1. Introduction to Java

o History of Java

→History :

Before Java : "OAK" :1991 : James Goasling: Console app :single user

Renamed "Java" from "OAK" : 1995 : James Goasling : multiuser : console app & Desktop app

o Features of Java (Platform Independent, Object-Oriented, etc.)

→Java Features :

1)simple

2)OO

3)interpreter : JVM : bytecode (classfile) to machine code

4)Robust :powerful

5)secure

6)dynamic

7)high performance :10x

8)multithreading :

9)platform independent :

10) portable :

o Understanding JVM, JRE, and JDK

→

JVM: Executes Java bytecode (runtime engine).

JRE: Provides the environment (including JVM and libraries) to run Java programs.

JDK: Provides everything needed to develop Java programs (includes JRE + development tools).

o Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)

→

1. **Install Java Development Kit (JDK)**

2. **Install Eclipse IDE**

3. Install IntelliJ IDEA

4. **Set up Java in the IDE**

5. **Verify the Setup**

o Java Program Structure (Packages, Classes, Methods)

→

Java programe Structure :

```
package com.simple;
class Methods
{
	String name="kishan Bharwad";
	public void getData()
	{
		System.out.println("Name is..."+name);
	}
}
public class ClassMethods
{
	public static void main(String[] args)
	{
		Methods m1=new Methods();
		m1.getData();
	}
}
```

2. Data Types, Variables, and Operators

→

❖ *Data Type :*

: to reserve the memory for that vaarible or

:which type of data you want to store in a variable that type you mention before the variable

:there are mainly 2 types

1) primitive : fixed size of data types

1.numeric

1.Integeral point: int, long,char,byte,etc..

2.floating point : float , double

2.non numeric

boolean , return

2) non primitive : no any fixed size of data type

class, array, interface etc..

❖ variable :

means is nothing but to store some value.

:token or identifire

- ❖ does not start with digit

2) does not allow reserved keyword as a variable name

3) does not allow space between variable name

4) followed with digit after any letter or "_" or “$”.

❖ Operator :

- ❖ Operator is use in two oprants.operator is symbol perform the to get the mathamatics operation.

Multiple operator in c language

- ❖ arithmetic operator: Addition , subtraction , multiplication , division, modulo etc. perform the mathematical operation use arithmetic operator.

Ex :

addition	+
subtraction	-
multiplication	*
division	/
modulo	%

o Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise

- relational operator: perform the two value comparison use relational operator.

Ex :

Double equal	==
Not equal	!=
Grater than	>
Less than	<
Less than equal	<=
Grater than equal	>=

- logical operator : perform the two or more condition check use logical operator.

Ex :&&(and) , ||(or) , !(not)

And	&&
Or	||
Not	!

- assignment operator : perform the assign value use assignment operator.

  Ex: += , -+, *= , /=

a-=b	a=a-b
a*=b	a=a*b
a/=b	a=a/b
a+=b	a=a+b

- increment/decrement :increment operator value+1,decrement operator value-1

  Prefix : ++a , --a postfix : a++ , a--

- bitwise operator : ex : & , | , << , >>

o Primitive Data Types in Java (int, float, char, etc.)

→

1) primitive : fixed size of data types

1.numeric

1.Integeral point: int, long,char,byte,etc..

2.floating point : float , double

2.non numeric

boolean , return

o Variable Declaration and Initialization

→

Int (data type) a(variable Declaration)=20(initialization);

o Type Conversion and Type Casting

→

Type conversion : convert from one data type to another data type mechanism

:there are mainly 2 types

1) implicit :automatically : convert from smaller data type in size convert into Bigger Data type

2) explicit :type casting : convert from Bigger data type in size convert into Smaller Data type

3. Control Flow Statements

o If-Else Statements

→

1. if_else statement :if_else is use to check the condition is true or false .

   Syntex : if(condition)

```
{
    //statement
}
else
{
    //statement
}
```

o Switch Case Statements

→

- switch statement:To make menu driven code.Never use relational op. by this. (n==1).switch can be used with only int, char, double data types.

  switch can be used with the keywords :

  switch, case, break, default.

  Syntex : switch(choice)

  {

  Case 1: // statement

  Break;

  Case 2 : //statement

  Break;

  Default : //statement

o Loops (For, While, Do-While)

→

- **While loop :** while(condition) { execute the code } .while loop condition is true loop is execute and condition is false loop is not execute. while loop is entry control loop, entry control loop is check condition before execute code. Exit the loop condition is false.

e.g :

While loop :

```
I = 1;                          //(initialization)

While(I <= 5)                   //condition
{
System.out.println ( i++);      //execute code
                                // increment/decrement
 }
```

- **For loop :** for(initialization , condition , increment/decrement). for loop condition is true loop is execute and condition is false loop is not execute. for loop is entry control loop, entry control loop is check condition before execute code. Exit the loop condition is false.

e.g :

For loop :

```
For
(I=1(initialization; I<=5(condition);i++(increment/decrement))
{

System.out.println (i);                 // execute code

 }
```

- **Do while loop :** do { execute the code } while(condition) . do while loop condition first time execute and after condition is true loop is execute and condition is false loop is not execute. do while loop is exit control loop, exit control loop is execute code before check condition. Exit the loop condition is false.

e.g :

Do while loop :

```
I = 1;(initialization)
 Do      {
         System.out.println (i);                 // execute code
         I++;                                                    // increment/decrement
         } While (I <= 5);              // (condition)
```

o Break and Continue Keywords

Break statement : Break is keyword and break keyword use the break the code, rest of the code will not executed .

Example :

```
{
 Int I;
 for(i=0;i<10;i++)
 {
         If(i==6)
         break;
         system.out.Println(I);
 }
```

Continue statements : The continue statement is used within loops to skip the remaining statements in the current iteration and proceed to the next iteration of the loop.

Example :

```
{
```

```
        for (int i = 1; i<= 10; i++)
        {
                if (i % 2 == 0)
                        {
                         continue; // Skip even numbers
                        }
                System.out.println( i);
         }
}
```

4. Classes and Objects

o Defining a Class and Object in Java

→

1) class : is an collection of data member(variables) and member function(methods, process) with its behaviors

sy:

```
        class classname
        {
                data member
                member function
        }
```

2) object : is a instancs of an class i.e.

:when you create class variables also called..

:its uses new keyword and class constructor to create object

:access whole properties of an class except private

sy:

classname objectname = new constructor();

o Constructors and Overloading

→

Constructor : is an special member function because its same name as a class name.

: does not return any value even void

: can be overloaded

: to initialized value of your data members at object creation time

: may used access modifire except private

: when your class object you create at that time to called constructor.

: there are mainly 2 types

1) default : no any argument in constructor

2) parameterized: may have one or more argument in constructor

Example :

```
package com.oops;
class Sum
{
	int x,y;

	public Sum()
	{
		x=10;
		y=20;
		System.out.println(x+y);
	}
	public Sum(int a)
	{
		x=a;
		y=a;
		System.out.println(x+y);
	}
```

```
	public Sum(int a,int b)
	{
		x=a;
		y=b;
		System.out.println(x+y);
	}

}
public class ContructorDemo1
{
	public static void main(String[] args) {
		new Sum();
		new Sum(123);
		new Sum(45,65);
	}

}
```

Overloading :
method overloading(compile time) : the two or more method name should be same in a single class but its behaviors(data types, arguments) are different .
Example :

```
package com.oops;

import java.util.Scanner;

class Area
{
	public void area(int a)
	{
		int ans=a*a*a;
		System.out.println("This is the area of cirecle"+ans);


	}
	public void area(float b,float h)
	{
		float ans=((b*h)/2);
		System.out.println("This is the area of tragulaer"+ans);
	}
}
public class OverLodingDemo
{
	public static void main(String[] args)
	{
		Area a=new Area();
		a.area(5);
```

```
                a.area(4f,5f);
        }
}
```

o Object Creation, Accessing Members of the Class

→

Object Creation :

Object(class name) ob(object name)=new(keyword)(class name) object();

AccessingMember of the class :

Ob(objectname).(method name)getData();

o this Keyword

→ this : when your class variable name and argument variable names are same at that time to seprate your class variable with using this keyword

Example :

```
package com.keywords;
class TDemo
{
        int no;
        String name;
        public void setData(int no,String name)
        {
                this.no=no;
                this.name=name;
        }
        public void display()
        {
                System.out.println("no is "+no);
                System.out.println("name is "+name);

        }
}
```

```
public class ThisDemo
{
	public static void main(String[] args)
	{
		TDemo t1=new TDemo();
		t1.setData(123,"kishan");
		t1.display();

	}
}
```

5. Methods in Java

o Defining Methods

→

❖ **Defining Methods**

A method in Java is defined by specifying its return type, method name, and any parameters. The body of the method contains the code that executes when the method is called.

Example:

```
public int addNumbers(int a, int b) {

  return a + b;

}
```

o Method Parameters and Return Types

→

❖ **Method Parameters and Return Types**

- **Parameters**: Parameters are variables passed into a method to provide input data. They are listed in parentheses after the method name.
- **Return Types**: The return type defines what type of data the method will return after execution. A method can either return a value or be declared as void if it does not return anything.

  Example :

```
public double calculateArea(double radius) {
   return Math.PI * radius * radius;  // Returns the area of a circle
}
```

o Method Overloading

→

❖ **Method Overloading**

the two or more method name should be same in a single class but its behaviors(data types, arguments) are different .

Example :

```
public int sum(int a, int b) {
   return a + b;
}
public double sum(double a, double b) {
   return a + b;
}
public int sum(int a, int b, int c) {
   return a + b + c;
}
```

o Static Methods and Variables

→

❖ **Static Methods and Variables**

A static method or variable belongs to the class, rather than to any specific instance of the class. Static methods can be called without creating an instance of the class. Similarly, static variables are shared across all instances of the class.

Example :

```
public class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }
}


public class Main {
    public static void main(String[] args) {
        // Call static method without creating an instance
        int result = MathUtils.add(5, 10);
        System.out.println("The sum is: " + result);
    }
}
```

6. Object-Oriented Programming (OOPs) Concepts

o Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction

→

encapsulation : wrapping up of data into single unit i.e. :data hiding

:private your data member and member function

inheritance : properties of parent class extends into child class

:properties of superclass extends into subclass

:main purpose is : Reusablity , extendsiblity

:to used "extends" keyword through create inheritance

:always called last child class to create object with access the properties of parent class except private

polymorphism : ability to take one name having many forms or different forms

:there are mainly 2 types

1) method overloading(compile time) : the two or more method name should be same in a single class but its behaviors(data types, arguments) are different i.e

2) method overriding(run time) : the whole signature of the method should be same in super class as well as in subclass but its behaviors (body part of the method) are different

abstract : only essential part should be display rest of the part will be hide : data hiding

1) using with class : we can not create object of that class

:must inherit into your child class

2) using with method : do no specify body part of the method

: your class must be also abstract

:must override your abstract method into your child class

o Inheritance: Single, Multilevel, Hierarchical

→

1) single : only one parent having only one child

2) multilevel : single inheritance having one another child

3) hierarchical : one parent having 2 or more child

o Method Overriding and Dynamic Method Dispatch

→

method overriding(run time) : the whole signature of the method should be same in super class as well as in subclass but its behaviors (body part of the method) are different

→

```
package com.oops;
class Overriding
{
	public  void display()
	{
		System.out.println("This is the first classs..");

	}
}
class Second extends Overriding
{
	public void display()
	{
		System.out.println("This is the second class..");
	}
}

public class OverridingDemo
{

	public static void main(String[] args)
	{
		Overriding or=new Overriding();
		or.display();
		or=new Second();
		or.display();


	}

}
```

Dynamic Method Dispatch : refers to the mechanism by which a call to an overridden method is resolved at runtime, rather than at compile time.

Example :

```
class Animal {

  public void sound() {

    System.out.println("Some generic animal sound");
```

```
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Meow");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Animal reference and object
        Animal myDog = new Dog();       // Animal reference but Dog object
        Animal myCat = new Cat();       // Animal reference but Cat object

        myAnimal.sound();  // Output: Some generic animal sound
        myDog.sound();     // Output: Bark
```

```
    myCat.sound();    // Output: Meow
  }
}
```

7. Constructors and Destructors

o Constructor Types (Default, Parameterized)

→

Constructor : is an special member function because its same name as a class name.

there are mainly 2 types

1) default : no any argument in constructor

→Example:

```
class Person {
  String name;
  int age;

  public Person() {                    // Default constructor
    name = "Unknown";          // Automatically initialized
    age = 0;
  }

  public void display() {
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
  }
}

public class Test {
  public static void main(String[] args) {
    Person person = new Person(); // Calls default constructor
    person.display();   }
}
```

2) parameterized: may have one or more argument in constructor

→Example :

```
class Person {
```

```
    String name;
    int age;

    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Test {
    public static void main(String[] args) {
        // Creating an object with parameterized constructor
        Person person = new Person("John", 25);
        person.display(); // Output: Name: John, Age: 25
    }
}
```

2. Destructors in Java

- **Java** does not have destructors in the same sense. In Java, **memory management** (including deallocation of objects) is handled automatically through **garbage collection**. When an object is no longer referenced, the **garbage collector** will automatically clean up the memory.

- **No Explicit Destructor**: Java does not provide a destructor method like C++. You don't need to explicitly free memory when you are done with an object.

o Copy Constructor (Emulated in Java)

→ **copy constructor** : is used to create a new object as a copy of an existing object.

Example :

```
class Student {
   String name;
   int age;

   // Constructor to initialize a new student
   public Student(String name, int age) {
      this.name = name;
      this.age = age;
   }

   // Copy constructor (emulated in Java)
   public Student(Student other) {
      this.name = other.name;  // Copy name from the other object
      this.age = other.age;    // Copy age from the other object
   }

   // Method to display student details
   public void display() {
```

```
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Test {
    public static void main(String[] args) {
        // Creating an original object
        Student student1 = new Student("Alice", 20);
        student1.display();

        // Creating a copy of student1 using the copy constructor
        Student student2 = new Student(student1);
        student2.display();
    }
}
```

o Constructor Overloading

→ **Constructor Overloading** : refers to the ability to define multiple constructors in a class with the **same name** but **different parameter lists** (i.e., the number or type of parameters must be different).

Example:

```
package com.oops;

import java.util.Scanner;

class Area
{
        public void Area(int a)
        {
                int ans=a*a*a;
                System.out.println("This is the area of circle"+ans);
```

```
        }
        public void Area(float b,float h)
        {
                float ans=((b*h)/2);
                System.out.println("This is the area of tragulaer"+ans);
        }
}
public class OverLodingDemo
{
        public static void main(String[] args)
        {
                Area a=new Area();
                a.area(5);
                a.area(4f,5f);
        }
}
```

o Object Life Cycle and Garbage Collection

→

1. Object Life Cycle in Java

The **object life cycle** refers to the process an object goes through from its creation to its destruction.

Stage of object life cycle :

1.Creation (Instantiation)

2.Initialization

3. Usage

4. Dereferencing (Becomes Eligible for Garbage Collection)

5. Destruction (Garbage Collection)

2. Garbage Collection in Java

Garbage collection (GC) is an automatic process in Java that manages memory by reclaiming memory from objects that are no longer needed. The primary goal of garbage collection is to **prevent memory leaks** and to ensure efficient use of memory.

Key Points about Garbage Collection:

1.Automatic Process

2.Heap Memory

3.Garbage Collector

4.No Guarantees of When GC Happens

5.Finalization

8. Arrays and Strings

o One-Dimensional and Multidimensional Arrays

→

1. One-Dimensional Arrays

A **one-dimensional array** is essentially a list of values stored in a single row. It's the simplest form of an array in Java and can be visualized as a sequence of elements.

Syntax :

datatype[] arrayName = new datatype[size];

2. Multidimensional Arrays

A **multidimensional array** is an array of arrays. In Java, the most common form of multidimensional arrays are **2D arrays** (two-dimensional), but Java supports arrays of any dimension (3D, 4D, etc.).

A two-dimensional array can be visualized as a table or matrix, where each element is accessed by two indices: one for the row and one for the column.

Syntax :

datatype[][] arrayName = new datatype[rows][columns];

o String Handling in Java: String Class, StringBuffer, StringBuilder

→

1. String Class

The String class in Java is immutable, meaning once a String object is created, its value cannot be changed. Any modification to a string results in the creation of a new string object.

2. StringBuffer Class

StringBuffer is a mutable sequence of characters. Unlike String, StringBuffer allows modification of the string without creating new objects each time, which makes it more efficient when performing a lot of string manipulations (like appending or inserting).

3. StringBuilder Class

StringBuilder is similar to StringBuffer but is **not thread-safe**. It provides better performance than StringBuffer when string manipulation is done in a single-threaded environment because there's no synchronization overhead.

o Array of Objects

→ **array of objects** : refers to an array where each element is an object of a specific class. Arrays in Java can hold any type of object, and using an array of objects allows you to store multiple instances of objects in a single data structure.

Example :

ClassName[] arrayName = new ClassName[size];

o String Methods (length, charAt, substring, etc.)

→

9. Inheritance and Polymorphism

o Inheritance Types and Benefits

→ inheritance : properties of parent class extends into child class

:there are mainly 5 types

1) single : only one parent having only one child

2) multilevel : single inheritance having one another child

3) hierarchical : one parent having 2 or more child

4) multiple : java does not support directly

5) hybrid: java does not support directly

Benefits of Inheritance

1. **Code Reusability**
   - Inheritance allows the child class to reuse the methods and properties of the parent class, reducing the need for redundant code.
   - This leads to more efficient and maintainable code.
2. **Extensibility**
   - You can extend existing functionality by adding new methods or overriding existing ones in the subclass without modifying the parent class.
3. **Maintainability**
   - Changes made to the parent class (such as bug fixes or improvements) can automatically be reflected in all subclasses, which simplifies maintenance.
4. **Modularity**
   - Inheritance promotes a modular approach to code design, where different components or features are organized into parent-child hierarchies, making it easier to manage and update code.
5. **Polymorphism**
   - Inheritance supports **polymorphism**, which allows objects of different classes to be treated as objects of a common superclass. This is particularly useful for writing more generic and reusable code.
6. **Abstraction**
   - Inheritance allows abstract classes (classes that cannot be instantiated) to define common interfaces for subclasses. This is useful in designing complex systems with clear hierarchies.

o Method Overriding

→

method overriding(run time) : the whole signature of the method should be same in super class as well as in subclass but its behaviors (body part of the method) are different.

Example :

```
package com.oops;
class Overriding
```

```
{
        public  void display()
        {
                System.out.println("This is the first classs..");

        }
}
class Second extends Overriding
{
        public void display()
        {
                System.out.println("This is the second class..");
        }
}

public class OverridingDemo
{

        public static void main(String[] args)
        {
                Overriding or=new Overriding();
                or.display();
                or=new Second();
                or.display();


        }

}
```

o Dynamic Binding (Run-Time Polymorphism)

→

Dynamic Binding(Run-Time Polymorphism) : This happens when the method call is resolved at runtime. It occurs in the case of **method overriding**, where the decision about which method to invoke is made based on the actual object's type, not the reference type.

Example :

```
// Parent class

class Animal {

   public void makeSound() {

      System.out.println("Animal makes a sound");
```

```
    }
}

// Subclass
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

// Main class to test dynamic binding
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Reference to Animal
        Animal myDog = new Dog();        // Reference to Animal, but object is Dog

        // Call makeSound() on both references
        myAnimal.makeSound();  // Output: Animal makes a sound
        myDog.makeSound();     // Output: Dog barks
    }
}
```

o Super Keyword and Method Hiding

→

Super Keyword : when your super class variablename and subclass variable name are same then you used superclass variable want to used value into subclass at that time used super keyword with variable

Example :

```
package com.keywords;
class SD
{
	int rollno;
	String name;
	public void getValue()
	{
		rollno=124;
		name="kishan";
	}
}
class Second  extends SD
{
	int rollno=100;
	public void  display()
	{
		System.out.println("rollno is "+super.rollno);
		System.out.println("name is "+name);
	}
}
public class SuperDemo {

	public static void main(String[] args)
	{
		Second s1=new Second();
		s1.getValue();
		s1.display();
	}

}
```

❖ **Method Hiding**

Method hiding is a concept that applies to **static methods**. method hiding occurs when a static method in a subclass has the same signature as a static method in the parent class. In this case, the method in the subclass hides the one in the parent class, and it is resolved at **compile time**, not runtime.

Example :

```
// Parent class

class Animal {
```

```
    static void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
// Child class
class Dog extends Animal {
    static void makeSound() {
        System.out.println("Dog barks");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal dog = new Dog();

        // Call static methods
        animal.makeSound();  // Animal's makeSound()
        dog.makeSound();     // Dog's makeSound() (method hiding)
    }
}
```

10. Interfaces and Abstract Classes

o Abstract Classes and Methods

→

abstract : only essential part should be display rest of the part will be hide : data hiding

1) using with class : we can not create object of that class

:must inherit into your child class

2) using with method : do no specify body part of the method

: your class must be also abstract

:must override your abstract method into your child class

o Interfaces: Multiple Inheritance in Java

→Interface : means same as class ,its collection of datamember and member function,data members are final or static by default.

Multiple Inheritance in Java with interface :which cannot extend more than one class (single inheritance), Java allows a class to implement multiple interfaces. This feature is Java's way of supporting **multiple inheritance**, without the complexity and ambiguity associated with multiple class inheritance.

Example :

```
interface Animal {
  void sound();
}
interface Mammal {
  void walk();
}
class Dog implements Animal, Mammal {
  @Override
  public void sound() {
    System.out.println("Woof!");
```

```
    }
  @Override
    public void walk() {
        System.out.println("Dog walks on four legs");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Woof!
        dog.walk();   // Output: Dog walks on four legs
    }
}
```

o Implementing Multiple Interfaces

→ **multiple inheritance** of **behavior** is achieved through **interfaces**. A class can implement more than one interface, thus inheriting the abstract methods declared in multiple interfaces. This allows a class to have multiple sets of functionality without being constrained by single inheritance, which is the case with classes.

Example :

```
interface Interface1 {
    void method1();
}
interface Interface2 {
    void method2();
}
class MyClass implements Interface1, Interface2 {
```

```
    @Override
    public void method1() {
        System.out.println("Method 1 from Interface1");
    }
   @Override
    public void method2() {
        System.out.println("Method 2 from Interface2");
    }
}
public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.method1();  // Output: Method 1 from Interface1
        obj.method2();  // Output: Method 2 from Interface2
    }
}
```

11. Packages and Access Modifiers

o Java Packages: Built-in and User-Defined Packages

→

Built-in Packages in Java

Java provides a rich set of built-in packages that offer a variety of functionality. These packages are part of the Java Standard Library (also called the Java API). Some common built-in packages include java.lang, java.util, java.io, and java.net.

Common Built-in Packages:

java.lang: This is the default package that is automatically imported in every Java program. It includes fundamental classes like String, Math, Object, Thread, and System.

java.util: Contains utility classes such as ArrayList, HashMap, Date, Collections, and others. It's commonly used for working with data structures and performing various utility tasks.

java.io: Provides classes for input and output operations like reading and writing files, including File, BufferedReader, FileWriter, etc.

java.net: Includes classes for networking, such as Socket, URL, and URLConnection, which allow you to work with web services and manage network connections.

Example:

package com.basic;

User-Defined Packages in Java

User-defined packages allow developers to group related classes, interfaces, and sub-packages to keep their code organized and avoid class name conflicts. A **package** is simply a namespace that organizes a set of related classes and interfaces.

Example :

import com.basic.Student

import com.basic.*;

o Access Modifiers: Private, Default, Protected, Public

→

Private: The `private` modifier makes the class member (variable, method, or constructor) **accessible only within the same class**.

Default : If no access modifier is specified (i.e., no private, protected, or public), the member has **default** access. This means that the member is **accessible only within classes that belong to the same package**.

Protected : The protected modifier allows access to members **within the same package** and to **subclasses** (even if they are in a different package).

Public : The public modifier makes the class member **accessible from any other class, regardless of the package**. There are no restrictions on visibility when public is used.

o Importing Packages and Classpath

→ **Importing Packages**

In Java, you can import classes and entire packages to use in your program. Importing allows you to access pre-written code from Java's standard library or other libraries you include in your project.

Example :

import java.util.*; // Import all classes from the java.util package

Classpath

The **classpath** is an environment variable or a configuration in your development environment that tells the Java Virtual Machine (JVM) and Java compiler where to look for classes when running a Java program. It specifies locations of the necessary .class files and libraries (JAR files).

Example :

java -cp /path/to/libs/*:. MyProgram

12. Exception Handling

o Types of Exceptions: Checked and Unchecked

→ **Checked Exceptions**

Checked exceptions are exceptions that **must** be explicitly handled by the programmer at compile time. The compiler checks whether these exceptions are either caught using a try-catch block or declared in the method signature using the throws keyword. If the programmer does not handle or declare the checked exception, the code will not compile.

Unchecked Exceptions (Runtime Exceptions)

Unchecked exceptions are exceptions that **do not** need to be explicitly handled or declared. They are typically caused by bugs in the code or situations that are unlikely to be recoverable. The compiler does not require that you handle or declare unchecked exceptions.

o try, catch, finally, throw, throws

→ 1) try : to find the error from the block

:when you know in my block some line of code having error

:in which line to find the error that line to reamining line in the block are skipped

:whatever error found in try block that error throw to catchblock

:try followed with catch , finally or both

2) catch : whatever error thrown by try block that error will handled with appropriate class

:catch can be multiple

3) finally :if error will come or not but my block always perfomed

4) throw : to create own Exception / custom exception / user defined exception

:to write inside the method

:used with new keyword and our own Exception constructor

:at a time only 1 exception to called

5) throws : to write with method or function signature

:used with userdefined or system defined exception

:called multiple exceptions by write using ","

o Custom Exception Classes

→ you handle specific error scenarios that are not covered by standard Java exceptions.

In Java, exceptions are subclasses of the Throwable class. There are two main types of exceptions:

- **Checked exceptions**: Subclasses of Exception, but not RuntimeException.
- **Unchecked exceptions (runtime exceptions)**: Subclasses of RuntimeException.

Extend the Exception or RuntimeException class:

- If your custom exception needs to be checked, extend the Exception class.
- If your custom exception should be unchecked (i.e., it doesn't need to be declared or caught), extend the RuntimeException class.

Define constructors:

- You can define multiple constructors to initialize the exception with a custom message and/or cause (another exception).

Optional: Add custom behavior:

- You can add methods to your custom exception class to carry extra data or implement custom behavior.

13. Multithreading

o Introduction to Threads

→ Thread : its an light weight process or processor

:its totally depends on process

:its self class

:its derived from java.lang package

:each and every programm must have a thread i.e. main()

:when you start your code for executing i.e

:every Java application has at least one thread—the **main thread** that starts when the program begins.

There are main two way to perform thread :-

1) by extending Thread class
2) by implementing runnable interface

o Creating Threads by Extending Thread Class or Implementing Runnable Interface

→

o Thread Life Cycle

→ Thread life cycle

1) newborn state : when you create object

2) runnable state: start()

3) running state: run()

4) blocked state: suspend()=>resume(), sleep(ms),wait()=>notify()

5) dead state: stop()

New State :-

- When you create Thead Object but not start yet.
- Syntax :- Thread t = new Thread();

Runnable State :-

- The Runnable State is Ready to run When you use to Start Method .
- Use start method.
- Syntax :- t.start();

Running State :-

- The thread is managed by the thread scheduler, which decides when it can run.

Blocked State :-

- Whenever A thread is temporary inactive , it could be blocked or waiting state.
- When you use t.join() , t.sleep() , t.wait() , t.suspend() the thread is switched to Blocked state.

Dead State :-

- the thread was terminated abnormally or the thread has either finished its execution.
- Use Stop method.
- Syntax :- t.stop();

o Synchronization and Inter-thread Communication

→

1.Synchronization : Synchronization is the process of controlling access to shared resources to prevent conflicts (such as data races) and ensure that only one thread can access a critical section of code or data at a time.

2. Inter-thread Communication in Java

Threads often need to communicate or synchronize their execution based on certain conditions. Java provides mechanisms like wait(), notify(), and notifyAll() for inter-thread communication, which are typically used in conjunction with synchronized blocks or methods.

a) wait(), notify(), and notifyAll()

These methods are defined in the Object class and are used for communication between threads that are waiting for some condition to be met.

- **wait()**: Causes the current thread to release the lock and enter the waiting state. It must be called from a synchronized context.
- **notify()**: Wakes up one thread that is waiting on the object's monitor (lock).
- **notifyAll()**: Wakes up all threads that are waiting on the object's monitor.

Example: Producer-Consumer Problem

14. File Handling

o Introduction to File I/O in Java (java.io package)

→ File input/output permenent to store data into file.

File it self class

You can create any type of file like .jpg , .txt, .docs, .xlsx

its derived from java.io package

having stream

- **Byte Stream**
  - **OutputStream:** to write data into files.
  - **InputStream:** to read data from files.

- **Character Stream**
  - **Writer: to write data into files.**
  - **Reader: to read data from files.**

o FileReader and FileWriter Classes

→

1)FileReader :-

- The FileReader class is used to read the content of text files .
- It part of the java.io package and is specifically designed for reading files as streams of characters.
- Syntax :- FileReader fr = new FileReader("output.txt")

Example :-

```
package com.File;

import java.io.FileReader;

import java.io.IOException;

public class FileReaderDemo1 {

	public static void main(String[] args)   {

	    try (FileReader fr = new FileReader("output.txt"))  {

	       int i;

	       while ((i = fr.read()) != -1)  {

	          System.out.print((char) i);

	       }

	    }

	    catch (IOException e) {

	       e.printStackTrace();

	   }
```

```
    }

}
```

2) File Writer :-

- The FileWriter class is used for writing characters to a text file.
- It is also part of the java.io package and is used when you need to write text-based data (strings or characters) to files.
- Syntax :- FileWriter fw = new FileWriter("output.txt")

Example :-

```
package com.File;

import java.io.FileWriter;

import java.io.IOException;

public class FileWriterDemo {

    public static void main(String[] args)  {

        try (FileWriter fw = new FileWriter("output.txt")){

            fw.write("Hello, World!");

            fw.write("\nThis is a test.");

            fw.flush();

            fw.close();

        }

        catch (IOException e) {

            e.printStackTrace();

        }

    }

  }
```

o BufferedReader and BufferedWriter

→ **BufferedReader** class wraps reader **FileReader** class.

BufferedWriter class wraps around writer **FileWriter** class.

o Serialization and Deserialization

→

- Serialization is a process of converting an object stat into a byte stream, and save , transmission , reconstructor deserialization
- Deserialization is a reverse process of serialization, converting byte stream to object stat.

15. Collections Framework

o Introduction to Collections Framework

→

Collection:

:is a group of object into single object

:its derived from java.util package

:Collection its self interface

o List, Set, Map, and Queue Interfaces

→

List interface :-

- The List Interface extends the collection interface and adds the method that are specific to lists , which are ordered collections that allow duplicate elements.
- Here are the some methods that are present in the list interface but not in the collection interface.

1) **ArrayList**
2) **LinkedList**
3) **Stack**
4) **Vector**

Set Interface :-

- represent a unordered Collection that are not allow duplicate elements.
- Set interface represents a unique elements. Set interface class is hashset class.

Map Interface :-

- Represents a collection of key-value pairs, where each key is unique and maps to a single value.
- It does not allow duplicate keys but allows duplicate values.
- Map interface class is HashMap class.

Queue :-

- Represents a collection designed for holding elements prior to processing. It follows the FIFO (First In, First Out) order.
- Queue follows FIFO order, used for holding elements before processing.

o ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap

→

1) ArrayList :its represent like dynamic array

:automatically shrink and grow both

:Default size is 0

:duplicate values are allow

:when you add some value in it , so same way display on console

:add() and remove()

:default symbol is "[]"

2) HashSet :auto implemented Set interface

:its represent like dynamic array

:automatically shrink and grow both

:Default size is 0

:duplicate values are not allow

:all the value has an hashkey

:all hashkey convert into hashcode

:all the value display via hashcode wise

:add() and remove()

:default symbol is "[]"

3) HashMap:auto implemented Map interface

:its represent like dynamic array

:automatically shrink and grow both

:Default size is 0

:its work look like in pair<k,v>

:duplicate pair are not allow,if key is same so value override

:all the pair has an hashkey

:all hashkey convert into hashcode

:all the pair display via hashcode wise

:put(),get() and remove()

:default symbol is "{}"

4)LinkedList : auto implemented List interface

:its represent like dynamic array/ doubly linked list

:automatically shrink and grow both

:Default size is 0

:duplicate values are allow

:when you add some value in it , so same way display on console

:add() and remove(),addfirst() and removefirst(),addlast() and removelast()

:default symbol is "[]"

5)TreeSet:

- **Auto-implemented** SortedSet interface.
- It represents a **sorted collection**.
- **Automatically grows and shrinks** as elements are added or removed.
- The **default size** is 0.
- **Duplicates are not allowed** in a TreeSet.
- **All values are displayed** in **ascending order**
- **All values have a unique position** based on their sorted order.
- **Elements are stored in a red-black tree** for efficient searching, insertion, and deletion. add() , remove() , first() , last()
- The **default symbol** used is [] (square brackets), but it is typically displayed in curly braces {} in the actual implementation.

6)TreeMap:

- **Auto-implemented** Navigable Map interface.
- It represents data as **key-value pairs**, stored in **sorted order** based on the keys.
- **Automatically grows and shrinks** as elements are added or removed.
- The **default size** is 0.
- It works by storing **pairs** in the form <key, value>.
- **Duplicate keys are not allowed**. If a key already exists, the **value is overridden** with the new one.
- **All pairs have a unique position** based on their sorted order (ascending or according to the comparator).
- **All keys have a hash key**, and they are converted into **hash codes** for efficient storage and retrieval.
- **Pairs are displayed in ascending order of keys**.
- **Elements are stored in a red-black tree** (a type of balanced binary search tree), which allows for **efficient searching, insertion, and deletion** operations.
- The **default symbol** used is {} (curly braces), representing key-value pairs as {key=value}.

o Iterators and ListIterators

→ iterators is object that use loop throws collaction arraylist , hashset etc..

Iterator is fetch the value from the collection object like a list , set and map.

Example :

```
Iterator<String> it=fruits.iterator();

while(it.hasNext()){
```

```
            System.out.println("Iterator "+it.next());

        }
```

ListIterators :-

- ListIterator allows traversal in both forward and backward directions . ListIterators only use in arraylist and linked list.

16. Java Input/Output (I/O)

o Streams in Java (InputStream, OutputStream)

→ Strems in java has two type :

1) byte stream : per byte
2) character stream : per character 2 byte

1.InputStream : to read the data from the file

:FileInputStream

:ObjectInputStream

: Buffered Input stream

2.OutputStream : to write the data into the file

:FileOutputStream

:ObjectOutputStream

: buffered output stream

1) file Input stream :- The input stream is read the data byte from a file.

Example :

```
public class FileInputStreamDemo {
```

```
    public static void main(String[] args) {
            try {
                    FileInputStream fs = new FileInputStream("fisrtFile.txt");
               int byteRead;
               while ((byteRead = fs.read()) != -1) {
                  System.out.print((char) byteRead);
               }
                 fs.close();
            }
            catch (IOException e) {
              e.printStackTrace();
            }
    }
}
```

FileOutputstream :- to write the data from byte stream in a file.

Example :-

```
public class FileOutputStreamDemo
{
    public static void main(String[] args) {
            File f1=new File("fisrtFile.txt");
            try {
                    FileOutputStream fio=new FileOutputStream(f1);
                    String s="Hello Devloper";
```

```
                    byte[] b=s.getBytes();

                    fio.write(b);

                    fio.flush();

                    fio.close();

                    System.out.println("Data Is write into file...");

                }

                catch (Exception e) {

                    e.printStackTrace();

                }

        }

    }
```

o Reading and Writing Data Using Streams

→ To read the file data

- FileInputStream
- BufferedInputStream
- FileReader
- BufferedReader

To write the file data

- FileOutputStream
- BufferedOutputStream
- FileWriter
- BufferedWriter

o Handling File I/O Operations

→

```
system.out.println("return Boolean value of file dorectory:"+f.isdirectory());
        system.out.println("return the file yes or no:"+f.isfile());
        system.out.println("return the file name is:"+f.getname());
        system.out.println("can file execute or not the file:"+f.canExecute())
        system.out.println("return the location of file:"+f.getAbsolutepath());
        system.out.println("file path is:"+f.getpath());
        system.out.println("file is readble or not return:"+f.canRead());
        system.out.println("file is writable or not return:"+f.canWrite());
        system.out.println("return the file size is:"+f.length());
```