OOPs through Java

Java

unit - I

History of Java –

Computer language innovation and development occurs for 2 reasons
i) To adopt changing environment & users
ii) To implement ordinarily environment & improvement in art of programming.

Creation of 'C'

'C' created, need for structured, efficient, high level language that could replace assembly code.

other languages prior to 'C' are –

FORTRAN – used for efficient & scientific applications

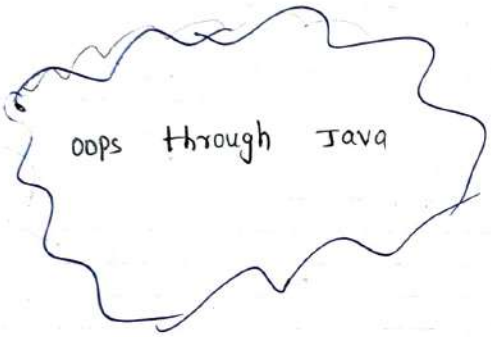
BASIC – not good for system code
easy to learn
not powerful

PASCAL – lack of structure
structured language
not designed for efficient

'C' languages

'Dennis Ritchie' – 1989's
powerful, efficient & structured lang

The next step is C++ language



Java

unit - I

History of Java –

Computer language innovation and development occurs for 2 reasons
i) To adapt changing environment & users
ii) To implement architectural improvement in art of programming.

Creation of 'C'

'Created, need for structured efficient, high level language that could replace assembly code.

other languages prior to 'C' are –
FORTRAN – used for efficient & scientific applications
Not good for system code
BASIC – Easy to learn
not powerful
lack of structure
PASCAL – structured language
not designed for efficient

'C' language

'Dennis Ritchie' – 1989's
'powerful', 'efficient' & structured language.
The next step is C++ language.

In ordu to avoid complexity in 'C' we introduce C++
Even in structured programming lang once project reaches certain size, its complexity increases
to solve this problem, new way to program was invented called "object oriented progeamwy" (oop)
It helps to organize complex programs through the use of Inheritance, Encapsulation & polymorphism.

C++ initially called 'c with classes' in 1983
- by adding object oriented feature

"Creation of Java"
By "James Gosling', 'patrick Naughton', 'chriswalth' 'Edfrank' and 'Mike sheridan' @ sunmicrosystem in 1991
Initially called 'oak' later renamed "Java" in year 1995

* Summary of oop (object oriented) concepts -
i) class
ii) object
iii) Encapsulation
iv) Inheritance
v) Abstraction
vi) polymorphism

* overview of Java -
oop is at the core of Java

Two paradigms - All computer program consist of 2 parts / 2 elements
1) code
2) Data
A program can be conceptually organized around its 'code' or 'data'

i) process oriented model -
- characterize a program as a series of linear steps (i.e. code)
- code acting on data
- 'c' employ this model
- problem with this approach as programs grows larger becomes more complex.

ii) object oriented programming -

organize a program around its data (i.e objects) & a set of well defined interfaces to that data.

→ To manage increasing complexity second approach came in to the world & "object oriented programming"

→ It is characterized as data controlling access to code.

→ achieves several organizational benefits.

'oop' principles

1) Encapsulation
2) Inheritance
3) polymorphism

i) Encapsulation — is the mechanism that binds together code and data and it keep both safe from outside interference & misuse

Encapsulation is basically to bind its data members & member functions into single unit called 'class'

It hide the details at the implementation level.

ii) Inheritance — is the process by which one object acquires the properties of another object.

it supports hierarchical classification

iii) polymorphism (Many forms) — is a feature that allows one interface to be used for a general class of actions

Because of polymorphism, in Java you can specify a general set of stacie routines that all share the same names.

A way of viewing World —

i) Agent —

A Java Agent is a regular Java class which follows set of strict conventions. The Agent class must be implemented a public static premain method.

After the JVM has initialized, each premain method will be called in the order Agents were specified.

The Agents development toolkits & agent programming languages are —

→ Agent identity
→ Agent host
→ Agent class

* OOP uses an approach of treating a real world agent as an object

2) Messages

Message contains a set of attributes & a 'content'. These messages need to be translated so they can be understood by end users around the world.

3) Methods

Method is a block of code which only runs when it is called. You can pass data, known as parameters into a method.

A method must be declared within a class. It is defined with the name of the method, followed by parenthesis ().

It provides predefined methods, such as System.out.println().

but you can also create your own methods to perform certain actions

* Steps to implement Java program

Implementation of Java program involves following steps

1. Creating the program
2. Compiling the program
3. Running the program

1) We can create a program using Text editor (Notepad) or IDE anyone and save filename as Sample.java

```
class Sample
{
public static void main(String [] args)
{
System.out.println("my first Java program");
}
}
```

2) Compile it by typing `javac Sample.java`

Javac → Java compiler translates source code into bytecode instructions

3) We need to use Java interpreter to run a program

Execute or run it by typing
"Java Sample"

Here Java interpreter converts the byte code into to Machine code

This Machine in computer is known as Java Virtual Machine (JVM)

In the Java program

① public → It means JVM can execute the method from anywhere

② static — It is a access specifier which indicates that main() can be called directly without creating an object for the class.

③ void — It indicates that the method main() doesnot return a value.

④ main() — It is a method which is a entry point into the Java program.
When you run Java program the main() is called first.

⑤ String args[] —
String is a class which belongs to Java.lang package
It can also be used as string data type in Java.
args[] — it is an array of string type.

⑥ system —
It is a class, which belongs to Java lang package.

⑦ out —
It's an output stream object, which is a member of system class

⑧ println() —
It is a method supported by the output stream object
It is used to display any kind of output on the screen.
It gives new line after printing the output.

⑨ print() —
It is similar to println() but it doesnot give new line after printing the output.

"Java Buzzwords" or "Features of Java"

(1) **simple** –
It follows the syntax of C & C++. it eliminates the complexities of C & C++. therefore Java has been made simple.

(2) **object-oriented** –
It is an object oriented programming lang the object model in Java is simple

(3) **platform independent** – (It is compiled by byte code that can run on any device)
platform is the combination of operating system and microprocessor
it is achieved by JVM (WORA)
↳ Write once read anywhere.

(4) **Robust** – (Ability to handle errors)
It is a strictly typed language it has a both compiler and an interpreter & it uses strong memory management.

(5) **Secure** –
Java developers have taken all care to make it a secure programming language.

(6) **Distributed** –
Java is designed for the distributed environment of the internet it handles TCP/IP protocols.

(7) **Multithreaded** –
Java was designed to meet the real-world requirement. To achieve this, Java supports multithreaded programming
it is the ability to run many any things simultaneously.

(8) **Dynamic** –
That is run time. This make it possible to dynamically link code in a safe.

(9) **Architecture-Neutral**
A central issue for the Java designers was that of code longevity & portability.
The JVM is an attempt to alter this situation the goal is to write once, run anytime anywhere, forever.

(10) **Interpreted & High performance** –
The Java byte code can be interpreted on any system that provides Java virtual Machine

The byte code would be easy to translate directly into native M/c code for every high performance by using a just-in-time compiler (JIT).

* Comments

8 types of comments defined by Java

// → single line comment.

/* */ → multiline comment.

/** */ → documentation comment.

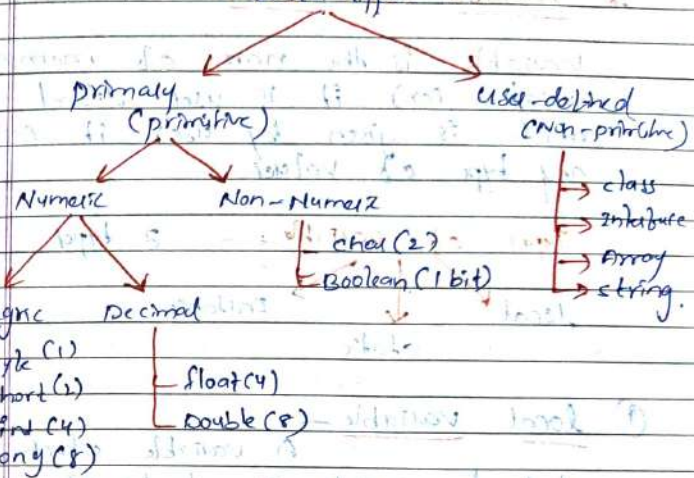
* Data types in ~~Java~~ Java

Data type specify the different sizes and values that can store in the variable

Data types are divided into 2 groups

(1) primitive — byte, short, int, long, float, double, boolean & char

(2) Non-primitive — Strings, class, Array etc.



	Datatype	Size	Description
1)	byte	1	-128 to 127
2)	Short	2	-32,768 to 32767
3)	int	4	-2,147,483,648 to 2,147,483,647.
4)	long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
5)	float	4	stores fractional no.'s storing 6 to 7 decimal digits
6)	double	8	stores fractional no.'s storing upto 15 decimal digits
7)	boolean	1 bit	stores true or false
8)	char	2	stores single character/ letter (or) ASCII value

* Variables in Java

variable is the name of memory location (or) it is used defined name which is given by user it can store any type of value.

Types of variable — 3 types

local, static, Instance

① **local variable** — A variable which is declared inside the body of the method (or) method parameter called local variable.

```
void fun(int x)      // x local variable.
{
    int n;           // n local variable.
}
```

② **Instance variable** — A variable which is declared inside the class but outside of all the methods called instance variable.

```
class A
{
    int a;  // instance variable
    public static void main(String[] args)
    {
    }
}
```

③ **static variable** — A variable which is declared with the help of static keyword called static variable.

```
static int n;   // 'n' is static
```

Ex:-

```
class A
{
    static int b = 20;  // 'b' static ('class')
    int c = 30;  // 'c' instance ('reference')
    public static void main(String[] args)
    {
        A rb = new A();
        int a = 10;  // local 'a'
        S.O.P(a);
        S.O.P(A.b);
        S.O.P(rb.c);
    }
}
```

* Classes and objects.

- Class is a group of objects
- It is a template (or) blueprint from which objects are created.
- It's a logical entity.

class contains
- fields (variables)
- methods
- constructors
- blocks
- nested class & interface

class is declared using keyword 'class' followed by class name

```
class class-name
{
public static void main (string args[])
{
    S.O.p ("Hello world");
}
}
```

Object
- It has state & behaviours
- It is a physical entity.
- It is an instance of class.

It has 3 characteristics
State, behaviour & identity.

new keyword is used to allocate memory at run time to create an object

```
class Student
{
    int id = 101;
    string name = "ABC";
    public static void main (string args[])
    {
        Student S1 = new student();  // create object of student
        S.O.p (S1.id); // accessing member through reference S1.
        S.O.p (S1.name);
    }
}
```

o/p

101
ABC.

* Arrays

→ Array is a group of similar typed variables that are referred by a common name
→ Array may be single / multidimensional
→ In Array elements are accessed by using index

1) one-Dimensional Array -

To create an array, you first must create an array variable

of desired type
format –
type var-name[];
Ex
int month[];
To link with month with an actual, physical array of integers you must allocate one using new and assign it to month
new is a special operator that allocates memory.
month = new int[12];

Ex
class AutoArray
{
public static void main (String args[])
{
int month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
S.O.p ("April has" + month[3] + "days");
}
}

O/p
April has 30 days.

Multidimensional Array –
Multidimensional arrays are actually arrays of arrays.
int 2D[][] = new int[3][5];

Ex
// 2D array
class TestArray1
{
public static void main (String args[])
{
int arr[][] = {{1,2,3}, {4,5,6}, {7,8,9}};
for (int i=0; i<3; i++)
{
for (int j=0; j<3; j++)
{
S.O.print(arr[i][j] + " ");
}
S.O.println();
}
}
}

O/p
1 2 3
4 5 6
7 8 9

* Operators in Java:

Operator is a symbol that is used to perform operations according to user requirement (variable & values).

Types:-

i) Arithmetic (+, -, *, %, /)
ii) Relational (<, <=, >, >=, ==, !=)
iii) Logical (&&, ||, !)
iv) Increment/decrement (++, --)
v) Assignment ← simple (=)
compound (+=, -=, *=, ...)
vi) Ternary (? :)
vii) Bitwise operator (AND, OR, XOR, complement)

i) Arithmetic operator

Ex:
```
import java.util.scanner;
class Arithmetic
{
public static void main(String args[])
{
int a, b;
S.O.p("enter 2 numbers :");
Scanner obj = new Scanner(System.in);
a = obj.nextInt();
b = obj.nextInt();
S.O.p("Addition:" + (a+b));
S.O.p("substraction:" + (a-b));
S.O.p("multi :" + (a*b));
}
}
```

o/p
Return result.

ii) Relational operator -

Ex
```
import java.util.scanner;
class Relational
{
public static void main(String args[])
{
int a, b;
S.O.p("enter 2 no.'s :");
Scanner obj = new Scanner(System.in);
a = obj.nextInt();
b = obj.nextInt();
S.O.p("true/false :" + (a<b));
S.O.p("true/false :" + (a>b));
S.O.p("true/false :" + (a<=b));
}
}
```

o/p - Return True/False.

iii) Logical operator -

Ex
```
class Logical
{
public static void main(String args[])
{
S.O.p("Logical AND, OR, NOT :");
S.O.p((10>5) && (2>1));
S.O.p((10>5) || (2>1));
S.O.p((10<5) || (2>1));
S.O.p(!(10>5));
}
}
```

o/p
Return true/false

iv) Increment / decrement operator

pre . post.
(++)(--) (++) (--)

Ex.

```
class prepost
{
 public static void main (String [] args)
 {
    int a=10;
    S.o.p (a);
    S.o.p ("prepost incr :");
    S.o.p (a++); //10 (11)
    S.o.p (++a); // 12
    S.o.p ("prepost decr :");
 }
}
```

Now we can also perform Assignment, Ternary & Bitwise operations

v) Assignment operator

```
class Assignment
{
 public ---
 {
    int a;
    a=10; = assignment (simple)
    S.o.p (a);
    a+=10; // compound assignment (a=a+10)
    S.o.p (a) // 20.
    - --
    - - - -
 }
}.
```

vi) Ternary : operator / conditional operator —

```
class Ternary
{
  public ----
  {
    int a=10, b=20, Max;
  Max=(a>b) ? a : b ;
    S.o.p (Max); // 20
  }
}
```

vii) Bitwise operator —

int a = 5 (0101)
int b = 7 (0111)

a	b	& AND	! OR	^ X-OR	(~a) ~ NOT
0	0	0	0	0	0
1	1	1	1	0	1
0	1	0	1	1	1
1	1	1	1	0	0
5	7	5	7	2	6.

0101
1
0110 (1's comp)

```
class Bitwise
{
    public static
    {
        int a=5, b=7
        S.o.p("AND"+ (a & b));  5
        S.o.p("OR" + (a | b));  7
        !
    }
}
```

* Expression in Java

Expression is a combination of operators, constants and variables

Types –
Constant Expression
Integral "
Floating "
Relational "
Logical "
pointer "
Bitwise "

An Expression may also use combinations of above expression. Such Expressions are known as compound Expression.

* Control statements in Java –

Java provides 3 types of control flow statements.

i) Decision Making statements
   * if statements
   * switch statement
     - simple if
     - if-else
     - ~~if else if ladder~~
     - ~~Nested if statement~~

ii) Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop

iii) Jump statements
   * break statement
   * continue statement.

* Inheritance in Java

It is a mechanism in which one object acquires all the properties and behaviours of a parent object. it is an imp concept of oops.

Why ?
→ For code Reusability
→ For method overriding.

Terms

- class
- sub class / child class
- super class / parent class
- Reusability.

Syntax –

```
class subclass_name extends superclass_name
{
    // methods & fields
}.
```

"extend" Keyword used in making new class from an existing class

Eg

```
class employee
{
    float salary = 40,000;
}
class programmer extends employee
{
    int bonus = 10,000;
    public static ----
    {
        programmer p = new programmer();
        S.O.p ("programmer salary is " + p.salary);
        S.O.p ("Bonus of programmer is " + p.bonus);
    }
}.
```

o/p

programmer salary is : 40,000
Bonus of programmer is : 10,000

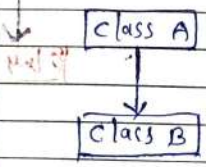
* Types of Inheritance –

In Java it supports only 3 type of inheritances

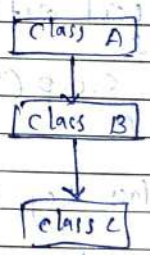
i) single inheritance
ii) multilevel inheritance
iii) Hierarchial inheritance

In Java the iv) Multiple and v) hybrid inheritance is achieved through interface only.

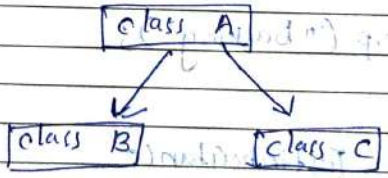
i) single



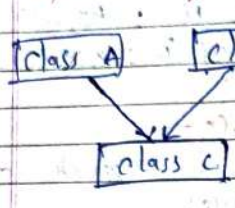
ii) Multilevel



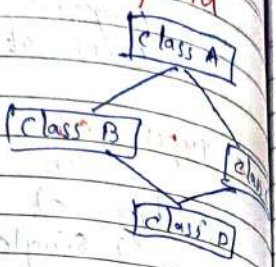
iii) Hierarchial



iv) Multiple



v) Hybrid



i) Single inheritance

when a class inherits another class, it is known as single inheritance




```
class Animal
{
  void eat()
  {
    S.O.P ("eating");
  }
}
class Dog extends Animal
{
  void bark()
  {
    S.O.P ("barking");
  }
}
class TestInheritance
{
  public static void main (String args[])
  {
    Dog d = new Dog();
    d.bark();
    d.eat();
  }
}
```

o/p

barking
eating.

ii) Multilevel inheritance

when there is a chain of inheritance, it is known as multilevel inheritance

In below ex, Babydog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.




```
class Animal
{
  void eat()
  {
    S.O.P ("eating");
  }
}
class Dog extends Animal
{
  void bark()
  {
    S.O.P ("barking");
  }
}
```

```
class Babydog extends Dog
{
 void weep()
 {
   S.O.p ("weeping");
 }
}

class Testinheritance
{
 public static void main (string
 {
  Babydog d = new Babydog();
  d.weep();
  d.bark();
  d.eat();
 }
}
```

o/p

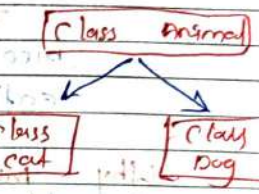
weeping
barking
eating

iii) Hierarchial inheritance

when two or more classes inherits a single class, it is known as hierarchial inheritance.

In below ex, Dog and cat class inherits the Animal class.

```
class Animal
{
  void eat()
  {
    S.O.p ("eating");
  }
}
class Dog extends Animal
{
  void bark()
  {
    S.O.p("barking");
  }
}
class Cat extends Animal
{
  void meow()
  {
    S.O.p ("meowing");
  }
}
class Testinheritance
{
  public static void main(string args
  {
    Cat c = new Cat();
    c.meow();
    c.eat();
    // c.bark(); error
  }
}
```

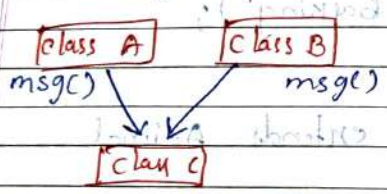


o/p

meowing
eating.

Why Multiple inheritance is not supported by Java.

consider a scenario where A, B and C are 3 classes. The class 'C' inherits A and B classes



If A and B classes have the same method and you call it from child class object, which method will be invoked (A or B) then the will Ambiguity cannot perform its task. So it returns 'Compile time error'

```
class A
{
  void msg()
  {
    S.o.p("Hello");
  }
}
```

```
class B
{
  void msg()
  {
    S.o.p("welcome");
  }
}
class C extends A, B
{
  public static void main(String args[])
  {
    C obj = new C();
    obj.msg(); // now which method should be invoked A/B ?
  }
}
```

Constructor

Constructor is a special method of a class which is invoked automatically whenever an object is created

It has same name that of a class

A constructor initializes an object immediately upon creation

A constructor has no return type not even void

It cannot be abstract, final, native, static or synchronized

"this" keyword refers to another constructor in same class.

A super keyword will call constructor of super class
Constructors are of 2 types.
i) Default constructor
ii) parameterised constructor.

i) Default - Constructor

→ It accepts no parameter
→ A default constructor is called when an instance is created for a class.
→ It automatically initialize all instance variable to zero.

Ex

```
class A
{
  int a;
  string b;
  Boolean c;
  A()  // default
  {
    a=100;
    b="ankit";
    c=true;
  }
  void Display()
  {
    S.o.p(a+" "+b+" "+c);
  }
}
```

```
class B
{ public static void main(string[] args) {
  A ref = new A();
  ref.Display();
}
```

o/p

100 ankit true

If constructor doesn't initialize the values by itself by default it will take values 0 for int
null for string
false for boolean.

ii) parameterized constructor —
A constructor through which we can pass one / more parameters is called parameterized constructor.

Ex

```
class A
{ int x,y;
  A(int a, int b) // parameterized
  {
    x=a; y=b;
  }
  void show()
  {
    S.o.p(x+" "+y);
  }
}
```

(PTO)

```
class B
{
    public static void main (String[] args)
    {
        A ref = new A (100, 200);
        ref.show();
    }
}
```

o/p

100 200.

Note: We can also create anonymous parameterized constructor but with different parameter data types.

* Access specifiers.

There are 4 types of Access specifiers of data in Java

They are

private

public

protected

default.

i) private

It can be applied to variables and methods.

The private members of a class are accessible from within the class only. They cannot be accessed from outside the class and its subclass.

Ex

```
class A
{
    private int data = 40;
    private void msg()
    {
        System.out.println("Hello Java");
    }
}
public class simple
{
    public static void main (String args[])
    {
```

```
A obj = new A();
S.O.p(obj.data); // Compile time error
obj.msg(); // Compile time error
}
}
```

o/p

Compile time error

Note: If you make any class constructor private, you cannot create the object of that class from outside the class.

ii) public –

It can be applied to variable (data members), methods & class.

public member of class can be accessed globally any other code.

They can be accessed from outside the class & sub class.

It can be accessed from any package.

Ex

```
class Alpha
{
  public int i;
  public void publicMethod()
  {
    System.out.println("In public method");
    System.out.println("i = "+i);
  }
}
```

```
class Beta
{
  public static void main(String args[])
  {
    Alpha a = new Alpha();
    a.i = 10;
    a.publicMethod();
  }
}
```

o/p

In public method

i = 10

iii) protected –

It can be applied to data members and methods.

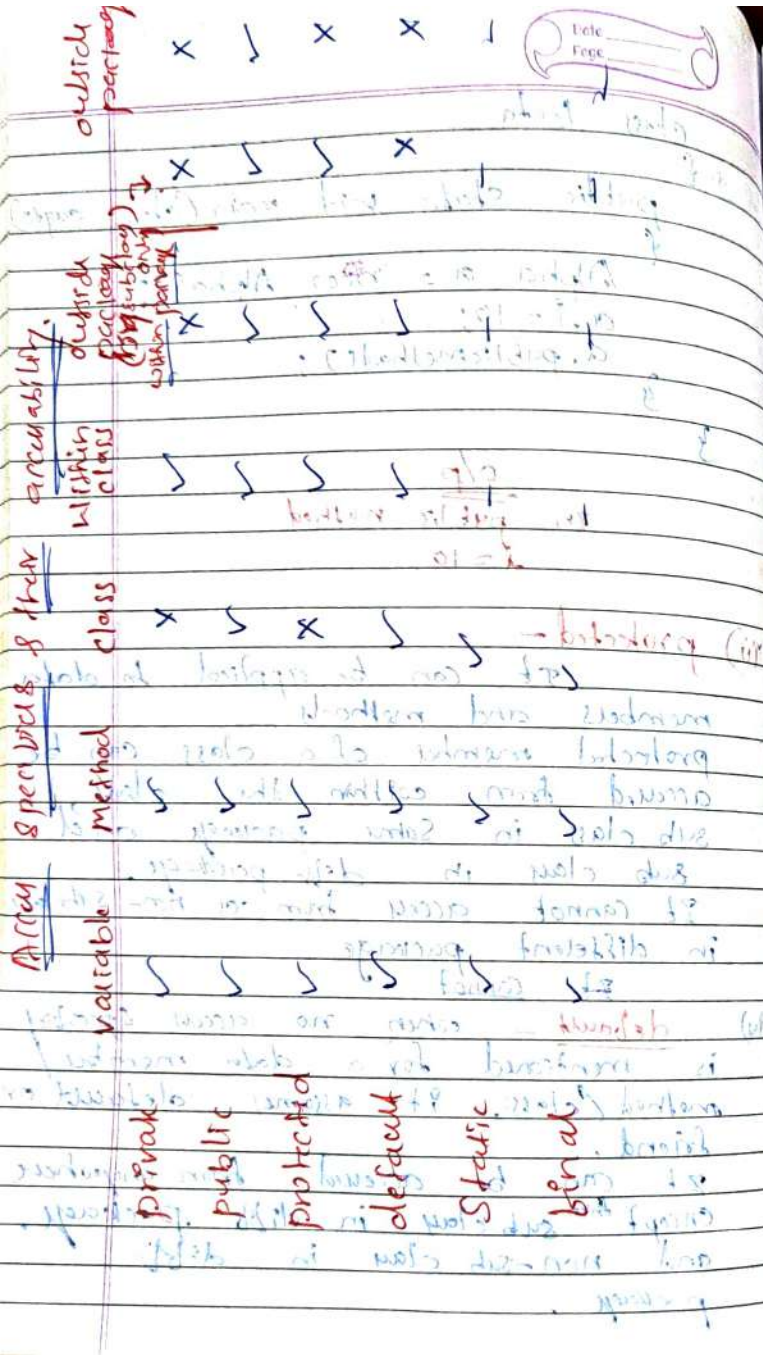
protected member of a class can be accessed from within the class, sub class in same package and sub class in diff packages.

It cannot access from a non-subclass in different package.

~~It~~ cannot

iv) default – when no access specifier is mentioned for a data member/method/class. it assumes default or friend.

It can be accessed from anywhere except from sub class in diff packages and non-sub class in diff packages.



* this keyword —

Sometimes a method will need to refer to the object that invoked it.

To allow this, Java defines this keyword. this can be used inside any method to refer to the current object.

That is, this is always a reference to the object on which the method was invoked.

① Ex —

```
class demo
{
    int x;
    int y;
    float z;
    demo(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    void display()
    {
        S.o.p("values of x, y and z are:"
            +x+" "+y+" "+z);
    }
}
```

```
class Demomain()
{
  public static void main (String args[])
  {
    demo d1 = new demo (1,2,3);   // parameterized constructor
    d1.display();
  }
}
```

Output

values of x, y and z are : 1 2 3

Method overriding and Method overloading in Java:

Method overriding — when the method signature (name & parameters) are the same in the superclass and the subclass (child), it is called overriding.

Method overloading — when two or more methods in the same class have the same "name" but different parameters, it is called overloading.

Method overriding —

It is used for runtime polymorphism.

Rules

i) The method (child) have the same name as in the parent class
ii) The method (child) have the same parameters as in the parent class.
iii) There must be an Inheritance relationship.

Eg

```
class vehicle
{
  void run()  // overridden method
  {
    S.O.P("vehicle is running");
  }
}
class Bike extends vehicle
{
  void run()  // overriding method
  {
    S.O.P("Bike is running safely");
  }
}
class Test()
{
  PSVM(String args[])
  {
    Bike obj = new Bike();
    obj.run();
  }
}
```

O/P

Bike is running safely.

/*

Vehicle obj = new Vehicle();
obj.run();

O/P - Vehicle is running

Vehicle obj = new Bike();
obj.run();

O/P - Bike is running safely

(a) Compiler checks whether run() method is available or in parent (Vehicle) or not (b) compile time

(a) run time in the child class (Bike) whether run() method is overriding or not "JVM" concludes

In overriding method resolution always take care by JVM not based on reference type.

It is also known as runtime polymorphism (or) Dynamic polymorphism (or) late binding.

* Method overloading -

If a class has multiple methods by the same name but different parameters, it is known as method overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Diff ways to overload the method.

2 ways

* By changing no. of arguments
* By changing the data type.

→ Ex by changing no. of arguments

```
class Calculation
{
  void sum(int a, int b)
  {
  } S.o.p(a+b);

  void sum(int a, int b, int c)
  {
  } S.o.p(a+b+c);

  psvm(String[] args)
{  Calculation obj = new Calculation();
   obj.sum(10, 10, 10);
} obj.sum(20, 20);
}
```

O/P 30
40.

* In this ex, a overloaded method with diff data type.

```
class Calculation1
{
  void sum(int a, int b)
  {
    S.o.p(a+b);
  }
  void sum(double a, double b)
  {
    S.o.p(a+b);
  }
  psvm(String[] args)
  {
    Calculation1 obj = new Calculation1();
    obj.sum(10.5, 10.5);
    obj.sum(20, 20);
  }
}
```

o/p
21.0
40

NOTE: In Java method overloading is not possible by changing return type of the method coz they may occur ambiguity.

* Super Keywords

It involves immediatly super class members, methods, construction. with help of super keyword.

Ex-

```
class one
{
  public void display()
  {
    S.o.p("I am from one class");
  }
}
class two extends one
{
  public void display()
  {
    super.display(); // immediate invoke super class
    S.o.p("I am from second class");
  }
}
public class third
{
  psvm(String args[])
  {
    one o = new two(); // one ref & 2 object
    o.display(); // runs in 2 class
  }
}
```

whenever classes are inherited and method overriding then only we can access 'super' keyword.

* Final Keyword -

The final keyword used to restrict the user to modify whenever the already assigned by parent class it can be applied on

a) → variable
b) → methods
c) → classes

If any class defined with final keyword it cannot inherit later it won't allow you to extend the class by inheritance

It just acts like a constant cannot be modified as the it may be variable / method / class.

The final modifier is used for finalizing the implementations of classes, methods & variables

Ex- a)

```
public class Example
{
    final int x = 10; // final variable
    String name = "xyz";
    public void change()
    {
        x = 20; // will give error
    }
}
```

b) A final method cannot be overridden by any sub class.

```
public class Example
{
    public final void change() // final method
    {
    }
}
```

c) The main use of final class. A class being declared as final is to prevent the class from being sub classed.

```
public final class example. // final class.
{
}
```

* Static & Dynamic Binding

Connecting a method call to the method body is known as binding.

Two types of binding

i) Static Binding (early binding)
ii) Dynamic Binding (late binding)

When type of object is determined at compiled time (by the compiler), it is known as "static binding" or method overloading

when type of object is determined at run time, it is known as dynamic binding. " method overriding

* Abstract class in Java

A class which is declared with abstract keyword is known as an abstract class

It can have abstract & non-abstract methods

Abstraction — is a process of hiding the implementation details and showing only functionality to the user.

It shows only essential things to the user and hides the internal details to the user

i) * We cannot create object for abstract class, but we can create reference to that.

Ex —
```
abstract class Animal
{
}
class Dog extends animal
{
}
class Demo
{
    Ps vm (String[] args)
    {
        Animal ob = new animal(); // error
        Animal ob = new Dog(); // compiles
    }
}
```

ii) It may or may not contain abstract methods

iii) It can have abstract & non-abstract methods

iv) To use an abstract class, you have to inherit from sub class

v) If a class contain partial implementation then we should declare a class as abstract.

Ex —
```
abstract class animal                    // cannot create object of class
{
    public abstract void sound(); // can't create object & method invoke.
}
class Dog extends animal
{
    public void sound()
    {
        S.o.p ("Dog is barking");
    }
}
class Lion extends animal
{
    public void sound()
    {
        S.o.p("Lion is roar");
    }
}
```

* object class in Java

The object class is the parent class of all the classes in Java by default

It is the topmost class of Java i, object is a super class of all classes. This means that a reference variable of type object can refer to an object of any other class.

Since arrays are implemented as classes a variable of type object can also refer to any array.

object defines following methods which means that they are available in every object.

```
object clone()
Boolean equals(Object object)
void finalize()
void notify()
void notifyAll()
String toString()
void wait()
Class <?> getClass()
int hashCode()
```

declared as final

* Exceptions

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program @ run time that disrupts the normal flow of the program's instructions.

Exceptions can be caught and handled by the program.

when an exception occurs within a method, it creates an object this object is called exception object it contains info about the exception, such as name & description.

* Here, we can try to find/catch exception

reasons for exception

invalid user input,
device failure,
loss of network connection,
physical limitations,
code errors,
opening an unavailable file.

Error — it is irrecoverable conditions such as
Jvm running out of memory,
memory leak,
stack overflow errors,
infinite recursion

* Here, application should try to catch

In Java, Exception handling is managed by 5 keywords.
try,
catch,
throw,
throws
& finally.

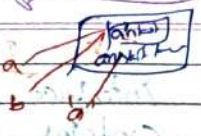
* String Handling in Java

Java strings are object that allows us to store sequence of characters which may contain alpha numeric values, which are enclosed in double quote(" ").

Note
* Strings are immutable in Java (cannot change).
* It contains methods that can perform certain operations on string
  eg- concat(),
  equals(),
  length()....

There are two ways to create a string object.
i) string literal
ii) new keyword.

i) string literal-




```
class A
{
    psvm (String[] args)
    {
        String a = "ankit";
        S.o.p(a); // prints ankit
        String b = "ankit";
        S.o.p(b); // ankit
        a.concat("kumar");
        S.o.p(a); // ankit
    }
}
```

In Java, string doesn't support any modification which is initially assigned @ beginning.

ii) new keyword.

```
class A
{
    public static void main(String[] args)
    {
        String a = new String("ankit");
        S.o.p(a); // ankit
        String b = new String("ankit");
        S.o.p(b); // ankit
        a.concat("kumar");
        S.o.p(a); // ankit
    }
}
// a = a.concat("kumar"); // avoid kumar display by o/p.
```

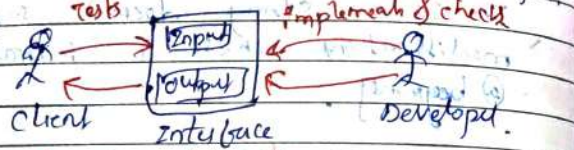
** Interface in Java

Interface is just like a class which contains only abstract method. To achieve interface Java provides a keyword "implements".

① - Interface methods are by default public & abstract.

② - Interface variables are by default public, static & final.

- Interface methods must be overrided inside the implementing class.
- It deals bet client & developer.



① import java.util.Scanner;

Ex- interface client
{
void input(); // public + abstract
void output(); // public + abstract
}
class Raju implements client
{
String name;
double sal;
public void input()
{
Scanner r = new Scanner(System.in);
S.o.p ("Enter username");
name = r.nextLine();
S.o.p ("Enter salary");
sal = r.nextDouble();
}
public void output()
{
S.o.p (name + " " + sal);
}
public static void main(String[] args)
{
client c = new Raju();
c.input();
c.output();
}
}

Ex-2) Bound on interface variable

interface customerraj
{
int iamt = 5; // public + static + final
void purchase(); // public + abstract.
}
class sellasarisu implements customerraj
{
public void purchase()
{
//amt = 7; // cannot reassign interface variable
S.o.p ("Raju needs " + amt + "kg rice");
}
}
class check
{

```
psvm (String[] args)
{
    customerraj C = new SellerSerivice();
    C.purchase();
    S.o.p(customerraj, amt);
}
}
```

polymorphism in Java

polymorphism means having many forms i.e has the ability of a message to be displayed in more than one form.
It is an imp feature of oops it can perform a single action in different ways.

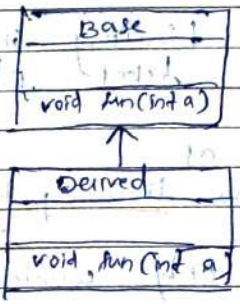
Types —

1) Compile time polymorphism (method overloading)



it is also known as "static polymorphism" OR "Early binding"

2) Runtime polymorphism (method overriding)



It is also known as "dynamic polymorphism" OR "late binding".

Adv

i) Increase code reusability by allowing objects of diff classes to be treated as objects of common class.
ii) Improves readibility & maintainability by reducing code.

Disadv

i) Difficult to understand the behaviour of object (bcz code is complex)
ii) performance issues,

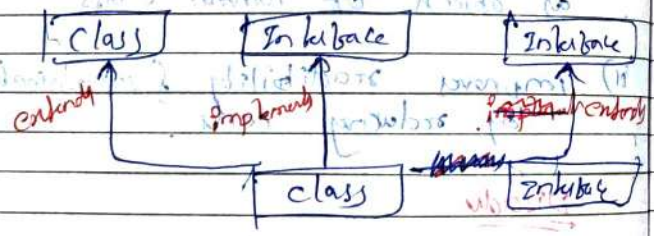
Ad-hoc polymorphism and pure polymorphism.

The method binding in adhoc polymorphism takes place during compilation
It's known as
"compile time polymorphism"

It is used only within a class

* Runtime method binding takes place in pure polymorphism
It's known as
"Run time polymorphism"

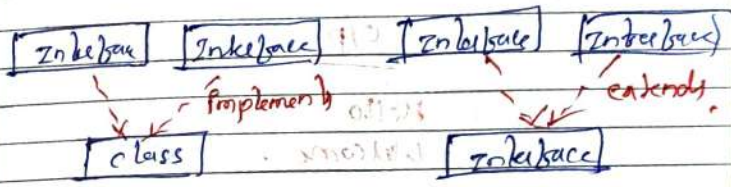
Interface (contd...)



Through interface we are going to achieve abstraction
There can be only abstract method in the java interface not method body.

It is achieved abstraction through multiple inheritance in Java

* Multiple inheritance in Java by interface.



Ex.
```
interface printable
{
  void print();
}
interface showable
{
  void show();
}
class A implements printable,showable
{
  public void print()
  {
    S.O.p ("Hello");
  }
  public void show()
  {
    S.O.p ("welcome");
  }
psvm (String args[])
{
```

```
A obj = new A();
obj.print();
obj.show();
}
}
```

O/p

Hello
Welcome.

* Forms of Inheritance

→ Specialization.
→ Specification
→ Construction
→ Extension
→ Limitation
→ Combination.

i) Specialization –
In this desired class is specialized form of the parent class & satisfies the specifications of the parent in all relevant aspects.

ii) Specification –
The child merely implements behaviours declared but not implemented in the parent.

The method implementation done in child & parent both are different.

iii) Construction –
A class often inherit almost all of its desired functionality from the parent class perhaps changing only the names of the methods called to interface to the class, it happens when new class and parent class failed to share the relationship.

iv) Extension –
In sub-classing for generalization modifies / enhances on the existing functionality of an object.
But sub-classing for extension lacks totally new abilities. It is just for method overriding.

v) Limitation –
Sub-classing for limitation occurs when the behaviour of the sub-class is smaller or more restrictive than its behaviour of the parent class.

vi) Combination –
When sub class may represent the combination of features from two or more parent classes.

* Benifits of Inheritance

→ s/w Reusability
→ Code sharing
→ Consistency of inheritance
→ s/w components
→ Information Hiding.

* Costs of Inheritance

→ Execution speed (reduce)
→ program size (increase lines of code)
→ message passing overhead
→ program complexity.

— As the program level of code increasing, execution speed decreases, message passing also play a vital role then automatically complexity of program increase.

PACKAGES
Unit - II

Java package —

Java package is a group of similar type of classes, interfaces and sub-packages.

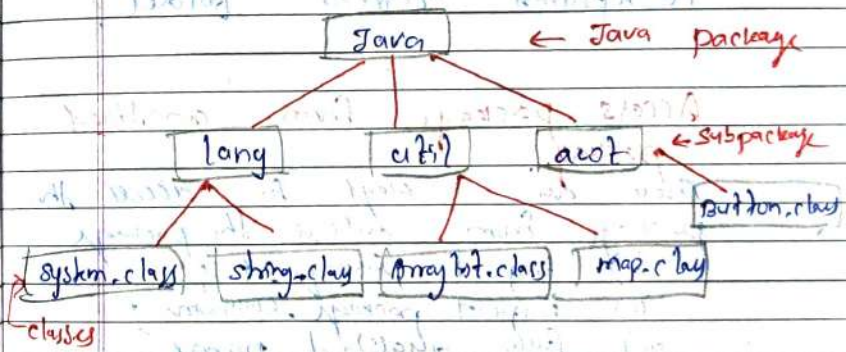
In Java, packages is categorized into 2 types they are

i) Built-in packages
ii) User-defined packages.

There are many built-in packages such as java.lang, awt, javax, swing, net, io, util, sql etc,

Adv

— provides area protection
— removes naming collision.



Example —

The package keyword is used to create a package in Java.

Inheritance

nheritance

ing.

nheritance
(reduce)
increase lines of code)
overhead
ty.
gram learning of
Execution speed
parsing extra play
automatically
gram increase.

PACKAGES
Unit - II

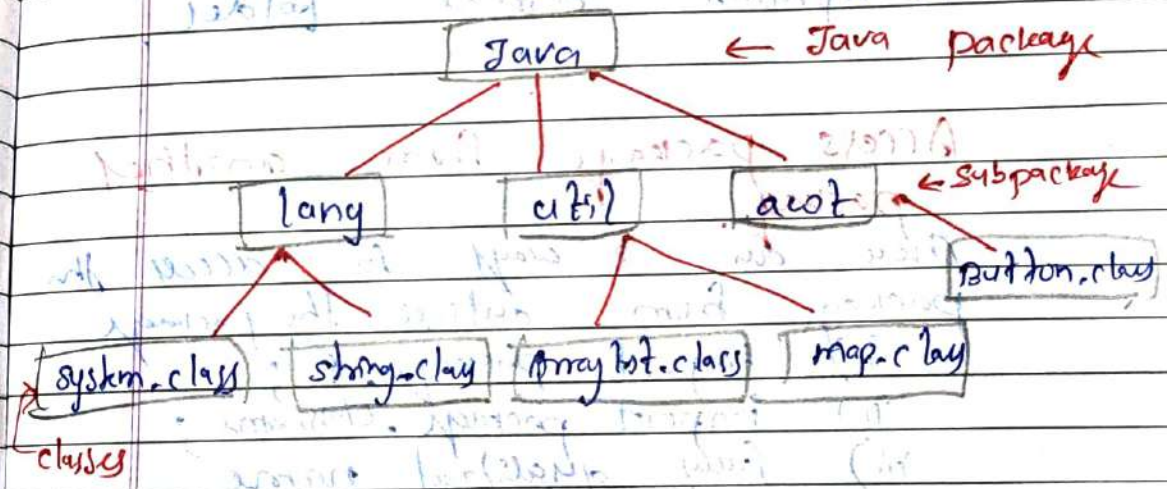
Java package –

Java package is a group of similar type of classes, interfaces and sub-packages.

In Java packages is categorized into 2 types they are
i) Built-in packages
ii) User-defined package.

There are many built-in packages such as java.lang, awt, javax, swing, net, io, util, sql etc.

Adv
1. provides access protection
2. remove naming collision



Example –

The package keyword is used to create a package in Java.

// sample.java

```
package mypack;
public class simple
{
  public static void main(String args[])
  {
    System.out.println("welcome to package");
  }
}
```

To compile: javac -d . simple.java

To run java mypack.simple

-d → specifies the destination where to put the generated class file
it represents current folder.

Access package from another package

There are 3 ways to access the package from outside the package

i) import package.*;
ii) import package.classname;
iii) fully qualified name.

i) If of package that import the package.* only interface of the class of the package that can be accessed not the classes.

ii) If of package that import the package.classname then only the mentioned class of this package can be accessed.

iii) fully qualified name behave like import as far as class is concerned other classes of the package are hidden and cannot be accessed.

iv) If of subpackage same as 1st

* → mean all the classes & interfaces of that package will be accessible but not subpackage

// A.java

```
package pack;
public class A
{
  public void msg()
  {
    S.O.P("Hello");
  }
}
```

// B.java

```
package mypack;
import pack.*;
class B
{
  psvm(String args[])
  {
    A obj = new A();
    obj.msg();
  }
}
```

importing package by class name → import pack.A

pack.A obj = new pack.A(); using fully qualified name

O/p: Hello

Access modifiers in Java

Access modifiers in Java are
- public (anywhere)
- private (within class)
- protected (within package & outside package subclass)
- default (within package)

Non-access modifiers in Java
- static
- final
- abstract
- volatile
- native.

Modifiers in Java



* Interfaces -
- Define
- Implement interface
- Nested interface
- applying interface
- variables in interface
- Extending interface

Definition

An interface is a completely abstract class. and it is a container of abstract methods and static final variables.

The variables declared in the interface cannot be modified by the class that implements the interface

Implementing interface

Once an interface has been defined, one or more classes can implement that interface

To implement interface, include the 'implements' keyword in a class definition and then create methods required by an interface.

The general form of a class that implements interface looks like this:

```
interface interface_name
{

}
class classname implements interface
{
    // class body declaring method body
        called in interface
}
```

Nested interface

The nested interface that declared inside another ~~class~~ interface must be accessed as outerinterface.Innerinterface.

```
class ~~outerclass~~
{
interface outerinterface
{
 void outermethod();
 interface innerinterface
 {
  void innermethod();
 }
}

class onlyouter implements outerinterface
{
 public void outermethod()
 {
  S.o.p(" this is outerinterface method only");
 }
}
class onlyinner implements outerinterface.
 innerinterface
{
 public void innermethod()
 {
  S.o.p(" This is innerinterface method");
 }
}.
```

```
public class NestedInterfaceEx
{
 psvm (--)
 {
  onlyouter obj = new onlyouter();
  onlyinner ob2 = new onlyinner();
  obj.outermethod();
  ob2.innermethod();
 }
}
```

o/p

This is outerinterface method only
This is innerinterface method only

* Extending interface

One interface can inherit another by use of the keyword 'extends'. The syntax is the same as for inheriting classes.

Ex

```
interface A
{
 void meth1();
 void meth2();
}
interface B extends A
{
 void meth3();
}
```

```
class Myclass implements B
{
    public void meth1()
    {
        S.o.p("Implemented meth1().");
    }
    public void meth2()
    {
        S.o.p("Implemented meth2().");
    }
}

class IFExtend
{
    psvm(---)
    {
        Myclass ob = new Myclass();
        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

Importing Packages

Java includes the import statement to bring certain classes / entire packages into visibility. Once imported, a class can be referred to directly, using only its name.

The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a dozens of classes in your application, however import statement will save a lot of typing.

In a Java source file, import statement occurs immediately following the package statement and before any class definations.

general form of import statement

```
import pkg1[.pkg2].(classname|*);
```

Here

pkg1 → name of top-level package

& pkg2 → name of subordinate package inside outer package separated by (.) dot

Finally you specify either an explicit classname (or) star (*).

↓

Indicates the Java compiler to import entire package.

Eg - program

```
package mypack;
public class Balance
{
  String name;
  double bal;
  public Balance (String n, double b)
  {
    name = n;
    bal = b;
  }
  public void show()
  {
    if (bal<0)
      System.out.println ("-->");
    else
      System.out.println (name + ":$" +
                          bal);
  }
}
```

```
import mypack.*;
class TestBalance
{
  psvm (String args[])
  {
    Balance test = new Balance
      ("Joseph" ,99.8);
    test.show();
  }
}
```

Its an example of user defined package.

Some Builtin packages provided for the programmer in order to reduce the burden some of the frequently used built in packages are —

* lang
* util
* io
* awt
* net
* applet.

To import all these packages syntax import java.util.*

(.* class name)

need to import all the class present in "util" package other than present in "java" package.

Eg — import java.util.Scanner

(Specific class)

java.lang →
language support classes. Java compiler links this & automatically imports. (primitive types — strings, maths, threads & exceptions).

java.util →
language utility classes such as (vectors, hash tables, random numbers, date etc.).

java.io →
Input/output support classes (input & output of data)

java.awt →
set of classes for implementing graphical user interface (GUI) - (windows, buttons, lists...)

java.net →
classes for networking. They include classes for communicating with local computers as well as with internet servers.

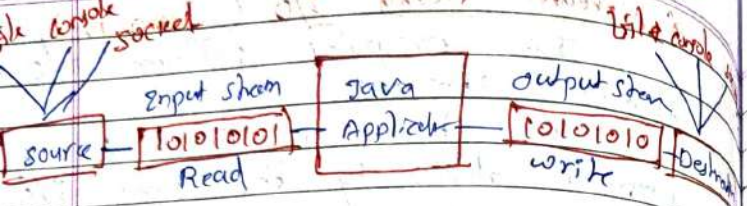
*** Stream based I/O (java.io)**
Java programs perform I/O through streams.
A stream is linked to a physical device by the Java I/O system.
Java I/O is used to process the input & produce the o/p.
It uses the concept of stream to make I/O operation fast.
The java.io package contains all the classes required for input and output operations.

Java defines 2 types of streams -
i) Byte stream
ii) Character stream

Byte stream provide a convenient means of handling input and output of bytes.

Character stream provide a convenient means of handling input and output of characters.

Byte stream classes -
Byte streams are defined by using 2 abstract classes
Input Stream (read)
output stream (write)



The byte stream classes are imported in your application by using
import Java.io.

Table 1: Byte stream classes in Java.

1) Input stream
2) output stream
3) Byte Array Input stream
4) Byte Array o/p stream
5) File i/p stream
6) File o/p stream
7) filter i/p stream
8) filter o/p stream
9) object i/p stream
10) object o/p stream
:
print stream (o/p stream containing print() & println()).

Data < i/p stream, o/p stream.

Buffered < i/p stream, o/p stream.

Character stream classes

Character streams are defined by using 2 abstract classes
Reader
Writer
these 2 classes handle unicode characters streams.
The character stream classes in Java (Java.io) are shown below.

Tab2: The character stream I/O classes in Java.io

1) Reader – abstract class that describes character i/p stream
2) Writer – abstract class that " " o/p stream
3) charArray < Reader, Writer
4) Buffered < Reader, Writer
5) File < Reader, Writer
6) Filter < Reader, Writer.
7) Reader < Input stream reader, o/p stream reader
8) String < reader, writer.
9) piped < reader, writer.
10) printwriter (o/p stream that contains print() & println()).

* predefined streams

All Java programs → Java.lang package (input) → class System (run-time environment)

System class contain 3 predefined stream variables in, out and err.

These fields are declared as public, static & final

System.out → refers to standard o/p stream (console) (default)

System.in → refers to standard input (keyboard) (default)

System.err → standard error stream (by default – console)

Java console class

console class is from – java.io package – used to read from and write to the console (character-based)

System.console() → used to get reference of system console.

* reading console input

In Java, there are 4 diff ways to read i/p from the user in command line argument (console)

1) using Buffered Reader class
2) using Scanner class
3) using Console class
4) using command line argument

1) using Buffered Reader class

This method is to take input by wrapping the system.in (standard i/p stream) in an InputStreamReader which is wrapped in a BufferedReader.

Ex –

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test
{
    p s v m (String[] args) throws exception
    {
        // Enter data using BufferedReader
        BufferedReader reader = new BufferedReader
        (new InputStreamReader(System.in));
        // Reading data using readline
        String name = reader.readline();
        // printing the read line
        S.O.P(name);
    }
}
```

I/p
Hello

O/p
Hello

ii) using scanner class —

```
import java.util.scanner;
class GetInput
{
  p svm( string args[])
  {
    Scanner in = new scanner (System.in);
    string s = in.nextLine();
    S.O.p ("you entred string" + s);
    int a = in.nextInt();
    S.O.p ("you enter integer" + a);
    float b = in.nextFloat();
    S.O.p ("you entred float" + b);
  }
}
```

O/p

I/p	O/p
Hello	you entred string Hello
25	" " integer 25
20.4	" " float 20.4

iii) using console class —
- most commonly used method
- console class defined in the java.io package.

```
import java.io.*;
class A
{
  public static void main (String[] args)
  {
    String str;
    char ch;
    Console obj = System.console();
    S.O.p ("enter username:");
    str = obj.readline();
    s.o.p ("enter password:");
    ch = obj.readpassword();
    S.O.p ("username:" + str);
    S.O.p ("password:" + ch);
  }
}
```

O/p

Enter username : abc
Enter password : - (not visible)

username : abc
password : @xyz

* Writing Console output using print, println & write methods.

output statement is used to print the output on display screen

- 3 main iostreams

Ex print method.

```
class print
{
  p s v m (String args[])
  {
    int a;
    for (a=1; a<=10; a++)
    {
      s.o.p(a);
    }
  }
```

O/p

1 2 3 4 5 6 7 8 9 10

Ex - println method

```
class println
{
  p s v m (---)
  {
    int a;
    for (a=1; a<=10; a++)
    {
      s.o.p(a);
    }
  }
```

O/p

1
2
3
4
5
6
7
8
9
10

Ex

```
class Write
{
  p s v m (---)
  { int a, b;
    a = 'q';
    b = 65;
    System.out.write(a);
    System.out.write('\n');
    System.out.write(b);
    System.out.write('\n');
  }
}
```

O/p

q
A

print & println methods defined by class printstream.

System.out → used for byte stream.

printstream → output stream
it has low level method "write()"
↓
used to write to console.

Simplest form of write() method defined by printstream is shown below

```
void write(int byteval)
```

* printwriter class (another method writing console o/p)

Specially used character-based class

Syntax is

```
printwriter(Output.stream  outputstream ,
                    boolean flushingon)
```
class → object

```
printwriter pw = new printwriter(
        ↓ system.out , true);
pw.println("This is a string");
```

eg

```
import java.io.*;
public class printwrite
{
  psvm(String args[])
  {
    printwriter pw = new printwriter(
                    system.out, true);
    pw.println("This is string");
    int i = -7
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

* Reading & Writing Files.

Java provides no. of classes and methods that allow you to read & write into files.

Main methods
FileInputStream } create byte streams
FileOutputStream } linked to files

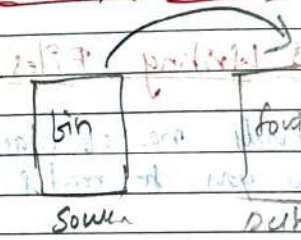
FileInputStream(String filename) throws FileNotFoundException

FileOutputStream(String filename) throws FileNotFoundException.

FileInputStream [Reading file]

Ex
```
int i;
FileInputStream fin;
fin = new FileInputStream(file1.txt);
  // throws exception if file not found
do
{
  i = fin.read();
  if(i != -1)
    S.o.p(char(i));
} while(i != -1);
fin.close();
```

The fileoutput stream [writing file]



Example for fileoutput stream, using write() method to copy file by one file to another.

Ex
```
int i;
FileInputStream fin = null;
FileOutputStream fout = null;
// declare 2 files one from another
fin = new FileInputStream(file1.txt);
fout = new FileOutputStream(file2.txt);
do
{
  i = fin.read();
  if(i != -1)
    fout.write(i);
} while(i != -1);
```

Random Access File operations

1) Modified Type — method
   → void — close()
   close random access file stream

2) FileChannel — getchannel()
   it returns unique FileChannel object associated with this file.

3) int — readInt()
   it reads a signed 32-bit integer from the file.

4) string — readUTF()
   reads in a string from file

5) void — seek(long pos)
   it sets filepointer offset, measured from beginning of this file.

6) void writeFloat (float v)

void writ(int b) — writes specified byte into the file

int read() — reads a byte from the

long length() — returns the length of the

void seek()

* Console class

To get i/p from console. This class provides methods to access the character-based console device, if any, associated with the current Java virtual m/c.

~~String text = system.console().readline();~~

Console class is used to get input from console & provides methods to read texts and passwords

string text = system.console().readline();
S.O.println("Text is :" + text);

Console ^class methods such as

Reader reader() — retrieve reader object

9) String readline() — used to read single line of text from console

3) char[] readpassword() — used to read password that is not being displayed on console

4 void flush() — used to flushes the console

Ex —

```
import.java.io.console;
class ReadStringTest
{
public static void main (String args[])
{
   Console c = system.console();
   S.O.p ("Enter your name");
   String n = c.readline();
   S.O.p ("welcome" +n);
}
}
```

O/p

Enter your name: Akhil
welcome Akhil.

```
//{
console c = system.console();
S.O.p ("enter psswd");
char[] ch = c.readpassword();
String pass = string.valueOf(ch); // converting char array into string
S.O.p ("password is" +pass);
}
```

O/p

enter password .....
password is : 123

* Serialization & Deserialization



Serialization is a mechanism of writing the state of object into a byte stream.

The reverse operation of serialization is called deserialization where byte stream is converted into an object.

For serializing the object, we call writeObject() method of objectOutputStream class and for deserilization we call readobject method of objectInputStream class.

Ex

Emp.java // file name Emp.

```
import java.io.Serializable;
public class Emp implements Serializable
{
    public string name;
    public string address;
}
```

// SerializableClass.java

```
import java.io.*;
public class SerializationClass
{
    psvm ( String[] args )
    {
        Emp emp = new Emp();
        emp.name = "Rosy";
        emp.addres = "Hyderabad";
        try
        {
            FileOutputStream fileout = new FileOutputS
                                ("e:\\files\\1.tx
            objectOutputStream out = new objectOutpu
                                stream(fileout);
            out.writeObject(emp);
            out.close();
            fileout.close();
            S.O.P("serialized data is saved in
                        f1.txt file");
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

// here object is created & values assigned those are going to write it on file using FileOutputStream, as we are dealing source as object so we need objectOutputStream

* Enumeration

Enum is a data type in Java which contains a fixed set of constants.

Java enum constants are static & final implicitly (is available in jdk 1.5) & declared

It is defined either inside a class/ outside a class.

Eg - outside class

```
enum Color
{
    RED,
    GREEN,
    BLUE;
}
public class Test
{
  psvm (String[] args)
  {
    Color C1 = Color.RED;
    s.o.p (C1);
  }
}
```

o/p

RED

Autoboxing —

The Automatic conversion of primitive data type to object

unboxing -

The Automatic conversion of object to primitive type.

As, Java is object oriented programming language but its primitive data types such as

byte
short
int
float, double — } primitive types.

are not objects so with the help of wrapper class we able to convert primitive data type to object

primitive data type	Wrapper class
byte	Byte
short	Short
int	Integer
float	Float
double	Double
char	Character
boolean	Boolean

Ex

```
class AutoBoxing
{
  psvm (String args[])
  {
    int a = 10;          // variable of int type
    Integer b = a;       // Auto boxing
    Integer c = Integer.valueOf(a);
    S.o.p(a+" "+b+" "+c);
  }
}
```

object — method.

// a → variable
b & c → objects

O/p

10 10 10

Ex

```
class UnBoxing
{
  psvm (String args[])
  {
    Integer a = new Integer(10);
    int b = a;           // unboxing
    int c = a.intValue(); // unboxing
    S.o.p(a+" "+b+" "+c);
  }
}
```

new a → object
b & c → primitive variable

O/p

10 10 10

Date Page

Java – Generics

Java generics allows us to create single class, interface and method that can be used with different types of data (objects).

Ex Create a generics class

```
class Main
{
  psvm(String[] args)
  {
    GenericClass<Integer> intObj = new
        GenericClass<>(5);
    S.o.p("Generic class returns:" + intObj.
        getData());
    GenericClass<String> strObj = new
        GenericClass<>("Java programming");
    S.o.p("Generic class returns:" + strObj.
        getData());
  }
}

class GenericClass<T>
{
  private T data;
  public GenericsClass(T data)
  {
    this.data = data;
  }
  public T getData()
  {
```

return this.data;
}
}

o/p

Generic class return : 5
Generic class return : Saurabh Pragne

This generic class is used to work with any type of data

class GenericClass <T> {...}

<> type parameter.

In above example generic class we have created 2 generic objects
Intobj → Here T, replaced by integer work with integer data
Stringobj → Here T, replaced by string, work with string data.

Generic method —
Like to generic class, we can also create generic method that can be used with any type of data such a class known as generic method.

Gn

class Container <T>
{
T value;
public void show()
{
s.o.p(value.getClass().getName());
}
}
public class GenericDemo
{
p s v m (String[] args)
{
Container <Integer> obj = new Container <>();
obj.value = 9;
obj.show();
}
}

o/p

java.lang.Integer

Why Generics ?

* To provide type safety and to resolve type casting problems

Unit- III

Exception Handling

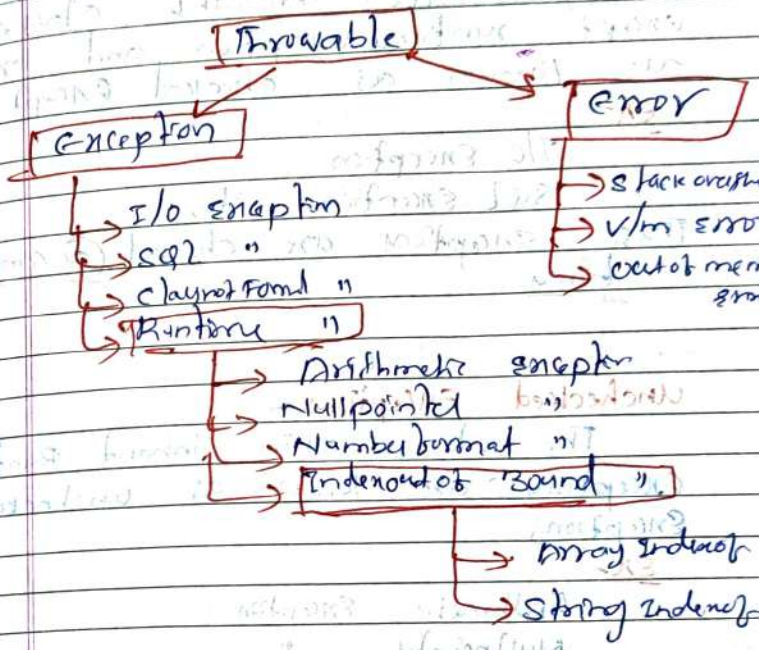
Exception –
is an event that disrupts the normal flow of program it's an abnormal condition in normal flow of program @ run time

Exception Handling
It is a mechanism to handle runtime errors such as
ClassNotFound exception,
IO exception
SQL exception
Remot Exception etc.,
By this mechanism we can maintain the normal flow of the application program.

Hierarchy of exception class

The Java.lang.Throwable class
↓
root class of Java exception
Again it is divided into
→ Exception
→ & Error

Hierarchy of Java Exception classes



There are two types of exceptions
→ Checked
→ Unchecked (error)

Acc. to oracle 3 types of Exceptions namely
→ checked Exception
→ Unchecked Exception
→ error

Checked Exceptions –

The classes which directly inherit Throwable class except runtime Exception and error are known as checked Exception.

Ex-
IO Exception
SQL exception etc.,

These exceptions are checked @ compile time

Unchecked Exceptions –

The classes which inherit Runtime Exception are known as unchecked Exceptions

Ex-
Arithmetic Exception
Nullpointer "
ArrayIndexOutOfBound exception etc

These exceptions are checked @ run time.

Error –

Error is irrecovarable

Ex-
OutOfMemory Error
VirtualMachine Error.

Java Exception Keywords

Java provide 5 keywords that are used to handle the exception.

i) try –

The 'try' keyword used to specify a block where we should place an exception code. Abu this it should be followed by either catch / finally block.

ii) catch –

The 'catch' block is used to handle the exception. it must be preceded by try block and followed by finally block later.

iii) finally –

The 'finally' block is used to execute the necessary code of the program whether an exception is raised or not.
it means it is the last line of Execution program.

iv) throw –

The 'throw' keyword is used to throw an exception.

v) throws –

The 'throws' keyword is used to declare Exceptions. it specifies

that there may occur an exception in the method. It doesnot throw an exception it is always used with method signature.

Example program

```
// command use of exception handling
public class TryCatchEx1
{
  public static void main (String[] args)
  {
    int data = 50/0; // throw exception
    S.o.p("rest of code");
  }
}
```

o/p

Exception in thread "main" java.lang.ArithmeticException: / zero.

Solution by using exception handling to print remaining part of program even though exception raised.

```
public class TryCatchEx2
{
  public static void main(String[] args)
  {
    try
    {
      int d = 50/0; // throw exception
    }
    // handling exception
    catch (ArithmeticException e)
    {
      S.o.p(e);
    }
    S.o.p("rest of code");
  }
}
```

o/p

java.lang.ArithmeticException: / zero
rest of code.

* Common Scenarios of Java Exception (unchecked)

① ArithmeticException occurs — divide any number by zero,

```
int a = 50/0;
```

② NullPointerException occurs — having null value in any variable

```
String s = null;
S.o.p(s.length());
```

3) NumberFormatException — if the formatting of any variable / number is mismatched

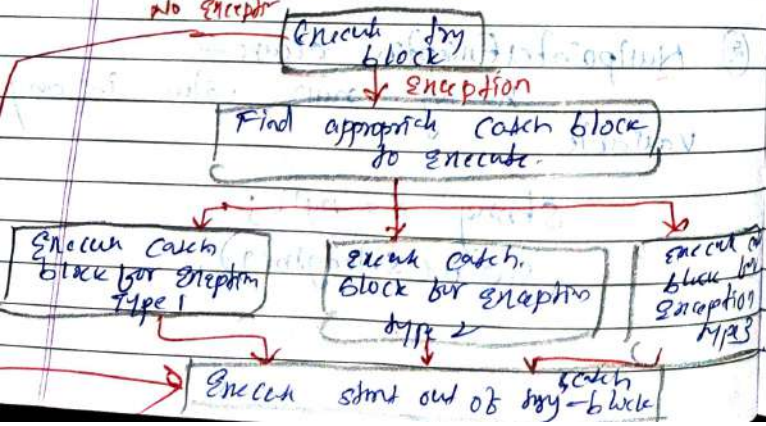
```
String s = "abc";
int i = Integer.parseInt(s);
```

4) ArrayIndexoutOfBoundsException — when an array exceeds to its size, this kind of exception occurs.

```
int a[] = new int[5];
a[10] = 50;
```

* Java catch Multiple Exceptions

A try block can be followed by one / more catch blocks. Each catch block must contain a diff exception handler. so, if you want to perform diff tasks at the occurrence of diff exceptions, use Java multi-catch blocks



ex

```
int a[] = new int[5];
s.o.p(a[10]);
```

o/p

ArrayIndexoutOfBound exception occurs

rest of code

```
public class MultiCatchBlock1
{
  psvm(String[] args)
  {
    try
    {
      int a[] = new int[5];
      a[5] = 30/0;
    }
    catch(ArithmeticException e)
    {
      s.o.p("Arithmetic exception occurs");
    }
    catch(ArrayIndexoutOfBoundException e)
    {
      s.o.p("ArrayIndexoutOf ---- occurs");
    }
    catch(Exception e)
    {
      s.o.p("parent exception occurs");
    }
    s.o.p("rest of code");
  }
}
```

o/p

Arithmetic exception occurs
rest of code

Java Nested try Block

In Java, using a try block inside another try block is permitted. It is called as nested try block.

Syntax -

```
...
// main try block
try
{   stmt 1;
    stmt 2;
    // another try block inside try block
    try
    {
     stmt 3;
     stmt 4;
     // try catch block within nested try block
     try
     {   stmt 5;
         stmt 6;
     }
     catch (Exception e2)
     {
       // exception msg
     }
    }
    catch (Exception e1)
    {
      // exception msg
    }
}
catch (Exception e3)
{
   // exception msg
}
.....
```

Example program

```
public class NestedTryBlock
{
 public static void main (String args[])
 {
   // outer try block
   try {
   // inner try block 1
   try {
     S.o.p ("going to divide by 0");
     int b = 39/0;
   }
   // catch block of inner try block 1
   catch (ArithmeticException e)
   {
     S.o.p(e);
   }
   // inner try block 2
   try
   {
     int a[] = new int[5];
     // assigning value out of array's bound
     a[5] = 4;
   }
```

outer try

```
// catch block of inner try block 2
catch (ArrayIndexOutOfBoundException e)
{
    S.O.p(e);
}
S.O.p("i other statement");
}
// catch block of outer try block
catch (Exception e)
{
S.O.p("handled the exception (outer catch)");
}
S.O.p("normal flow");
}
}
```

throw

- It is a keyword used to create an exception and throws it explicitly
- Throw our own exception using 'Throw' keyword
- The flow of execution stops immediately after the throw statement and any subsequent statements are not executed

Syntax

```
throw new ExceptionType("content");
```

Keyword — operator — Any type of exception name

Ex -

```
import java.lang.*
class Throw
{
 p svm (String args[])
{
try
{
throw new ArithmeticException("This is my own exception");
} S.O.p ("This is try block");
catch (ArithmeticException e)
{
  S.O.p (e.getMessage());
}
S.O.p ("program over");
}
}
```

(class, my own exception, object)

O/p

This is my own exception
program over.

// this keyword is used to create our own exception and throw it explicitly.

throws

- It is a keyword, it contains the list the type of exceptions that a method may throw.

- Throws keyword is used when doesnot want to handle the exception and try to send the exception to the Jvm or called

Syntax

returntype Methodname( ) throws Exceptionlist
{
==
}.

Ex:-

```
import java.lang.*;
class ThrowsExample
{
  static void method() throws ArithmeticException
  {
    int x=10;
    int y=0;
    int res = x/y;
    S.o.p(res);
  }
  p s v m (string args[])
  {
    try
    {
      chitti();
    }
    catch(ArithmeticException e)
    {
      S.o.p("Division by zero");
    }
  }
}
```

method throws exception is is handled by caller

O/P

ArithmeticException is raised in main division by zero.

Throw	V/s	Throws
1) used to throw an exception by object explicitly		used to declare an exception by pass to the caller/jvm

2) present inside method body
void method()
{
throw new ArithmeticException("Hello");
}
		always with method signature.
		void method() throws ArithmeticException
		{
		==
		}.

3) throw can handle 1 exception at a time		we can handle multiple exceptions through throws - checked & unchecked both can handle.
4) checked exceptions cannot handle		

finally

It is a keyword used to create a block of code that will be executing after a try/catch block

whether the exception is handle or not.

- If an exception occurs then the catch block executes after that finally block will be executed.
- If there is no exception also finally block must be executed.

Syntax

```
try
{ // doubtful code
}
catch(e)
{ // handling code
}
finally
{ // executable code
}
```

Ex

```
import java.lang.*;
class FinallyExample
{
  psvm(String args[])
  {
    try
    { int x, y, res;
      x=10;
      y=0;
      res = x/y; // division by zero
      S.o.p(res);
    }
    catch(ArithmeticException e)
    {
      S.o.p("Division by zero");
    }
    finally
    {
      S.o.p("this is finally block");
    }
    S.o.p("program ends");
  }
}
```

o/p

Division by zero
This is finally block
program ends.

* Built-in Exceptions in Java

Java defines several Exception class inside the standard package java.lang.

The most general of these Exceptions are sub classed of the standard type RuntimeException.

Since java.lang is implicitly imported into all java programs, most exceptions derived from RuntimeException are automatically available.

Types of Built-in Exceptions
- checked (compile time)
- unchecked (run time)

Java defines several other types of Exception that relate to its various class libraries.
Following is the list of Java unchecked & checked Runtime Exceptions.

- unchecked exceptions

(1) ArithmeticException (unch)
Arithmetic error, such as divide by zero.

(2) ArrayIndexOutOfBoundsException. (unch)
Array index out-of bound range.

(3) ArrayStoreException —
Assignment to an array element of an incompatible type.

(4) ClassCastException —
Invalid cast

(5) IllegalArgumentException —
Illegal argument used to invoke a method.

(6) IllegalMonitorStateException —
Illegal monitor operation, such as waiting on an unlocked thread.

7) IllegalStateException —
Environment / application in incorrect state

8) IndexOutOfBoundException —
Some type of index is out-of-bounds

9) NegativeArraySizeException —
Array created with a negative size.

10) NullPointer

11) NumberFormat

12) SecurityException.

Checked Exceptions (Java.lang)

1) ClassNotFoundException
2) CloneNotSupported ,,
3) IllegalAccessException.
4) InstantiationException
5) Interrupted Exception
6) NoSuchFieldException
7) NoSuchMethod Exception.

Ex checked
// InterruptedException.

```
class Test
{
  psvm (String args[])
  {
    Thread t = new Thread();
    t.sleep(1000);
  }
}.
```

Ex: unchecked

// Null pointer Exception

```
Class Nullpointe
{
public static void main (string args[])
{
   try {
      string a = null; //null value
      S.o.p(a.charAt(0)); }
   catch (NullpointeException e)
   {
      S.o.p ("Nullpointe Exception");
   }
}
}
```

* User-defind Exception

In this, user is going to raise an Exception and throw it to the program which are basically derived from class exception

Creating our own exception is known as a "custom exception" (or) "user-defined Exception".

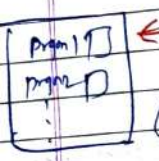
Ex:

```
class InvalidException extends Exception
{
   InvalidAgeException (string msg)
   {
      S.o.p(msg);
   }
}
class Test
{
   public static void main (string args[])
   {
      try {
         vote(12);
      }
      catch (Exception e)
      {
         S.o.p(e);
      }
}
```

* Multithreading

Multitasking → performing multiple tasks simultaneously.

Two ways to perform multitasking
- → process-based (more than 1 program run at a time)
- → Thread-based. (more than 1 thread run at a time)
  - ↳ (Multithreading)

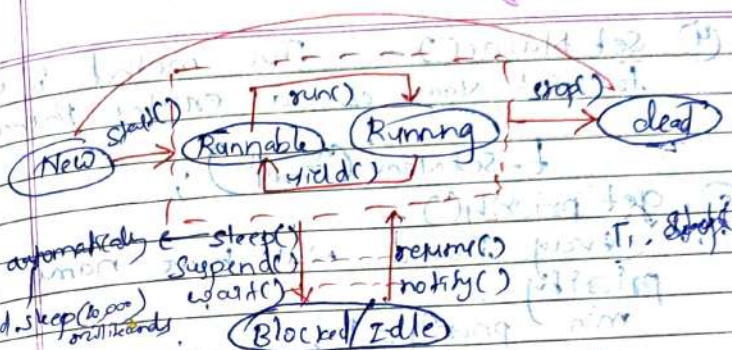


Thread —
- Some part of a program
- sequence of instruction which run independently



Executing more than 1 thread at a time called "Multithreading" (within a same program)

Thread life cycle [Thread model]
1) New.
2) Runnable
3) Running
4) Blocked
5) Stop.



* Thread Methods

(1) start() — This method is used to executing of a thread.

```
Thread t = new Thread();
t.start();
```

by using Thread class object t is created then object only call start method.

~~Thread t = me.~~

(2) run() — After executing start method it will implicitly call the run method.

```
public void run()
{
    //code.
}
```

(3) getName() — This method is used to get name of a thread.

```
S.O.p(t.getName()); // Thread
```

After creating thread display the thread thread name.

(4) **Set Name()** – This method is used to set name of a existing thread

```
t.setName("Thread1");
```

(5) **get priority()** –
Every thread will have name & priority
min priority = 1
Normally = 5 (always recommended)
Max = 10.

(6) **Set priority()** –
This method is called to set the priority of a existing thread

```
t.setpriority(10);
S.O.p(t.getpriority());
```

(7) **Sleep(milliseconds)** –
If we want to block/suspend thread for some time then use this sleep method in terms of milliseconds
It is a static method

```
t.sleep(2000);
```

Always use sleep() method in try catch blocks

(8) **suspend()** – suspending thread for certain time. If you suspend thread then after to resume it resume() method should call.

(9) resume() – suspended thread will resume
(10) wait() –
(11) notify() –
(12) notifyall()
(13) yield() – To send a thread from running to runnable then use this method
(14) Join()

```
t1.start() - 1,2,3
t1.join() 3x
t2.start() - 1,2,3
```

After finishing t_1 thread only if we want to start t_2 thread then use this method.
By this method - 2 threads cannot start run simultaneously.
(15) stop()
After executing all threads in certain program then to stop the process call stop() method
(16) isAlive()
To know the thread status used is alive method().

Thread creation example:

```
class Thread1 extends Thread
{
  public void run()
  {
    S.O.p("Thread1 started");
```

```
for (int i=1 ; i<=20 ; i++)
{
    S.O.p ("Thread1 : " + i);
    if (i==5)
        stop();
}
}

class Thread2 extends Thread
{
    public void run()
    {
        S.O.p ("Thread2 started");
        for (int i=1 ; i<=20 ; i++)
        {
            S.O.p ("Thread2 :" + i);
            if (i==10)
                yield();
        }
    }
}
class example
{
    p s v m (String args[])
    {
        Thread1 t1 = new Thread1();
        Thread2 t2 = new Thread2();
        t1.start();
        t2.start();
    }
}
```

Difference between Thread based and process based Multitasking.

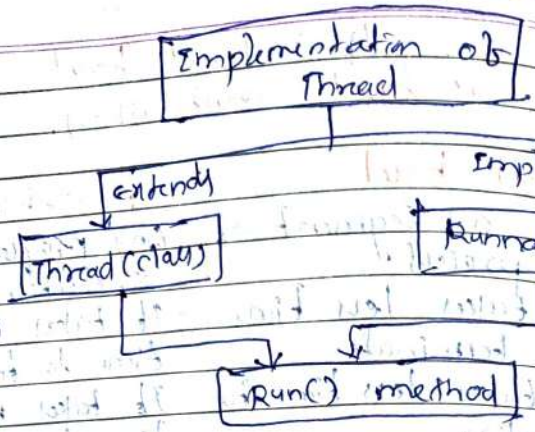
Thread based	process-based
1) It is a segment of a process	- Any program in Execution
2) It takes less time to terminate	- It takes more time to terminate
3) It takes less time to creation	- It takes more time to creation
4) More efficient in term of communication	- less efficient in term of comm.
5) share memory	- process is isolated
6) Light-weight (shares data, code & resources)	- Heavy-weight process
7) No system call is involved	- System call is involved in it.
8) It shares data with each other	- It doesn't share data with each other.

* Creating a thread:

Java defines 2 ways by which a thread can be created

i) By Implementing Runnable interface

ii) By extending Thread class



i) Implementing the Runnable Interface:
The easiest way to create a thread is to create a class that implements runnable interface. The class needs to implement the run() method (ie, public void run()).

Ex
```
class T1 implements Runnable
{
  public void run()
  {
    S.o.pln("Concurrent thread started running...");
  }
  p s v m (String args[])
  {
    T1 obj = new T1();
    Thread t = new Thread(obj);
    t.start();
  }
}
```

o/p
Concurrent thread started running.

ii) Extending Thread class:
In this method to create a thread, we have to create a new class that extends Thread and then to create an instance of that class.

Ex
```
class T2 extends Thread
{
  public void run()
  {
    S.o.p("one thread created");
  }
  p s v m (String args[])
  {
    T2 obj1 = new T2();
    obj1.start();
  }
}
```

o/p
one thread created

* Thread priority

In Java, each thread as its priority represented between 1 & 10. If not also Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method (Thread priority)

→ public final int getpriority()
The Java.lang.Thread.getpriority() method returns the priority of a given thread

→ public final void setpriority()
The Java.lang.Thread.setpriority() method updates (or) assigns the priority of the existing thread.
It throws an IllegalArgumentException if the value newpriority goes out of the range, min = 1 & max = 10.

3 constants defined

public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY

Example

```
import java.lang.*;

public class Threadpriority extends Thread
{
public void run()
{
 S.o.p ("inside run() method");
}
puble s v m (string args[])
{
 Threadpriority t1 = new Threadpriority();
 Threadpriority t2 = new Threadpriority();
 Threadpriority t3 = new Threadpriority();
 S.o.p ("priority of thread t1:" +
          t1.getpriority());
 S.o.p ("priority of thread t2:" + t2.getpriority());
 S.o.p ("priority of thread t3:" + t3.getpriority());
  t1. setpriority (6);
  t2. setpriority (3);
  t3. setpriority (9);
 S.o.p ("priority of thread t1:" + t1.getpriority());
 S.o.p ("priority of thread t2:" + t2.getpriority());
 S.o.p ("priority of thread t3:" + t3.getpriority());
 S.o.p ("currently executing the thread:");
          + Thread.currentThread().getName());
 S.o.p ("priority of main thread is:" +
          Thread.currentThread().getpriority());
 Thread.currentThread().setpriority(10);
 S.o.p ("priority of main thread is:" +
          Thread.currentThread().getpriority());
}
}
```

o/p

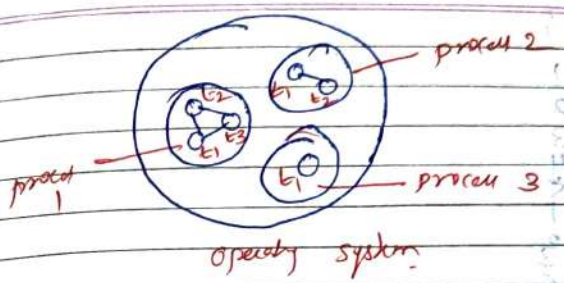
priority of Thread t_1 : 5
priority of Thread t_2 : 5
" " " t_3 : 5
priority of Thread t_1 : 6
" " t_2 : 8
" " t_3 : 9
Currently executing the thread : main
priority of main thread is : 5
priority of main thread is : 10

* Creating Multiple Threads

Creating more than one thread to perform multiple tasks is called "multithreading in Java".

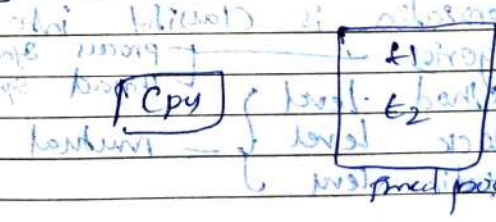
In multiple threading programming multiple threads are executed simultaneously that improving the performance of cpu because cpu is not idle if other threads are waiting to get some resources.

Multiple threads share same address space in the heap memory. It is good to create multiple threads to execute multiple tasks rather than creating multiple processes.



Ex

```
class MyThread extends Thread
{
  public void run()
  {
    for(int i=1; i<10; i++)
    {
      System.out.println(i);
    }
  }
}
public class ThreadDemo1()
{ psvm(String args[])
  {MyThread t1= new MyThread();
   MyThread t2 = new MyThread();
   t1.start();
   t2.start();
  }
}
```



O/P

1
2
3
4
5
1
8
3
6
4
5
6

* synchronization of Threads:

Synchronization is a technique through which we can control multiple threads or among the no. of threads only one thread will enter into the synchronized area.

The main purpose is to overcome the problem of multithreading which multiple threads are trying to access same resource at same time on that situation it may provide some wrong result.

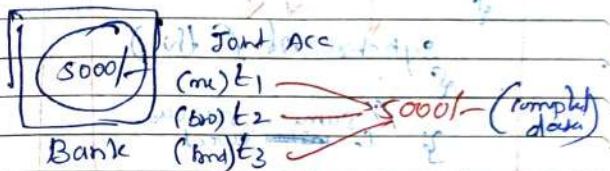
Synchronization is classified into two categories:
- process synchronization
- thread synchronization

Thread synchronization:
- i) method level
- ii) block level
- iii) static level

— mutual exclusive

eg: method level synchronization

```
public void run()
{
    // same resource
}
t1.start();
t2.start();
t3.start();
```



If t1, t2 & t3 wants to debit the amount at same time then we should have 15,000/- to 02 each thread should receive 5000/- but it is not possible so it shows the wrong result.

To overcome this multithreading concept we introduced synchronization

```
public synchronized void run()
{
    same resource
}
t1.start()
t2.start()
t3.start()
```

synchronized keyword will allow only 1 thread at a time

In synchronized block only one thread is allowed at a time only it allows another thread if t1 withdraw 5000/- then t2 & t3 cannot access that resource ∴ it shows insufficient balance

ii) Block level synchronization —

```
public void hotel()
{
  synchronized (this)
  {
    // same resource
  }
}
t1.start();
t2.start();
t3.start();
```

It can be used to perform synchronization on any specific resource of the method.
synchronized block is used to block of one object for any shared resource.

iii) Static synchronization

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

Why synchronization
→ To prevent thread interface
→ To prevent consistency problem.

contin... static synchronized

Eg — Static synch —

```
class Thsample
{
  synchronized static void print(int n)
  {
    for (int i=1; i<=5; i++)
    {
      S.o.p(n+i);
      try
      {
        Thread.sleep(200);
      }
      catch (Exception e)
      {
      }
    }
  }

class Thread1 extends Thread
{
  public void run()
  {
    Thsample.print(5);
  }
}

class Thread2 extends Thread
{
  public void run()
```

5,10

5+1	6	T1
2	7	
3	8	
4	9	
5	10	
10+1	11	T2
2	12	
3	13	
4	14	
5	15	

∴ may more than one thread be accessed the its static

```
    {
      Thsample.print(10);
    }
}
public class Example
{
  public static void main(String args[])
  {
    Thread1 t1 = new Thread1();
    Thread2 t2 = new Thread2();
    t1.start();
    t2.start();
  }
}
```

O/p

```
6
7
8
9
10
11
12
13
14
15
```

(t1 sleep, t2 starts)

Ex synchronization - method

```
class Thsample
{
  synchronized void print(int n)
  {
    for (int i=1; i<=5; i++)
    {
      S.o.p(i+n);
      try
      {
        Thread.sleep(200);
      }
      catch (exception e)
      {
      }
    }
}

class Thread1 extends Thread
{
  Thsample s;
  Thread1(Thsample s);
  {
    this.s = s;
  }
  public void run()
  {
    s.print(10);
  }
}

class Thread2 extends Thread
{
  Thsample s;
  Thread2(Thsample s)
  {
    this.s = s;
  }
  public void run()
  {
    s.print(100);
  }
}
```

```
public class Test
{
  psvm (String args[])
  {
    Thsample TS = new Thsample();
    Thread1 t1 = new Thread1(TS);
    Thread2 t2 = new Thread2(TS);
    t1.start();
    t2.start();
  }
}
```

O/p

```
11
12
13
14
15
101
102
103
104
105 ...
```

Ex - Synchronized Block ~~Method~~

```
class Thsample
{
  Synchronized void print(int n)
  {
    for (int i=1; i<=5; i++)
    {
      S.o.p(i+n);
      try
```

Ex - Synchronization - Block

```
class sample
{
  void print(int n)
  {
    synchronized (this)
    {
      for (int i=1; i<=5; i++)
      {
        S.o.p(n*i);
        try
        { Thread.sleep(200);
        }
        catch (exception e)
        {
        }
      }
    }
  }
}
```

```
class Thread1 extends Thread
{
  Sample s;
  Thread1 (Sample s)
  {
    this.s = s;
  }
  public void run()
  {
    s.print(5);
  }
}
class Thred2 extends Thread
{
  Sample s;
  Thread2 (Sample s)
  {
```

```
        this.s = s;
    }
    public void run() {
        s.print(10);
    }
}
public class Example
{
    public static void main(String args[])
    {
        Sample s = new Sample();
        Thread1 t1 = new Thread1(s);
        Thread2 t2 = new Thread2(s);
        t1.start();
        t2.start();
    }
}
```

o/p

```
5
10
15
20
25
10
20
30
40
50
```

Inter-thread communication:-

Inter-thread communication is nothing but 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 in the same critical section to be executed by following methods of object class.

- wait()
- notify()
- notifyall()

} - java.lang package

wait()

This method in Java notify 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 throws InterruptedException

various forms of wait() method :-

→ public final void wait()

→ public final void wait(long ms) throws InterruptedException.

→ public final void wait(long milliseconds, long nanosecond) throws InterruptedException.

notify() method –

This method wake up a single thread that called wait() on the same object.
If more than one thread is waiting it will awake any 1 of them.

syntax –

```
public final void notify()
```

notifyAll() method –

This method is used to wake up all threads that called wait() method on the same object.
The thread having highest priority will run fast

syntax –

```
public final void notifyAll()
```

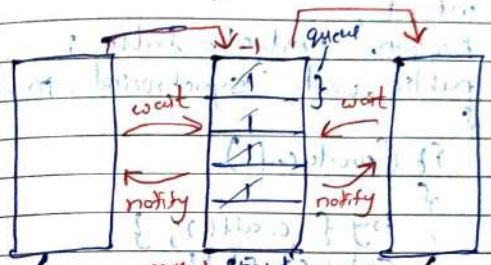
* used to allow synchronized threads to communicate with each other.

Inter thread communication

Example

* producer – consumer problem

It is commonly used in multi-process synchronization



– producer will produce items and add into shared buffer pool
– consumer will consume items from buffer pool
– producer can't add data into buffer if it is full
– consumer ~~will~~ won't consume data from pool if it is empty

if it is empty (shared pool)
↓
reaches → 1

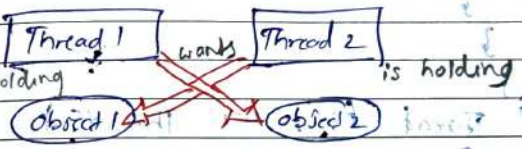
if it is full → reaches max-1

So this problem can be solved by producer & consumer ~~by~~ by applying inter thread communication.

```
class item
{
int i;
boolean produced = false;
public void synchronized produce(int x)
{
if (produced)
{ try { wait(); }
catch() { }
}i=x;
produced = true;
notify();
}
consume()
{
if (!produced)
{
try { wait(); }
catch () { }
}
produced = false;
notify();
}
}
```

* Deadlock in Java

Deadlock is a part of Multithreading. it can occur in a situation when a thread is a waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread



If 2 threads are waiting for each other forever such type of situation is called "deadlock"

Ex

```
public class TestDeadlockEx
{
  psvm(String[] args)
  {
    final String resource1 = "ratan";
    final String resource2 = "vimal";
    Thread t1 = new Thread()
    {
      public void run()
      {
        synchronized(resource1)
        {
          S.o.p("Thread1: resource1 locked");
```

```
try
{
  Thread.sleep(100);
}
catch (exception e)
{ }
synchronized (resource 2)
{
  S.o.p ("Thread 1: locked resource 2");
}
}
};
Thread t2 = new Thread()
{
  public void run()
  {
    synchronized (resource 2)
    {
      S.o.p ("Thread 2: locked resource 2");
      try
      { Thread.sleep(100);
      catch (exception e) { }
      synchronized (resource 1)
      {
        S.o.p ("Thread 2: locked resource 1");
      }
    }
  }
};
t1.start();
t2.start();
}}
```

O/P

Thread 1: locked resource 1
Thread 2: locked resource 2.

How to avoid Deadlock

i) Avoid nested locks
ii) Avoid unnecessary locks
iii) using Thread join

Suspend() & resume() methods in Java

Suspend() — The main purpose of suspend method is to put the thread from running state to waiting state

Ex

```
A t1 = new A();
A t2 = new A();
A t3 = new A();
t1.start();
t2.start();  X
t3.start();
t2.suspend();
```

Resume() — Is used to resume a suspended thread from waiting state to runnable state

Ex `t2.resume();`

Ex

```
class A1 extends Thread
{
  public void run()
  {
    String name = Thread.currentThread().getName();
    for (int i=1; i<=3; i++)
    {
      S.O.p(name);
    }
  }
  public static void main (String[] args) throws
    InterruptedException
  {
    A t1 = new A();
    A t2 = new A();
    A t3 = new A();
    t1.setName("Raj");
    t2.setName("Ram");
    t3.setName("Sam");
    t1.start();
    t2.start();
    t2.suspend(); // pause
    t3.start();
    t2.resume(); // restart.
  }
}
```

O/P

Raj
Sam
Ram

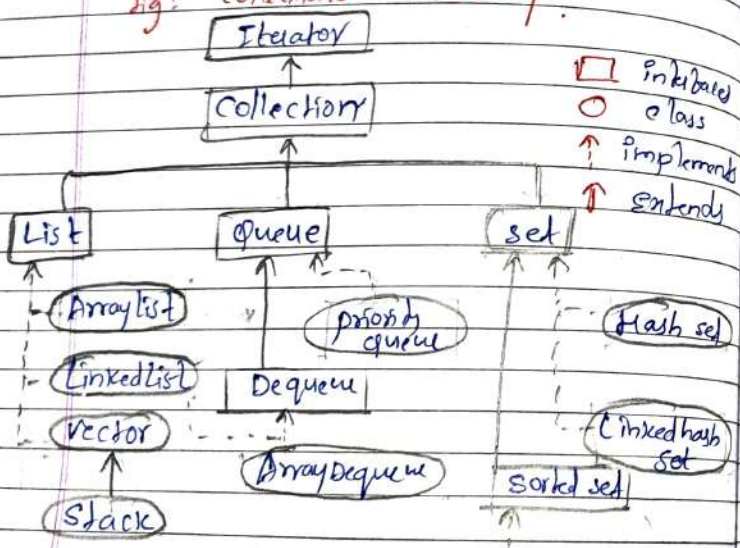
Unit - IV

The collection Framework

Collections overview

Java collections are set of predefined classes & interfaces that helps programmer to perform different kinds of data structures operations like - sorting, searching, traversing, storing & pattern processing data efficiently.

Fig: Collections Hierarchy.



methods declared in collection interface

1) public boolean add(E e)
used to insert an element in this collection

2) public boolean addAll(Collection<? extends E> c)
used to insert the specified collection elements in the invoking collection

3) public boolean remove(Object element)
used to delete an element from the collection

4) public boolean removeAll(Collection<?> c)
used to delete all the elements of the specified collection from the invoking collection

5) public int size()
return the total number of elements in the collection.

6) public void clear()
remove all the total no of elements from the collection

7) public boolean contains(Object element)
It is used to search an element.

public Iterator iterator()
It returns an iterator.

* Iterator Interface –
It provides the facility of iterating the elements in a forward direction only.

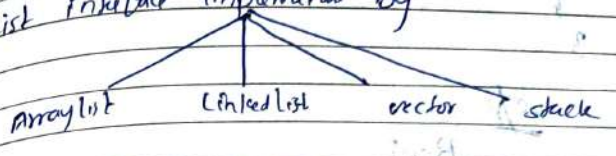
3 – methods

1) public boolean hasNext()
2) public boolean next()
3) public void remove()

* Collection Interface
is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have.

List Interface
it is the child interface of collection interface
it inhibits a list data type structure in which we can store the ordered collection of objects
it can have duplicate values

List interface implemented by




```
List <data_type> list1 = new ArrayList();
List <data type> list2 = new LinkedList();
List <data type> list3 = new vector();
List <data type> list4 = new stack();
```

These are various methods in list interface that can be used to insert, delete and access the elements from the list.

ArrayList

```
import java.util.*;
class TestJavaCollection1
{
  public static void main(String args[])
  {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Ravi");
    list.add("Vijay");
    list.add("Ravi");
    list.add("Ajay");
    Iterator itr = list.iterator();
    while(itr.hasNext())
    {
```

System.out.println(itr.next());
}
}
}

op:
Ravi
Vijay
Ravi
Ajay

① ArrayList
- it implements list interface
- it uses dynamic array to store the duplicate element of different data types.
- maintain insertion order and is not synchronized.
- elements stored in ArrayList can randomly accessed

② LinkedList
- it implements collection interface
- it uses doubly linked list internally to store the elements
- maintains insertion order. it is not synchronized.
- manipulation of data is fast because shifting is required.

ex -
// same as ArrayList except only change in creation of linkedlist after the main function //

LinkedList<String> al = new LinkedList<String>();

③ Vector -
- vector uses dynamic array to store the data elements
- like to arraylist
- it is synchronized and contains many methods that are not part of collection framework.

ex

public class ExampleVector
{
psvm (String args[])
{
Vector<String> v = new Vector<String>();
v.add("Arun");
v.add("Anil");
v.add("Amit");
:
Iterator<String> itr = v.iterator();
while (itr.hasNext())
{
S.O.p (itr.next());
}
}

(4) Stack –

- it is a subclass of vector
- LIFO Data structure
- it contains all methods which class has and also contains boolean
- push(), peek(), ---
- child class of vector

Stack s = new Stack();

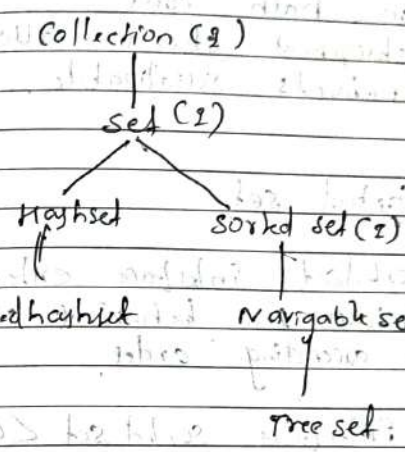
methods –

1) push() (add object)
2) pop() (remove top object)
3) peek() (– return top of Stack)
4) empty()
4) search()

program eg –

```
import java.util.*;
class stackdemo
{
  psvm (String args[])
  {
    Stack s = new Stack();
    S.push("A");
    S.push("B");
    S.push("C");
    S.o.p(S);                    // [A, B, C]
    S.o.p(S.search("A"));        // [3]
    S.o.p(S.search("z"));        // [-1]
  }
}
```

Hash set



Set

- Duplication are not allowed
- Insertion order is not maintained
- set is child interface of collection
- set doesn't contain any new methods so we have to use collection interface methods only.

i) Hashset

- Hashtable (underlined data structure for hash set)
- Duplicate not allowed.

- If we want to add duplicate object through add() method it just returns false.
- Insertion order is not preserved
- all objects are inserted based on hash code
- Heterogenous objects allowed
- implements serializable.

Sorted set

- SortedSet interface extends Set & declares the behaviour of a set sorted in ascending order.

interface sorted set <E>

E → specifies the type of objects that the set will hold

methods -

1) E First() - returns the first element in sorted set.
2) SortedSet <E> headset (E end) - returns sorted set containing those elements less than end that are contained in invoked set
3) E last() - returns the last element in the sorted set.

Unit-V

GUI PROGRAMMING

Called.

GUI

(command user interface)

(limitations overcome by)

GUI (graphical user interface)

→ The Abstract Window Toolkit (AWT) was Java's first GUI framework.

→ AWT classes are contained in the java.awt package.

- user friendly interface (GUI main role) to creat window based applications

→ "AWT is the first library Java introduced for giving GUI"

After AWT — java.awt

↓

swings — java.swing

↓

Applet — java.applet

AWT

→ Java's first GUI framework.

→ Awt classes are contained in the java.awt package.

- All the classes are components

The individual components — (Button, label, checkbox, choice, list...)

- Awt is used to create containers

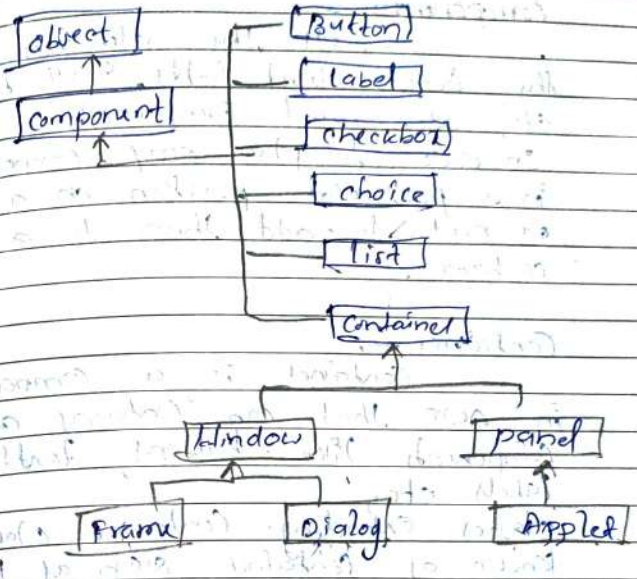


fig: Hierarchy for Java AWT.

Awt

- is a platform-dependent API to develop GUI or window based applications in Java
- It was developed by Sun microsystem in 1995
- It is heavy-weight in use
- It contains large no. of classes and methods used for creating & managing GUI.
- AWT provides various components like button, label, checkbox, etc, used as objects inside a Java program

Components —
All the elements like the button, text fields, scroll bars, etc, are called components. In order to place every component in a particular position on a screen we need to add them to a container.

Container —
Container is a component in AWT that can contain another components like buttons, textfields, labels etc.
classes extending Container class are known as container such as Frame, Dialog and panel.

Types of containers
There are 4 types of containers in Java AWT:
1) window
2) panel
3) Frame
4) Dialog.

1) Window —
Window is a container that have no borders and menu bars, you must use frame, dialog or another window for creating a window we need to create instance of window class to create this container

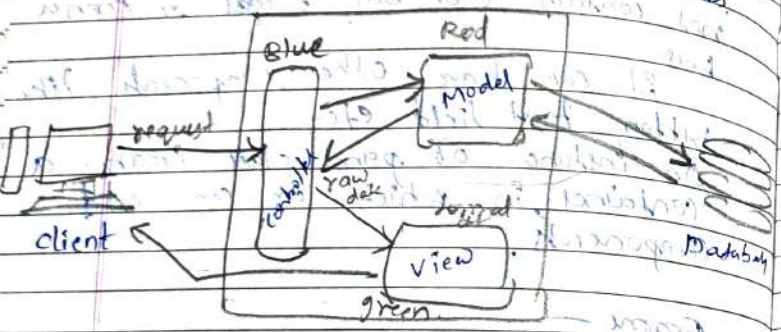
2) panel —
panel is the container that does not contain title bar, border or menu bar.
It can have other components like button, text field etc.
An instance of panel class creates a container, in which we can add components.

3) Frame —
Frame is the container that contains title bar and border and can have menu bars.



* MVC Architecture

- MVC stands for Model view controller
- It is an Architectural pattern
- MVC divides a s/w application in 3 parts - model, view & controller.



Model - Interacts with database
Executes business logic
→ its a java class
→ used by programmers/developers.
- backend database

view - what user sees on the screen
generated UI for the user
- JSP page.
- uses components like pie charts, bar graphs, tables etc
- shows data

Controller - Takes user i/p (request params)
Interacts with model & view.
- Servlet / Servlet filter.
- Interface bet" model & view.

- MVC is one of the most frequently used industry standard web development framework to create scalable & extensible projects.

* Layout Managers in Java

The Layout managers enable us to control the way in which visual components are arranged in our GUI forms by determining the size and position of components within the containers.

Types

① FlowLayout
It is a simple layout manager that arranges components in a row, left to right, wrapping to next line as needed

② BorderLayout
BorderLayout divides the container into 5 regions: North, south, east, west and center. Components can be added to these regions, and they will occupy the available space accordingly.

③ GridLayout
GridLayout arranges components in a grid with specified no. of rows & columns.

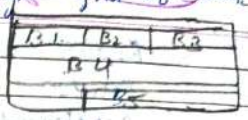
each cell in the grid can hold a component.
it is used in creating a uniform grid of components such as calculator / game board.

④ Card layout —
Cardlayout allows components to be stacked on top of each other like a deck of cards, only one component is visible at a time, and you can switch between components using methods like next() and previous().
This card is useful in creating wizards or multi step processes.

⑤ Groupplayout :—
It a resizable & complex layout manager that provides precise control over the positioning and sizing of components.
It arranges components in a hierarchical manner using groups.
This is mostly used in GUI building like the one in NetBeans IDE.

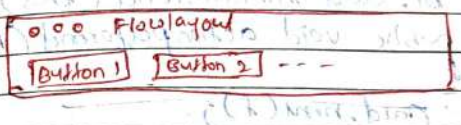
⑥ GridBagLayout —
This is powerful layout manager allows you to create complex layouts by specifying constraints for each component.

(6) cont:
— it is used to align components horizontally & vertically along their base line
— components may not be of same size
— it maintains dynamic rectangular grid of cells

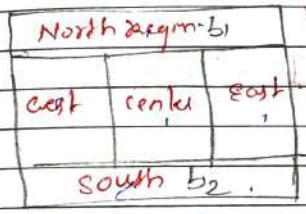


① Flowlayout —
— it is default layout

```
Frame f = new Frame("Flowlayout");
f.setLayout(new FlowLayout());
f.add(new Button("Button 1"));
f.add(new Button("Button 2"));
```



② Borderlayout —
This layout has 5 regions
Each region can contain only one component

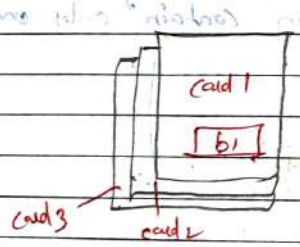



```
Frame f = new Frame("layout");
f.setLayout(new BorderLayout());
f.add(b1, "North");
f.add(b2, "South");
```

③ Cardlayout —

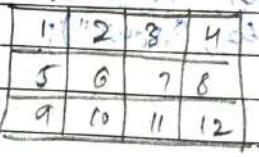
```
Frame f = new frame();
Button b1, b2;
CardLayout card;
card = new CardLayout (40, 30);
f.setSize
{
f.add ("card1", b1);
f.add ("card2", b2);
b1.add ActionListener (this);
public void actionperformed (ActionEvent ae)
{
card.next(f);
}
```

used to display next card.



(4) Grid layout :-

3 rows, 4 columns



f.setLayout (new GridLayout(3,2));

f.add(b1);
:
f.add(b6);

b1	b2
b3	b4
b5	b6

* AWT - creating Frame.

2-ways to creat AWT frame

1) By creating Frame object

Ex Frame f = new Frame();

2) By Extending Frame class

eg class Myframe extends Frame
{
Myframe()
{
}
}

Default setting of Frame.

- created Frame is invisible
  f.setVisibile(true);
- can't close.
  ctrl+C to close.
- Size of frame is 0,0.

f.setSize(500,300)

- No title
  f.setTitle("My Frame");
- Background colour is white.

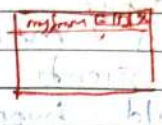
① Creating frame using object creation

```
import java.awt.*;
public class MyFrame
{
    public static void main(String args[])
    {
        Frame f = new Frame();
        f.setVisible(true);
        f.setSize(500,300);
        f.setTitle("My Frame");
    }
}
```

② Creating a frame by extending frame class

```
import java.awt.*;
public class myframe extends Frame
{
    myframe()
    {
        this.setVisible(true);
        this.setSize(500,300);
        this.setTitle("My Frame");
    }
}
```

O/P



* Event Handling

Event — changing the state of an object (or) behaviour by performing actions.

Ex:- click on button
Changing mouse

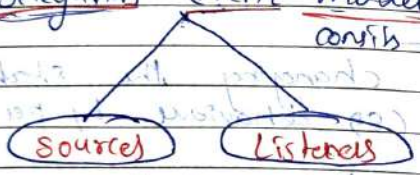
"java.awt.event" — package provides various event classes.

Classification of Events
- Foreground Events
- Background Events

1) Foreground Events — require user interaction to generate
Ex — clicking button, Scroll bar, mouse/cursor movements

ii) Background event →
Events that don't require interaction of users to generate
Ex- OS failures / interrupts ...

Event Handling
It is a mechanism to control the events and to decide what should happen after an event occur.
To handle events, Java follow "Delegation Event model" consis of



Sources →
Events are generated from source, The various sources like buttons, checkboxes, list, menu item, choice, scrollbar etc.

Listeners →
Listeners are for handling the events which are generated from the source
Each listener represents an interface

To perform the event Handling we need to register the source with the Listener

Registering the source with Listener
Diff classes provide diff registration methods
Ex
For key event we use addKeyListener() to register
For Action event we use addActionListener() to register.

Event classes in Java — Listener Interfaces
ActionEvent — ActionListener
AdjustmentEvent — AdjustmentListener
ComponentEvent — ComponentListener
ContainerEvent — ContainerListener
(Event classes)

interfaces contain abstract methods need to implemented by registred class to handle Events.

Diff methods defined by interfaces
(1) ActionListener — actionPerformed()

Mouse Events → mouse listener & mouse motion listener

1) Mouse clicked
2) Mouse Entered
3) Mouse Exited
4) Mouse pressed
5) Mouse Released
6) Mouse moved
7) Mouse Dragged

1–5 → mouse listener interface
6–7 → mouse motion listener interface

MouseEvent class
↓
getX() } – coordinates of window.
getY() }

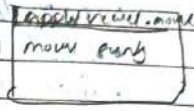
Steps to perform mouse events in Java code

1) import packages — awt
   awt.event
   applet
2) Class should be extends applet implements interfaces
3) Applet code
4) init() — set foreground
   – set background
   – Register listener interface
5) event method definations.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code = mouse width = 600 height
   =500>
</applet> */

public class Mouse extend Applet implements
mouseListener, mouseMotionListener.
{
    int x=30, y=30;
    String msg = "Mouse Events";
    public void init()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
        SetBackground(Color.black);
        SetForeground(color.white);
    }
    public void mouseEntered(MouseEvent
            me)
    {
        SetBackground(Color.magenta);
        showStatus("mouse entered");
        repaint();
    }
    public void mouseExited(MouseEvent
    {
        SetBackground(Color.red);
        showStatus("mouse Exited");
        repaint();
```

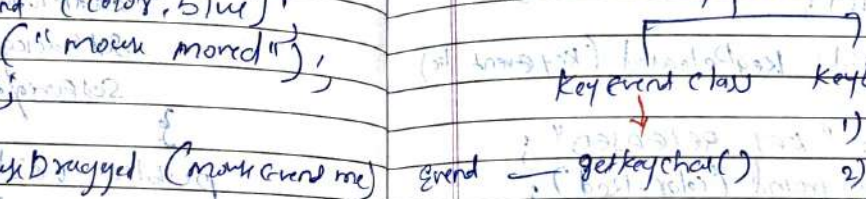
```
public void mousePressed (MouseEvent me)
{
    SetBackground (Color.yellow);
    ShowStatus ("a mouse pressed");
    repaint();
}
public void mouseReleased (MouseEvent me)
{
    SetBackground (Color.green);
    ShowStatus ("mouse Released");
    repaint();
}
public void mouseMoved (MouseEvent me)
{
    X = me.getX();
    Y = me.getY();
    SetBackground (Color.blue);
    ShowStatus ("mouse moved");
    repaint();
}
public void mouseDragged (MouseEvent me)
{
    SetBackground (Color.pink);
    ShowStatus ("mouse Dragged at
    ("+me.getX()+","+me.getY()+")");
    repaint();
}
public void mouseClicked (MouseEvent
(or me)
{
    SetBackground (Color.green)
    ShowStatus ("mouse clicked");
    repaint();
}
}
```



Key Events

The Java KeyListener is notified whenever you change the state of key. It is notified against key event.

The KeyListener port is used in Java AWT to listen for keyboard events, such as key pressed and key releases.



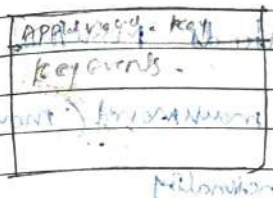
* Handling key events in Java.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/* <applet code = key width = 600 height
    = 500>
</applet> */
```

```
public class Key extends Applet implements
  KeyListener
{
   int x=40, y=40;
   msg = "KEY EVENTS";
   public void init()
   {
      SetBackground (color.black);
      setForeground (color.yellow);
      addKeyListener (this);
   }
   public void keyPressed (KeyEvent ke)
   {
      msg = "KEY PRESSED";
      setBackground (color.pink);
      showStatus ("key pressed");
   } repaint();
   public void keyReleased (KeyEvent ke)
   {
      msg = "KEY RELEASED";
      SetBackground (color.Red);
      showStatus ("key Released");
   } repaint();
   public void keyTyped (KeyEvent ke)
   {
      msg = "Key TYPED is";
      msg += ke.getKeyChar();
      setBackground (color.black);
      showStatus ("KeyTyped");
   } repaint();
```

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

O/p



Adapter classes

- It is used to simplify the process of event handling.
- provide empty implementation of all methods in an event listener interface.
- Defines a new class to act as event listener by extending one of the adapter classes and implementing only those methods that you want to use in your code.

Ex

```
class adapter1 extends Frame
{
   Frame f;
   adapter1()
```

```
{
    f = new Frame(" ");
    f.addMouseMotionListener(new mi());
    f.setVisible(true);
    f.setLayout();
    ---
    ---
}
class mi extends MouseMotionAdapter
{
    public void mouseMoved(MouseEvent m)
    {
        // functionality
    }
}
```

Adapter classes are nothing but an extension of interface for event-handling that contains blank definition of all the methods in the interface.

* Inner class in Java

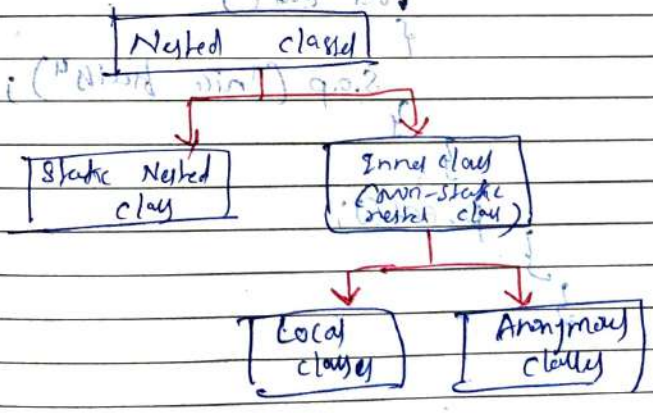
Java inner class (or) nested class is a class that is declared inside the class or interface.

Syntax

```
class Java_outer_class
{
    class Java_inner_class
    {
        // code
    }
}
```

There are two types of nested classes
→ non-static (inner class)
→ static nested

Types	Description
1) member Inner class	A class created within a class & outside method.
2) Anonymous inner class	A class created for implementing an interface / extending class Java compiler decides its name
3) Local Inner class	A class which is created within method
4) Static Nested class	A static class was created within the class.
5) Nested interface	An interface created within class / interface.



Java Anonymous Inner class

Java Anonymous inner class is a inner class without a name and for which only a single object is created.

This class is created in 2 ways
1) class (abstract / concrete)
2) Interface

Eg

```
abstract class person{
  abstract void eat();
}
class TestAnonymousInner
{
  public static void main(String args[])
  {
    person p = new person(){
      void eat()
      {
        S.o.p("nice fruits");
      }
    };
    p.eat();
  }
}
```

O/p

nice fruits.