**JBDL notes Contd…**

**16th Nov 2024**

***(I)Introduction to JAVA:***

● HighLevel Language(designed to be easy for humans to read and write.)

● Class Based Object Oriented Language(uses classes as blueprints to create objects, encapsulating data and behavior, which allows for modular, reusable, and organized code that models real-world entities effectively.)

● Write Once Run Anywhere(Platform Independent because java code is converted into bytecode)

***(II)How Java is Platform Independent:(Behind the scene)***

● Java Virtual Machine

● Java RunTime Environment

● Java Development Kit

javac converts the code into bytecode which is same for all platforms and then jvm after verfication converts to machine code that gets executed

Just-In-Time (JIT) compiler is in jvm before interpretation of bytecode to machine code it goes through jit it optimizes the redundant codes so that redundant codes are not interpretated again and again .

Compiling means source code to bytecode which is the intermediate code .Running or execution is the interpretation line by line from bytecode to machine code.

***(III)Classes and Objects in Java:***

● **Classes**: Blueprint of objects that describes the states and behaviors of any

class’s object.

● **Objects**: Entities which have states, behaviors and entities.

● **Instance Variable:** Within a class but outside any method.

● **Static Variable:** Outside any method within a class with a static Keyword.

● **Local Variable:** inside the method and this gets destroyed once the method is

completed.

**Accessing the Instance Level variabl**e is only possible with an object.

**Accessing the Static Variable** is only possible using the classname.name of the variable.

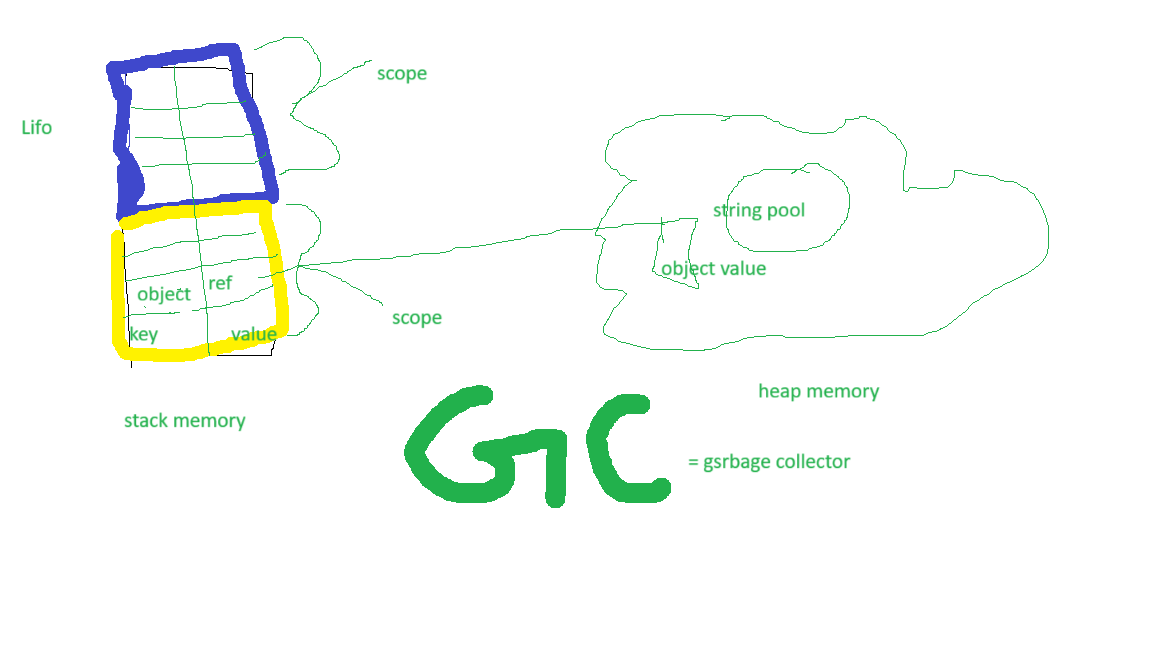
| **Property** | **Local Variable** | **Instance Variable** | **Static Variable** |
| --- | --- | --- | --- |
| **Definition** | Declared within a method, constructor, or block. | Declared within a class but outside any method, constructor, or block. | Declared using the static keyword, shared across all instances of the class. |
| **Lifetime** | Exists only during the execution of the method, constructor, or block. | Exists as long as the object containing it exists. | Exists as long as the class is loaded into memory. |
| **Initialization** | Must be explicitly initialized before use. | Automatically initialized to default values (e.g., 0 for int, null for objects) if not explicitly initialized. | Automatically initialized to default values if not explicitly initialized. |
| **Scope** | Limited to the method, constructor, or block in which it is declared. | Tied to the instance of the class and can be accessed via the object reference. | Tied to the class and can be accessed directly using the class name or through any instance. |
| **Access Modifiers** | Cannot have access modifiers (e.g., public, private). | Can have any access modifier (public, protected, private, or default). | Can have any access modifier (public, protected, private, or default). |
| **Copy per Object** | No copy is associated with the object (stack-bound variable). | Each object gets its own copy of the instance variable. | A single copy is shared across all instances of the class. |
| **Memory Location** | Stored in the **stack** memory. | Stored in the **heap** memory as part of the object. | Stored in the **Metaspace** as part of the class metadata (or heap in some JVM implementations). |
| **Access** | Accessible only within the method, constructor, or block where it is defined. | Accessible via the object reference or this keyword within instance methods. | Accessible using the class name (e.g., ClassName.variableName) or an object reference. |
| **Example** | int sum = 0; (inside a method). | int age; (inside the class, outside any method). | static int count; (inside the class, marked as static). |

***(IV)Memory Management of Java:***

● **Stack Memory** : Short lived and method specific values are kept inside it.

● **Heap Memory**: Dynamic memory allocation for objects at run time.

● **Meta Space Memory**: Memory for class metaData.(static block print)



1. Every mthod has scope in in stack memory it consist of local variables and object refrences in a key value format.
2. Object value is in heap memory and it is linked with stack string which are not defined like objects are in String pool
3. As end of scope reaches in a lifo style scope and its variables and object refrences starts getting destroyed.
4. After the stack memory destruction Garbage collector look for all the objects and string and array whose refrence link is destroyed and is then destroyed and cleared of heap memory.
5. There is something called meta space memory containing static variables and meta data like names return type etc of class

***(V)Constructors:***

 **Purpose**: A constructor is a special method used to initialize objects of a class.

 **Name**: It has the same name as the class.

 **No Return Type**: It does not have a return type (not even void).

 **Automatic Invocation**: Automatically called when an object is created using the new keyword.

 **Overloading**: Can be overloaded with multiple constructors having different parameter lists.

 **Default Constructor**: If no constructor is explicitly defined, the compiler provides a default (no-arg) constructor.

 **Parameterized Constructor**: Can take arguments to initialize fields with specific values.

● No argument constructor also possible

 **Chaining**: Can call other constructors of the same class using this() or parent class constructors using super().

 **Modifiers**: Can have access modifiers (public, private, protected, or default).

// 1. Default Constructor // Automatically provided by the compiler if no constructors are explicitly defined. // However, if any constructor is defined, the default constructor is NOT provided.

Example() { }

// 2. No-Argument Constructor (explicitly defined)

Example() {

this.value = 0; // Default value

this.name = "Default Name";

System.out.println("No-Argument Constructor called"); }

// 3. Parameterized Constructor

Example(int value, String name) {

this.value = value; // Set to provided value

this.name = name; // Set to provided name

System.out.println("Parameterized Constructor called"); }

***(VI)O.O.P.s(java don’t follows pure oops since it allows non primitive data types as well pure eg. smalltalk)***

Object-Oriented Programming (OOP) is a programming paradigm where real-world entities are modeled as objects, and problems are solved by interacting with these objects. It focuses on data (attributes) and the behaviors (methods) of objects, rather than functions and logic. ***(VII)Encapsulation And Data Hiding:***

Hiding internal state(from the outside classes) and requiring all interactions to be performed

through an object’s publicly exposed methods is known as Encapsulation.

Wrapping the code into a single unit to provide you the control over data. We can make

read-only and write-only classes with this. Easy to test, can add custom functionalities while

getting and setting the data.

(Bundling of state and behavior into a unit class hiding important things making visibe only few things that are required to ensure functionality and desired work.)

**To achieve encapsulation:**

1. To make private variables and public getter setters to access and update the

values.

To achieve Data Hiding:

1. No public getter and setters for any private instance variable and that can only be accessed within the same class or package.

***(VIII)Access Modifiers:***

Below are the modifiers for classes, attributes, methods and constructors:

**Modifiers Description**

public That code can be accessed from anywhere in the code

protected That code can be accessed from within the declared same

package or subclass from some outside package

private That code can be accessed from within the declared class

default That code can be accessed from within the same declared

package

Access Levels:

Modifier Class Package Subclass World

public Y Y Y Y

protected Y Y Y N

default Y Y N N

(no modifier)

private Y N N N

example of encapsulation:-

package AccessModifier;  
  
public class AccessModifier {  
 private String id;  
  
 public String getDesc() {  
 return desc;  
 }  
  
 public void setDesc(String desc) {  
 this.desc = desc;  
 }  
  
 private String desc;  
  
 public String getId() {  
 return id;  
 }  
  
 public void setId(String id) {  
 this.id = id;  
 }  
}

package AccessModifier;  
public class Main {  
 public static void main (String args[]){  
 AccessModifier obj = new AccessModifier();  
 //System.out.println(obj.id); it wil cause error  
 obj.setId("23");  
 obj.setDesc("Average performer");  
 System.*out*.println(obj.getId()+" "+obj.getDesc());}}

***(IX)Advantages of Encapsulation:***

● Data Hiding (only getter and setter at most available)

● Flexibility to make class readable, writable(can have both getter and setter or only getter or only setter)

● Reusability (circle class radius private but getArea and getCircumference are public that can be reused)

● Code testing becomes easier

***(X)Singleton class pattern:***

**Defination :-**Class which has setting done in a way only one object for it can be made.

This is because there are times where we want such thing specially in multithreading .

How to implement :-

1. A) Lazy instantiation

Singleton Class:-

public class Singleton {

// Static variable to hold the single instance

private static Singleton instance;

// Private constructor to prevent instantiation from other classes

private Singleton() {}

// Public method to provide the single instance

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton(); // Create the instance lazily

}

return instance;

}

// Example method

public void showMessage() {

System.out.println("Hello from Singleton!");

}

}

Another Class Using the Singleton:-

public class AnotherClass {

public void useSingleton() {

// Access the Singleton instance

Singleton singleton = Singleton.getInstance();

// Call a method on the Singleton instance

singleton.showMessage();

}

public static void main(String[] args) {

AnotherClass obj = new AnotherClass();

obj.useSingleton();

}

}

OUTPUT:-

Hello from Singleton!

B) We can make multiple object but they will be same

public void useSingleton() {

// Access the Singleton instance

Singleton singleton = Singleton.getInstance();

Singleton singleton2 = Singleton.getInstance();

// Call a method on the Singleton instance

singleton.showMessage();

singleton2.showMessage();

System.out.println(singleton == singleton2); // true

}

OUTPUT:-

Hello from Singleton!

Hello from Singleton!

true

C) Parameterized

Singleton Class:-

public class Singleton {

private static Singleton instance;

private String value;

// Private parameterized constructor

private Singleton(String value) {

this.value = value;

}

// Parameterized getInstance method

public static Singleton getInstance(String value) {

if (instance == null) {

instance = new Singleton(value); // Create instance with the parameter

}

return instance;

}

// Method to display the stored value

public void showMessage() {

System.out.println("Value: " + value);

}

}

Main Class:-

public class Main {

public static void main(String[] args) {

// First call to getInstance with a parameter

Singleton singleton1 = Singleton.getInstance("First Instance");

singleton1.showMessage(); // Output: Value: First Instance

// Second call to getInstance with a different parameter

Singleton singleton2 = Singleton.getInstance("Second Instance");

singleton2.showMessage(); // Output: Value: First Instance (unchanged)

// Verify both references are the same

System.out.println(singleton1 == singleton2); // Output: true

}

}

OUTPUT:-

Value: First Instance

Value: First Instance

true

D) Updating we need to have public method to update value

Singleton Class:-

public class Singleton {

private static Singleton instance;

private String value;

// Private parameterized constructor

private Singleton(String value) {

this.value = value;

}

// Parameterized getInstance method

public static Singleton getInstance(String value) {

if (instance == null) {

instance = new Singleton(value); // Create instance with the parameter

}

return instance;

}

// Method to display the stored value

public void showMessage() {

System.out.println("Value: " + value);

}

// Method to update the value

public void updateValue(String newValue) {

this.value = newValue;

}

}

Main Class:-

public class Main {

public static void main(String[] args) {

// First call to getInstance with a parameter

Singleton singleton1 = Singleton.getInstance("First Instance");

singleton1.showMessage(); // Output: Value: First Instance

// Update the value using the updateValue method

singleton1.updateValue("Updated Value");

singleton1.showMessage(); // Output: Value: Updated Value

// Get the Singleton instance again (it will be the same instance)

Singleton singleton2 = Singleton.getInstance("Second Instance");

singleton2.showMessage(); // Output: Value: Updated Value

// Verify both references are the same

System.out.println(singleton1 == singleton2); // Output: true

}

}

OUTPUT:-

Value: First Instance

Value: Updated Value

Value: Updated Value

true

2)Early Instantiation

public class Singleton {

// Early instantiation: Instance is created when the class is loaded

private static final Singleton instance = new Singleton();

// Private constructor to prevent instantiation from other classes

private Singleton() { System.out.println("Singleton instance created"); }

// Public method to provide access to the instance

public static Singleton getInstance() {

return instance; } }

3)Bill Pugh Singleton (Best Practice) (thread safety )

public class Singleton {

private Singleton() {}

private static class SingletonHelper {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHelper.INSTANCE;

}

}

***17th Nov 2024***

***(I)Inheritance:***

Receiving state and behavior from parent to child I java .

**Terminologies for Inheritance:**

**SuperClass:** Parent class/Base class from which features and behaviors are getting inherited.

**SubClass:** Child class/Extended class/ Derived class which is inheriting the parent class. It can

have other functionalities as well.

**Reusability:** We are always re-using the fields and methods into the subclass.

**Types of Inheritance in JAVA:**

1. Single Level : Inherit features and methods from one superclass.

2. Multi Level: Derived class is acting as parent class for some other child class.

3. Hierarchical: One Base class is acting as parent for more than one child class.

4. Multiple Inheritance: One class has more than one superclass.(Not allowed using

classes in java)(diamond problem child don’t know which one to inherit)

5. Hybrid Level: Mix of any two types of above discussed Inheritance.

**extends** keyword followed with parent class name in child class to implement parent class.

Super keyword for refrencing parent . **super()** for contructor, **super.value** & **super.speak()**

**Features of Inheritance in JAVA:**

● Default SuperClass : object class is superclass to everything.

● SuperClass can only be one : due to multiple inheritance not allowed, can have

only one superclass. And also cyclic thing cant be done .

● Inheriting constructor : constructors do not get inherited but can be called via

subclass constructor. Using **super()**

● Private member inheritance : child class does not inherit private members of

parent class.

**Important points about inhertiance:**

**1.** Parent Constructor gets called first when we create child class object.

Example:-

package JBDL\_2\_17nov2024;  
  
public class Animal {  
 public Animal() {  
 System.*out*.println("Inside Animal class");  
 }  
  
 public void speak(){  
 System.*out*.println("Animal speaking");  
 }  
}

package JBDL\_2\_17nov2024;  
  
public class Dog extends Animal{  
 public Dog() {  
 System.*out*.println("Inside Dog class");  
 }  
  
 public static void main (String args[]){  
 Dog tom = new Dog();  
 tom.speak();  
 }  
}

OUTPUT:-

Inside Animal class

Inside Dog class

Animal speaking

2. Object class is default superClass to all classes.The memory is created to child object and

inside that we have section for parent class. Eg. Use of this is every class’s object can use .hashCode() method which is a method of Object class.

3. Overloading(same method name different parameters) and overriding(same method in child and parent child is given prefrence)

EXAMPLE:-

package JBDL\_2\_17nov2024;

class Animal {

public Animal() {

System.out.println("Inside Animal class");

}

// Overloaded methods in the Animal class

public void speak() {

System.out.println("Animal speaking");

}

public void speak(String sound) {

System.out.println("Animal makes sound: " + sound);

}

}

class Dog extends Animal {

public Dog() {

System.out.println("Inside Dog class");

}

// Overriding the speak method from the Animal class

@Override

public void speak() {

System.out.println("Dog barking");

}

// Overloaded method in the Dog class

public void speak(int times) {

System.out.println("Dog barks " + times + " times");

}

public static void main(String[] args) {

Dog tom = new Dog(); // Constructor calls

tom.speak(); // Calls overridden method in Dog

tom.speak("woof"); // Calls overloaded method in Animal

tom.speak(3); // Calls overloaded method in Dog

// Using Animal reference to call speak

Animal genericAnimal = new Animal();

genericAnimal.speak(); // Calls speak in Animal

genericAnimal.speak("growl"); // Calls overloaded method in Animal

}}

OUTPUT:-

Inside Animal class

Inside Dog class

Dog barking

Animal makes sound: woof

Dog barks 3 times

Inside Animal class

Animal speaking

Animal makes sound: growl

Inheritence has **Is-a relation** child is a parent . it will have access to all parent methods.

Also there is **tight coupling** any changes in parent will have change in child.

**Has-a** **relation**

I wanna have few function in class which I don’t want to give to another class.

Eg. There is class Student and Address . Address has 2 instance variable city and pincode . Since all students are from same city not necessary to give it to Student class. **Association** ->is a relationship between two or more classes in object-oriented programming, where objects of one class can interact with objects of another class. It defines how objects are connected to each other and how they communicate.

***(II)Aggregation & Composition:-***

● When two classes have weak relations between them i.e Aggregation(child itself also make sense)

package JBDL\_2\_17nov2024.Has\_a\_aggregration;  
  
public class Address {  
 String city;  
 String pincode;  
  
 public Address(String pincode, String city) {  
 this.pincode = pincode;  
 this.city = city;  
 }  
  
 public String getPincode() {  
 return pincode;  
 }  
  
 public void setPincode(String pincode) {  
 this.pincode = pincode;  
 }  
  
  
 public String getCity() {  
 return city;  
 }  
  
 public void setCity(String city) {  
 this.city = city;  
 }  
   
}

package JBDL\_2\_17nov2024.Has\_a\_aggregration;  
  
public class Student {  
 String name;  
 Address address;  
  
 public Student(String name, Address address) {  
 this.name = name;  
 this.address = address;  
 }  
  
 public void printPincode(){  
 System.*out*.println(address.pincode);  
 }  
}

package JBDL\_2\_17nov2024.Has\_a\_aggregration;  
  
public class Main {  
 public static void main(String args[]){  
 Address address = new Address("831009","jamshedpur");  
 Student student = new Student("Shivanshu",address);  
 student.printPincode();  
 }  
}

● When two classes have strong relations between them i.e. Composition. Sort of

Parent class object created child object created and child object can’t live without parent object. .(for child to exist parent should exist without parent it has no significance and it dies with parent)

package JBDL\_2\_17nov2024.Has\_a\_composition;  
  
public class Car {  
 String name;  
 Wheel wheel;  
  
 public Car(String name) {  
 this.name = name;  
 this.wheel = new Wheel();  
 }  
}//wheel gets destoyed when Car class gets destroyed

package JBDL\_2\_17nov2024.Has\_a\_composition;  
  
public class Wheel {  
 String name;  
}

|  |  |
| --- | --- |
| **Is-a Relation** | **Has-a Relation** |
| 1) Can access by creating an object in extended class | 1) Can access by direct creating an object in any class |
| 2) Reusability is the advantage | 2) Accessing any class by creating an object |
| 3) Tightly Coupled | 3) Non Tightly Coupled |

***(III)Polymorphism:-***

One name, many forms. A fundamental concept of object-oriented programming (OOP) that allows objects to take on multiple forms.

1. Compile Time Polymorphism (static polymorphism or OverLoading)

Method overloading occurs when multiple methods in the same class share the same name but have different parameter lists (type, number, or both).

Example:-

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println(calc.add(5, 10)); // Calls int version

System.out.println(calc.add(5.5, 10.5)); // Calls double version

}

}

2. RunTime Polymorphism (Dynamic Method Dispatch or OverRiding)

 Achieved using **method overriding** and **upcasting**.

 A child class provides its implementation for a method defined in the parent class.

 A parent class reference can point to a child class object.

Polymorphism in Java is a concept that allows objects of different classes to be

treated as objects of a common class.

example

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal a;

a = new Dog(); // **Upcasting**

a.sound(); // Calls Dog's implementation

a = new Cat(); // **Upcasting**

a.sound(); // Calls Cat's implementation

}

}

3. Upcasting and Downcasting :- we have seen upcasting above but what is downcasting when do we do it and how to do it correctly?

If you try to call a method that is only defined in the Dog class, the code will result in a **compile-time error**. This is because the method you are trying to call is not part of the Animal class, and Java checks the availability of methods based on the **reference type** (Animal in this case) during compilation.

Example:-

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

void dogSpecificMethod() {

System.out.println("Dog wags tail");

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Dog();

a.sound(); // Works: Calls Dog's implementation of sound()

// a.dogSpecificMethod(); // Compile-time error

// Correct way to call Dog-specific method

if (a instanceof Dog) {

Dog d = (Dog) a; //**Downcasting converting to child**

d.dogSpecificMethod(); // Works

}

}

}

***(IV)Interfaces:-***

In Java, an **interface** is a blueprint of a class that contains **abstract methods** (methods without a body) and **static or default methods** (introduced in Java 8). It helps in abstraction and multiple inheritence

EXAMPLE:-

interface Animal {

// Abstract method

void sound();

// Default method (introduced in Java 8)

default void sleep() {

System.out.println("Sleeping...");

}

// Static method (introduced in Java 8)

static void info() {

System.out.println("This is an Animal interface");

}

}

class Dog implements Animal {

@Override

public void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Calls overridden method

dog.sleep(); // Calls default method from interface

Animal.info(); // Calls static method from interface

}

}

**Output**:

Dog barks

Sleeping...

This is an Animal interface

**Solving multiple inheritence**

interface A {

void methodA();

}

interface B {

void methodB();

}

class C implements A, B {

public void methodA() {

System.out.println("Method A");

}

public void methodB() {

System.out.println("Method B");

}

}

public class Main {

public static void main(String[] args) {

C obj = new C();

obj.methodA();

obj.methodB();

}

}

**Output**:

Method A

Method B

**Diamond problem solution:-**

interface A {

default void display() {

System.out.println("Display from A");

}

}

interface B {

default void display() {

System.out.println("Display from B");

}

}

class C implements A, B {

// No implementation for display() here will cause a compile-time error.

}

Solution

class C implements A, B {

@Override

public void display() {

// Explicitly choose which default method to call

A.super.display();

B.super.display();

System.out.println("Display resolved in C");

}//or simply write return A.super.display(); or return B.super.display();

}

public class Main {

public static void main(String[] args) {

C obj = new C();

obj.display();

}

}

**default** using it I can make function in interface with body which if not implemented in child also it wont hamper and can be overridden in child and also default does not mean access specifier here

EXAMPLE1:-

interface Animal {

default void sound() {

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

}

}

class Dog implements Animal {

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

dog.sound(); // Calls the overridden method in Dog

}

}

OUTPUT

Animal makes a generic sound

EXAMPLE2:-

interface Animal {

default void sound() {

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

}

}

class Dog implements Animal {

@Override

public void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Animal dog = new Dog();

dog.sound(); // Calls the overridden method in Dog

}

}

OUTPUT

Dog barks

**Static** methods belong to the interface and cannot be overridden.

interface Utils {

static void printMessage() {

System.out.println("Static method in interface");

}

}

public class Main {

public static void main(String[] args) {

Utils.printMessage(); // Calls static method directly

}

}

OUTPUT:-

Static method in interface

Interface can have **private methods** and can be used in other classes using a solution:-

interface Helper {

default void show() {

log("Default method called");

}

private void log(String message) {

System.out.println("Log: " + message);

}

}

public class Main {

public static void main(String[] args) {

Helper helper = new Helper() {};// its **anonymous inner class**

//interface cannot be instantiated directly, you cannot simply write Helper helper = new //Helper(); it either have something like this or override the default method here

helper.show();

}

}

**Variables** in interfaces are **public static final** by default.

Interfaces can **extend** other interfaces **but not class**. And the one that implement the child has to implement the functions of parent and child eg.

interface Animal {

void eat();

}

interface Pet extends Animal {

void play();

}

class Dog implements Pet {

public void eat() {

System.out.println("Dog eats");

}

public void play() {

System.out.println("Dog plays");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat();

dog.play();

}

}

Since overriding is happening therefore interfaces also help to achieve **Polymorphism**

***(V)Abstraction:***

using some interfaces or abstract classes and show only definition of method to the outer world that is known as abstraction in java. **Its hiding of implementation details.**

There are some points that need to be considerd for its use properly:-

1. Abstract method is a method with no body .
2. Abstract class can have concrete or normal method and abstract methods or both no problem but abstract method cannot be in non abstract method.
3. It is necessary for child of abstract class to have abstract method implemented
4. Abstract class cant have its object made but have refrence eg.

**Animal tom = new Dog("Tom");**. Abstract class can have static variables and constructors also It can have private, protected, public variables and methods

1. No two abstract classes can be extended by a class.

Example:-

// Abstract class

abstract class Shape {

protected String color; // Instance variable

// Constructor to initialize color

public Shape(String color) {

this.color = color;

}

// Abstract method

abstract double area();

// Non-abstract method to access the variable

public String getColor() {

return color;

}

}

// Concrete subclass

class Circle extends Shape {

private double radius;

// Constructor

public Circle(String color, double radius) {

super(color); // Initialize the color variable in the parent class

this.radius = radius;

}

// Implement abstract method

@Override

double area() {

return Math.PI \* radius \* radius;

}

// Access the parent class variable

public void printDetails() {

System.out.println("Color: " + color); // Accessing the instance variable

System.out.println("Radius: " + radius);

}

}

// Main class

public class Main {

public static void main(String[] args) {

Circle circle = new Circle("Red", 5.0);

// Accessing instance variables through methods

System.out.println("Color of circle: " + circle.getColor());

// Accessing instance variables directly (if they are protected/public)

circle.printDetails();

}

}

Output:

Color of circle: Red

Color: Red

Radius: 5.0

When to use Abstract Classes or Interfaces:

1. When reading problem we can have idea classes are related(inheritance style relation) and will have shared functionality then use abstract class when we expect that unrelated classes would define contracts or capabilities for unrelated classes we use interface (same function in unrelated classes relationship ).
2. If we want multiple inheritance we will have Interface
3. I we wanna make like this **Animal tom = new Dog("Tom");**. And constructors We use abstract class
4. Want to have instance variables, static variables, and constants Abstract class for Interface only have constants static final.
5. Can use all access modifiers (private, protected, public) (also public will always be public in child while protected can be public in child class use and both cant be made private we can only increase the visiblity) Abstract class for interface its always public

***(VI)Inner Class:-***

An **inner class** is a class defined within another class (the **outer class**). Inner classes are often used when the logic of the inner class is tightly coupled with the outer class and won't be reused elsewhere.

#### **Characteristics:**

1. Inner classes have access to the private and public members of the outer class.
2. They are used to logically group classes that are used only by the outer class.

Code Example for Inner Classes

class OuterClass {

private String privateMessage = "Outer private message";

public String publicMessage = "Outer public message";

// Inner Class

class InnerClass {

void displayMessages() {

// Accessing outer class private and public variables

System.out.println("Private Message: " + privateMessage);

System.out.println("Public Message: " + publicMessage);

}

}

}

public class Main {

public static void main(String[] args) {

// Creating an OuterClass instance

OuterClass outerClass = new OuterClass();

// Creating an InnerClass instance

OuterClass.InnerClass innerClass = outerClass.new InnerClass();

// Calling inner class method

innerClass.displayMessages();

}

}

\*If the inner class is declared **static**

OuterClass.InnerClass innerClass = new OuterClass.InnerClass();

\* System.out.println(innerClass.privateMessage); // Compile-time error

A **final class** is a class that **cannot be extended** (inherited) by any other class.

final class FinalClass {

void display() {

System.out.println("This is a final class.");

}

}

// This will cause a compile-time error:

// class SubClass extends FinalClass { }

public class Main {

public static void main(String[] args) {

FinalClass finalClass = new FinalClass();

finalClass.display();

}

}

***(VII)Enum:-***

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

Every enum in Java **implicitly extends the java.lang.Enum class**. Therefore, enums **cannot extend any other class** but can implement interfaces.

Enums can have fields, constructors, and methods to associate additional information with each constant.

enum Day {

SUNDAY(0, "Sun"),

MONDAY(1, "Mon"),

TUESDAY(2, "Tue");

private int id;

private String abbreviation;

// Enum constructor

Day(int id, String abbreviation) {

this.id = id;

this.abbreviation = abbreviation;

}

// Getter methods

public int getId() {

return id;

}

public String getAbbreviation() {

return abbreviation;

}

}

public class EnumExample {

public static void main(String[] args) {

// Accessing enums

Day today = Day.SUNDAY;

// Accessing fields and methods

System.out.println("Day: " + today); // Prints SUNDAY (name of the constant)

System.out.println("ID: " + today.getId()); // Prints 0

System.out.println("Abbreviation: " + today.getAbbreviation()); // Prints Sun

}

}

Output:-

Day: SUNDAY

ID: 0

Abbreviation: Sun

|  |
| --- |
| values() |

|  |
| --- |
| Returns an array of all enum constants. |

for (Day day : Day.values()) {

System.out.println(day);

}

|  |
| --- |
| valueOf(String) |

|  |
| --- |
| Returns the constant matching the given name; throws an exception if not found. |
| Day day = Day.valueOf("MONDAY");  System.out.println(day); // Prints MONDAY  Day firstDay = Day.values()[0] |
| |  | | --- | | ordinal() |  |  |  | | --- | --- | | Returns the position (index) of the enum constant. | | | System.out.println(Day.SUNDAY.ordinal()); // Prints 0  System.out.println(Day.TUESDAY.ordinal()); // Prints 2 | | | name() |  |  |  | | --- | --- | | Returns the name of the constant as declared. | | | toString() |  |  | | --- | | Returns the name of the constant (can be overridden). | | System.out.println(Day.MONDAY.name()); // Prints MONDAY  System.out.println(Day.MONDAY.toString()); // Prints MONDAY | |

\***Enums Implementing Interfaces**

interface Greeting { void greet(); }

enum Day implements Greeting { SUNDAY, MONDAY, TUESDAY;

@Override public void greet() { System.out.println("Hello! Today is " + this.name()); } }

Eg. MALE,FEMALE

***24th November 2024***

#### **What is an Exception?**

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions.

## Exceptionis an abnormal Condition, We should handle the Exception.

## Case when normal flow gets interrupted.

|  |  |
| --- | --- |
| Exception | Error(irrecoverable conditions ) |
| **In control of program** | **Outside the control of program** |
| **Possible to catch the exception** | **Not possible to catch** |
| **Best Practice to tell user about the problem occurred and change in flow accordingly** | **Best Practice is to exit the program nicely and log the error.** |

Exapmples of errors :- Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

### 1. ****Compile-Time Errors****

These errors occur **before the program is executed**, during the compilation phase. They prevent the program from being compiled successfully.

#### Causes of Compile-Time Errors:

* **Syntax errors**: Issues in the syntax of the code (e.g., missing semicolons, incorrect use of keywords).
* **Type mismatches**: Assigning incompatible data types.
* **Unresolved references**: Using variables, methods, or classes that are not defined.
* **Access violations**: Trying to access private methods or variables inappropriately.
* **Missing exceptions handling**: Not handling checked exceptions.

// Syntax Error: Missing semicolon

int number = 10

// Type Mismatch

String s = 5; // Cannot assign an integer to a String

// Unresolved Reference

System.out.println(undeclaredVariable); // Variable is not declared

**Solution**: These errors must be fixed before the program can run, as Java won't generate bytecode (class files) for programs with compile-time errors.

### 2. ****Runtime Errors****

These errors occur **while the program is running**, after successful compilation. They are typically caused by invalid operations or unexpected conditions during execution.

#### Causes of Runtime Errors:

* **Arithmetic errors**: Dividing by zero.
* **Null pointer exception**: Accessing a method or property on a null object.
* **Array index out of bounds**: Trying to access an invalid array index.
* **Invalid user input**: Input data not matching the expected format.
* **Class cast exception**: Invalid type casting.
* **IllegalArgumentException:** It is thrown to indicate that a method has received an **argument that is invalid or inappropriate**.

// Arithmetic Error: Division by zero

int result = 10 / 0; // Throws ArithmeticException

// Null Pointer Exception

String str = null;

System.out.println(str.length()); // Throws NullPointerException

// Array Index Out of Bounds

int[] arr = {1, 2, 3};

System.out.println(arr[5]); // Throws ArrayIndexOutOfBoundsException

Object obj = "Hello"; // obj is a String

Integer number = (Integer) obj; // Trying to cast a String to Integer

if (percentage < 0 || percentage > 100) { throw new IllegalArgumentException("Discount percentage must be between 0 and 100: " + percentage); }

These are mostly exceptions that can be handled using try catch but there are errors as well that cannot be handled example

 **OutOfMemoryError**: When the JVM runs out of memory.

 **StackOverflowError**: Due to deep or infinite recursion.

 **VirtualMachineError**: Indicates critical issues with the JVM itself.

### 3. ****Logical Errors****

These errors occur when the program compiles and runs but produces **incorrect or unintended results** due to flaws in the logic of the code. Logical errors are often the most difficult to detect because they do not produce error messages.

#### Causes of Logical Errors:

* **Incorrect algorithms**: Using wrong formulas or algorithms.
* **Wrong conditions in loops or if statements**: Leading to infinite loops or skipped logic.
* **Incorrect variable assignments**: Assigning wrong values or swapping logic unintentionally.

// Incorrect Algorithm: Calculating the average without dividing

int sum = 100;

int count = 5;

int average = sum; // Forgot to divide by count (average should be 20)

// Incorrect Loop Condition

for (int i = 0; i <= 5; i--) { // Infinite loop because `i--` decreases instead of increasing

System.out.println(i);

}

// Wrong Conditional Logic

int x = 5, y = 10;

if (x > y) { // This condition is incorrect; x is not greater than y

System.out.println("x is greater than y");

}

**Solution**: Logical errors require **thorough debugging and testing** to detect and fix. Tools like debuggers, unit tests, and code reviews can help identify these issues.

|  |  |
| --- | --- |
| **Checked Exception** | **Unchecked Exception** |
| Checked at compile Time. | Checked at RunTime |
| If a method tells it throws an exception, we need to handle it using try, catch block or throws keyword. | We can handle it,by using try-catch or throws but it's not required by the compiler. |

**Checked Exceptions**: Enforced by the compiler. Handle or declare them.

**Unchecked Exceptions**: Not enforced by the compiler. Fix program logic to avoid them.

**Examples of Checked Exceptions:**

* **IOException** Trying to read from a file that doesn't exist.
* **FileNotFoundException** The file you are trying to access does not exist.
* **SQLException** thrown when there is an issue with accessing a **database**
* **ClassNotFoundException** Attempting to load a class that doesn't exist

Example for exception handelling and use of **try**-**catch** and **throws** keyword

import java.io.\*;

public class Main {

public static void main(String[] args) {

try {// here is where error might occur

readFile("nonexistent.txt");

} catch (IOException e) { // Handle all exceptions in one place

System.out.println("An error occurred: " + e.getMessage());

}}

public static void readFile(String fileName) throws IOException { // Declaring exception

FileReader file = new FileReader(fileName); // Might throw IOException

file.close();

}}// throws propagate all exceptions to the caller i.e. By using throws, you leave the responsibility of handling the exception to the method that calls your method. they can be handled in one central location, making the code cleaner and easier to maintain. If a method simply cannot handle an exception itself, throws avoids unnecessary redundant try-catch blocks. Moreover we can use both throws and try-catch to same function wont get any error but not good practice.

The **finally** block in Java is used to execute **cleanup code** that must be run **whether an exception is thrown or not**. It is primarily used for releasing resources (like closing files, database connections, etc.). It always executes example

**throw** explicitly throwan exception from your code. It can throw both checked and unchecked exceptions.

Example for finally :-

public class Main {

public static void main(String[] args) {

java.io.FileReader reader = null;

try {

reader = new java.io.FileReader("file.txt");

// Perform file operations

} catch (Exception e) {

System.out.println("Exception occurred: " + e.getMessage());

} finally {

if (reader != null) {

try {

reader.close(); // Ensure resource cleanup

} catch (Exception e) {

System.out.println("Error while closing the file.");

}

}}}}

Example for throw :-

public class Main {

public static void main(String[] args) {

int age = 15;

try {

checkAge(age);

} catch (IllegalArgumentException e) {

System.out.println("Caught exception: " + e.getMessage());

}

}

public static void checkAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be 18 or older."); // Explicitly throwing an exception

}

}

}

when there is exception and it goes to catch does it stops the execution or just prints the message in catch and goes on? It depends on program if there is no proper handling aka proper catch then it terminates while if it has proper handling it will continue example if there is ArthimaticException occurring and or catch is like catch(NullPointerException e) then it terminates it need to be either catch(Exception e ) or catch(ArthimaticException e )

**Try-with-Resources**

import java.io.\*;

public class Main {

public static void main(String[] args) {

try (BufferedReader reader = new BufferedReader(new FileReader("testfile.txt"))) {

String line;

while ((line = reader.readLine()) != null) {

System.out.println(line);

}

} catch (IOException e) {

System.out.println("IOException: " + e.getMessage());

}

// The BufferedReader is automatically closed at the end of the try block. Key //requirement is The object being used in the **try-with-resources** must implement the //AutoCloseable interface (or its subclass java.io.Closeable for I/O-related resources). This //interface defines a close() method that is automatically called when the block exits, ensuring //that resources are cleaned up. Another example only using FileReader (. implements Closeable)

}}

**Custom Exception**

class CustomCheckedException extends Exception {

public CustomCheckedException(String message) {

super(message);

}

}

public class Main {

public static void main(String[] args) {

try {

throw new CustomCheckedException("This is a checked exception");

} catch (CustomCheckedException e) {

System.out.println("Caught exception: " + e.getMessage());

} }}//for checked exceptions These exceptions must either be handled with a try-catch block or declared in the method signature using throws

class CustomUncheckedException extends RuntimeException {

public CustomUncheckedException(String message) {

super(message);

}

}

public class Main {

public static void main(String[] args) {

throw new CustomUncheckedException("This is an unchecked exception");

}

}//for unchecked exceptions

having 2 types of exception

\*public void methodName() throws Exception1, Exception2 { // Method implementation }

\*one try multiple catch best way is some catch and at last catch (Exception e)

**Equals(.equals())**: Method of Object class, by default checks if two objects are stored at the same location. It do Reference comparison.

Example:-

class Person {

private String name;

private int age;

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter methods

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

public class PersonTest {

public static void main(String[] args) {

Person person1 = new Person("John", 30);

Person person2 = new Person("John", 30);

Person person3 = new Person("Jane", 25);

Person person4 = person1;

System.out.println(person1.equals(person2)); // Reference comparison op->false

System.out.println(person1.equals(person3)); // Reference comparison op->false

System.out.println(person1.equals(person4)); // Same reference op->true

System.out.println(person2.equals(person4)); // Different reference op->false

}

}

.equals() can be overridden example

class Person {

private String name;

private int age;

// Constructor to initialize Person objects

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter methods for name and age

public String getName() {

return name;

}

public int getAge() {

return age;

}

// Setter methods for name and age

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

// Override equals() method to compare two Person objects

@Override

public boolean equals(Object obj) {

if (this == obj) return true; // Check if both references point to the same object

if (obj == null || getClass() != obj.getClass()) return false; // Null check and class type check

Person person = (Person) obj; // Cast the object to a Person

return age == person.age && name.equals(person.name); // Compare the attributes of both objects

}

}

public class PersonTest {

public static void main(String[] args) {

Person person1 = new Person("John", 30);

Person person2 = new Person("John", 30);

Person person3 = new Person("Jane", 25);

Person person4 = person1;

System.out.println(person1.equals(person2)); // op->true

System.out.println(person1.equals(person3)); //op->false

System.out.println(person1.equals(person4)); //op->true

System.out.println(person2.equals(person4)); //op->true

}

}

**Hash Code**: The hashCode() method is an integer value (a 32-bit signed integer) that is generated based on the internal state of the object, typically derived from the object’s fields. This value is used by hash-based collections to place objects into "buckets" for quick lookup. The **hash code** is **not** the memory address of an object

Can be overridden

In Person class

@Override public int hashCode() { return Objects.hash(name, age); // Hash based on name and age }

In PersonTest

// Comparing objects using equals() method System.out.println("person1.equals(person2): " + person1.equals(person2)); // true System.out.println("person1.equals(person3): " + person1.equals(person3)); // false System.out.println("person1.equals(person4): " + person1.equals(person4)); // true System.out.println("person2.equals(person4): " + person2.equals(person4)); // true

// Displaying hash codes

System.out.println("person1.hashCode(): " + person1.hashCode()); System.out.println("person2.hashCode(): " + person2.hashCode()); System.out.println("person3.hashCode(): " + person3.hashCode()); System.out.println("person4.hashCode(): " + person4.hashCode());

Output:-

person1.equals(person2): true

person1.equals(person3): false

person1.equals(person4): true

person2.equals(person4): true

person1.hashCode(): 2025260571

person2.hashCode(): 2025260571

person3.hashCode(): 149536222

person4.hashCode(): 2025260571

equal objects have same hashCode this is the **contract**

**Summary of equals() and hashCode() Contract:**

1. **If two objects are equal (equals()), they must have the same hash code**.
2. **equals() should be reflexive(true for same), symmetric(x==y so y==x), and transitive(x==y & y==z so x==z)**.
3. **equals() should return false when comparing to null**.
4. **hashCode() must be consistent for equal objects**.
5. **hashCode() may produce the same value for unequal objects, but this should be avoided**.

This unequal objects can still have the same hash code, but for hash-based collections, having different hash codes is preferred because it reduces hash collisions (when two objects with different values are placed in the same bucket).

**Iterable**

It is the root interface for all collection types that provide iteration functionality and also To provide a standard way to obtain an Iterator from a collection..

import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> names = new ArrayList<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

// Using for-each loop (Iterable is implicitly used)

for (String name : names) {

System.out.println(name);

}

}

}//Here, ArrayList implements Iterable, so it can be iterated using a for-each loop.

//ArrayList, HashSet, LinkedList, TreeSet, HashMap keys or values etc.

**Iterator** is an **interface** that provides methods to iterate over a collection in a sequential manner.

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

ArrayList<String> names = new ArrayList<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

Iterator<String> iterator = names.iterator(); // Get an iterator

while (iterator.hasNext()) { // Check if there are more elements

String name = iterator.next(); // Get the next element

System.out.println(name);

}

}

}

**remove()** method in an **Iterator**. This method is used to remove the last element returned by the next() method from the underlying collection during iteration.

Example:-

while (iterator.hasNext()) {

String name = iterator.next(); // Get the next element

if (name.startsWith("C")) {

iterator.remove(); // Remove the element and is void don’t return any thing

System.out.println("Removed: " + name); } }

System.out.println("Modified List: " + names); // Print the modified list

//Use is providing uniformity to different collections

import java.util.Iterator;

public class IterableDemo implements Iterable<String>{

String[] arr = new String[10];

@Override

public Iterator<String> iterator() {

return new MyIterator(0);

}

public class MyIterator implements Iterator<String>{

int index = 0;

public MyIterator(int index) {

this.index = index;

}

@Override

public boolean hasNext() {

return arr[index]!=null && index<10;

// return index<1;

}

@Override

public String next() {

String current = arr[index];

index++;

return current;

}

}//inner class

public static void main(String[] args){

IterableDemo iterableDemo = new IterableDemo();

iterableDemo.arr[0]="abcd";

iterableDemo.arr[1]="def";

iterableDemo.arr[2]="ghi";

//to give unfiormaity

Iterator<String> iterator = iterableDemo.iterator();

while(iterator.hasNext()){

System.out.println(iterator.next());

}

}

}