', 'b', 'c', 'd', 'e', 'f'};

String s1 = new String(chars);
String s2 = new String(chars, startIndex, numchars)
(or) String s2 = new String(chars, 2, 3);
System.out.println("s1" + s1);
System.out.println("s2" + s2);
}
```

O/p:

s1 abcdef

s2 cde



Java String class methods (or) String operations (or) String methods:

(1) charAt()
(2) length()
(3) format()
(4) substring()
(5) contains()
(6) join.
(7) equals()
(8) isEmpty()
(9) concat()
(10) replace()
(11) Split()
(12) intern()
(13) indexOf()

etc.

Character Extraction:

* String class provides number of ways to extract characters from given string. String uses the index/ offset concept similar to arrays. String index begin at 0.

character extraction methods / functions:

1. charAt()
2. getChars()
3. getBytes()
4. toCharArray()

1. **charAt():** To extract a single character from a string. It returns the character, present at particular index.

Syntax: `char charAt(int location)`

↑ index of the character to be extracted

Ex:

```
String Name = "Ajithkumar";
char ch = Name.charAt(5);
System.out.println("Index 5 has "+ch);
```

O/P:

```
Index 5 has k
```

If the specified index not available, it throws StringOutOfBoundsException.

2. getChars(): (used to get substring)

If you want to extract more than one character at a time, getChars() method is used.

Syntax:

```
void getChars(int start, int end, char target[],
                                  int targetStart);
```

start → index of the beginning of substring.
end → index of end of the substring.
target → ~~input~~ output string.

Ex:

```
String s = "This is a string method";
int start = 10;
int end = 14;
char target[] = new char[end-start];
s.getchars(start, end, target, 0);
}
}
```

O/p:

strin

---

3. getBytes() — Alternative to getChars() that stores the characters in an array of bytes. Used to ~~be~~ in character to byte conversions.

→

Syntax:

byte[] getBytes()

This method is useful when we export a String into an environment which not supports 16 bit Unicode characters.

Ex: Internet protocols & text file formats uses 8bit ASCII for all text interchange.

Example program:

```
String S1 = "ABCDEFG";
byte[] ass = S1.getBytes();
for (int i=0; i<ass.length; i++)
    System.out.println (ass[i]);
```

output:

```
65
66
67
68
69
70
71
```

4. toCharArray():- To convert all the characters in a String object into a character array. - It returns array of characters for the entire String.

Syntax:

char[] toCharArray()

Example:

```
String name = "Rajini";
char list[] = name.toCharArray();
for (int i=0; i < list.length; i++)
    System.out.println(list[i]);
```

O/P:

```
R
a
j
i
n
i
```

String Comparision:

String class has methods to compare Strings (or) SubStrings within Strings.

→

methods / functions for String comparision:

1. equals()
2. equalsIgnoreCase()
3. regionMatches()
4. startsWith()
5. endsWith()
6. equals() and ==
7. compareTo()

(1) equals():

It is used to compare two Strings. ~~and and are equal or not~~ It is used to check whether the two Strings are equal, or not.

If all the characters from the two Strings are same, it returns true.

If any of the characters is different, it returns false

Syntax: boolean equals(String str)

Example1:

```
String S1 = "Hello";
String S2 = "Hello";
if (S1.equals(S2))
        System.out.println("Same");
else    System.out.println("not Same");
```

Hello = Hello.

O/P: Same.

Example 2:

HELLO ≠ Hello.

```
String s1 = "HELLO";
String s2 = "Hello";
System.out.println(s1.equals(s2));
```

O/P: false.

2. equalsIgnoreCase(): It compares two strings, ignoring lower case and upper case differences. It will not consider lower (or) upper case.

Syntax: boolean equalsIgnoreCase(String str);

Example:

```
String s1 = "HELLO";
String s2 = "Hello";
System.out.println(s1.equalsIgnoreCase(s2));
```

O/P: True.

3. regionMatches(): This method is used to check/test if two String regions are equal.

This method has two variants.

1. Case sensitive.
2. Ignores Case ~~sens~~

Syntax:

1. boolean regionMatches (int toffset, String other,
   int ooffset, int len);

2. boolean regionMatches (boolean ignoreCase, int toffset,
   String other, int ooffset, int len);

ignoreCase → if 'true', case is ignored for comparison

toffset → the starting offset of the subregion in this string.

other → the string argument being compared

ooffset → the starting offset of the subregion in the string argument.

len → the number of characters to compare

Return value:

* if the substring is present / match occurs, answer is true.

*

Example:

```
String str1 = "HELLO";
String str2 = "LLO";
String str3 = "Hello";
System.out.println("Comparing String1 and String2");
System.out.println(str1.regionMatches(2, str2, 0, 5));
System.out.println(str1.regionMatches(2, str2, 0, 3));
```

Example:

```
String s1 = " Robot";
String s2 = " Enthiran";
System.out.println(s1.startsWith("Ro"));
System.out.println(s2.startsWith("Ro"));
```

O/P:

true
false.

---

endsWith():- endsWith() method checks whether the String ending with specified character(s)/suffix.

Syntax:

```
boolean endsWith(String chars)
```

↑ Suffix / character(s) to test

Example:

```
String s1 = "Robot";
String s2 = "Enthiran";
System.out.println(s1.endsWith("bot"));
System.out.println(s2.endsWith("ran"));
System.out.println(s2.endsWith("t"));
```

→

o/p:

true
true
false.

---

equals() Vs == :

* equals and == operator perform different jobs.

(*) equals() method compares the characters inside string object.

(*) == operator compares two object references, whether they refer to the same instance.

Example:

```
Class Sample
{ public static void main (String args[])
String S1 = "Hello";
String S2 = new String (S1); // "Hello"
System.out.println ( S1.equals (S2)); // true
System.out.println ((S1 == S2)); // false.
}}
```

o/p:

true // S1 Hello, S2 Hello. equals ✓
false.

Hello (S1) ≠ Hello (S2) ✗

Explanation:

S1 ≠ S2, because reference to these string objects are differing even though they have same characters.

4. CompareTo()

* It compares the given strings lexicographically or in dictionary order.

* It can return, positive, negative (or) zero value.

if $s1 > s2$, it returns positive number

if $s1 < s2$, it returns negative numbers.

if $s1 == s2$, it returns 0.

value	meaning
less than zero (or) -ve	Invoking string is less than str.
Greater than zero (or) +ve	Invoking string is greater than str.
zero (or) 0	Two ~~Invoking~~ strings ~~is~~ are equal.

* Need of compareTo() method:

1. It is not enough to know whether two strings are identical.
2. For sorting applications, we need to know less than, greater than and equal.
3. CompareTo() method provides this facility.

Example:

lessthan: apple < banana. [as per dictionary order]

greaterthan: pineapple > lemon. ["]

equal: cherry = cherry.

Example Program: // assrange the strings in alphabetical order.

```
String arr[] = { "Now", "is", "the", "time",
"for", "all", "good", "men", "to", "come", "to", "the",
"aid", "aid", "of", "their", "country" };

for (int i = 0; i < arr.length; i++)
{
    for (int j=0+1; j < arr.length; j++)
    {
        if (arr[i].compareTo (arr[j])< 0)
        {
            String temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
}
```

// to display output.

```
for ( int i = 0; i < ass.length; i++)
    System.out.println( ass[i]);
```

CompareToIgnoreCase ( ):

Syntax: int compareToIgnoreCase (String str)

This method does all the work of compareTo( ) and ignores case.

Special String Operations:

1. String literals
2. String concatenation (or) +
3. String Conversion and toString.

String concatenation:

(*) '+' operator concatenates two Strings and produces a String object.

(*) Example1:

```
String s = "My name is" + "Billa";
System.out.println ( s);
```

O/P: My name is Billa.

* Example 2:

```
String s1 = "My name is";
String s2 = "Billa";
String s3 = s1 + s2;
System.out.println(" s3);
```

O/P: My name is Billa.

* Example 3:

```
String s1 = "My age is";
String s2 = s1 + "9";
```

O/P: My age is 9.

toString() method:

* It is used to represent any object as String.
  (or) It gives string representation of an object.
* To print any object, java compiler internally invokes the toString() method.
* toString() method can be overridden. It means the definition of this method can be changed as per program requirement.

* also we can provide our own representation for the toString() method.

* To implement toString(), simply return a String object that contains the human-readable string that describes an object of your class.

Example program:

```
class Box {
    int width;
    int depth;
    int height;
    Box(int width, int height, int depth)
    {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }

    public String toString() {
        return "Dimensions of the box are" +
               width + "," + height + "," +
               depth + ".";
    }
```

→

```
class demo {
public static void main (String args[])
{
        Box b = new Box (10, 12, 14);
        String s = "Box b:" + b ; // Concatenating box object.
        System.out.println(b); // Invokes toString()
        System.out.println(s); "
}}
```

O/P:

Dimensions of the box are 10, 12, 14.

Box b : Dimensions of the box are 10, 12, 4.

Searching Strings:

The String class provides two methods that allow you to search a string for a specified character (or) substring.

(1) indexOf(): Searches the first occurrence of a character (or) Sub String.

(2) lastIndexOf(): Searches the last occurrence of the character (or) substring.

* These methods can return -1, if the character (or) substring not found, else return the index position.

* Syntax:

(1) int indexOf(char ch);

→ To search the first occurrence of the character

(2) int indexOf(String str);

→ To search the first occurence of the substring str and returns its index position.

(3) int indexOf(char ch, startIndex);

→ To search the ~~first~~ occurrence of the given character from the startIndex.

~~(4) int Index~~

(4) int lastIndexOf(char ch);

→ To search the last occurrence of the characters.

(5) int lastIndexOf(String str)

→ To search the last occurrence of the substring str and gives the index position.

(6) int lastIndexOf(String str, startIndex);

→ To search the occurrence of the given ~~character~~ substring from the startIndex.

Example:

```
String s = "Welcome to java world";
System.out.println(s.indexOf('j'));
System.out.println(s.indexOf("to"));
System.out.println(s.indexOf('j', 15));
System.out.println(s.lastIndexOf('l'));
System.out.println(s.lastIndexOf("java");
System.out.println(s.lastIndexOf('l', 15));
System.out.println(s.lastIndexOf("java", 15));
System.out.println(s.indexOf("java", 15));
```

Modifying a String:

1. substring() → Extracting a substring from original string
2. concat() → joining two (or more) strings.
3. replace() → replaces a char/string with another char/string
4. trim() → Gives the original string after removing leading and trailing white spaces.

Data Conversion:

1. valueOf() → converts the data from its internal format to human readable (String) format.

change case():

1. toLowerCase() → Converts all the characters to lowercase
2. toUpperCase() → converts all the characters to uppercase

LectureNo. String buffer class.

Topic(s) to be covered	constructors – length(), capacity(), charAt(), getChars() – append(), insert(), reverse(), delete(), replace(), substring.

	Lecture Outcome (LO)	
	At the end of this lecture, students will be able to	Bloom's Level
LO1	Know the use of StringBuffer	understand.
LO2	Compare the new methods with String class	understand.
LO3.	discuss about the new methods in StringBuffer	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

StringBuffer:

* It is the peer class of String, provides more functionalities.
* normally, a string represents fixed-length, immutable character sequences.
* StringBuffer makes the String, mutable. It means growable and writable character sequences.
* StringBuffer used to create modifiable (or) mutable strings.

StringBuffer Constructors: to create strings.

(1) StringBuffer() → default constructor, 16 characters size
(2) StringBuffer(int size) → size specifies number of characters
(3) StringBuffer(String str) → It creates string using str and allocates 16 more spaces.
(4) StringBuffer(CharSequence chars): → It creates string object using the data available in chars.

length() and capacity():

* length of the StringBuffer can be found using length()
* The total allocated capacity can be found through the capacity() method.

Syntax:

```
int length()
int capacity()
```

* Example:

```
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer" + sb);
System.out.println("length" + sb.length());
System.out.println("capacity" + sb.capacity());
```

* O/P:

```
buffer  Hello
length  5
capacity 21.
```

ensureCapacity(): void ensureCapacity(int capacity);

* It is useful to know in advance when we need to append a large numbers of small strings to a StringBuffer.

* when we want to preallocate room for a certain numbers of characters, after a StringBuffer has created, we will use ensureCapacity() to set the size of the buffer.

setLength(): void setLength(int len);
↳ should be non negative

* To set the length of the String within a StringBuffer.

* It is indirectly used to increase (or) decrease the length of the string.

* If size increase happens, null characters are added at the end.

* If size decrease happens, the characters after the specified length are lost.

→

Example:

```
StringBuffer sb = new String ("Hello");
System.out.println(sb);  // Hello
System.out.println(sb.length()); // 5   len=
sb.setLength(2);  // len=2.
System.out.println(sb); // He.
```

CharAt() and setCharAt():

* charAt() method returns the value of the character present at given index location.

Syntax: char charAt(int index)

Example:
```
String str = "Java";
System.out.println(str.charAt(3));
```
O/P: v

* setCharAt(): set a value of character at a particular index.

Syntax: void setCharAt(int index, char ch)

Example:
```
String str = "Java";
str.setCharAt(3,'b'); S.O.P(str);
```
O/P: Jaba

insert(): It inserts one string into another.

Syntax:

```
StringBuffer insert (int index, String str)
StringBuffer insert ( int index, char ch)
StringBuffer insert ( int index, Object obj)
```

Example:

```
StringBuffer sb = "Welcome Java";
sb.insert (8, "to");
System.out.println (sb);
```

O/P:

Welcome to Java.

reverse():

* this method reverses the characters present in StringBuffer.

* Syntax:

```
StringBuffer sb = "APPLE";
sb.reverse ();
System.out.println (sb);
```

* O/P: ELPPA

Use: This reverse() method can be used in palindrome checking operation.

getChars(): * to copy a substring of a StringBuffer into an array, getChars() used.

* It copies the set of characters from start to end position, into target array.

Syntax: void getChars(int start, int end, char target[], int targetStart);

append():

* This method joins (or) concatenates any data to the end of the StringBuffer.

Syntax:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj
```

Example:

```
String s = "My age is";
int a = 52;
StringBuffer sb = new StringBuffer(40);
s = sb.append(a);
System.out.println(s);
```

O/P: My age is 52.

delete() and deleteCharAt():

* To delete character(s) within a StringBuffer ~~by using~~ delete() method is used.
* Syntax:

  StringBuffer delete(int start, int end);
* Example:

```
StringBuffer sb = "Test";
sb.delete(1,3);
S.o.p(sb);
```

* O/P: T

* deleteCharAt() method delete a character at a particular location (or) index.

Syntax:

StringBuffer ~~sb~~ deleteCharAt(int indexloc);

Example:

```
StringBuffer sb = "Test";
sb.deleteCharAt(0);
S.o.p(sb);  // ans: est.
sb.deleteCharAt(1); // ans: et.
```

'40);

replace(): Replace a set of characters, (or) Substring with another set of characters/substring within StringBuffer.

Syntax:

StringBuffer replace (int start, int end, String str)

The substring being replaced is specified by start and end-1

Example:

```
StringBuffer sb = new StringBuffer ("My dream World");
sb.replace (3, 7, "joy");
System.out.println (sb);
```

O/p: My joy World.

SubString(): To obtain a portion of a StringBuffer by using SubString() method.

Syntax:

(i) String subString (int start)

(ii) String substring (int start, int end)

(i) returns a substring from starting position to end of the StringBuffer.

(ii) returns a substring from start to end-1 position.

Example:

```
StringBuffer sb = "My dream World";
sb.substring(7);  // m World.
sb.substring(7, 10);  // m Wo.
```

Additional methods:

(1) indexOf(String str): gives the index of first occurrence of str in stringBuffer.

2. lastIndexOf(String str): gives the index position of last occurrence of str in StringBuffer.

Example:

```
StringBuffer sb = new StringBuffer("one two one");
int i = sb.indexOf("one");
System.out.println("first occurrence:" + i);
i = sb.lastIndexOf("one");
System.out.println("last occurrence:" + i);
```

O/p:

first occurrence: 0

last occurrence: 8.

Unit-V

LectureNo. Java Event Basics.

Topic(s) to becovered	Event Basics – Event class – Event handler interface – Event dispatch chain – Button control – Actionevent – Example Program.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	understand to event basics	Understand
LO2	know event class and interface	Understand
LO3	explain the button control usage	Understand.
LO4	explain about action event	Understand

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate / Discuss / Peer to Peer Learning / Quiz / Role Play / Any other

Lecture Notes

JavaFx Event Basics:

* JavaFx uses the delegation event model appeoach to event handling.
* A source (or) control that generates an event and sends it to one (or) more listeners which handles the event.
* To receive an event, the handler for the event must be first registered with the event source.

* when the event occurs, the handler is called. Handler must respond to the event and return. Thus the user interface element delegates the processing of an event to a seperate event handler.

* Event class:

▲ The base class for JavaFx events is the Event class, which is packaged in javafx.event.

▲ Event inherits java.util.EventObject. which means the JavaFx events share the same basic functionality as other Java events.

▲ When the JavaFx event is generated, it is encapsulated within an Event instance.

▲ Event supports several methods to help us to manage events.

Example: Action Event : It is generated by several JavaFx controls including push button.

The EventHandler Interface:

Events are generated by implementing the EventHandler interface, which is also in javafx.event. It is a generic interface with following syntax.

`Interface EventHandler <T extends Event>`

T → specifies the type of event that the handler will handle. It defines one method called handle(), which receives the event object as an argument.

`void handle (T eventObj)`

eventObj is the event that was generated. Event-handlers are implemented through anonymous inner classes or lambda expressions. One event handler can handle events from more than one source.

Two ways of specifying a handler for event:

1. First, you can call addEventHandler() which is defined by Node class among others.

2. Second, in many cases, you can use Convenience method. It uses a prefix "setOn" and sets an event handler property.

→

The Event Dispatch chain:

* In JavaFx, the events are processed via an event dispatch chain.
* It is a path from the top element in the scene graph to the target of the event, which is the control that generated the event.
* When an event is processed, two main phases occur.

a) The event is passed from the top element down the chain to the target of the event. This is called event capturing.

b) After the target nodes processes the event, the event is passed back up the chain, thus allowing parent nodes a chance to process the event. This is called Event bubbling. Event handlers are called during event bubbling phase.

Event Filter:

* It is a special type of event handler. Event filters are called during the event capturing phase. Thus the event filter is working before calling event handler. If a filter consumes the event, the event handler will not execute.

Button control:

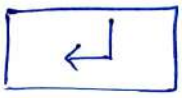
* most commonly used control is Button in all JavaFx applications.
* Button class provides push button control.

- It is available under javafx.scene.control package.
- Button inherits a fairly long list of base classes that includes ButtonBase, Labeled, Region, Control, Parent and Node.

* Button contains text, graphics or both.

ex:

Submit button (text)



(graphics) Enter button.

both. (Text & graphics)

* Button supports array of options, but default form is used widely.

Constructors:

1) Button() : default constructor, creates Button without any text.

2) Button(String str): str is the message to be displayed on button.

ex: Button("View")

Event for Button class:

* Action Event: when a button is pressed, an action event is generated. It is packaged in javafx.event.

* It defines two types of action events.

  i) ~~ACTION~~ ACTION: only one event

  ii) ALL: It refers to all action event types

* ACTION is used to register a handler for an action event via the addActionHandler() method.

* setOnAction() method is also used to register an action event handler. It is a convenience method.

Syntax:

```
setOnAction(EventHandler<ActionEvent> handler)
```

The handler to be registered.

Example Program for Button:

This program uses 2 buttons and a label. They are First button & second Button and response.

```
// handler for first button.
btnFirst.setOnAction(new EventHandler<ActionEvent>()
{
    public void handle(ActionEvent ae)
    {
        response.setText("First button Pressed");
    }
});
// handler for second button
btnSecond.setOnAction(new EventHandler
    <ActionEvent>()
    {
        public void handle(ActionEvent ae)
        {
            response.setText("Second Button Pressed");
        }
    }));
// add controls to scene.
root:getchildren().addAll(btnFirst, btnSecond,
response);
// display stage
    myStage.show();
}
}
```

Each time a button is pressed, the content of the label is set to display which button was pressed.

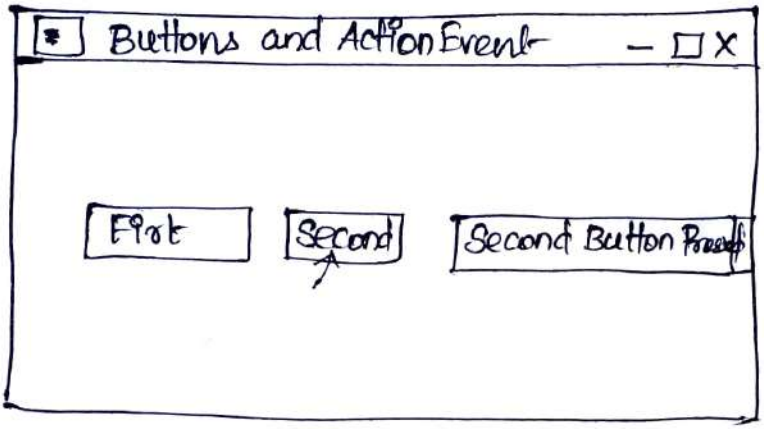
Coding:

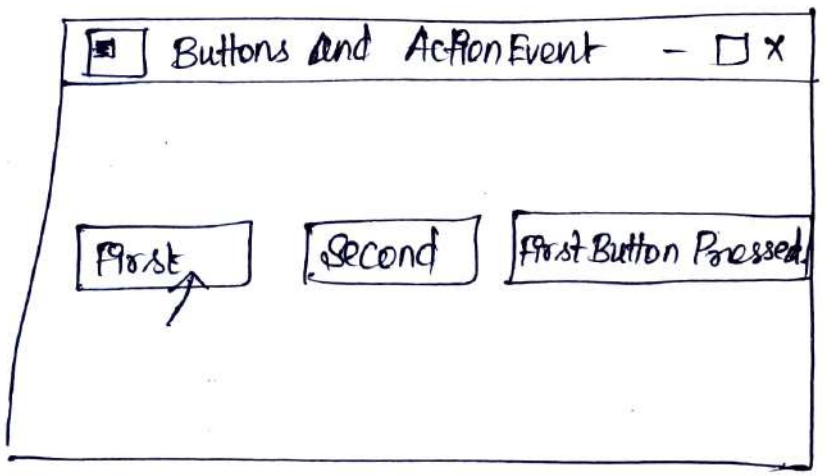
```
import javafx.applications.*;
        :
        :

public class JavaFxEventDemo extends Application
{
 Label response;

 public void start (Stage myStage)
 {
  myStage.setTitle ("Buttons and ActionEvent");

     FlowPane root = new FlowPane (10,10);
     root.setAlignment (Pos.CENTER);
     Scene myScene = new Scene (root, 300,
                                100);
     myStage.setScene (myScene);

     response = new Label ("Press/Push a Button");
     Button btnFirst = new Button ("First");
     Button btnSecond = new Button ("Second");
```

→

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

Sample Output:





Suggested Questions / Assignments / Home works / any other
1) what is event handling? 2) what is event handler? what is actionEvent? 3) what is event dispatch chain? 4) Explain about button control with a sample program.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill
2.	Introducing JavaFX 8 programming, 1st Edition.	Herbert Schildt	Oracle Press.

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Handling key and mouse events Controls.

Topic(s) to becovered	keyevents:– types– methods– Sample program. Mouse Events: types – Handless+ methods – Example.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	explain types and methods of key Event	understand
Lo2	explain the types & handlers of Mouse Events	Understand
Lo3	discuss about MouseEvent with example.	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

key events:

* when a key is typed on the keyboard, a keyEvent is generated.

* keyEvents can be handled by instances of various classes and both the Node and Scene classes define convenience methods that support keyEvents.

* when a keyevent is handled by a Node, events are received only when that node has input focus.

* keyevent is available in javafx.scene.input.

* 3 types of keyevents:

	fields
▶ a key is pressed	KEY_PRESSED
▶ a key is released	KEY_RELEASED
▶ a key is typed	KEY_TYPED.

* The fields are defined by KeyEvent as objects of EventType.

* A KeyPressed Event is generated when a key is pressed on the keyboard. Ex: SHIFT, ALT, ARROW keys.

* A key released Event is generated when a key is released.

* A key typed Event is generated when a normal character such as k, 9 or +, - ; etc is typed from keyboard.

Both the Node and scene define convenience methods that make it easier to register an event handler for

the various types of key events.

methods:

1) void setOnKeyPressed (EventHandler <? Super KeyEvent> handler)

2. void setOnkeyReleased (EventHandler <? Super KeyEvent> handler)

3. void setOnkeyTyped (EventHandler <? Super KeyEvent> handler)

handler specifies the eventhandler. when a key typed event is received, we can obtain a character using getCharacter() on the Event.

when a keyPressed event or key released event is received, we can obtain keycode associated with the event. key codes are enumeration constants defined by keyCode. and packaged under javafx.scene.input.

keycode	meaning
RIGHT	RIGHT ARROW key.
LEFT	LEFT ARROW key.
KP_RIGHT	The RIGHT ARROW key on the number pad.

KP_LEFT	The LEFT ARROW key on the number pad.
F1	F1 key.
F10	F10 key.
ALT	ALT key.
CTRL	CONTROL key.
SHIFT	SHIFT key.

getCode(): To obtain keycode of the key Pressed/released, on the keyevent.

```
KeyCode getCode();
```

Sample Program: description

This program registers key-typed and key-pressed event handlers on the program's scene. KeyTyped handler simply displays the character in a label. The keyPressed handler simply displays a character in a label. The key Pressed handler recognizes the RIGHT and LEFT arrow keys, F10 and ALT.

If the key Pressed is one of these keys, that is reported. Otherwise, no other action takes place.

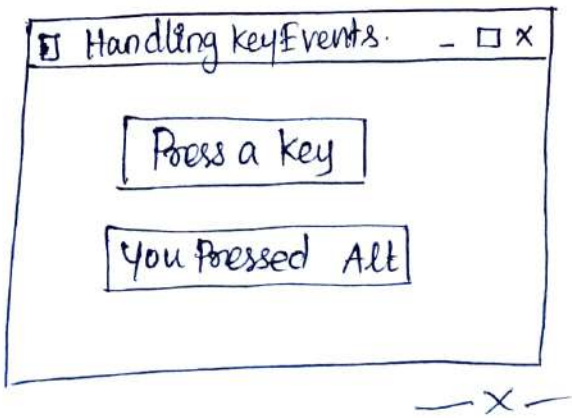
Coding:

```
import javafx.application.*;
:
:
```

```
public class keyEventDemo extends Application
{
    Label prompt;
    Label showkey;

    public void start ( Stage myStage)
    {
        myStage.setTitle ("Handling keyEvents");
        FlowPane root = new FlowPane ( orientation,
                                       VERTICAL, 0, 10);
        root.setAlignment (Pos. CENTER);
        Scene myscene = new Scene ( root, 300, 100);
        myStage.setScene (myScene);
        prompt = new Label ("Press a key");
        showkey = new Label ("");

        // handler for keyTyped event
        myscene.setOnkeyTyped (new EventHandler <keyEvent>
                               () {
            public void handle ( KeyEvent ke)
            {
                showkey.setText ("You Typed" + ke.getCharacter());
            }

        // handler for key_Pressed event
        myScene.setOnkeyPressed (new EventHandler
                                 <keyEvent> () {
```

```
public void handle (KeyEvent ke)
{
    Switch (ke.getCode())
    {
      case RIGHT: showkey.setText("you pressed
                                  Right arrow"); break;
      case LEFT: showkey.setText("you presses F10");
                 break;
      Case ALT: showkey.setText("you pressed ALT");
                break;
    }
}
});
root.getChildren().addAll(prompt, showkey);
myStage.show();
}
public static void main (String [] args)
{ launch(args);
}
}
```

Out Put:



—x—

Mouse Events:

* mouse events represented by MouseEvent class
* Mouse event class is under 'javafx.scene.input' package.
* MouseEvent can be handled by instances of various classes, and the Node and Scene classes define the convenience methods for mouse events.
* when a mouse event handled by Node, mouse events are received only when node has input focus.
* when a mouse event handled by Scene, mouse events are received only when ~~no~~ scene has i/p focus.

* different types of mouse events:
  (1) when a mouse is moved.
  (2) when a button is clicked (or) released
  (3) when a mouse enters (or) exits an element
  (4) when a mouse is dragged

A numbers of EventType objects are defined by MouseEvent.
ex: MOUSE_CLICKED & MOUSE_MOVED, MOUSE_RELEASED, MOUSE_ENTERED, etc.

Definition of MouseEvent:

* It defines registration Event handlers for MouseEvent:

1) setOnMouseClicked (EventHandler <? Super MouseEvent> handler)

2) setOnMouseMoved (")

Here handler is the event handler.

Methods of MouseEvent: It defines a number of methods to help us determine what has occurred.

1) getButton ( ): when the mouse is clicked, you can find out which button was used, by calling getButton ( ) method.

Syntax: final MouseButton getButton ( )

	return values
Left button was clicked	→ MouseButton.PRIMARY
Right Button was clicked	→ MouseButton.SECONDARY
Middle Button was clicked	→ MouseButton.MIDDLE.
If no Button was clicked	→ MouseButton.NONE

→

getClickCount(): To obtain detail about numbers of times a mouse button has been clicked.

	return value
single click	→ 1
double click	→ 2.
no click	→ 0.

getSceneX() & getSceneY(): To obtain the location of the mouse in the screen (scene), when event occurs. (ie) (x,y) position.

consider the origin (0,0) is the upper left corner.

Syntax:

```
final double getSceneX()
final double getSceneY()
```

example: mouse is clicked @ (10,5) position in the scene (x = 10, y = 5)

returns

getSceneX() → 10.

getSceneY() → 5

→

Example program for Mouse Event:

This program reports when the mouse is clicked, what button was clicked and how many times it was clicked. It also displays the location of the mouse relative to the scene. (x,y).

Coding:

```
import javafx.application.*;
        :
public class MouseEventDemo extends Application
{
  Label showEvent;
  Label showLocation;

  public void start(Stage myStage)
  {
     myStage.setTitle("Handling Mouse Events");

     FlowPane rootNode = new FlowPane(
           Orientation.VERTICAL, 0, 10);
     rootNode.setAlignment(Pos.CENTER);
     Scene myScene = new Scene(rootNode, 300, 100);
     myStage.setScene(myScene);
```

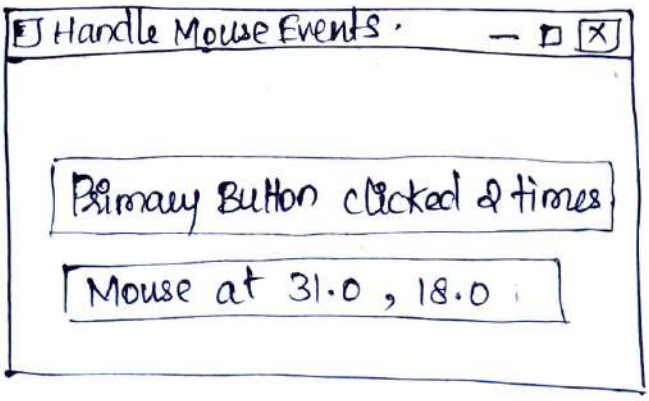
```
Scene myScene = new Scene (rootNode, 300, 100);
myStage.setScene (myScene);
ShowEvent = new Label ("Use the Mouse");
Show Location = new label ("");

// Handler for mouse event on the scene
myScene. setOnMouseClicked (new EventHandler <
    MouseEvent>() {
        public void handle (MouseEvent me) {
            int clickcount = me.getClickCount();
            String times = "time";
            if (clickcount > 1)
                times += "s";
            Switch (me.getButton())
            {
            Case PRIMARY:
                ShowEvent.setText ("Primary button
clicked" + clickcount + " " + times); break;
case MIDDLE:
    ShowEvent.setText ("Middle Button Clicked" + clickcount
    + " " + times); break;
Case SECONDARY:
    ShowEvent.setText ("Secondary button clicked" +
            clickcount + " " + times); break;
} } };
```

→

// Handler for mouse move event on the scene.

```
myScene.setOnMouseMoved(new EventHandler
    <MouseEvent>() {
        public void handle(MouseEvent me)
    {
    showLocation.setText("Mouse at" +
            me.getSceneX() + "  " +
            me.getSceneY()};
    } };

    rootNode.getChildren().addAll(showEvent,
                                  showLocation);
    myStage.show();
  }
}

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

output:



Suggested Questions / Assignments / Home works / any other
1) what is key event? 2) what is mouse event? 3) what are the different types of key event is? 4) what are the types of mouse events? 5) explain the handling of key and mouse events.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill
2.	Introducing JavaFx 8 Programming, 1st Edition	"	Oracle Press.

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Controls in JavaFx.

Topic(s) to becovered	Javafx controls – CheckBox: Constructor, methods, event, example; – ToggleButton:– methods, event, example – RadioButton: Constructor, event – example.

	Lecture Outcome (LO)	Bloom's Level
	At the end of this lecture, students will be able to	
LO1	List out various Javafx controls	understand
LO2	understand the working of toggle button	understand
LO3	know the methods and event in checkbox	understand
LO4	apply the concept of Radio button in programs	Apply

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

Javafx Controls:

1) CheckBox
2) ToggleButton
3) RadioButton
4) ListView
5) ComboBox
6) ChoiceBox.
7) TextControls.

} Available in javafx.scene.control package

CheckBox:

* CheckBox encapsulates the functionality of the CheckBox.
* Superclass: ButtonBase
* CheckBox refers to special type of Button. It supports 3 states.
    1. checked
    2. unchecked
    3. Intermediate (undefined).

* Intermediate state: It enables the checkbox to better fit.
  Ex: It can indicate that the state of some option has not been set (or) that the ~~state of some~~ option is not relevant to a specific situation.

* Constructors:
    1) CheckBox(): default constructor, creates empty checkbox.
    2. CheckBox(String str): It creates a checkbox that has the text specified by str as Label.

* ActionEvent: CheckBox generates an action event when it is clicked. you can set the action event handler on a CheckBox by calling SetOnAction().

* isSelected(): To obtain the state of the checkbox

Syntax: final boolean isSelected()

- true: Check box is selected
- false: check box is not selected.

* Example Program:

- Checkbox let the user to select various input devices options. Each time a checkbox is clicked, an action event is generated.

Coding:

```
import javafx.applications.*;
    :
public class CheckBoxDemo extends Application
{
    CheckBox    keyboard;
    CheckBox    mouse;
    CheckBox    scanner;

    Label   response;
    Label   selected;

    String inputDevices = " ";

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

```
public void start ( Stage myStage)
{
    myStage . setTitle (" Demonstration of CheckBox"
                        ~~CheckBox Demo~~");
    FlowPane root = new FlowPane ( Orientation . VERTICAL ,
                                   0, 10);
    root. setAlignment ( Pos . CENTER_LEFT);
    root . setPadding (new Insets (0,0,0,10);
    Scene myScene = new Scene ( root , 300, 180);
    myStage. setScene (myScene);
    Label heading = new Label ("Select Input Devices");
    response = new Label (" No devices selected");
    selected = new Label ("Support devices : <none>");
```

```
keyboard = new CheckBox ("keyBoard ");
mouse = new CheckBox(" Mouse");
scanner = new checkBox(" Scanner");
```

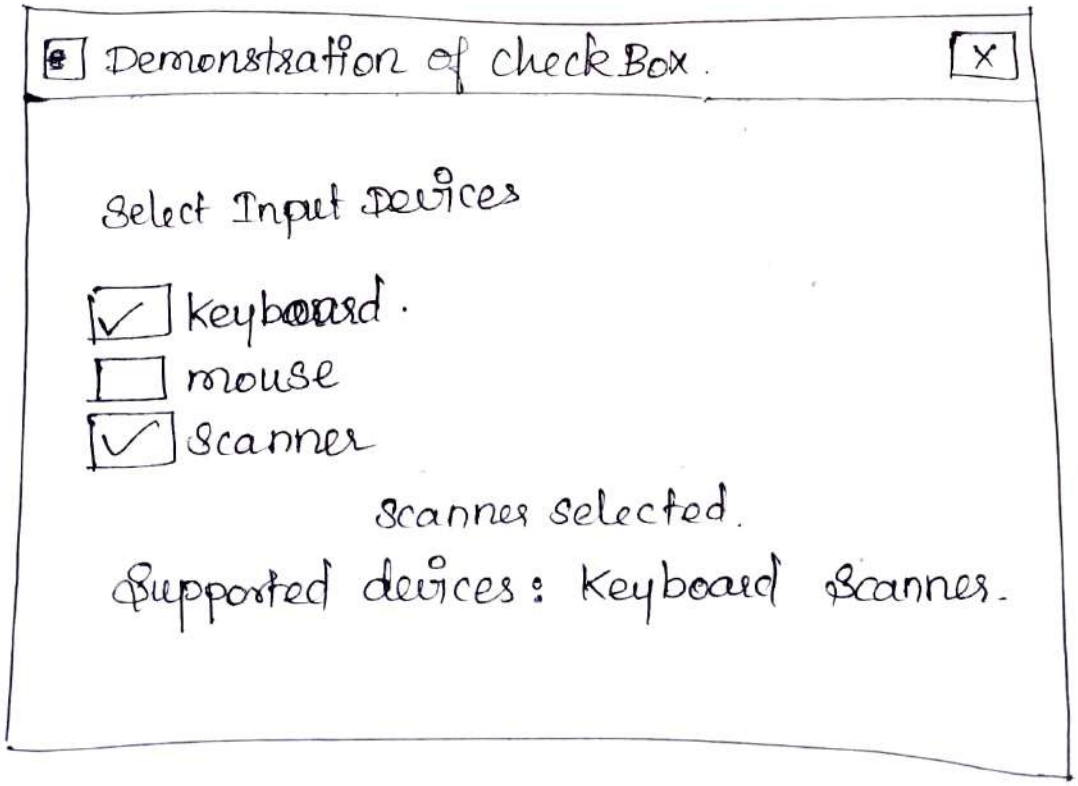
```
keyboard . setOnAction( new EventHandler <ActionEvent>
    () {
        public void handle ( ActionEvent ae)
        {   if (keyboard . isSelected ())
                response . setText ( "keyboard selected")
            else
                response . setText (" keyboard cleared")
        ShowAll ();  } } );
```

```
mouse.setOnAction(new EventHandler<ActionEvent>()
{
    public void handle(ActionEvent ae)
    {
        if(mouse.isSelected())
            response.setText("Mouse selected");
        else
            response.setText("Mouse Cleared");
        showAll();
    }
});

scanner.setOnAction(new EventHandler<ActionEvent>
                    ()
{
    public void handle(ActionEvent ae)
    {
        if(scanner.isSelected())
            response.setText("Scanner selected");
        else
            response.setText("Scanner cleared");
        showAll();
    }
});

root.getChildren().addAll(heading, keyboard, mouse,
    scanner, response, selected);
    myStage.show();
}
```

→

```
void showAll()
{
    Input Devices = "";
    if (keyboard.isSelected()) InputDevices += "KeyBoard";
    if (mouse.isSelected()) InputDevices += "mouse";
    if (scanner.isSelected()) InputDevices += "scanner";
    Selected.setText("Supported devices:" + InputDevices);
}
}
```

Sample Output:



Explanation:

steps:

1. Start the javafx application by calling launch.
   launch (args).
2. override the start method.
3. give the stage title. using setTitle ()
4. use a flowpane for root and set verticalgap 10.
5. center the ~~nodes~~ controls vertically, left align them horizontally. using setAlignment (Pos. CENTER_LEFT);
6. Set a padding value of 10 on the left for the flow pane using setPadding ()
7. create a scene by creating object of Scene. and set the size (300, 180)
8. Set the scene on the stage. using setScene.
9. create a label 'heading' to report the state of the selected checkbox.
10. create a label 'selected' to report all selected input devices.
11. create 3 Checkboxes keyboard, mouse, scanner using CheckBox () Constructor.
12. Handle action events for each checkbox.

→

13. add all contsols to the scene graph using
root. getChildren. addAll ( )

14. show the stage and its scene using
myStage. show( ).

15. update and show the input devices list using
showAll ( ).

— × —

Toggle Button:

* Javafx Provides a variation on the push button called a toggle button.

* It is on/off type button.

* It has 2 states < Pressed
Released.

When you press togglebutton, it stays pressed state rather than popping back up as a regular push button.

when you press togglebutton second time, it releases (or) pop ~~back~~ up.

* It is base class is ButtonBase. and Implements Toggle interface.

→

* 2 constructors:

1) ToggleButton() → empty ToggleButton

2) ToggleButton(String str) :→ ToggleButton with text displayed on button.

* ActionEvent:

- It generates ActionEvent when it is clicked.
- when the button is pressed, the option is selected.
- when the button is released, option deselected.
- To check whether the button is selected (or) not use isSelected() function.

> true → button selected pressed.
> false → button released.

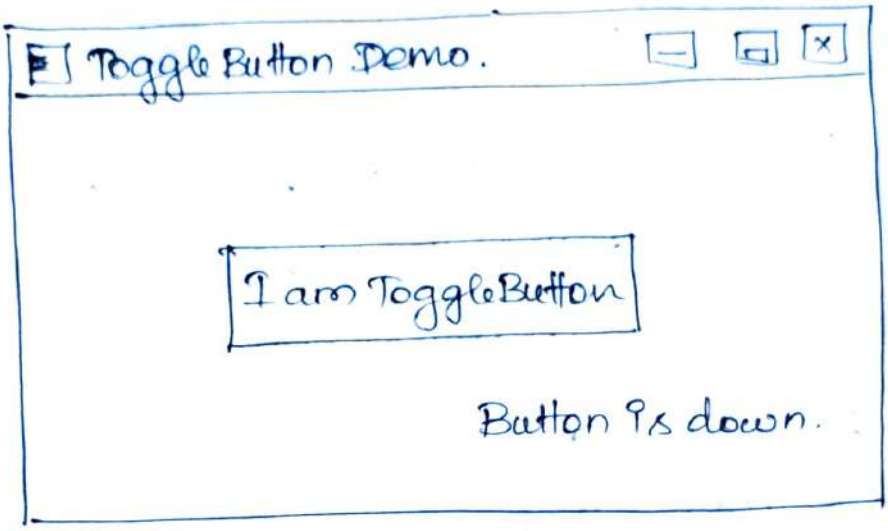
* Example:

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
```

```
Public class ToggleButtonDemo extends Application.
{
    ToggleButton toggle;
    Label release;
    public static void main(String args[])
    {
        launch(args);
    }
    public void start(Stage myStage)
    {
        myStage.setTitle("ToggleButton Demo");
        Flowpane root = new Flowpane(10,10);
        root.setAlignment(Pos.CENTER);
        Scene myScene = new Scene(root, 200, 200);
        myStage.setScene(myscene);
        response = new Label("Push the button");
        toggle = new ToggleButton("I am ToggleButton");
        toggle.setOnAction(new EventHandler<
            ActionEvent>(){
            public void handle(ActionEvent ae)
            {
                if(toggle.isSelected())
                    response.setText("Button is down");
                else
                    response.setText("Button is up");
            }
        });
```

```
root.getChildren().addAll(toggle, response);
myStage.show();
}}
```

Output:



Radio Button.

* They used to manage a group of mutually exclusive buttons.
* It means only one button can be selected at any one time.
* Base class : ButtonBase
  Interface : Toggle Interface.
* Radio buttons are used when the user must select only one option among several options.

* Constructors:

1) RadioButton ( ): Empty radio button, default Constructor

2) RadioButton (String str): Radio button created with a text or label.

* The Radio buttons must be grouped, so that only one of the buttons in the group can be selected at any time.

* Buttongroup Class: ToggleGroup ( ).

* To add Radiobuttons to a toggle group by calling setToggleGroup ( ) method.

Syntax: `setToggleGroup (ToggleGroup tg)`

* setSelected ( ): one of the button is selected when the group is first displayed in the screen.

Syntax: `void setSelected ( boolean state)`

State: true → button is selected
false → button is not selected.

* fire ( ): If the button was not selected previously, then fire ( ) is called and action event is generated.

to select the button.

→

Example:

```
import javafx.application.*;
        :
public class RadioButtonDemo extends Application
{
    Label response, prompt;
    RadioButton keyboard, mouse, scanner;
    ToggleGroup tg;

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

    public void start (Stage myStage)
    {
        myStage.setTitle(" Radio button Demo");
        Flowpane root = new Flowpane (
    Orientation.VERTICAL, 0, 10);
        root.setAlignment (Pos.CENTER_LEFT);
        root.setPadding (new Insets (0, 0, 0, 10));
      Scene myscene = new Scene (myScene);
      prompt.newLabel ("Select primary I/p device")
      response = newLabel (" ");
```

```
keyboard = new RadioButton ("key Board");
mouse = new RadioButton ("mouse");
scanner = new RadioButton ("scanner");
```
// Creating Radio buttons.

```
tg = new ToggleGroup();
```

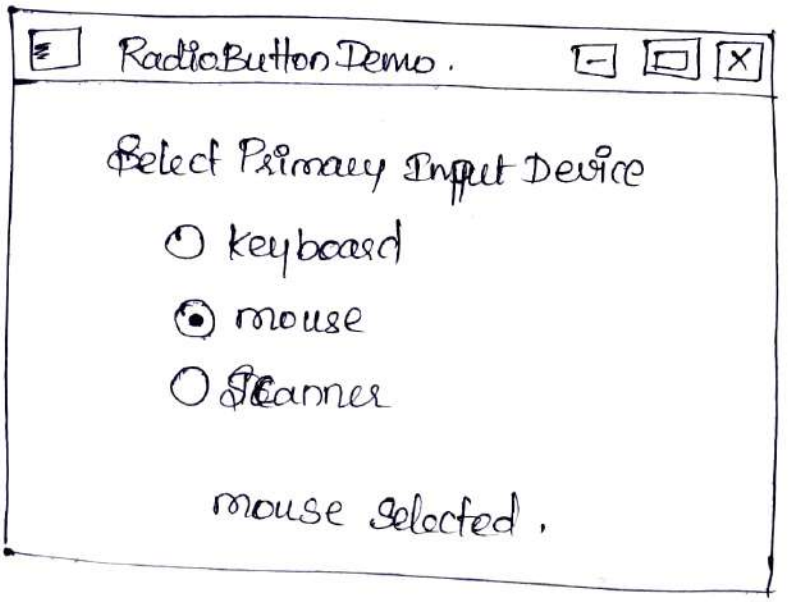
```
keyboard.setToggleGroup(tg);
mouse.setToggleGroup(tg);
Scanner.setToggleGroup(tg);
```
// grouping the Radio buttons

// Action Events.

```
keyboard.setOnAction (new EventHandler <ActionEvent>
                      () {
    public void handle (ActionEvent ae)
    {
        response.setText ("KeyBoard selected");
    } });

mouse.setOnAction (.....                    () {
        public void handle (ActionEvent ae)
    {   response.setText ("mouse selected");

    } });

Scanner.setOnAction (....) () {
        public void handle (ActionEvent ae)
        {
            response.setText ("Scanner selected");
        } });
```

```
// Fire the event for first selection.
// so radio button is selected and actionevent caused.
        keyboard.fire();

root.getChildren.addAll(prompt, keyboard, mouse,
scanner, response);
        myStage.show();
}}
```

Output:



—x—

LectureNo. ListView, ComboBox and ChoiceBox.

Topic(s) to becovered	ListView: constructors, methods, example ComboBox: constructors, methods, exaupple ChoiceBox: Constructors, methods, example.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	explain the methods, constructors of ListView.	Understand
Lo2	define Combobox and Choice box	Understand.
Lo3	use combobox, choice & list view in programs.	Apply.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate /Discuss /Peer to Peer Learning /Quiz /Role Play / Any other

Lecture Notes

ListView:

* ListView are controls that display a list of entries from which we can select one (or) more.
* no. of. entries in the list can increase/ decrease during execution.
* It make efficient use of limited screen space.
* It is alternative to other selection Controls.

Syntax:

```
Class ListView <T>
```

ListView is the generic class.

T → type of entries stored in listview.

Constructors of ListView:

1) ListView ( ) → empty listview is created with default size.
2) ListView (ObservableList <T> list)
   - → list of observable objects.
   - → inherits java.util.List.
   - → packaged under javafx.Collections ();

Methods of ListView:

1) setPrefHeight (double height): to set preferred height.
2) setPrefWidth (double width): to set preferred width.
3) setPrefSize (double width, height): to set both dimensions at the same time.

Other details:

* If listview exceeds the size, it will automatically provide scroll bars
* If length exceeds, horizontal scroll bar will be added.

* ListView allows only one item in the list to be selected at any one time. (Single selection model)

* To select multiple items, change the selection mode.

* Two basic ways of using ListView:

1) Ignore events generated by the list and simply obtain the selection in the list.

2). monitor the list for changes by registering a change listener.

* Methods in ListView:

1) getSelectionModel(): To listen the change of events, obtain the selection model used by the listview. It returns a reference to the model.

Syntax:

final MultipleSelectionModel <T> getSelectionModel()

→ model for multiple selections.
→ multiple selection mode should be turn on to enable multiple selection

2). Two ways of identifying the list item selected.

a) obtain the reference to the item

b) obtain the index.

→

* getSelectedItemProperty(): to obtain the reference to the selected item. It defines what takes place when an element in the list is selected.

* addListner(): to add the change listener to this property using addListner()

Example:

Creates a listview to display various types of apples and allowing user to select one. Display the selection.

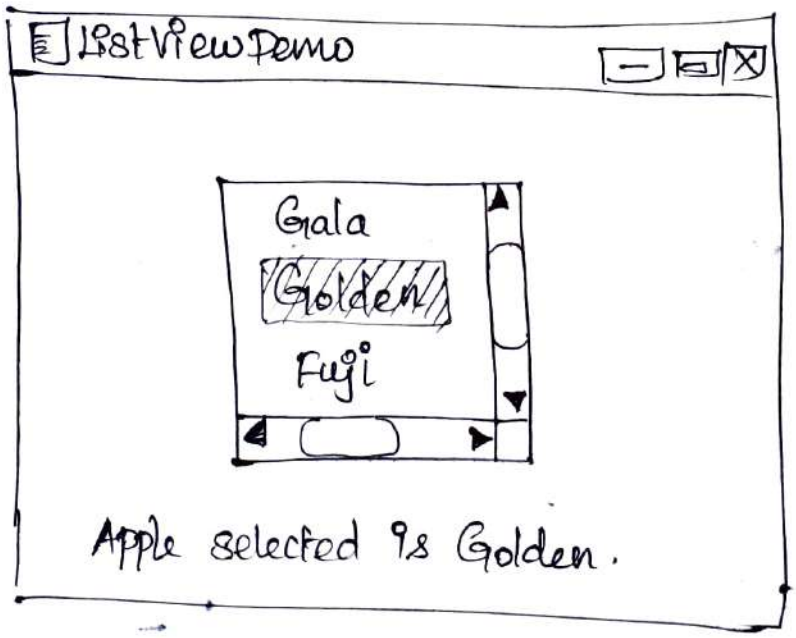
```
import javafx.application.*;
    :
public class ListViewDemo extends Application
{
    Label lblresponse;
    ListView<String> Apple;
    public void start(Stage myStage)
    {
        myStage.setTitle("ListView Demo");
        FlowPane root = new FlowPane(10,10);
        root.setAlignment(Pos.CENTER);
```

```
Scene myScene = new Scene( root, 220, 140);
myStage.setScene(myScene);
lblresponse = new Label("Select your Apple type");

// create list of entries.

    ObservableList < String> appleTypes =

        FxCollections.observableArrayList ( "Gala",
    "Golden", "Fuji", "Jonathan");

// Create ListView

    Apple = new ListView <String> (appleTypes);
    Apple.setPrefSize ( 80, 80);
    MultipleSelectionModel <String> model1 =
        Apple.getSelectionModel();

// adding listener.

model1.selectedItemProperty().addListener {
        new ChangeListner < String> () {
public void changed (observable values < ? extends
        String > changed String ov, String nv)
{
  lblresponse.setText ( "Apple Selected is" + nv);
} } );
```

→

```
root.getChildren().addAll(Apple, lblResponse);
    myStage.show();
  }
}

public static void Main(String args[])
{
    launch(args);
  }
}
```

Output:



To enable multiple selection:

```
model.setSelectionMode(SelectionMode.Multiple);
for(String item: getselectedItems())
    selItems += "\n" +item;
response.setText("All selected Apples" + selItems);
```

Combo Box

* A Combo box displays one selection, but it will also display a drop-down list that allows user to select a different item.

* It allows user to edit a selection.

* ComboBox inherits ComboBoxBase.

* ComboBox is designed for single selection.

* Generic class:

```
Class ComboBox <T>
```

T → type of entries.

* Constructors:

(1) ComboBox(): default constructor, creates empty Combobox.

(2) ComboBox(Observable<T> list): specifies set of entries in a list to be displayed in Combobox.

list → object of type ObservableList.

ObservableList: set of observable objects. It inherits Java.util.List.

* observableArrayList() method is defined by FXCollections class, used to create observable list.

* ComboBox generates an action event, when its selection changes. It will also generate a change event.

* It is possible to ignore events and display simply the obtain the current selection.

getValue(): To obtain the current selection from Combobox

setValue(): To set the value of a combobox under program control.

Syntax: `setVal (T newVal)`

* Example Program:

```
import javafx.application.*;
    :
public class ComboBoxDemo extends Application {
    ComboBox <String> places;
    Label lblResponse;
    public static void main (String args[])
    { launch (args); }
```

→

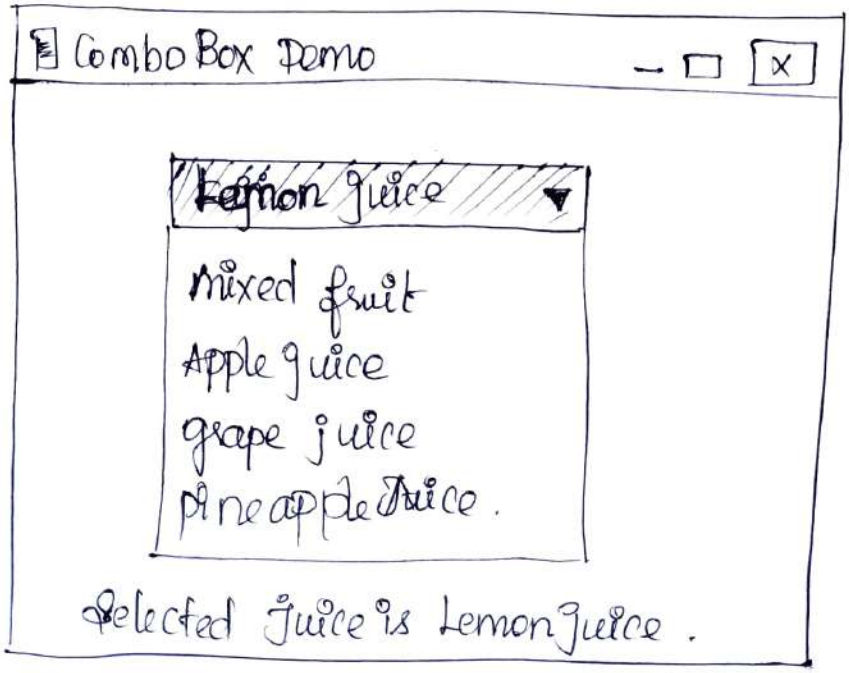
```
public void start (Stage myStage)
{
    myStage.setTitle ("Combobox demo");
    FlowPane root = new FlowPane(10,10);
    root.setPadding (new Insets(10,0,0,10));
    root.setAlignment (Pos.TOP_CENTER);
    Scene myScene = new Scene (root, 240, 120);
    myStage.setScene (myScene);
    lblRespone = new Label();
    // create a list of juices to display in Combo box.
    ObservableList <String> juiceTypes =
        FXCollections.observableArrayList ("Lemon juice",
                                           "mixedfruit",
                                           "Apple juice",
                                           "Grape Juice",
                                           "Pine Apple juice");
// Creates Combobox.
    juices = new ComboBox <String> (juiceTypes);
// set default value.
    juices.setValue ("Lemon juice");
    lblResponse.setText ("Selected juice type is" +
                         juices.getValue());
```

→

// ActionEvents of ComboBox.

```
juices.setOnAction(new EventHandler<ActionEvent>()
{
    public void handle(ActionEvent ae)
    {
        lblResponse.setText("Selected juice is" +
                            juices.getValue());
    }
});

rootNode.getChildren.addAll(juices, lblResponse);
myStage.show();
}
```

Sample Output:



Choice Box

* It allows the user select an option from a drop-down list. The selection is shown as checked (✓).

* It is similar to ComboBox but without the editing capabilities.

* Single selection only supported.

* ChoiceBox is a useful alternative to radio buttons when space is an issue.

* Class. ChoiceBox <T>

T → specifies the type of entries.

* Two constructors:

(1) ChoiceBox(): default constructor creates empty ChoiceBox.

(2) ChoiceBox(ObservableList <T> list): creates list of entries to be displayed in the ChoiceBox.

* A ChoiceBox is managed like a ListView. We can monitor the list of changes by registering a →

a change listner on the selection model as in ListView.

* To listen for change events, obtain the selection model by calling getSelectionModel() on choiceBox.

Syntax: SingleSelectionModel <T> getSelectionModel()

SingleSelectionModel is a class that defines the model used for single selections. ChoiceBox supports single selection. If you need multiple selection, use ListView. Add the change listner to the object returned by selectedItemProperty() when called on the selection model.

* setValue(): set the current value of the control

* getValue(): to obtain the selected value.

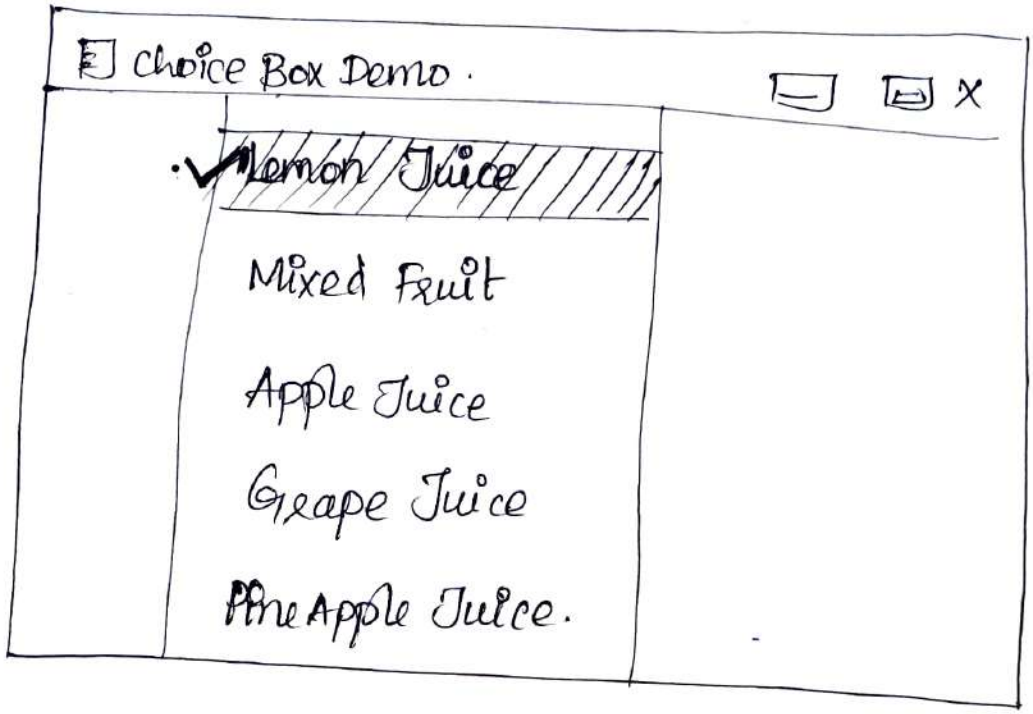
* setPrefWidth() & setPrefHeight() (or) setPrefSize() used to set the size of the combobox explicitly.

Example Program:

```
import javafx.application.*;
        :
        :
public class ChoiceBoxDemo extends Application
```

```
{
    Label lblResponse;
    ChoiceBox <String> juices;
    public static void main(String args[])
    {
        launch(args);
    }
public void start (Stage myStage)
{
    myStage.setTitle("ChoiceBox Demo");
    FlowPane root = new FlowPane(10,10);
    root.setAlignment(Pos.CENTER);
    Scene myScene = new Scene(root, 220, 140);
    myStage.setScene(myscene);
    lblResponse = new Label("Select your juice");
    ObservableList <String> juicetypes =
        FxCollections.ObservableArrayList("Lemon Juice",
        "Mixed Fruit", "Apple Juice",
        "Grape Juice", "PineApple Juice");
    juices = new ComboBox <String> (juicetypes);
```

Output Screen:



```
juices = new ChoiceBox <String> (Juicetypes);

juices.setValue ("lemon juice");

SingleSelectionModel <String> selModel = juices.
                    getSelectionModel();

selModel.selectedItemPeopesty () addListner {
    new ChangeListener < String > () {

public void changed ( Observable Value < ? extends
                    < String > changed , String oldval , String
                                                newVal ){

    lblresponse.setText ("Selected Juice" + newval1)}
    } );

rootNode.getChildren ().addAll ( lblResponse , Juices);
 myStage.show ();

}

}.
```

Suggested Questions / Assignments / Home works / any other
1) what is choicebox? 2) what is combobox? 3) what is listview? 4) Explain the usage of Choicebox & listview. 5) Explain the working of Combobox.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. Scroll Pane and Text Controls.

Topic(s) to becovered	Scroll Pane: uses – constructors – methods – Example. Text Controls: types – constructors – events – methods – Example Program.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	know the usage of Scrollpane	understand.
Lo2.	explain how scrollPane is working	understand.
Lo3.	know the usage of Text Control	understand.
Lo4.	discuss about methods and events of Text Control	understand.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate / Discuss / Peer to Peer Learning / Quiz / Role Play / Any other

Lecture Notes

Scroll Pane:

* Javafx ScrollPane provides scrollable view of User Interface (UI) elements / controls.

* It is used when we need to show large content within limited space. It has a viewport which will show a portion of the contents and provides a scroll bar when necessary.

Example: i) ListView, ComboBox and TextArea controls need to pack large data within small space. so ScrollPane is added to these controls when their contents exceed the dimensions.

2). A large graphics image/photo may not fit within reasonable boundaries (or) text in a label is longer than will fit within the size allotted for it.

* Scroll pane uses:

i) when scroll pane added to a node/control, the scroll bars automatically implemented that scroll the contents of the wrapped node.

2). scrolling capabilities are added to a node, when wrapping the node in a ScrollPane.

* Constructors of ScrollPane:

1) ScrollPane() → default constructor to Create ScrollPane.

2) ScrollPane(Node content): → we can specify a node that we want to scroll.

Content → Information to be scrolled.

SetContent() Method:

* when we use default constructor of ScrollPane, we can add the content (or) node to be scrolled by using setContent() method.

* Syntax: void setContent(Node content)

ViewPort: It is the viewable area of scrollPane. It is the area in which the content being scrolled is displayed. The scroll bar scrolls ~~thro~~ the content through the viewport.

Syntax: To set width

void setPrefViewportHeight(double height)

void setPrefViewportWidth(double width)

Vertical / HoriZontal ScrollBar:

* If the component is taller than the viewport, a vertical scroll bar is added.

* If the component is wider than the viewport, a hor horizontal scrollbar is added.

* If the contents are fit with in viewport, the scrollbars are removed.

setPannable(boolean enable): ScrollPane offers the ability to pan its contents by dragging the mouse. To turn on this feature, use setPannable().

enable < true → Panning allowed.
false → Panning disabled.

setVvalue(x);

setHvalue(): we can set the value of the scroll bar.

The new horizontal position specified using this function.

Syntax: void setHvalue(double newHval)

setVValue(): new vertical position of scroll bar.

Syntax: void setVvalue(double newHVal)

default scroll bar positions start at zero.

Example:

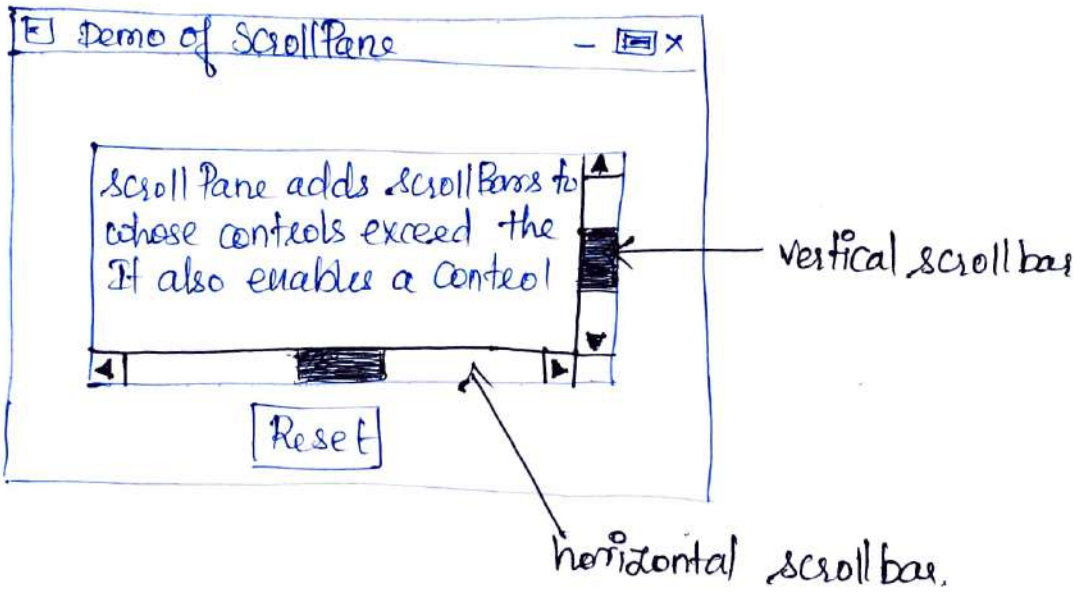
```
import javafx.application.*;
    :
public class ScrollPaneDemo extends Application
{
    ScrollPane scrollpane;
    Button btnReset;
```

```
public void start( Stage myStage)
{
    myStage.setTitle("Demo of scrollPane");
    FlowPane root = new FlocoPane(10,10);
    root.setAlignment(Pos.CENTER);
    Scene myscene = new Scene(root, 200, 200);
    myStage.setScene(myScene);

    Label label1 = new label("ScrollPane adds
        Scrollbars to a Node whose contents exceed
        the allotted space. It also enables a conteol
        to fit smaller space than it and offers us
        an elegant solution");
```

```
ScrollPane sclPane = new ScrollPane(label1);
sclPane.setPrefViewPostWidth(130);
sclPane.setPrefViewPostHeight(80);
sclPane.setPannable(true);
```

```
    btnReset =new Button("Reset");
    btnReset.setOnAction( new EventHandler<
        ActionEvent>(){
            public void handle(ActionEvent ae)
            {
                sclPane.setVValue(0);
                sclPane.setHvalue(0);
    });}
```

```
root.getChildren().addAll(sclPane, btnReset);
myStage.show();
}
Public static void main(String args[])
{
    launch(args);
}
}.
```



Text Controls:

* JavaFx provides text controls which allows us to enter text. Those controls allows us to enter string/ text of our own wish rather than selecting predefined options.

* Three forms of text controls:
  1. TextField : one line of text supported
  2. TextArea : multiline of text supported
  3. PasswordField: used to enter passwords & what typed is not shown.

all these three controls should inherit TextInputControl.

* TextField: It is used to enter one line of text.
  - It defines two constructors.
  - 1. default constructor: creates an empty text field that has the default size.

Syntax: `TextField()`

Example: `TextField tf1 = new TextField();`

→

▶ (2) Constructor with default text size:

The second constructor of the text field allows us to set default text to the TextField.

Syntax: `TextField ( String text)`

Example: `TextField tf2 = new TextField ("CSE");`

Other methods:

(1) setPrefColumnCount(int columns): used to specify the size of the textfield.

(2) setText (String str): used to set the textfield with some text.

(3) getText(): used to get (or) return the content of the textfield.

Other capabilities of TextField:

* Keyboard commands for cut, copy and paste are supported by Text.
* delete and insert operations
* getCaretPosition(): To obtain the position of caret
* getAnchor(): Starting point of selected text
* clear(): clear the content of the text field.

* setPromptText(): when no text has been entered inside the TextField, it will set a prompting message.

Syntax: setPromptText(String str)

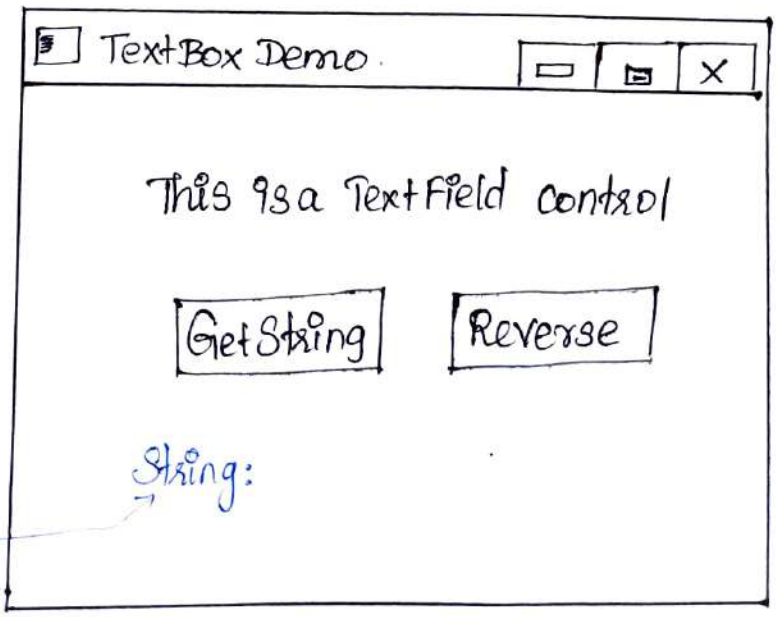
(*) ActionEvent: when we press [enter] key inside a TextField, an action event will be generated.

Example program:

This program creates a TextField and requests a String. When the user presses [ENTER] key (or) when the user presses [GetString] Button, the string is obtained and displayed. It will prompt a message when the TextField is empty.

When the user presses [Reverse] button, the reversed string is shown on TextField.

Design:



```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {

    TextField tf;

    Label response;
    Button btnGetText;
    Button btnReverse;

    public static void ~~main~~ start(Stage myStage)
    {

        // Stage title
        myStage.setTitle("TextBox Demo");

        // use a FlowPane for the root node. Here
        // vertical and horizontal gaps of 10.
        FlowPane rootNode = new FlowPane(10,10);

        // to set the controls on screen center
        rootNode.setAlignment(Pos.CENTER);
```

```
// Create a Scene.
Scene myScene = new Scene (rootNode, 230, 140);
// set the scene on the stage.
myStage.setScene(myScene);
// Create a label that will display the string.
response = new Label ("String");
// Create button that gets the text.
btnGetText = new Button ("Get String");
// Create button for reversing the text
btnReverse = new Button ("Reverse");
// Create a text field.
tf = new TextField ();
// Set the prompt message.
tf.setPrompt ("Enter a String");
// Set size.
tf.setPrefColumnCount (15);
// Event handling part : Action event
// when user presses ENTER button, the text in the field
// is obtained and displayed
```

String:

Get String

Reverse

→

```
tf.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
    response.setText("String" + tf.getText());
}});

// GetText from the text field when the button is
// pressed and display it.
btnGetText.setOnAction(new EventHandler<ActionEvent>
() {
    public void handle(ActionEvent ae)
    {
        response.setText("string" + tf.getText());
        tf.setText(str.reverse().toString());
    }
});

// Use a seperator to better organize the layout.
Seperator seperator = new Separator();
seperator.setPrefWidth(200);
// Add the controls to the scene graph.
rootNode.getChildren().addAll(tf, btnGetText,
btnReverse, seperator, response);

// Show the stage and its scene.
myStage.show();
}}
```

Suggested Questions / Assignments / Home works / any other
1) what is scroll Pane ? 2. Explain about scroll Pane with example program. 3. what is Text Field / Text Conteol? 4. Explain about how Text Field is working.

	Text Books / Reference Books / Any other suggested Materials		
S.No	Title	Author	Publisher
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Lecture No. Layouts, AnchorPane & BorderPane.

Topic(s) to becovered	Layouts - types - AnchorPane:- constructors - methods - Example. BorderPane:- Constructor - methods - Example.

	Lecture Outcome (LO)	
	At the end of this lecture, students will be able to	Bloom's Level
LO1	define what is layout	remember
LO2	List out the types of layouts	understand
LO3	Explain about AnchorPane	understand
LO4	Explain about BorderPane	understand

Teaching Learning Material	Student Activity
Chalk & Talk, ICT Tool	Discuss.

Lecture Notes

Layout:

Layouts are the toplevel container classes that define the UI styles for scene graph objects. They are the parent of other nodes, organizes scene graph nodes.

Layouts:

* The layout of the components on the screen is managed by a layout pane.
* Eight general purpose layout panes available.
* package: javafx.scene.layout.

→

* Layout panes types: (or) styles:

1. Anchor pane
2. Border pane
3. Flowpane
4. Grid Pane
5. Stack pane
6. Title pane
7. HBox
8. VBox.

* Base class of all layouts inherits from Pane class.

* To provide foundational support for controls and layouts inherits from class Region.

1. Anchor Pane: It is one of the javafx layout.

* It allows the edges of child nodes to be anchored to an offset from the anchor pane's edges. If the anchor pane has a border and/or padding set, the offsets will be measured from the inside edge of these insets.

* It inherits Pane class.

* Constructors:

1) AnchorPane(): Creates a new AnchorPane.

2). AnchorPane(Node.... c): creates a anchor pane with specified nodes.

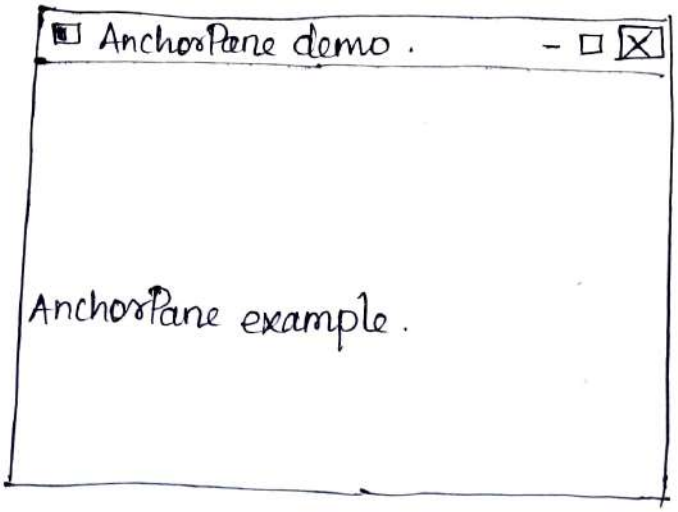
Commonly used methods:

1) getBottomAnchor(Node c)
2) getLeftAnchor(Node c)
3) getRightAnchor(Node c)
4) getTopAnchor(Node c)
5) SetBottomAnchor(Node c, Double v)
6) setLeftAnchors(Node c, Double v)
7) SetRightAnchors(Node c, Double v)
8) setTopAnchor(Node c, Double v)

Example:

```
Stage.setTitle("Anchor Pane demo");
Label label = new Label("Anchor Pane example");

AnchorPane anchorPane = new AnchorPane(label);
AnchorPane.setRightAnchor(label, 10.0);
AnchorPane.setTopAnchor(label, 10.0);
   ;;      .setLeft ;;  (label; 10.0);
   ;;      .setBottom ;; (label; 10.0);
Scene scene = new Scene(anchorPane);

Stage.setScene(scene);
stage.show();
```

Output:



2. BorderPane:

* Popular layout of javafx.
* It implements a layout style that defines five locations to which an item can be added.

1) center
2) top
3) left
4) bottom
5) Right

} borders. (or) four sides of the scene.

* Borderpane is quite useful when we want to organize a window that has a header and footer, content and various controls on the left and or right.

* Constructors:

1) BorderPane(): default constructor.

when we use this constructors, nodes can be assigned to a location by using following methods:

a) setCenter(Node item)
b) setTop(Node item)
c) setLeft(Node item)
d) setRight(Node item)

item: Node to be added in the layout.

2. BorderPane(Node centerPos):

3. BorderPane(Node centerPos, Node topPos, Node rightPos, Node bottomPos, Node leftPos)

These 2 constructors used to assign a node in indicated positions.

* setAlignment(Node What, Pos how): To set the alignment of ~~this~~ a node.

what specifies which node to be aligned.
how specifies which type of alignment.

* Pos → It is a enumeration (enum) that specifies alignment constants such as

1. Pos.CENTER
2. Pos.BOTTOM_RIGHT
3. Pos.TOP_LEFT

* package of Pos is javafx.geometry.

* setMargin(Node what, Insets margin):

The margins provide the gap between the edges of the node and edges of a border position to prevent from crashing.

→

what → the node for which margin will set

margin → the margin to be set

Example for BorderPane:

This program puts labels into top and bottom locations, a textfield in the center, a slider on the right and a VBOX on the left.

Coding:

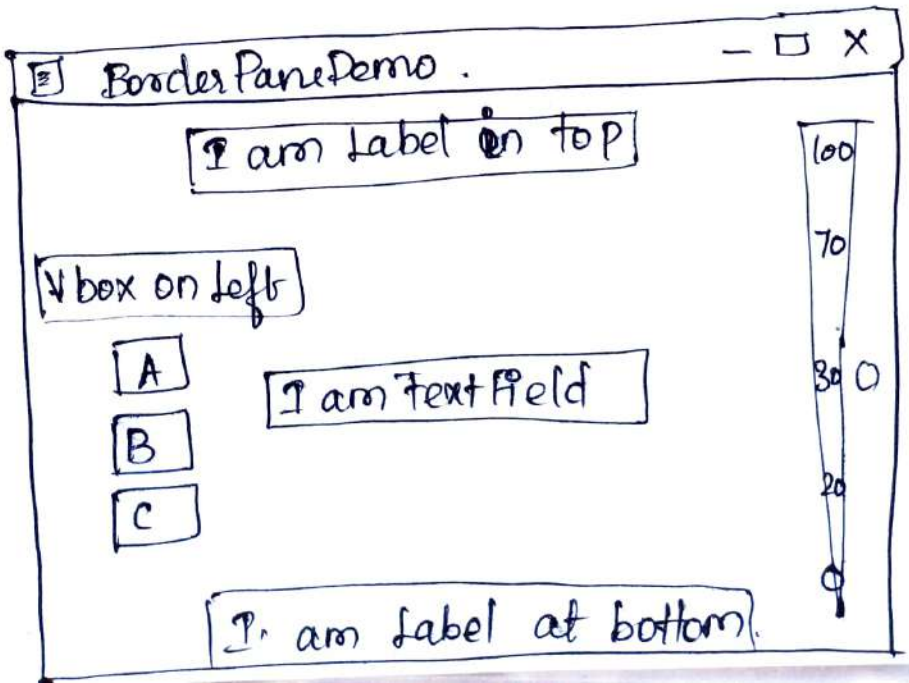
```
import javafx.Application.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.geometry.*;

Class BorderPaneDemo extends Application
{
 public void start(Stage myStage)
 {
    myStage.setTitle("BorderPane Demo");
    BorderPane root = new BorderPane();
    Scene myScene = new Scene(root, 340, 200);
```

→

```
myStage.setScene(myScene);
TextField tf1 = new TextField("I am TextField");   //to Put Textfield @Center
BorderPane.setAlignment(tf1, Pos.CENTER);
root.setCenter(tf1);

// Right (to put slider on the Right)
Slider sli = new Slider(0.0, 100.0, 50.0);
sli.Right.setOrientation(Orientation.VERTICAL);
sli.Right.setPrefWidth(60);
sli.Right.setShowTickLabels(true);
sli.Right.setShowTickMarks(true);
BorderPane.setAlignment(sli.Right, Pos.CENTER);
root.setRight(sli.Right);

// left (to put VBox on left).
    Button btn1 = new Button("A");
    Button btn2 = new Button("B");
    Button btn3 = new Button("C");

    btnAlpha.setPrefWidth(60);
    btn2.setPrefWidth(60);
    btn3.setPrefWidth(60);

    Label label1 = new Label("VBox on left");

    VBox vb = new VBox(10);
    vb.getChildren().addAll(label1, btn1, btn2, btn3);

root.setLeft(vb);
BorderPane.setAlignment(vb, Pos.~~LEFT~~ CENTER);
```

```
// Set margin for left location.
BorderPane.setMargin(vb, new Insets(10,10,10,10));

// Top
Label label2 = new Label("I am label on top");
BorderPane.setAlignment(label2, Pos.CENTER);
root.setTop(label2);

// Bottom
Label label3 = new Label("I am label at bottom");
BorderPane.setAlignment(label3, Pos.CENTER);
rootNode.setBottom(label3);
myStage.show();
}
public static void main(String args[])
{
launch(args);
}
}
```

Output:



Suggested Questions / Assignments / Home works / any other
1. What is layout? 2. What are the different styles (or) types of layout? 3. Explain in detail about AnchorPane with suitable example. 4. Explain in detail about BorderPane with suitable example.

	Text Books / Reference Books / Any other suggested Materials		
S.No	**Title**	**Author**	**Publisher**
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

LectureNo. FlowPane , HBox & VBox Layouts.

Topic(s) to becovered	FlowPane : types, Peoperties, methods example. HBox & VBox: properties, methods and examples.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
Lo1	define flowPane & its types.	Remembes.
Lo2.	define HBox & VBox	understand.
Lo3	use the flowPane on application	apply
Lo3	use the HBox & VBox in Application	apply.

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate / Discuss / Peer to Peer Learning / Quiz / Role Play / Any other

Lecture Notes

FlowPane: (. line by line arrangement)

* This layout pane organizes the nodes on a flow that are wrapped at the flowpane's boundary.
* Horizontal flow pane organizes the nodes on a a row and wrap them according to flowpane's width.
* Vertical flowPane arranges the nodes on a

Column and wrap them according to the flowpane's height.

* It is represented in javafx.scene.layout.FlowPane class.

Properties:	Description	method.
1) alignment	overall alignment of flowpane's content	setAlignment (Pos value)
2. orientation	~~Horizon~~ orientation of flowpane	setOrientation (Orientation value)
3. hgap	Horizontal gap b/w columns.	setHgap (Double value)
4. Vgap	Vertical gap among rows.	setVgap (Double value)
5. prefWrapLength	to set width and height of flowpane.	setPrefWrapLength (double value)
6. ColumnHAlignment	Horizontal alignment of the nodes within columns.	setColumnHAlignment (HPos value)
7. rowVAlignment	Vertical alignment of the nodes within rows	setRowVAlignment (VPos value).

Constructors:

1) FlowPane ( ) → default

2) FlowPane (Double Hgap, Double Vgap)

3) FlowPane (Node ... children)

4) FlowPane (Orientation orientation).

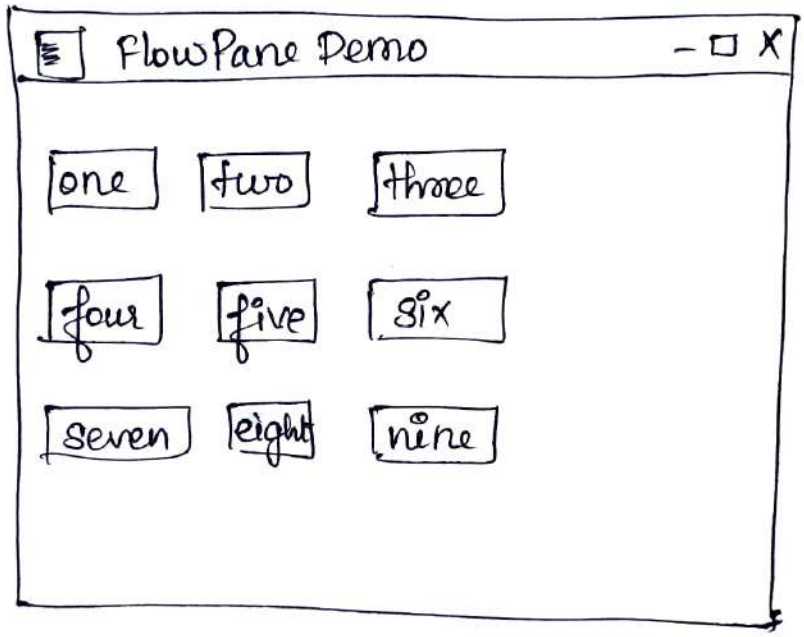
Example:

```
.
.
.
myStage.setTitle ( " FlowPane Demo");
Label one = new Label ( " one");
Label two = new Label ( " two");
.
.
.
Label nine = new Label ( "nine");

FlowPane flPane = new FlowPane (10, 10);
flPane.getChildren ( ).addAll ( one, two, three, four,
                    five, six, seven, eight, nine);

flPane.setPrefWrapLength (150);
FlowPane root = new FlowPane ( );
root.getChildren ( ).add ( flPane);
```

→

```
Scene myScene = new Scene(root, 230, 140);
myStage.setScene(myScene);
myStage.show();
}
```

Output:



If we change size in setPrefWrapLength(), then output will change. It specifies flowPane WrapLength.

HBox and VBox layouts:

HBox: This layout arranges the nodes in a single row. It is represented by javafx.scene.layout.HBox HBox arranges the controls in Horizontal line.

→

* to create HBox layout, just create object of HBox class.

Properties	Description	setter methods
alignment	represents alignment of the nodes	setAlignment(Double val)
fill-Height	It is boolean value. If true, height of the nodes become equal to height of HBox	setFillHeight(Double value)
spacing	represents the space b/w the nodes in the HBox.	setSpacing(Double val)

Constructors:

1) HBox(): create HBox layout with 0 (Zero) spacing.

2) HBox(Double ~~space~~ hGap): creates HBox layout with spacing b/w controls.

Example:

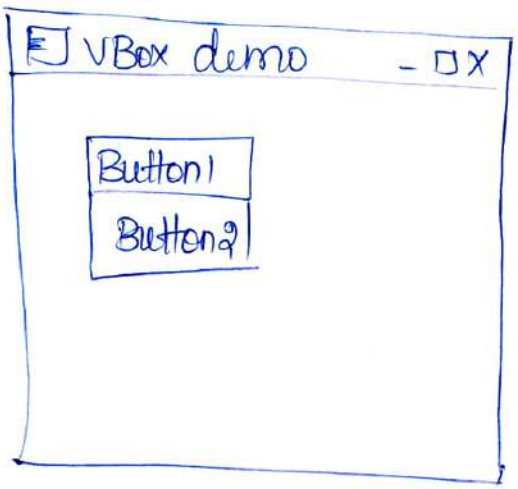
```
import javafx.application.*;
import javafx.scene.layout.HBox;
:
:
```

Example:

Write the same program of HBox here. Change

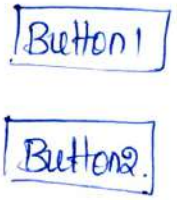
```
VBox root = new VBox();
```

O/P:



To add gap/space:

```
root.setSpacing(40);
```



Suggested Questions / Assignments / Home works / any other
1) what is flowpane? Explain it with example. 2) what is HBox and VBox? Explain them with suitable examples.

	Text Books / Reference Books / Any other suggested Materials		
S.No	Title	Author	Publisher
1	The Complete Reference Java, 7th Edition	Herbert Schildt	McGrawHill

YouTube	Reader may use the link to listen to the video of this lecture
	Reader may use the link to assess their understanding of the lecture. Teachers may use the question for conducting activity in the class

Property	Description	Setter method.
alignment	It represents the default alignment of children within the StackPane's width and height.	1) setAlignment (Node child, Pos value) 2) setAlignment (Pos value)

Constructors:

1) StackPane () → default constructor.

2) StackPane (Node ? children) → It allows us to specify the list of nodes to be added in stackpane.

Example:

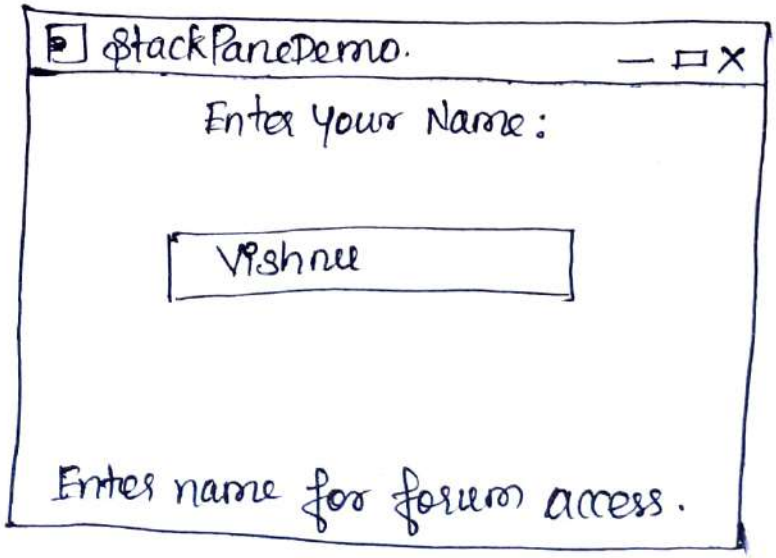
This program creates a textField and two labels. The labels are positioned at the top of the pane and at the bottom. The TextField placed in center.

Coding:

```
import javafx.application.*;
    :
    :
public class StackPaneDemo extends Application
{
    public void start (Stage myStage)
    {
```

```
mystage.setTitle(" StackPane Demo");
TextField tf = new TextField();
tf.setMaxWidth(120);
Label label1 = new Label(" Enter your name:");
Label label2 = new Label(" Enter Name for forum access.");
StackPane root = new StackPane();
StackPane.setAlignment(label1, Pos.TOP_CENTER);
StackPane.setAlignment(label2, Pos.BOTTOM_CENTER);
root.getChildren().addAll(tf, label1, label2);
Scene myScene = new Scene(root, 200, 120);
mystage.setScene(myScene);
mystage.show();
}
public static void main(String args[])
{
    launch(args[]);
}
}
```

Output:



GridPane:

* To layout controls using a row/column format, JavaFx provides GridPane.

* we have to specify row and column index in which we want a control to be placed.

* Grid pane positions controls at specific locations within a 2-D grid.

Constructor:

1. GridPane() → default constructor.

Methods:

1) setConstraints(Node what, int column, int row):

* row and column indices specifies the location at which a child node is added to the grid.

2) add():

void add(Node child, int column, int row):

Another way of specifying row and column is by using add() method.

3) setVgap(): To set the vertical gap around a control.

```
void setVgap(double gap);
```

4) setHgap(): To set the horizontal gap around a control.

```
void setHgap(double gap);
```

5) setPadding(): To set padding within the pane.

Example:

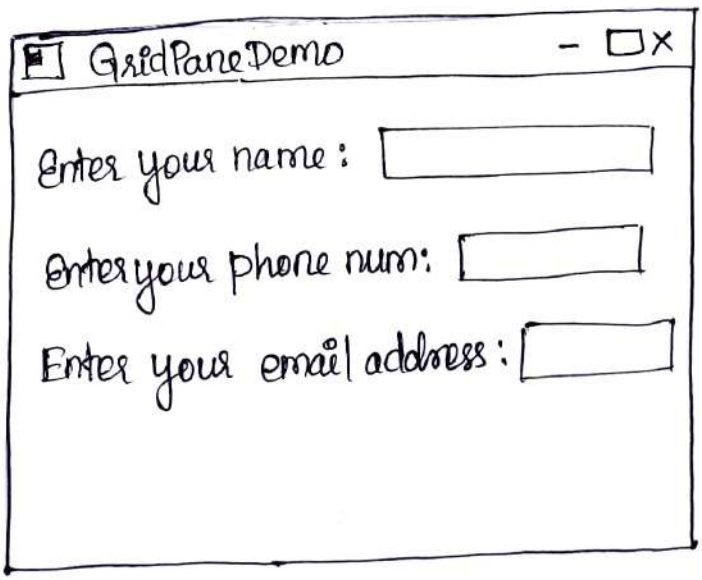
This program creates three labels and three text fields. It uses the grid to organize the layout so each label is linked with its text field.

Coding:

```
import javafx.application.*;
    :
public class GridPaneDemo extends Application
{
  public void start(Stage myStage)
  {
    myStage.setTitle("GridPane Demo");
    TextField tf1 = new TextField();
    tf1.setMaxWidth(120);
    TextField tf2 = new TextField();
    tf2.setMaxWidth(120);
  Label label1 = new Label("Enter your Name:");
  Label label2 = new Label("Enter your phone num:");
  Label label3 = new Label("Enter email address:");
  GridPane root = new GridPane();
  root.setPadding(new Insets(10,10,10));
    TextField tf3 = new TextField();
    tf3.setMaxWidth(120);
```

```
root.setVGap(10);
root.setHgap(20);
root.add(label1, 0,0);
root.add(label2, 0,1);
root.add(label3, 0,2);          // add first column.

root.add(tf1, 1,0);
root.add(tf2, 1,1);
root.add(tf3, 1,2);             // add second column.

Scene myScene = new Scene(root, 300, 120);
myStage.setScene(myScene);
myStage.show();
}
public static void main(String args[])
{
    launch(args);
}
}
```

Output:



* Other methods of GridPane :

1. setVgrow() :
2. setHgrow() :

} If an application needs a particular row (or) column to grow if there is extra space, it may set its grow Priority on Row & Col Constraints.

3. setRowSpan() :
4. setColumnSpan() :

} to control span of more than one column (or) row

5. setGridLinesVisible() : if true, it will display the grid lines of the grid Pane (ie) rows & columns.

* By default, the grid Pane will resize rows/columns to their preferred sizes, even if the gridPane is resized larger than its preferred size.

LectureNo. Menu Bar.

Topic(s) to becovered	Menu – Menu Basics – Menu classes – methods available in MenuBar, Menu, MenuItem, Constructors, Creating menu – example – Mnemonics, accelerator & Image.

	Lecture Outcome (LO) At the end of this lecture, students will be able to	Bloom's Level
lo1	Understand the basics of menu	understand
lo2.	Know the methods & constructors of Menu	Understand.
lo3	explain menu creation with example	understand
lo3	add mnemonics, accelerators & images to menu	Apply

Teaching Learning Material	Student Activity
Chalk & Talk / ICT Tool / Any other	Listen / Participate / Discuss / Peer to Peer Learning / Quiz / Role Play / Any other

Lecture Notes

Menu:

* JavaFx menu system supports menubar which is main menu for an application.

* Standard menu: Contains, either menu items to be selected (or) submenus.

Context menu: Activated by right clicking the mouse.
Other name: popup menu.

* we can enable menu items without using mouse and with the use of Accelerator keys from keyboard.

example: Copy → Ctrl + V
Cut → Ctrl + X
:

* Mnemonics: It allows menu item to be selected by the keyboard once the menu options are displayed.

* MenuButton: It lets you to create a button that activates a menu.

Menu Basics:

* package for menu 'javafx.scene.control'.

* To create a main menu for an application, create an instance of MenuBar. It is a container for menus (or) containers for instances of menus.

* Each menu object contains one (or) more selectable items. The items displayed by a Menu are the objects of type MenuItem

Menu Classes.

class	Description.
CheckMenuItem	A check (✓) menu Item.
ContextMenu	A popup menu activated by right clicking the mouse.
CustomMenuItem	A menu Item that can contain any type of Node.
Menu	A standard menu. A menu consists of one or more MenuItems.
MenuBar	An object that holds the top-level menu when pressed.
MenuItem	An object that populates menus.
RadioMenuItem	A radio button menu item.
SeperatorMenuItem	provides a visual seperator b/w menu items.
SplitMenuItem	Both a button & dropdown menu.

Menu Event Handling:

* when a menu item is selected, an action event is generated. when using one action event handler to process all menu selections, one way to determine which item was selected.

An overview of MenuBar, Menu & MenuItem:

(or)

Methods available in MenuBar, Menu & MenuItem:

1) MenuBar:

* Container for menus.
* It supplies the main menu of an application.
* It inherits Node. It can be added to scene graph.
* Two constructors of MenuBar:

(1) default constructor (ie) MenuBar(): creates empty menuBar and we can add menus.

(2) MenuBar (MenuItem1, MenuItem2, ...)

* Methods of menuBar:

(1) getMenus(): It returns a list of menus managed by the menu bar. You can add menus that you create to this list.

Syntax: final ObservableList <Menu> getMenus()

2) Menu:

A menu instance is added to this list of menus by calling add(). Also we can use addAll() to add 2 (or) more Menu instances in a single call.

* void add (int idx, Menu menu): This method is used to add menu at the index specified by idx.

* Menu can contain menuItems (or) submenus.

* Constructors of Menu:

1) Menu (String name): creates menu with specified name

2) Menu (String name, Node image): here image specifies the image to be displayed.

Menu ~~Item~~ contains both image & text.

* Methods of Menu:

1) setText(): add (or) change name of the menu.

2). setGraphic(): add (or) change image "

3) getItems(): returns list of items associated with the menu.

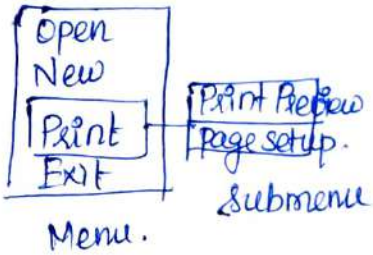
4). add((or) addAll(): add() → to add one item, addAll(): to add many items to menu.

5). remove(): remove an item from menu

6) size(): To get the size of the list.

3) MenuItem:

* It encapsulates elements of menu.
* each item can cause a selection linked to some action. example: Save (or) close.
  (or)
* Each item can cause a submenu to be displayed. Ex: File



* Three constructors of MenuItem:

a) MenuItem() : empty menu Item.

b) MenuItem(String name): Creates menu Item with name.

c) MenuItem(String name, Node Image):
It creates menu item with name & image.

* A menuItem generates an action event when selected.

→

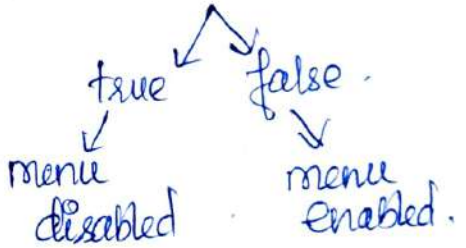
* Methods of MenuItem:

1) setOnAction(): to register an action event handler for event

2) fire(): to fire an event on a menu item.

3) setDisable(): used to enable / disable a menu item.

setDisable(boolean disable)



Creating a menu:

Step1: Create MenuBar instance to hold menus.

Step2: Construct each menu that will be in the menu bar.

Step3: Add menuItems to menu. (or) add submenus to menu.

Example Program:

```
import javafx.application.*;
```

⋮

→

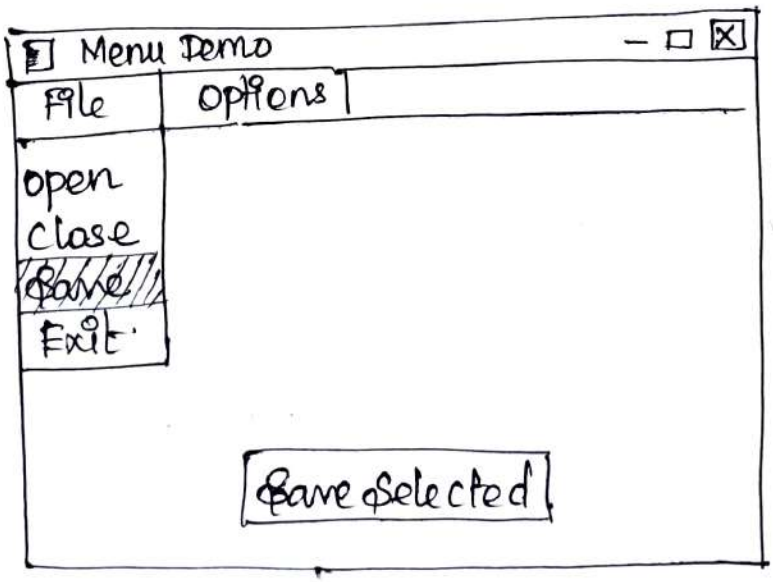
```
public class MenuDemo extends Application
{
    Label lblResponse;

    public void start ( Stage myStage)
    {
        myStage.setTitle (" Menu Demo");
        BorderPane root = new BorderPane();
    Scene myScene = new Scene ( root, 300, 300);
    myStage.setScene (myScene);
    lblRespone = new Label (" Menu example");
    MenuBar mb = new MenuBar ();

    // create the File menu

    Menu fileMenu = new Menu ("File");
    MenuItem open = new MenuItem("open");
    MenuItem close = new " ("close");
        "    save = " " ("save");
        "    exit = " " ("exit");

    fileMenu.getItems().addAll ( open, close, save, exit);
    mb.getMenus.add ( fileMenu);
    Menu optionsMenu = new Menu ("options");
```

→

```
Menu devicesMenu = new Menu("Input Devices");
MenuItem keyboard = new MenuItem("keyboard");
"        mouse = new "("mouse");
"        scanner = new "("scanner");

devicesMenu.getItems().addAll(keyboard, mouse,
                              scanner);
optionsMenu.getItems().add(devicesMenu);

// set action event handlers for the menu items.
open.setOnAction(MEHandler);
close.SetOnAction(MeHandler);
exit.  "  (  "  );
save.  "  (  "  );
keyboard.  "  (  "  );
mouse.  "  (  "  );
scanner.  "  (  "  );

// Event Handler.
EventHandler<ActionEvent> MeHandler = new
        EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
    String name = (MenuItem) ae.getTarget()).
                  getText();
```

```
// If Exit is chosen.
    if (name.equals("Exit"))
        Platform.exit();
    response.setText(name += "selected");
// add menubar @ top of BorderPane.
    rootNode.setTop(mb);
    rootNode.setCenter(response);
// show the stage and its scene.
    myStage.show();
}
public static void main(String args[])
{
    launch(args);
}
}
```

output:



Add Mnemonics and Accelerators to menu items:

~~Mnemonics:~~ Accelerator:

* open.setAccelerator(KeyCombination.keyCombination("Shortcut + O"));

It is equal to CTRL + O → File → Open.

Mnemonics:

Menu fileMenu = new Menu("_File");

we can choose file menu using Alt + F

Add Image to MenuItem:

MenuItem keyboard = new MenuItem("keyboard", new ImageView("keyboard.png"));