**Conditional Statements:**

In Java, conditional statements are used to make decisions based on conditions. They help control the flow of a program by executing certain blocks of code when specific conditions are true.

Here’s a simple breakdown:

**1. if statement:**

- The `if` statement is used to execute a block of code only if a specified condition is `true`.

Syntax:

if (condition) {

// Code to be executed if the condition is true

}

**2. else statement:**

- The `else` statement runs if the condition in the `if` statement is `false`.

Syntax:

java

if (condition) {

// Code to be executed if the condition is true

} else {

// Code to be executed if the condition is false

}

**3. else if statement:**

- If there are multiple conditions, you can use `else if` to check them in sequence.

Syntax:

if (condition1) {

// Code to be executed if condition1 is true

} else if (condition2) {

// Code to be executed if condition 2 is true

} else {

// Code to be executed if both conditions are false

}

**4. switch statement:**

- The `switch` statement is used when you want to check a variable against multiple specific values. It’s an alternative to multiple `else if` statements.

Syntax:

switch (variable) {

case value1:

// Code to be executed if variable equals value1

break;

case value2:

// Code to be executed if variable equals value2

break;

default:

// Code to be executed if variable doesn't match any case

}

**Program:**

package practice;

public class ConditionalStatements {

public static void main(String[] args) {

if\_else();

looping();

jump\_statements();

exceptionHandling();

}

public static void if\_else() {

String weather = "Winter";

if (weather == "Winter") {

System.out.println("It's cold outside");

}

else if (weather == "Summer") {

System.out.println("Wear light clothes");

}

else if(weather == "Rainy"){

System.out.println("It's raining outside");

}

else{

System.out.println("Weather is not known");

}

}

public static void looping() {

System.out.println("For Loop");

for (int i = 1; i <= 5; i++) {

System.out.print(i+" ");

}

System.out.println();

System.out.println("While Loop");

int i = 6;

while (i <= 10) {

System.out.print(i+" ");

i++;

}

System.out.println();

System.out.println("Do While Loop");

int j = 11;

do {

System.out.print(j+" ");

j++;

} while (j <= 15);

System.out.println();

}

public static void jump\_statements() {

System.out.println("Jump Statements");

for (int i = 1; i <= 10; i++) {

if (i == 7) {

break;

}

if (i == 3) {

continue;

}

System.out.print(i+" ");

}

System.out.println();

}

public static void exceptionHandling() {

int a = 10;

int b = 0;

System.out.println("Exception Handling");

try {

int c = a/b;

System.out.println(c);

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero");

}

}

}

**Composition of Objects:**

Composition in Java is a design principle where one object is made up of other objects, creating a "has-a" relationship. It’s a way to build complex classes by combining simpler, smaller classes. Instead of inheritance, where one class is a specialised version of another (an "is-a" relationship), composition allows a class to contain objects from different classes, making it more flexible and modular.

Example of Composition:

Think of a car. A car has an engine, has wheels, and has seats. Each of these parts (engine, wheels, seats) is its own independent object, but together, they form a car.

How Composition Works:

1. Class Containing Objects: A class can have one or more fields (variables) that are objects of other classes.

2. "Has-a" Relationship: This indicates that the main object is composed of several smaller objects. For instance, a `Car` has an `Engine`, `Wheels`, and `Seats`, but they are not subclasses of the car. They are separate objects that the car depends on.

3. Reusability: The objects that make up a class (like engine, wheels, seats) can be reused in other contexts, such as in different types of vehicles.

**Program:**

package practice;

class Gmail{

String mail;

Gmail(String mail){

this.mail = mail;

}

void welcomeGmail() {

System.out.println("Welcome to GMail");

}

void mailDetails() {

System.out.println("This is your mail: "+mail);

}

}

class Gphotos{

int n;

Gphotos(int n){

this.n = n;

}

void welcomeGphotos() {

System.out.println("Welcome to Google Photos");

}

void photoDetails() {

System.out.println("The no.of photos in ur account are: "+n);

}

}

class Gpay{

double balance;

Gpay(double balance){

this.balance = balance;

}

void welcomeGpay() {

System.out.println("Welcome to GPay");

}

void gpayDetails() {

System.out.println("The balance in your account is: "+balance);

}

}

class Google{

Gmail email;

Gphotos photos;

Gpay pay;

Google(Gmail email,Gphotos photos,Gpay pay){

this.email = email;

this.photos = photos;

this.pay = pay;

}

void welcoming() {

System.out.println("Google welcomes you");

email.welcomeGmail();

photos.welcomeGphotos();

pay.welcomeGpay();

}

public Gmail getMail() {

return email;

}

public Gpay getGpay() {

return pay;

}

public Gphotos getPhotos() {

return photos;

}

}

public class CompositionofObjects{

public static void main(String[] args) {

Gmail obj1 = new Gmail("obj1@gmail.com");

Gphotos obj2 = new Gphotos(100);

Gpay obj3 = new Gpay(5000);

Google main = new Google(obj1,obj2,obj3);

main.welcoming();

System.out.println();

main.getMail().mailDetails();

System.out.println();

main.getGpay().gpayDetails();

System.out.println();

main.getPhotos().photoDetails();

}

}

**OOP’s Concepts:**

**1. Class and Objects**

- A class is a blueprint or template that defines the properties and behaviours (methods) that an object can have. Think of a class like a recipe—it tells you how to make something, but it’s not the actual thing itself.

- An object is an instance of a class. It’s the actual thing created using the class. If a class is the blueprint, an object is the actual house built using that blueprint.

**2. Interfaces**

- An interface defines methods that a class needs to implement, but it doesn’t provide any code for these methods.

- In Java, interfaces are used to ensure that different classes follow the same method structure, even if they implement the methods differently.

**3. Inheritance**

- Inheritance allows one class to inherit properties and methods from another class. This creates a parent-child relationship where the child class reuses the code of the parent class and can also add new features.

- Parent class: The class that passes on its features.

- Child class: The class that inherits and can extend or modify the features of the parent class.

**4. Abstraction**

- Abstraction means hiding the complex details and showing only the necessary parts of an object or concept. It focuses on what something does, rather than how it does it.

- In programming, abstraction hides complex code logic and only exposes essential functions to the user.

**5. Polymorphism**

- Polymorphism means "many forms." It allows one action to behave differently based on the object that performs it. There are two main types of polymorphism: compile-time (method overloading) and run-time (method overriding).

- Method Overloading: Multiple methods with the same name but different parameters, allowing them to perform similar tasks with variations.

- Method Overriding: A child class can modify the behaviour of a method inherited from the parent class.

**6. Encapsulation**

- Encapsulation is the concept of bundling variables and methods that operate on the data into a single unit, and restricting direct access to some of the object’s components. This ensures that an object controls how its data is used and modified.

-This keeps the internal data safe and prevents outside code from tampering with it directly.

**Program:**

package practice;

//--------------------------------Classes and Objects-------------------------------------------------

class Student{

String name;

int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

public void getStudentDetails(){

System.out.println("Student name: "+name+" age: "+age);

}

}

//--------------------------------Encapsulation-------------------------------------------------------

class Calculator{

private double a;

private double b;

Calculator(double a, double b) {

this.a = a;

this.b = b;

}

public void addition() {

System.out.println("Sum of the two numbers is: "+(a+b));

}

public void subtraction() {

System.out.println("Difference of the two numbers is: "+(a-b));

}

public void multiplication() {

System.out.println("Product of the two numbers is: "+(a\*b));

}

}

//------------------------------Inheritance------------------------------------------------------------

class Shape{

String name;

public Shape(String name) {

this.name = name;

}

public void getDetails() {

System.out.println(name+" has 4 sides");

}

}

class Square extends Shape {

Square(String name){

super(name);

}

void getangles() {

System.out.println("sum of angles in a square is 360");

}

}

//------------------------------------Polymorphism-----------------------------------------------------

class Circle extends Shape{

Circle(String name){

super(name);

}

@Override

public void getDetails() {

System.out.println(name+"has no sides");

}

void getangles() {

System.out.println("The complete angle of a circle is 360");

}

}

//------------------------------------Abstraction--------------------------------------------------------

abstract class Pet{

String name;

Pet(String name){

this.name = name;

}

abstract void details();

}

class Dog extends Pet{

Dog(String name){

super(name);

}

@Override

void details() {

System.out.println(name+"is a dog");

}

}

//----------------------------------Interfaces------------------------------------------------------------

interface Area{

void calculateArea();

}

class Rectangle implements Area{

int l,b;

Rectangle(int l, int b){

this.l = l;

this.b = b;

}

@Override

public void calculateArea() {

System.out.println("Area of the Rectangle is: "+(l\*b));

}

}

public class OOPsConcepts{

public static void main(String[] args) {

System.out.println("1. Classes and Objects:");

Student st = new Student("Damon",22);

st.getStudentDetails();

System.out.println();

System.out.println("2. Encapsulation:");

Calculator calc = new Calculator(2.67,3.85);

calc.addition();

calc.multiplication();

System.out.println();

System.out.println("3. Inheritance");

Square sq = new Square("square");

sq.getDetails();

sq.getangles();

System.out.println();

System.out.println("4. Polymorphism");

Circle crcl = new Circle("circle");

crcl.getDetails();

crcl.getangles();

System.out.println();

System.out.println("5. Abstraction");

Dog dog = new Dog("Snoopy");

dog.details();

System.out.println();

System.out.println("6. Interfaces");

Rectangle rect = new Rectangle(5,6);

rect.calculateArea();

}

}

**Types of Constructors**

In Java, a constructor is a special type of method used to initialise objects. When you create an object from a class, the constructor is automatically called to set up the initial state of the object. Constructors have the same name as the class and don’t have a return type.

There are two main types of constructors in Java:

**1. Default Constructor**

- A default constructor is a constructor that doesn’t take any arguments.

- If you don’t explicitly write a constructor in your class, Java provides a default constructor automatically. This constructor initialises the object with default values (e.g., integers as `0`, booleans as `false`, and objects as `null`).

- It’s useful when you want to create an object without needing to provide any initial values.

**2. Parameterized Constructor**

- A parameterized constructor allows you to pass parameters (arguments) to the constructor when creating an object. This way, you can give the object specific values right at the time of creation.

- This type of constructor is used when you want to initialise an object with specific data, like setting the name and age of a person, or the make and model of a car.

**Program:**

package practice;

class Employee{

String name;

int age;

double salary;

String dept;

Employee(){

System.out.println("This is default constructor with no parameters\n");

}

Employee(String name){

this.name = name;

System.out.println("For this object only name is given\n");

}

Employee(String name, int age){

this.name = name;

this.age = age;

System.out.println("Here only 2 parameters are given\n");

}

Employee(String name, int age, double salary){

this.name = name;

this.age = age;

this.salary = salary;

System.out.println("In this constructor 3 out of 4 parameters are given\n");

}

Employee(String name, int age, double salary, String dept){

this.name = name;

this.age = age;

this.salary = salary;

this.dept = dept;

System.out.println("In this constructor all 4 out of 4 parameters are given\n");

}

@Override

public String toString() {

return "Employee name: "+name+"\nEmployee age: "+age+"\nEmployee salary: "+salary+"\nEmployee department: "+dept;

}

}

public class TypesofConstructors{

public static void main(String[] args) {

Employee emp1 = new Employee("Gabriel",32,50000,"HR");

Employee emp2 = new Employee("Chris",33,40000);

Employee emp3 = new Employee("Henry",30);

Employee emp4 = new Employee("Alex");

Employee emp5 = new Employee();

System.out.println(emp1);

}

}

**Access Specifiers**

Access specifiers also called access modifiers, in Java control who can access a class, its variables, and its methods. They help manage the visibility of code components and ensure that certain parts of your code are protected or accessible depending on our needs.

There are four main types of access specifiers in Java:

**1. Public**

- Can be accessed by anyone, from anywhere in the program, across different classes, even outside the package.

- If you want a method, variable, or class to be available universally throughout your entire application, you mark it as “public”.

**2. Private**

- Can only be accessed within the same class

- If you want to restrict access to variables or methods so that they can only be used within the same class, you mark them as “private”. No other class, even within the same package, can access them.

**3. Protected**

- Can be accessed within the same package, and also by subclasses, even if they are in different packages.

- Usage: This is useful when you want to allow access to a field or method within the same package or to classes that inherit from your class.

**4. Default**

- If you don’t specify any access specifier, it’s considered default.

- This restricts access to within the same package. It can’t be accessed from outside the package, but all classes within the same package can use it.

**Program:**

package practice;

class superClass{

public int publicVar = 10;

private int privateVar = 20;

protected int protectedVar = 30;

int defaultVar = 40;

public void publicDetails() {

System.out.println("In public function of same class");

System.out.println("Public variable in same class: "+publicVar);

System.out.println("Private variable in same class: "+privateVar);

System.out.println("Protected variable in same class: "+protectedVar);

System.out.println("Default variable in same class: "+defaultVar);

}

private void privateDetails() {

System.out.println("In private function of same class");

System.out.println("Public variable in same class: "+publicVar);

System.out.println("Private variable in same class: "+privateVar);

System.out.println("Protected variable in same class: "+protectedVar);

System.out.println("Default variable in same class: "+defaultVar);

}

protected void protectedDetails() {

System.out.println("In protected function of same class");

System.out.println("Public variable in same class: "+publicVar);

System.out.println("Private variable in same class: "+privateVar);

System.out.println("Protected variable in same class: "+protectedVar);

System.out.println("Default variable in same class: "+defaultVar);

}

void defaultDetails() {

System.out.println("In default function of same class");

System.out.println("Public variable in same class: "+publicVar);

System.out.println("Private variable in same class: "+privateVar);

System.out.println("Protected variable in same class: "+protectedVar);

System.out.println("Default variable in same class: "+defaultVar);

}

}

class subClass extends superClass{

@Override

public void publicDetails() {

System.out.println("In public function of extended class");

System.out.println("Public variable in extended class: "+publicVar);

System.out.println("Private variable in extended class is not accessible");

System.out.println("Protected variable in extended class: "+protectedVar);

System.out.println("Default variable in same extended: "+defaultVar);

}

private void privateDetails() {

System.out.println("In private function of extended class");

System.out.println("Public variable in same class: "+publicVar);

System.out.println("Private variable in extended class is not accessible");

System.out.println("Protected variable in extended class: "+protectedVar);

System.out.println("Default variable in extended class: "+defaultVar);

}

@Override

protected void protectedDetails() {

System.out.println("In protected function of extended class");

System.out.println("Public variable in extended class: "+publicVar);

System.out.println("Private variable in extended class is not accessible");

System.out.println("Protected variable in same class: "+protectedVar);

System.out.println("Default variable in same class: "+defaultVar);

}

@Override

void defaultDetails() {

System.out.println("In default function of extended class");

System.out.println("Public variable in same class: "+publicVar);

System.out.println("Private variable in extended class is not accessible");

System.out.println("Protected variable in same class: "+protectedVar);

System.out.println("Default variable in same class: "+defaultVar);

}

}

class normalClass{

superClass superclass = new superClass();

public void publicDetails() {

System.out.println("In public function of same class");

System.out.println("Public variable in same class: "+superclass.publicVar);

System.out.println("Private variable in different class is not accessible");

System.out.println("Protected variable in same class: "+superclass.protectedVar);

System.out.println("Default variable in same class: "+superclass.defaultVar);

}

private void privateDetails() {

System.out.println("In private function of same class");

System.out.println("Public variable in same class: "+superclass.publicVar);

System.out.println("Private variable in different class is not accessible");

System.out.println("Protected variable in same class: "+superclass.protectedVar);

System.out.println("Default variable in same class: "+superclass.defaultVar);

}

protected void protectedDetails() {

System.out.println("In protected function of same class");

System.out.println("Public variable in same class: "+superclass.publicVar);

System.out.println("Private variable in different class is not accessible");

System.out.println("Protected variable in same class: "+superclass.protectedVar);

System.out.println("Default variable in same class: "+superclass.defaultVar);

}

void defaultDetails() {

System.out.println("In default function of same class");

System.out.println("Public variable in same class: "+superclass.publicVar);

System.out.println("Private variable in different class is not accessible");

System.out.println("Protected variable in same class: "+superclass.protectedVar);

System.out.println("Default variable in same class: "+superclass.defaultVar);

}

}

public class AccessSpecifier1{

public static void main(String[] args) {

superClass superCls = new superClass();

subClass subCls = new subClass();

normalClass nrmlcls = new normalClass();

superCls.publicDetails();

System.out.println();

// superCls.privateDetails();

System.out.println("Private class is not accessible in other class");

System.out.println();

superCls.protectedDetails();

System.out.println();

superCls.defaultDetails();

System.out.println();

subCls.publicDetails();

System.out.println();

// subCls.privateDetails();

System.out.println("Private class is not accessible in other class");

System.out.println();

subCls.protectedDetails();

System.out.println();

subCls.defaultDetails();

System.out.println();

nrmlcls.publicDetails();

System.out.println();

// nrmlcls.privateDetails();

System.out.println("Private class is not accessible in other class");

System.out.println();

nrmlcls.protectedDetails();

System.out.println();

nrmlcls.defaultDetails();

}

public void publicProgramForOtherPackage() {

System.out.println("This is public function in other package");

}

private void privateProgramForOtherPackage() {

System.out.println("This is private function");

}

protected void protectedProgramForOtherPackage() {

System.out.println("This is protected function");

}

void defaultProgramForOtherPackage() {

System.out.println("This is default function");

}

}

package practice2;

import practice.AccessSpecifier1;

//import practice.normalClass; => only public classes can be imported into other package

//import practice.subClass; => only public classes can be imported into other package

//import practice.superClass; => only public classes can be imported into other package

public class AccessSpecifier2{

public static void main(String[] args) {

AccessSpecifier1 obj = new AccessSpecifier1();

obj.publicProgramForOtherPackage();

System.out.println();

// obj.privateProgramForOtherPackage();

// obj.protectedProgramForOtherPackage();

// obj.defaultProgramForOtherPackage();

System.out.println("Only public access specifier will work in other package, no other access specifiers will work");

}

}

**Files I/O:**

1. FileReader and BufferedReader: This method reads the file line by line, offering efficient reading with the help of buffering.

2. FileInputStream: Reads the file byte by byte, displaying each character as it’s read.

3. Scanner: Reads the file line by line and is useful for simple file reading operations.

4. Files.readAllLines (Java NIO): Reads all lines from the file into a list, which is then printed.

5. Files.lines with Stream API: Uses Java 8’s Stream API to read and process the file line by line in a more modern, functional style.

**Program:**

package practice;

import java.io.\*;

import java.nio.\*;

import java.util.\*;

public class JavaFiles{

public static void main(String[] args) {

String filename = "file.txt";

String content = "This is the context in the file.";

writeFile(filename,content);

System.out.println("1. Using FileReader and BufferedReader: ");

readWithBufferedReader(filename);

System.out.println("\n2. Using FileInputStream:");

readWithFileInputStream(filename);

System.out.println("\n3. Using Scanner:");

readWithScanner(filename);

System.out.println("\n4. Using Files.readAllLines (Java NIO):");

readWithFilesReadAllLines(filename);

System.out.println("\n5. Using Files.lines (Java NIO) with Stream API:");

readWithFilesLines(filename);

fileOperations(filename);

}

private static void fileOperations(String filename){

File file = new File(filename);

System.out.println("\nFile Operations:");

System.out.println("Exists: " + file.exists());

System.out.println("Is File: " + file.isFile());

System.out.println("Is Directory: " + file.isDirectory());

System.out.println("Can Read: " + file.canRead());

System.out.println("Can Write: " + file.canWrite());

System.out.println("Absolute Path: " + file.getAbsolutePath());

// Rename the file

File newFile = new File("new\_" + filename);

if (file.renameTo(newFile)) {

System.out.println("File renamed successfully.");

} else {

System.out.println("Failed to rename the file.");

}

// Delete the file

if (newFile.delete()) {

System.out.println("File deleted successfully.");

} else {

System.out.println("Failed to delete the file.");

}

}

private static void readWithFilesLines(String filename){

try (Stream<String> stream = Files.lines(Paths.get(filename))) {

stream.forEach(System.out::println);

} catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage());

}

}

private static void readWithFilesReadAllLines(String filename){

try {

Files.readAllLines(Paths.get(filename)).forEach(System.out::println);

} catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage());

}

}

private static void readWithScanner(String filename){

try (Scanner scanner = new Scanner(new File(filename))) {

while (scanner.hasNextLine()) {

System.out.println(scanner.nextLine());

}

} catch (FileNotFoundException e) {

System.err.println("File not found: " + e.getMessage());

}

}

private static void readWithFileInputStream(String filename){

try (FileInputStream fis = new FileInputStream(filename)) {

int content;

while ((content = fis.read()) != -1) {

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

}

System.out.println();

} catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage());

}

}

private static void readWithBufferedReader(String filename){

try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {

String line;

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

System.out.println(line);

}

} catch (IOException e) {

System.err.println("Error reading file: " + e.getMessage());

}

}

private static void writeFile(String filename, String content) {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {

writer.write(content);

System.out.println("Content written to file successfully.");

} catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage());

}

}

}

**Exceptions:**

**Program:**

package practice;

import java.util.Scanner;

class InsufficientFundsException extends Exception {

private double amount;

public InsufficientFundsException(double amount) {

super("Insufficient funds. You need " + amount + " more to complete this transaction.");

this.amount = amount;

}

public double getAmount() {

return amount;

}

}

class InvalidAccountException extends RuntimeException {

private String accountId;

public InvalidAccountException(String accountId) {

super("Invalid account ID: " + accountId);

this.accountId = accountId;

}

public String getAccountId() {

return accountId;

}

}

public class BankAccountExceptionDemo {

public static void main(String[] args) {

BankAccount account = new BankAccount("12345", 1000.0);

Scanner scanner = new Scanner(System.in);

while (true) {

try {

System.out.print("Enter account ID: ");

String accountId = scanner.nextLine();

System.out.print("Enter amount to withdraw: ");

double amount = Double.parseDouble(scanner.nextLine());

account.withdraw(accountId, amount);

System.out.println("Withdrawal successful. New balance: " + account.getBalance());

break;

} catch (InsufficientFundsException e) {

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

System.out.println("You need $" + e.getAmount() + " more. Would you like to try again? (y/n)");

if (!scanner.nextLine().equalsIgnoreCase("y")) {

break;

}

} catch (InvalidAccountException e) {

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

System.out.println("Please try again with a valid account ID.");

} catch (NumberFormatException e) {

System.out.println("Error: Invalid amount entered. Please enter a valid number.");

} catch (Exception e) {

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

e.printStackTrace();

break;

} finally {

System.out.println("Transaction attempt completed.");

}

}

scanner.close();

}

}

// Bank account class with methods that may throw exceptions

class BankAccount {

private String accountId;

private double balance;

public BankAccount(String accountId, double initialBalance) {

this.accountId = accountId;

this.balance = initialBalance;

}

public double getBalance() {

return balance;

}

// This method demonstrates throwing both custom checked and unchecked exceptions

public void withdraw(String accountId, double amount) throws InsufficientFundsException {

// Validate account ID (may throw unchecked exception)

if (!this.accountId.equals(accountId)) {

throw new InvalidAccountException(accountId);

}

// Check for sufficient funds (may throw checked exception)

if (amount > balance) {

double shortfall = amount - balance;

throw new InsufficientFundsException(shortfall);

}

balance -= amount;

}

}