Green University of Bangladesh

Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Full,Year:2024),B.Sc.in CSE (Day)

KSA 2 (Class Notes)

Course Title: Object Oriented Programming

Course Code: CSE201 Section:232-D4

Student Details

	Name	ID
1.	MD.SHAJALAL	223002088

Submission Date : 11-12-2024

Course Teacher's Name : Md. Parvez Hossain

Project Status	
Marks: ……	Signature:.....
Comments:.....	Date:.....

Class Note Midterm

Note

→ Content

- Overview of Java
- Data types, Variables and Arrays.
- Operators
- Control Statements.

→ Overview of Java ←

What is Java?

An object Oriented Programming language developed at Sun Microsystems.

Java was Conceived by James Gosling, Patrick Naughton, Chris Worth, Ed Frank, and Mike Sheridan at Sun Microsystems.

A set of standardized class libraries (Packages) that support:

i. Creating graphical user interfaces
ii. Communicating over networks
iii. Controlling multimedia data

Features of Java:

- platform independent
- Object Oriented
- Robust
- Secure
- Multi-threaded

Platform Independent

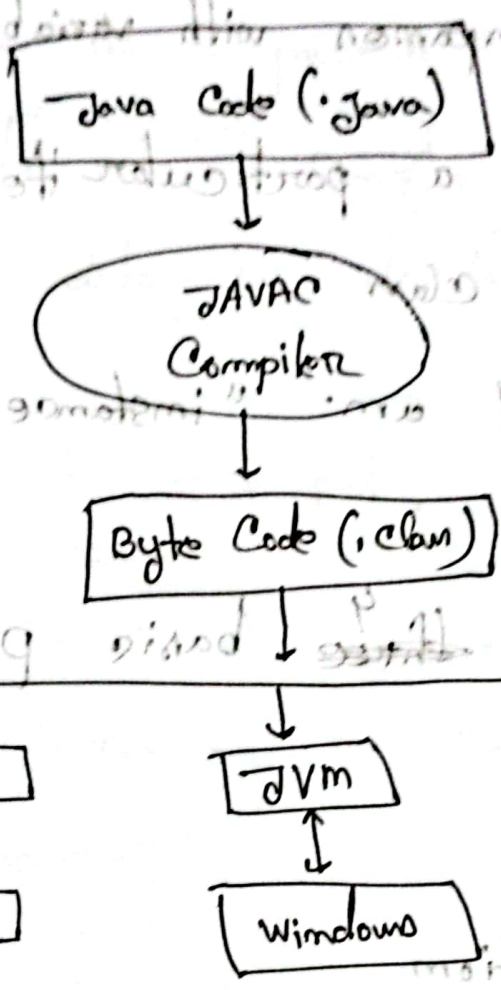
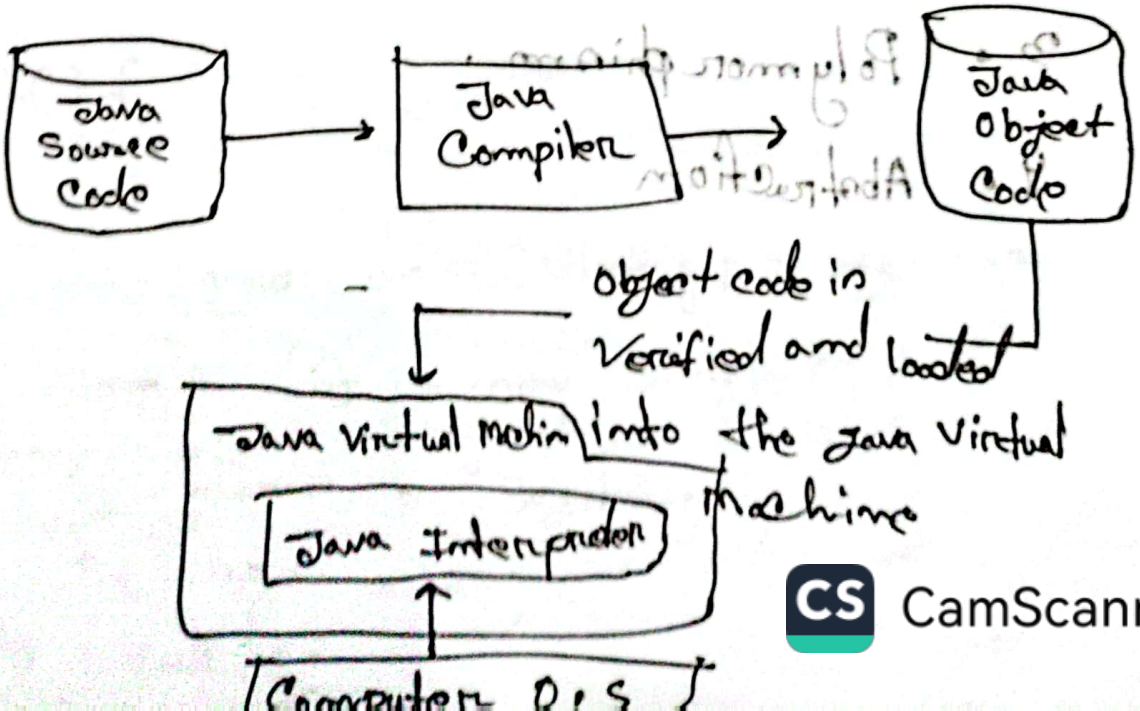


Diagram of JVM



→ "Class" means a Category of things

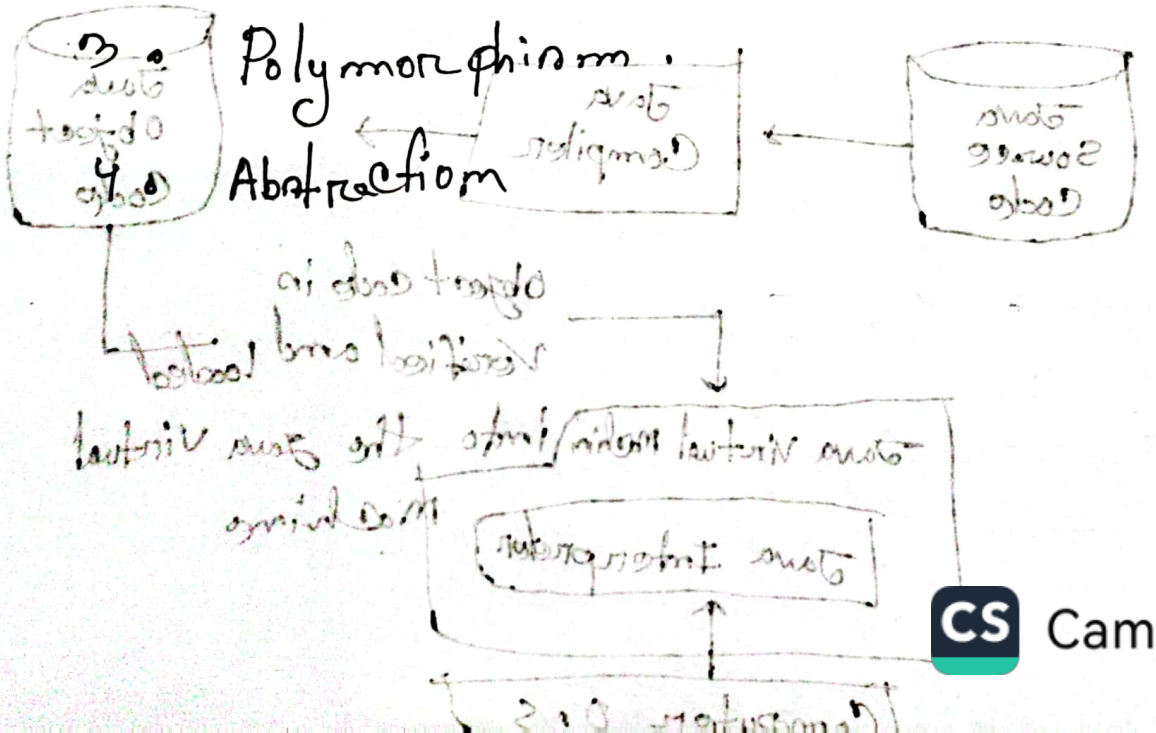
- A Class encompasses with variables and methods.

→ "Object" means a particular item that belongs to a class

- Also Called as "instance"

→ There are ~~three~~ 4 basic principles beyond OOP

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction



→ Simple Java Program

```
package hello;

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

main method must declare

---

public : to get access to other Command Line interface instructions.

Static : to get access without creating any instance.

void : main do not return any value.

→ Data types, Variables and Arrays

The Primitive Types

→ Java defines eight primitive types of data; byte, short, int, long, char, float, double and boolean.

→ These can be put in four groups.

- Integers : byte, short, int, long
  8bit 16bit 32bit 64bit
- Floating point numbers : float, double.
  32bit 64bit
- Characters : char.
  16bit
- Boolean : boolean. value can be just true or false.

Operators

→ Arithmetic Operators

+ +=
- -=
* *=
/ /=
% %=
++ --

→ Bitwise Operators

~ NOT
& AND
| OR
^ XOR
>> SHIFT right
<< SHIFT left

→ Assignment operators

=

→ Logical operators

&& → AND
|| → OR
== → Equal to
!=

→ Relational

<
>
>=
<=

→ Ternary

?:

⟹ Java Array ⟸

→ An array is a collection of variables of the same type.

→ Using array

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

⟹ Declaring array

```
datatype[] arrayName;
```

Example

```
int[] number;
String[] names;
```

⟹ Creating array

- For creating an array you can use new operator

```
int[] number = new int[10]
```

→ • Array Initialization

• int [ ] number = new int [10];

number[0] = 10;
number[1] = 20;
number[2] = 45;

⇒ Array Examp

package array_demo;

public class Array1 {

public static void main (String[] args) {

int[ ] number = new int[5];

number[0] = 10;
number[1] = 12;
number[2] = 15;
number[3] = 200;
number[4] = 90;

System.out.println(number[3]);

}

output

20

```
class Array {
    p s v m (String[] args){
        int[] age = new int[5];
        for (int i = 0; i<5; i++){
            System.out.println(age[i]);
        }
    }
}
```

⟹ Content

- Java Class
- Java Objects.
- Example : Java class and objects
- Java methods
- Java Constructors.

→ Java class and object

- A class is a user defined blueprint or prototype from which objects are created.
- Java class represents the set of properties or methods that are common to all objects of one type.

→ Java Class

```
class ClassName {
    // variables
    // methods
}
```

⟹ Java Object

Class_Name Obj_Name = new Class_Name();

⟹ Java Methods

```
package Stud ;
public Class Student {
    String name ;
    int id ;
void set ( String n , String d , int i ) {
        name = s ;
        id = 2020 ;
    }
void Pri.( ) {
  System . out . print ln ("Name : " + name
```

method

Methods

⟹ Java Constructors

```
public static void main (String[] args) {
    Student S1 = new Student ("Kahkasha", 20);
    S1 = PrintInfo();
}
```

Ex

⟹ Content

- Recursion
- Call by value & Call by Reference.
- Variable - Length Arguments.

⟶ Recursion ⟵

```
public class Fotorial {
    int fact(int m) {
        if (m == 1)
            return 1;
        else
            return m * fact(m-1);
    }

    public static void main(String[] args) {
        Fdorial ob = new Folorial();
        int result = ob.fact(5);
        System.out.println("Featorial = " + result);
    }
}
```

- Recursive Call
- Base Case

→ Call by value & Call by Reference.

Call by value

- In Java, call by value refers to calling a method by paming the value in the parameter.
- In Java, Call by value pamer a copy of the variable to the method, So all changes are reflected only in that method; thus, not changes are reflected in the main method.
- when we pam a variable with primitive data typer. it is considered a call by values in Java, So any changes to the variable will not be reflected in the caller's scope.

Exam

```
public class CallByValue {

    void Change (int i) {
        i = 20;
    }
}

public class Test {
    p s v m (String[] args) {

        CallByValue ob = new CallByValue();
        int x = 10;
        System.out.println("x before call : "+x);
        ob.Change(x);
        System.out.println("x after call : "+x);
    }
}
```

output

10

10

Call by Reference

- If we call a method by passing a reference type data (object, String etc.) then it is known as call-by-reference.
- Changes to that formal parameter does affect the actual parameter.
- In call-by-reference original value gets Change

Example

```
class Reference {
    String name;
    void Chang(Reference R2){
        R2.name = "Rubel";
    }
}
class Test {
```

```
psVm. (s, a ) {
Reference rc1 = new Reference

rc1 . name = "A Sojib";

S . O . P ("before :" + rc1 . name );

rc1 : change (rc1);

S . O . P ("after :" + rc1 . name );

}
```

output

Sojib

Rubel

Variable Length Arguments (Varargs)

→ Variable length arguments → Varargs method

→ A method that takes a variable number of arguments is called a varargs method

Exam

```
public class AddDemo {
    void add (int ... num) {
        int sum = 0;
        for (int x : num) {
            sum = sum + x;
        }
        System.out.println (sum);
    }
public class Test
    p s v m (String[] args)
    AddDemo ob = new AddDemo();
    ob.add (10, 20)
    ob.add (10, 30, 40, 50);
}
```

Bye

1. Encapsulation

→ Encapsulation is a process of Packaging variables and methods into a single unit.

→ Protecting data by declaring them as private.

* Single unit example:

```
Package oop;

public class Single {
    String name;
    int age;
    void eat() {

    }
    void talk() {

    }
}

public static void main(String[] args) {
    Single S1 = new Single();
    S1.name = "Sojib";
    S2.age = 22;
    S1.talk();
}
```

Encapsulation – আরো একটি নিয়ম

Data Private

Data Private এর মানে হলো Private keyword ব্যবহার করা, যেমন

```
Package java;
Public Class Data {

    Private String name;
    Private int ID;

    void eat() {

    }
    void talk() {

    }

p s v m (String[] args) {

    Data S1 = new Data();
    S1.name = "Shayalal";
    S1.ID = 11012;
    S1.talk();

}
```

Private keyword দেওয়া থাকলে সেই ভেরিয়েবল আমি ইচ্ছা মতো ব্যবহার করতে পারবো না

Private থাকা ফাংশন বা void ফাংশন ব্যবহার করতে হলে, method এর মধ্যে Private keyword ব্যবহার করতে হবে,

এইখানে যেকোনো ভেরিয়েবল ব্যবহার করা যাবে না।

Private method দ্বারা ফাংশন ব্যবহার করা যায় না।

get Name() , Set Name() method

Example

```
p.c. Jav {
Private String name;
Private int id;
  Public String getName() {
        return name;
  }
Public void setName(String name) {
    this.nam = name;
  }
p s v m (String[] args) {
    Jav N1 = new Jav();
    N1.SetName("Shajalal");
  System.out.println(N1.getName());
}
```

How to do encapsulation?

1. Declare the variables as private.
2. Provide public setter and getter method to modify and get the variables value.

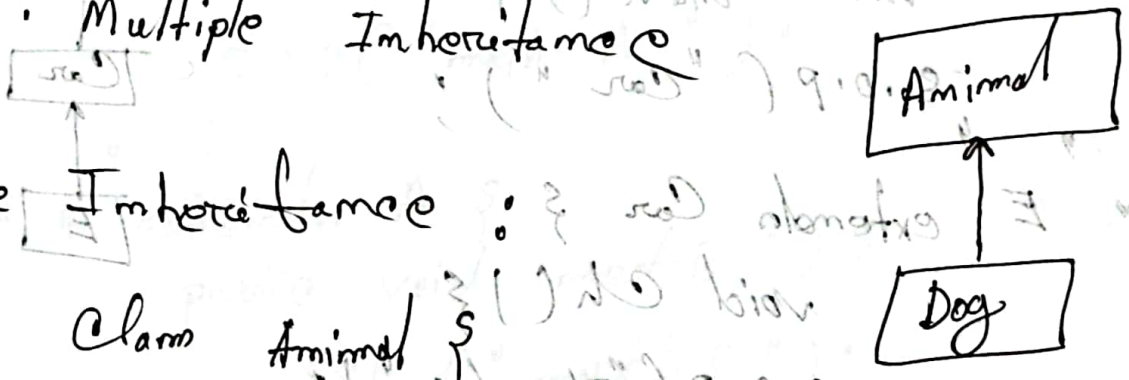
→ Exercise – 1 write a Java Program to Create a Class called Person with private instance variables name, age, and country. Provide public getter and setter methods to access and modify these variables.

```
Class Person {
    Private String name;
    Private int age;
    Private String country;

    Public String getName() {
        return name;
    }
    Public int getAge() {
        return age;
    }
    Public String getCountry() {
        return country;
```

```
Public void setName (String name) {
    this.name = name; }
Public void setAge (int age) {
    this.age = age; }
Public void setCountry (String country) {
    this.country = country;
}
public static void main (String[] args) {
    Person S1 = new Person();
    S1.setName ("Shagalala");
    S1.setAge (22);
    S1.setCountry ("BD");
String name = S1.getName();
int age = S1.getAge();
String country = S1.getCountry();
System.out.println ("Name " + name);
}
}
```

Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance

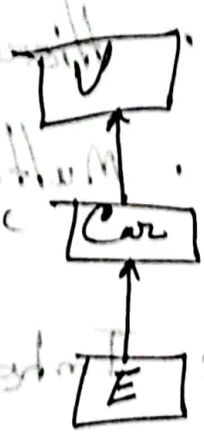


1. Single Inheritance :

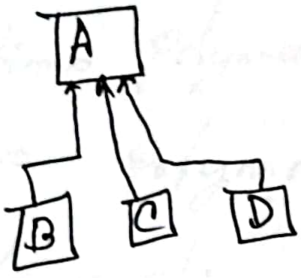
```
class Animal {
    void eat() {
        System.out.println("This animal");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("The dog");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog N = new Dog();
        N.eat()
        N.bark()
    }
}
```

→ Multilevel Inheritance Syntax

```
class V {
    void start() {
    System.out.println("Vehicle");
    }
}
class Car extends V {
    void drive() {
    S.o.p("Car");
    }
}
class E extends Car {
    void ch() {
    S.o.p("Electric");
    }
}
public class Main {
    public static void main(String[] args) {

    E s1 = new E();

    s1.start();
    s1.drive();
    s1.charge();

    }
}
```



→ Hierarchical Inheritance




```
class A {
    public void met() {
        S.O.P("math A");
    }
}
class B extends A {
    public void metB() {
        SOP("math B");
    }
}
class C extends A {
    public void Main() {
        SOP("math C");
    }
}
class D extends A {
    public void N() {
        SOP("Math D");
    }
}
public class Test {
    P S V m (String args[]) {
        B N1 = new B();
        C N2 = new C();
        D N3 = new D();
        N1.met();
        N2.metB();
        N3.Main();
```

2. Inheritance

→ Inheritance is the process by which one object acquires the properties of another object.

→ Inheritance extends keyword

```
public class Animal {
    int age;
    String name;
}
public class Dog extends Animal {
}
```

→ Why do we need inheritance?

- For Code Reusability
- For method overriding
- To implement parent-child relationship

→ In Java, we use the extends keyword to inherit from a class.

Access Modifier

	Class	package	Subclass	World
public	Yes	Yes	Yes	Yes
Private	Yes	No	No	No
Protected	Yes	Yes	Yes	No

Java Method Overriding

→ Declaring a method in subclass which is already present in superclass is known as method overriding.

Java Method overriding -এরমাধ্যমে কোড রিইউজ করা হয়।

- Code reuse
- One interface, multiple implement
- Run time polymorphism.

What are the rules for Method overriding

1. Name, Signature type, parameter must be same.
2. If a method can't be inherited, then it can't be overridden.
3. A method declared as final or static can't be overridden.
4. Constructor can't be overridden.

Exam Method overriding

```
public class Person {
    String name;
    int age;
    void displayInform() {
        System.out.println("Name :" + name);
        System.out.println("Age :" + age);
    }
}
public class Teacher extends Person {
    String qualification;
    @Override
    void displayInform() {
        S.o.p("Name :" + name);
        S.o.p("Age :" + age);
        S.o.p("Qu :" + qualification);
    }
}
Class Test
    P.s.v.m(String[] args) {
        Teacher S1 = new Teacher();
        S1.name = "Parvez";
        S1.age = 27;
```

```
s2.displayInformation();
}
}
```

output

Name : Parvez Hossain
Age : 27
Qu : BSC in CSE

→ Same Question ←

Can we overrided static method ?

Ans : No, Static method can't be overridden, Because Static method is bound to class, on the other hand method is bound to object

Can we overrided Java main method ?

Ans : No, because main is a static method.

Overriding Vs Overloading

- Don't Confuse the Concepts of overloading and overriding
- overloading deals with multiple methods with the same name in the same class, but with different Signatures.
- overloading deals with two methods, one in a Parent Class and one in a Child Class that have the same Signature.
- Overloading lets you define a similar operation in different ways for different data.
- Overriding lets you define a similar operation in different ways for different object types.

Overloading example

```
public class overload {

    void add (int a, int b) {
        System.out.println(a+b);
    }
    void add (int a, int b, int c) {
        System.out.println(a+b+c);
    }
    void add () {
        System.out.println("Nothing to add");
    }
}

class main {
```

⊞ Super Keyword ⊞

→ Super Keyword কোথায় ব্যবহার করা হয়?

Ans: "super" keyword is used to refer immediate super class object.

→ Use of super keyword in Java

1. Use of super with variables
2. methods
3. Constructors

Exam

Variables

```
class Var {
    int number = 140;
}
class Car extends Var {
    int number = 120;
    void display() {
        System.out.println("Max Speed: "
            + super.number);
    }
}
```

```
Class Test {
    public static void main(String[] args)
    {
        Car i$_1$ = new Car();
        new.display();
    }
}
```

output

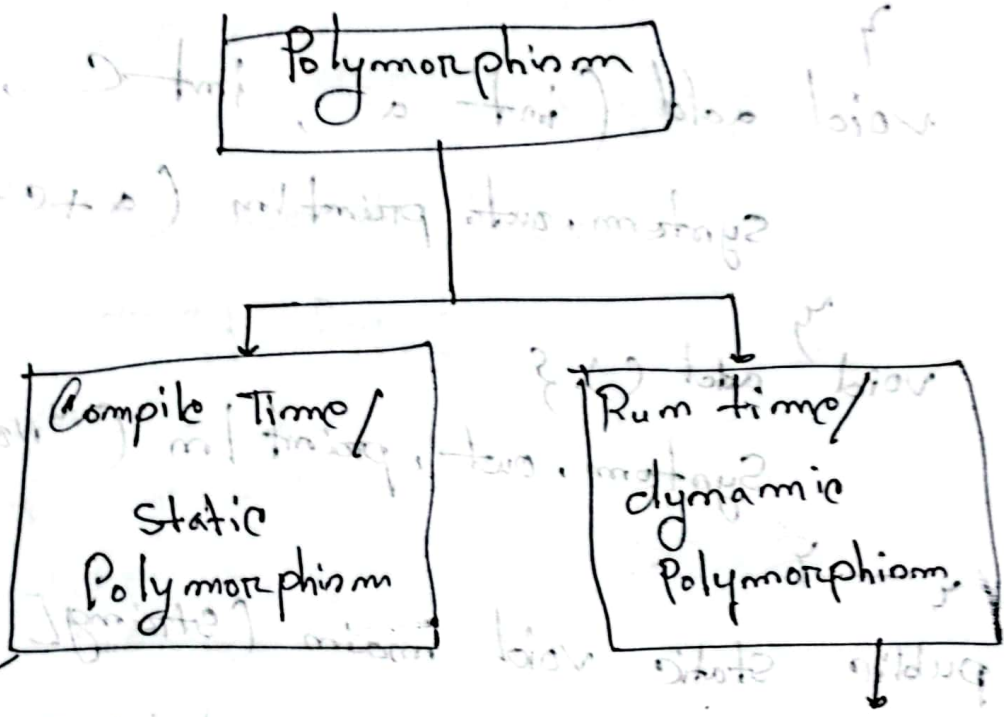
number = 140

Example Methods

```
class Person {
    void mem()
    {
        S.O.P("This in person class");
    }
}
class Student extends Person {
    void mem()
    {
        S.O.P("This.in.Student class");
    }
    void display()
    {
```

3. Polymorphism

→ Polymorphism means ability to have many different form. "one interface, multiple methods".



Method Overloading

Constructor overloading

Method Overriding

→ Polymorphism

1. Compile-time Polymorphism
2. Runtime Polymorphism

1. Compile-time Polymorphism

```
Class → Cal {
    int Add (int a, int b) {
        return a+b;
    }
    int Add (int a, int b, int c) {
        return a+b+c;
    }
    double Add (double a, double b) {
        return a+b;
    }
}
public class Test {
    ps vm (String[] args) {
        Cal c1 = new Cal();
        System.out.println("Sum " + cal.add(2,3));
        System.out.println("Sum2 " + cal.add(1,2,4));
        System.out.println("Sum3 " + cal.add(22.4,1.23));
    }
}
```

1. WAP in Java to print all odd numbers from 1 to n

```
import java.util.Scanner;
public class OddNum{
    public s v m (String[] args){
        Scanner s1 = new Scanner(System.
        S.o.p("Enter n :");
        int n = scanner.nextInt();
        S.O.P("odd num "+n+":");
        for(int i=1; i<=n; i++){
            if(i%2!=0){
                S.o.p(i+" ");
            }
        }
    }
```

2. WAP in Java to find the factorial of a given number

```
import java.util.Scanner;
public class F {
    p s v m (String[] args) {
        Scanner N = new Scanner(System.in);
    S o p ("Enter ");
        int num = scanner.nextInt();
        long factorial = 1;
    for (int i = 1; i <= num; i++) {
        factorial = factorial * i;
    }
System.out.p ("The " + num + " is: " + factorial
```

4. Abstraction

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

⇒ How to achieve Abstraction?

There are two ways to achieve abstraction in Java.

1. Abstract class (0 to 100%)
2. Interface (Achieve 100% abstraction)

Abstract class

- যখন তুমি কোনো class তৈরি করবে

class mob {

- যদি Abstract class তৈরি করি তাহলে class এর আগে Abstract লিখতে হবে।

abstract class mob {

}

Abstract method

→ Non Abstract method:

```
void manage() {
    ------
    ------
}
```

→ Abstract method

```
abstract void manage();
```

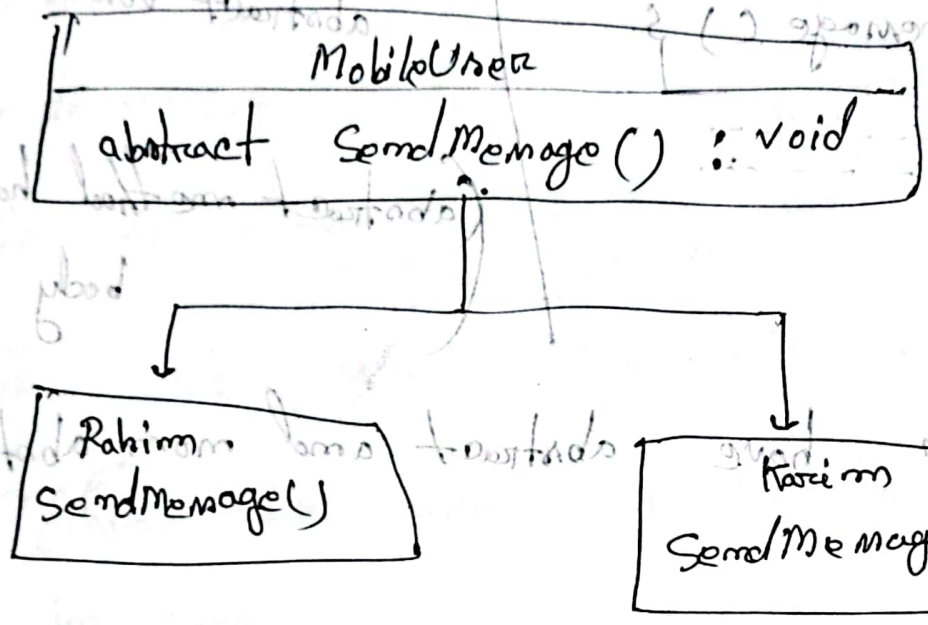
(abstract method has no body)

→ Abstract class have abstract and non abstract method.

Examp

```
abstract class Mobile {
    void call() {
        System.out.println("Hello");
    }
    abstract void sendMem();
}
```

⟹ Abstraction Example ⟸




```
abstract class MobileUser {
    abstract void sendMessage();
}

class Rahim extends MobileUser {
    @Override
    void sendMessage() {
        System.out.println("Hi, this is Rahim");
    }
}

class Karim extends MobileUser {
    @Override
    void sendMessage() {
        S.o.p("Hi, this is Karim");
    }
}
```

```
public Class Test {
    p s v m (String[] args) {
        MobileUser mp;
        mp = new Rahim();
        mp.SendMessage();
        mp = new Karim();
        mp.SendMessage();
    }
}
```

→ You are tasked with developing a Payment processing System for an e-Commerce platform. The System must handle different types of payments, including Credit Card Payments, Bank Transfers, and Digital Wallet Payments. To simplify the process and maintain a clean interface for payment processing, use method overloading to implement multiple versions of a processPayment method in a class called PaymentProcessor. Why would method overloading be more effective in this scenario compared to using different method names?

N.B. Process payments using a credit card, requiring a credit card number, expiration date, and CVV, Process bank transfers using bank account details (account number and routing number) and process digital wallet payments using the wallet's email address and payment amount.

Code

```
class PaymentProcessor {
    public void processPayment(String creditCardNumber,
                               String expirationDate,
                               String cvv) {
        System.out.println("Processing credit card
                            payment...");
    }
    public void processPayment(String accountNumber,
                               String routingNumber) {
        System.out.println("Processing bank transfer...");
    }
    public void processPayment(String emailAddress, double amount) {
        System.out.println("Processing digital
                            wallet payment...");
    }
}
```

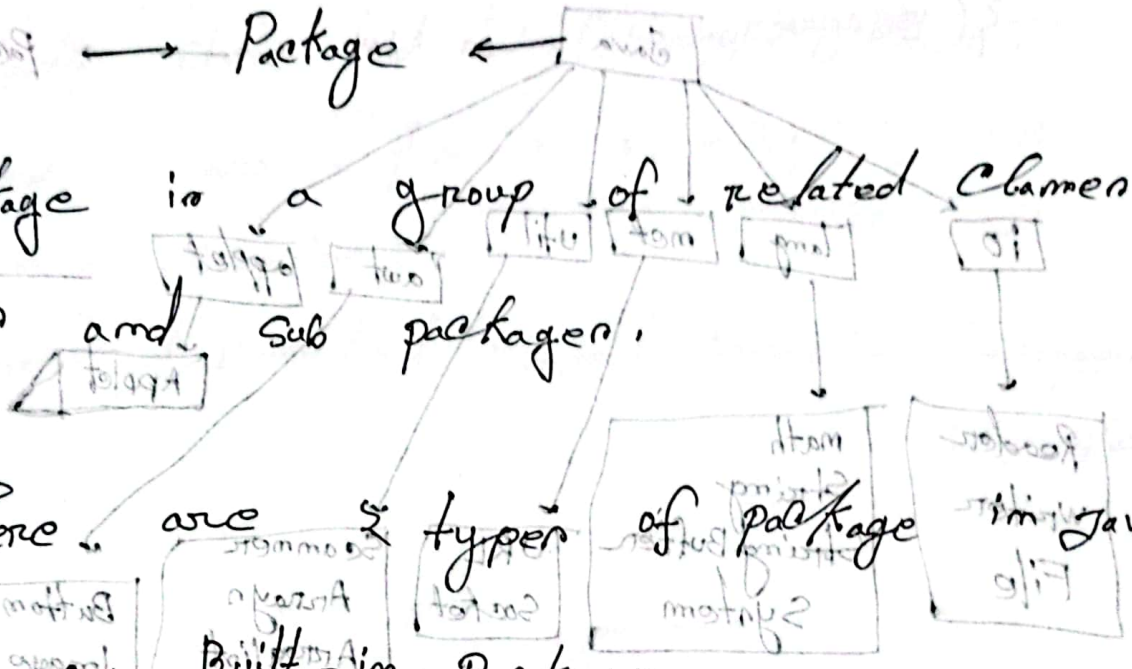
```
public class Main {
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();
        processor.processPayment("1234-5678-9012-3456",
                                 "12/25", "123");
        processor.processPayment("1273-4759910", "98765");
        processor.processPayment("user@gmail.com", 10650.75);
    }
}
```

Clam Note Final

→ Object Oriented Programming, Final Slides ←

slide - 8

→ Package ←



- A package is a group of related classes, interfaces and sub packages.

→ There are 2 types of package in Java

1. Built-in Package
2. User-defined package

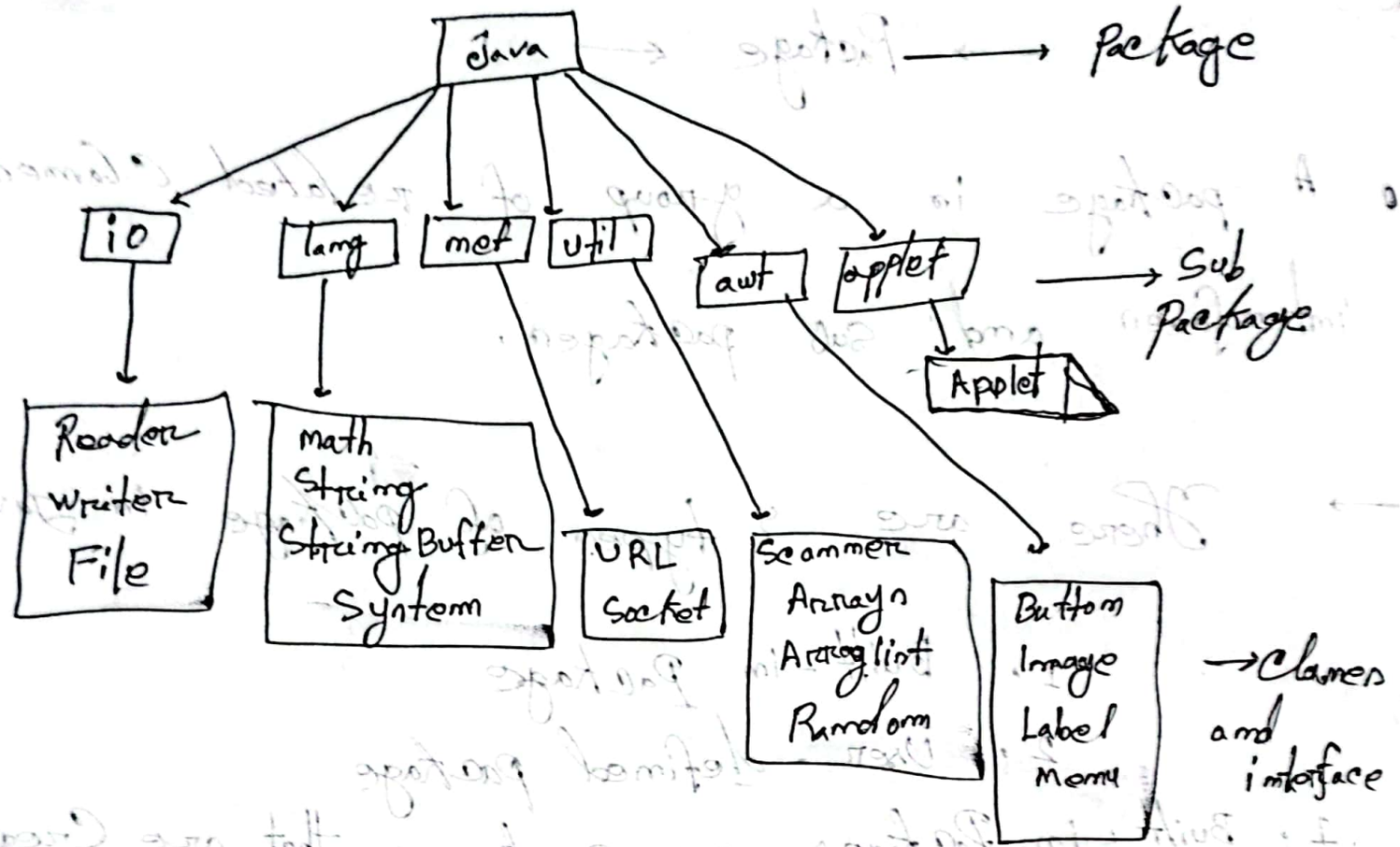
1. Built-in Packages are packages that are Created that we will only use

Example

```
import Java.Util.Scanner;
```

2. User-defined package is the things we make ourselves

→ Built-in-package ←



→ Package

→ Sub Package

→ Classes and interface

Example

```
import java.util.Scanner;
import java.lang.Math;
import java.io.Reader;
```

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

→ Built-in packages are existing java packages that come along with the JDK.

- example: java.lang, java.util, java.io etc.

```
import java.util.Arrays;
public class myClass {
    public static void main(String[] args) {
        int[] arr = {10, 12, 8, 9};
        Arrays.sort(arr);
        System.out.println("Sorted Array is = " + Arrays.
                toString(arr));
    }
}
```

ex

```
import java.util.ArrayList;
class Array {
    public static void main(String[] args) {
        ArrayList<Integer> myList = new ArrayList<>(3);
        myList.add(3);
        myList.add(2);
        myList.add(1);
    }
}
```

→ User - defined package ←

Java also allows you to Create packages as per your need. These packages are called user defined package

→ To define a package in Java, you use keyword package.

Package packageName;

```
package Content;
class Test {
    public static void main (String[] args) {
        System.out.println ("Hello");
    }
}
```

→ How to import packages in Java?

Java has import statement that allows you to import an entire package (as in earlier examples), or use only certain classes and interfaces defined in the package.

For example,

```
import java.util.Date;
import java.io.*;
```

slide-2

Content

→ Java Interface

→ implements Keyword in Interface.

→ Why use Interface

→ extends Keyword in Interface.

→ Abstract Class vs Interface.

→→ Java Interface ←←

A Java Interface is a programming Construct in Java that acts as a blueprint or Contract. It is used to define a set of methods that a class must implement ensuring a Consistent Structure across multiple Classes.

```
interface polygon {
    public void getArea();
}
```

→ Here, polygon is an interface. we have used the interface keyword to declare an interface.

→ abstract method getArea()

Definition of Java Interface

→ An interface is a reference type in Java

→ It can only contain abstract methods (methods without a body) and static final fields (constants).

→ Interfaces are used to achieve multiple inheritance and abstraction in Java.

Why Use an Interface

1. Multiple Inheritance

Since Java does not support multiple inheritance through classes, interfaces provide a way to achieve it.

2. Abstraction

Interfaces enforce a structure by ensuring that implementing classes define certain methods.

3. Code Reusability : Interfaces allow you to define shared behaviors across multiple classes.

→ Example of an Interface in Java

```
interface Animal {
    void eat();
    void sleep();
}
class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }
    @Override
    public void sleep() {
        System.out.println("Dog is sleeping");
    }
}
public class Test {
    public static void main(String[] args) {
        Animal s1 = new Dog();
        dog.eat();
        dog.sleep();
    }
}
```

→ implements keyword in Interface ←

- like abstract classes, we cannot create objects of interfaces.
- However, we can implement interfaces in other classes.
- In Java, we use the implements keyword to implement interfaces.

example

```
interface Student {
    void getStudent();
}

class Teacher implements Student {
    public void getStudent(int ID, String name) {
        int ID = 2088;
        String name = "Sojib";
        System.out.println("Student" + ID, name);
    }
```

class test {

public static void main (String[] args) {

teacher S1 = new teacher();

S1.getStudent("2088", "sojib");

}

}

→ Extends keyword in interface ←

→ Similar to classes, interfaces can extends other interfaces.

→ The extends keyword is used for extending interfaces.

example interface Animal {

}

interface Dog extends Animal {

}

→ Abstract Class Vs Interface ←

Interface	Abstract Class
1. Can only have abstract methods.	1. Can have abstract and non abstract methods.
2. It supports multiple inheritance.	2. It doesn't supports multiple inheritance.
3. Can only have Static and final variable	3. Can have Static, non static, final and non final variable.
4. Fully abstract	4. Partial abstraction

5. Example
interface Animals{
void eat();
}
	5. Example :
	abstract Class Animals{
	abstract void eat();
	}

→ Exception ←

- Java Exception
- Type of Exception
- Exception & Handling
- Example : Java Exception Handling

Java Exception

In Java, an exception is an unexpected event or error that occurs during the execution of a program, disrupting its normal flow. Exceptions are objects that represent these errors, and Java provides a robust mechanism to handle them, ensuring that the program does not crash abruptly and can recover gracefully.

Types of Exception

1. Runtime Exception
2. IO Exception

1. → Runtime Exception

A runtime exception happens due to a programming error. They are also known as Unchecked Exception.

These exceptions are not checked at Compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API
- Null pointer access
- out-of-bounds array access
- Dividing a number by 0

2. IO Exception

An IOException is also known as a Checked exception. They are checked by the Compiler at the Compile-time and the programmer is prompted to handle these exceptions.

Some of the examples of checked exceptions are:

- Trying to open a file that doesn't exist results in (FileNotFoundException)
- Trying to read past the end of a file.

Java Exception Handling

Java provides the following keywords for exception Handling

1. try
   → Defines a block of code to monitor for exceptions.
2. catch
   → Defines a block of code to handle the exception.
3. finally
   → Defines a block of code that executes whether an exception occurs or not
4. throw
   → Used to explicitly throw an exceptions
5. thrown
   → Used in method declarations to specify exceptions that a method might throw.

→ Example of Exception Handling

```
try {
    // code
}
catch (ExceptionType e) {
    // catch block
}
finally {
    // finally block
}
```

→ Java try... Catch Example

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

try {
    int divideByZero = 5/0;
    System.out.println ("Rest of code in try block");
}
Catch (ArithmeticException e) {
    System.out.println ("A => " + e.getMessage());
}
}
```

Example : Java try ... finally block

```
class Main {
  public static void main (String[] args) {
    try {
      int di = 5/0;
    }
    finally {
      System.out.println ("Finally block");
    }
  }
}
```

→ Java try... Catch... finally block

```
import java.io.*;
class L {
    Private int[] list = {5, 6, 7, 10, 2};
    public void writefilist() {
        PrintWriter out = null;
    try {
     System.out.println("Entering try S");
     out = new PrintWriter(new FileWriter("outputfile.txt"));
    for (int i=0; i<7; i++) {
        out.println("v at:" + i + "=" + list[i]);
    }
catch (Exception e) {
    System.out.println("Exception => " + e.getMessage());
    }
finally {
    if (out != null) {
     S.o.p("class in");
    out.close();
    }
```

```
else {
    s o p ("pra mo u");
}
}
}

@lam main {
    p s v m (string[] args) {
        L l = new L();
        l.writeToInt();
    }
}
```

→ Multiple Catch blocks

```
class L {
    public int[] arr = new int[10];
    public void wiri() {
    try {
        arr[10] = 11;
    }
catch ( Num... e1) {
    System.out.println(" N.... => " + e1.getMessage());
}
catch ( I.... e2) {
    s o p ("Im .... => " + e2.getMessage());
}
}
class main {
    p s v m(String[] args) {
        L s1 = new L();
        s1.writeInt();
    }
}
```

→ String ←

String is a class in Java, which is immutable and holds a sequence of characters. It comes under the java.lang package.

— Declaring Strings in Java

Strings can be declared in two ways:

1. Using String Literals:

Example

```
String str1 = "Hi";
String str2 = "Hello";
```

2. Using the new keyword.

```
String str = new String("Hello");
```

→ String Methods ←

string charAt( )

Example:

```
public class CharAtExa {
    p s v m (String args[ ]) {
        String name = "Sojib";
        char ch = name.charAt(3);
        sop(ch);
    }
}
```

string length( )

Example:

```
public class L {
    psvm ( ) {
        String N1 = "Sojib";
        String N2 = "Shahjalal";
        SOP ("S Length " + N1.length());
        SOP ("S " + N2.length());
```

→ Content ←

- Java Applet
- Advantage of Applet
- Hierarchy of Applet
- Lifecycle of Java Applet
- Simple example of Applet

→ Java Applet

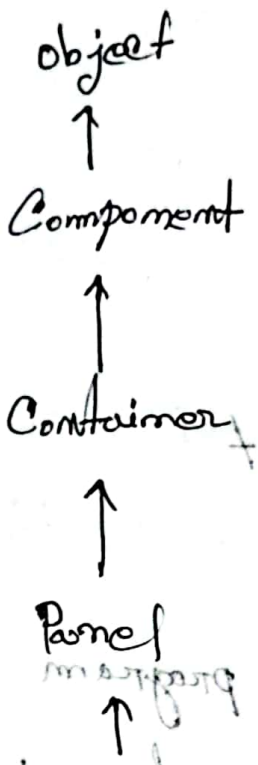
Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

→ Advantage of Applet

There are many advantages of applet. They are as follows:

It works at client side so less response time.
Secured
It can be executed by browsers running under many platforms, including Linux, windows, mac os etc.

→ Hierarchy of Applet :



→ Lifecycle of Java Applet

- Applet is initialized
- u Started
- u Painted
- u Stopped
- u Destroyed

→ Simple example of Applet

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet {
    public void paint(Graphics g) {
        g.drawString("welcome", 150, 150);
    }
}
```

Content

- Java File Handling
- Methods of File Class
- Java Create and write To Files
- Get File Information
- Java Delete Files

→ Java File Handling ←

File handling is an important part of any application.

Java has several methods for Creating, reading, updating, and deleting files.

Example:

```
import java.io.File; (Import the File Class)

File myobj = new File ("filename.txt");
                      (Specify the filename)
```

→ The File Class has many useful methods for creating and getting information about file.

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not.
canWrite()	Boolean	Tests whether the file is writable or not.
createNewFile()	Boolean	Creates an empty file
getName()	String	Returns the name of the file.
getAbsolutePath()		
length()	Long	
list()	String[]	
mkdir()	Boolean	
delete()	Boolean	
exists()	Boolean	

→ Java Create and write To File ←

To Create a file in Java, you can use the createNewFile() method. This method returns a boolean value, true if the file was Succesfully Created and false if the file already exists. Note that the method is enclosed in a try... Catch block. This is necessary because it thrown an IOException if an error occurs.

→ Create File -

```
import java.io.File;
import java.io.IOException;
public class CreateFile {
    public static void main(String[] args) {
        try {
            File s1 = new File("filename.txt");
            if (s1.CreateNewFile()) {
                SOP("F C:" + s1.getName());
            } else {
                SOP("File al ex;");
            }
        } Catch (IOException e) {
            SOP("Am error oc");
```

→ Write To a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method.

```
import java.io.FileWriter;
import java.io.IOException;
public class WriteToFile {
        psvm() {
    try {
    FileWriter S2 = new FileWriter("filename.text");
    S2.write("File in Java");
    S2.close();
     sop("succesfully");
    }
    Catch (IOException e) {
     sop("An error.");
    e.printStackTrace();
}
}
```

→ Java Read File

In the previous chapter, you learned how to create and write to a file.

In the following example, we use the Scanner class to read the contents of the text file we created in the previous chapter.

Example

```
import java.io.File;
import java.io.FileNotFoundEx;
import java.util.Scanner;

public class ReadFile {
    psvm (String[] args) {

    try {
      File S1 = new File ("filename.txt");
      Scanner S2 = new Scanner (S1);
    while (S2.hasNextLine ()) {
      String data = S2.nextLine()
        System.out.println(data);
    S2.close ();
}
```

```
catch (FileNotFoundEx e){
    sop ("An error occ");
    e.printStackTrace();
}
}
```

→ Get File Information

To get more information about a file, use any of the File methods:

Ex

```
import java.io.File;
    public class {
        psvm ( ) {
    File N1 = new File ("Filename.txt");
if ( s1.exists()) {
    sop ("File name" + N1.getName())
    sop ("  A      " + N1.getAbsolutepath());
}else {
    sop ("The file does not ");
}
}
}
```

→ Java Delete Files ←

To delete a file in java, use the delete() method.

example

```
import java.io.File;
public class DeleteFile {
    psvm ( ) {
    File s1 = new File("filename.txt");
if (s1.delete()) {
        sop("D:" + s1.getName());
    } else {
        sop("Failed to delete");
    }
}
```

→ Delete a Folder

you can also delete a folder. However it must be empty:

Example

```
import java.io.File;
public class D{
    psvm(){
File s1 = new File("C:\Users\myName\Test");
if (s1.delete()){
    sop("Deleted " + s1.getName());
}else{
    sop("F ");
}
}
```

Thread

Multithreaded Programming

→ When a Java program starts up, one thread begins running immediately.

→ Other child threads are spawned from the main thread.

→ often it must be the last thread to finish execution.

Useful Methods of Thread.

1. Static Thread currentThread()

→ returns a reference to the thread in which it is called.

2. Static void sleep (long millisecond)

→ cause the thread from which it is called to suspend execution for the specified period of milliseconds.

3. final void setName (String threadName)

→ set the name of the thread from which it is called. It to the name given as parameter.

4. final String getName()

→ return the name of the thread from which it is called.

Creating Threads

→ Two ways

1. Implement the Runnable Interface:
2. Extend the Thread class.

First Approach - Implement Runnable

→ Create a class that will implement Runnable.

To implement Runnable, the class need only to implement:

public void run()

— Establishes the entry point for another, Concurrent thread.

→ Introduce an object of type Thread from within the same class.

→ Call the start() method of the new object of type Thread.

Example - First Approach

```
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        t = new Thread(this, "demo thread");
        System.out.println("Child thread: " + t);
    }
    public void run() {
        try {
            for (int n = 5; n > 0; n--) {
                System.out.println("Child Thread: " + n);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println("Child Interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
```

```
class ThreadDemo {
    public static void main (String args[]) {
        new NewThread();
        try {
            for(int n=5; n>0; n--) {
                sop ("main Thread : "+n);
                Thread.Sleep(1000);
            }
        }
        catch ( InterruptedException e ) {
            sop ( " main " );
        }
        sop ("Exiting main");
    }
}
```

Second Approach - Extends Thread

→ Create a new class that extends Thread and then create an instance of that class.

→ The extending class must override the run() method.

→ It must also call the start() method to begin execution of the new thread.

Example:-

```
class NewThread extends Thread {
    NewThread() {
        super("demo thread");
        System.out.println("Child Thread: " + this);
        start();
    }
}
```

Creating Multiple Thread

In Java multiple threads means running multiple tasks concurrently using separate threads. A thread is lightweight subprocess that enables a program to perform multitasking, where different tasks can run independently and simultaneously

what is Multiple Threading

→ multiple threading involves running two or more threads simultaneously within a single program.

→ It allows efficient utilization of the CPU by performing tasks concurrently.

→ Threads share the same memory space, making inter-thread communication easier and faster.

Why use Multiple threads :

→ Improved Performance : Enables parallel processing, making programs faster.

→ Concurrency : Executes multiple tasks simultaneously, improving responsiveness

→ Shared Resources

→ Responsiveness

Multiple Example

```
class Multiple implement Runnable
    String name;
    Thread t;
Multiple (String tmp) {
    name = tmp;
    t = new Thread (this, name);
    SOP ("New th. :" ++);
    t.start()
}
```

```
public void run() {
    try {
        for (int i = 5; i > 0; i--) {
            sop(name + " : " + i);
            Thread.sleep(1000);
        }
    }
    catch (InterruptedException e) {
        sop(name + " " + e);
    }
}

class Demo {
    psvm() {
        new Multiple("One");
        new Multiple("Two");
        new Multiple("Three");
    }
}
```

→ isAlive():

- final boolean isAlive()
- it returns true if the thread upon which it is called is still running.

→ join()

- final void join() throws InterruptedException
- This method waits until the thread on which it is called terminates.

Thread Priority

→ Higher - priority threads get more CPU time than lower - priority threads (Condition apply)

→ final int getPriority ()

→ final int setPriority (int level)

→ level is in limit between MIN_PRIORITY (1) and MAX_PRIORITY (5).

→ Default priority is NORM_PRIORITY which is 5.

Thread Synchronization

→ Two or more threads need access to shared resource.

→ Need some way to ensure that the resource will be used by only one thread at a time. — Synchronization.

→ Monitor Synchronization

→ A monitor is object that is used as a mutually exclusive lock.

→ Only one thread can own a monitor at a given time.

→ Threads attempting to enter a locked monitor will be suspended until first thread exits.

→ Other threads are waiting for the monitor

Inter Thread Communication

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

→ notify(): wakes up a thread that called wait() on the same object.

→ notifyAll(): wakes up all the threads that called wait() on the same object. One of the threads will be granted access.