GEORGE KITUKU MUIA

SCT121-0421/2020

JAVA PROGRAMMING ASSIGNMENT

1. **What are the different logical operators available in Java? Provide examples of how they are used**

**&& (Logical AND):** Evaluates to true if both operands are true. If one or both operands are false, it returns false.

int a = 10;

int b = 20;

boolean result = (a > 5) && (b > 15); // true && true = true

System.out.println(result); // Output: true

**|| (Logical OR):** The || operator returns true if at least one of the operands is true. If both operands are false, it returns false.

int a = 10;

int b = 5;

boolean result = (a > 5) || (b > 15); // true || false = true

System.out.println(result); // Output: true

**! (Logical NOT):** The ! operator is a unary operator that inverts the value of a boolean expression. If the expression is true, ! will make it false, and vice versa.

boolean isTrue = true;

boolean result = !isTrue; // !true = false

System.out.println(result); // Output: false

**^ (Logical XOR):** The ^ operator, also known as the logical XOR (exclusive OR), returns true if one operand is true and the other is false. If both operands are the same, it returns false.

boolean a = true;

boolean b = false;

boolean result = a ^ b; // true ^ false = true

System.out.println(result); // Output: true

1. How does the && (logical AND) operator differ from the & (bitwise AND) operator in Java? Write a small program to demonstrate the difference.

**(Logical AND)**:

* Used in conditional statements.
* It evaluates the second operand only if the first operand is true.
* Short-circuits, meaning if the first operand is false, the second operand is not evaluated.

**(Bitwise AND)**:

* Used to perform bitwise AND operations on integer types (like int, long, etc.).
* It evaluates both operands, regardless of the first operand's value.
* Can also be used in conditional statements, but it does not short-circuit.

public class AndOperatorDemo {

public static void main(String[] args) {

// Example 1: Using && (Logical AND)

int a = 5;

int b = 10;

// Logical AND: The second condition (b > 15) is not evaluated because a < 10 is false

if (a < 10 && b > 15) {

System.out.println("Logical AND: Both conditions are true.");

} else {

System.out.println("Logical AND: One or both conditions are false.");

}

// Example 2: Using & (Bitwise AND)

int x = 6; // Binary: 110

int y = 3; // Binary: 011

// Bitwise AND: Compares each bit of x and y

int result = x & y; // Binary result: 010 (which is 2 in decimal)

System.out.println("Bitwise AND: The result of x & y is " + result);

// Example 3: Using & (Bitwise AND) in a conditional statement

int c = 5;

int d = 10;

// Bitwise AND in a conditional: Both conditions are evaluated

if (c < 10 & d > 15) {

System.out.println("Bitwise AND in conditional: Both conditions are true.");

} else {

System.out.println("Bitwise AND in conditional: One or both conditions are false.");

}

}

}

1. Explain the short-circuit behavior of the && and || operators in Java. How does it impact the performance of conditional statements? Provide a code example

**(Logical AND) Short-Circuit Behavior:**

* If the first operand is false, the overall expression cannot be true, so the second operand is not evaluated.
* **Impact on Performance:** Short-circuiting can improve performance by avoiding unnecessary evaluations, especially if the second operand is complex or involves a method call.

**(Logical OR) Short-Circuit Behavior:**

* If the first operand is true, the overall expression is true, so the second operand is not evaluated.
* **Impact on Performance:** Similar to &&, short-circuiting can optimize performance by skipping unnecessary evaluations.

public class ShortCircuitDemo {

public static void main(String[] args) {

int a = 5;

int b = 10;

// Logical AND (&&) Short-Circuit

if (a > 10 && checkCondition()) {

System.out.println("This won't be printed because a > 10 is false.");

} else {

System.out.println("Logical AND: Short-circuit occurred, second condition not evaluated.");

}

// Logical OR (||) Short-Circuit

if (a < 10 || checkCondition()) {

System.out.println("Logical OR: Short-circuit occurred, second condition not evaluated.");

}

}

public static boolean checkCondition() {

System.out.println("checkCondition() method is called.");

return true;

}

}

1. What is the difference between the equals() method and the == operator in Java? Write a program to compare two objects using both.

The == operator checks if two references point to the same memory location, while the equals() method checks if two objects are meaningfully equivalent (contents-wise).

class Person {

String name;

Person(String name) {

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Person person = (Person) obj;

return name.equals(person.name);

}

}

public class Main {

public static void main(String[] args) {

Person person1 = new Person("John");

Person person2 = new Person("John");

Person person3 = person1;

// Using == operator

System.out.println(person1 == person2); // false

System.out.println(person1 == person3); // true

// Using equals() method

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

System.out.println(person1.equals(person3)); // true

}

}

1. How should the equals() method be overridden in a custom class to ensure proper comparison of objects? Write a Java class that demonstrates this.

To properly compare objects of a custom class, override the equals() method to define what makes two instances equal.

class Employee {

int id;

String name;

Employee(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Employee employee = (Employee) obj;

return id == employee.id && name.equals(employee.name);

}

public static void main(String[] args) {

Employee emp1 = new Employee(1, "Alice");

Employee emp2 = new Employee(1, "Alice");

Employee emp3 = new Employee(2, "Bob");

System.out.println(emp1.equals(emp2)); // true

System.out.println(emp1.equals(emp3)); // false

}

}

1. Why is it important to override the hashCode() method when overriding the equals() method? Provide a practical example to illustrate the importance.

When equals() is overridden, hashCode() must also be overridden to maintain the contract between them: equal objects must have the same hash code.

import java.util.HashMap;

import java.util.Map;

class Product {

int id;

String name;

Product(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Product product = (Product) obj;

return id == product.id && name.equals(product.name);

}

@Override

public int hashCode() {

return 31 \* id + name.hashCode();

}

public static void main(String[] args) {

Product prod1 = new Product(1, "Laptop");

Product prod2 = new Product(1, "Laptop");

Map<Product, String> productMap = new HashMap<>();

productMap.put(prod1, "Electronics");

System.out.println(productMap.get(prod2)); // Electronics

}

}

1. What are some of the key features of Java that make it a widely-used programming language?

**Platform Independence**: Java programs can run on different platforms without modification.

**Object-Oriented**: Encourages modular and reusable code.

**Robust**: Strong memory management, exception handling, and type checking.

**Security**: Features like bytecode verification and sandboxing.

**Multithreading**: Supports concurrent execution of code.

**Rich API**: Extensive library support for various functions.

1. How does Java achieve platform independence? What role does the Java Virtual Machine (JVM) play in this? Write a short program and explain how it runs on different platforms.

Java achieves platform independence through the Java Virtual Machine (JVM). The JVM interprets compiled Java bytecode and executes it on the host machine, making the same Java program runnable on different platforms.

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

1. Explain the concept of garbage collection in Java. How does it help in memory management? Write a program that triggers garbage collection.

Garbage collection in Java automatically manages memory by reclaiming memory occupied by objects that are no longer in use.

public class GarbageCollectionExample {

public static void main(String[] args) {

GarbageCollectionExample obj = new GarbageCollectionExample();

obj = null;

System.gc(); // Requesting JVM to run Garbage Collector

System.out.println("Garbage collection triggered");

}

@Override

protected void finalize() throws Throwable {

System.out.println("Garbage collector called");

}

}

1. What is the difference between static and non-static methods in Java? Provide examples in a Java class.

Static methods belong to the class and can be called without creating an instance. Non-static methods belong to instances of the class.

public class MethodExample {

static void staticMethod() {

System.out.println("Static method called");

}

void nonStaticMethod() {

System.out.println("Non-static method called");

}

public static void main(String[] args) {

MethodExample.staticMethod(); // Calling static method

MethodExample obj = new MethodExample();

obj.nonStaticMethod(); // Calling non-static method

}

}

1. Can a static method access instance variables in Java? Why or why not? Write a program to demonstrate this.

Static methods cannot directly access instance variables since they belong to the class, not instances.

public class StaticDemo {

int instanceVariable = 10;

static void staticMethod() {

// System.out.println(instanceVariable); // Error: Cannot access instance variable

}

void instanceMethod() {

System.out.println(instanceVariable); // Can access instance variable

}

public static void main(String[] args) {

StaticDemo obj = new StaticDemo();

obj.instanceMethod();

}

}

1. Write the syntax for creating a static method and a non-static method in Java. Create a class with both types of methods and explain their differences.

public class MethodSyntax {

static void staticMethod() {

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

}

void nonStaticMethod() {

System.out.println("Non-static method");

}

public static void main(String[] args) {

MethodSyntax.staticMethod(); // Calling static method

MethodSyntax obj = new MethodSyntax();

obj.nonStaticMethod(); // Calling non-static method

}

}

1. How do instance variables differ from arrays in Java? Provide an example to illustrate the differences.

Instance variables store single values, while arrays store multiple values of the same type.

public class InstanceVsArray {

int instanceVar = 10;

int[] arrayVar = {1, 2, 3, 4, 5};

public static void main(String[] args) {

InstanceVsArray obj = new InstanceVsArray();

System.out.println("Instance Variable: " + obj.instanceVar);

System.out.println("Array Elements: ");

for (int i : obj.arrayVar) {

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

}

}

}

1. Can arrays in Java hold different data types? Explain with an example program.

Java arrays cannot hold different data types directly, but arrays of type Object can hold different object types.

public class MixedArray {

public static void main(String[] args) {

Object[] mixedArray = {1, "Two", 3.0, '4'};

for (Object obj : mixedArray) {

System.out.println(obj);

}

}

}

1. What are the advantages of using arrays over individual instance variables? Write a program that demonstrates these advantages.

Arrays allow storing multiple values in a single variable and provide indexed access to elements.

public class ArrayAdvantages {

public static void main(String[] args) {

int[] numbers = {1, 2, 3, 4, 5};

int sum = 0;

for (int number : numbers) {

sum += number;

}

System.out.println("Sum of array elements: " + sum);

}

}

1. What is a generic class in Java? Provide an example of how to define and use one.

A generic class allows you to define a class with type parameters, providing flexibility and type safety.

// Defining a generic class

class Box<T> {

private T item;

public void setItem(T item) {

this.item = item;

}

public T getItem() {

return item;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<>();

integerBox.setItem(123);

System.out.println("Integer Box contains: " + integerBox.getItem());

Box<String> stringBox = new Box<>();

stringBox.setItem("Hello Generics");

System.out.println("String Box contains: " + stringBox.getItem());

}

}

1. How do generic methods differ from generic classes in Java? Provide a use case and write a program to demonstrate it.

Generic methods allow you to create methods with type parameters, providing flexibility and reusability within the method.

public class GenericMethodExample {

// Defining a generic method

public static <T> void printArray(T[] array) {

for (T element : array) {

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

}

System.out.println();

}

public static void main(String[] args) {

Integer[] intArray = {1, 2, 3, 4, 5};

String[] stringArray = {"A", "B", "C", "D"};

// Using generic method

printArray(intArray);

printArray(stringArray);

}

}

1. Why is it beneficial to use generics in Java? Explain with an example program that shows the advantages of using generics.

Generics provide type safety, reduce code duplication, and improve code readability.

import java.util.ArrayList;

import java.util.List;

public class GenericsAdvantage {

public static void main(String[] args) {

List<String> stringList = new ArrayList<>();

stringList.add("One");

stringList.add("Two");

// Without generics, casting is needed

String item = stringList.get(0);

System.out.println("First item: " + item);

}

}

1. What is static binding in Java? When does it occur? Provide an example to illustrate static binding.

Static binding (or early binding) occurs at compile-time. It is used for static, private, and final methods.

class StaticBinding {

private void display() {

System.out.println("Static Binding Example");

}

public static void main(String[] args) {

StaticBinding obj = new StaticBinding();

obj.display(); // Static binding

}

}

1. Explain dynamic binding in Java with an example. How is it different from static binding? Write a program to demonstrate dynamic binding.

Dynamic binding (or late binding) occurs at runtime. It is used for overridden methods.

class Animal {

void sound() {

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

}

}

class Dog extends Animal {

@Override

void sound() {

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

}

}

public class DynamicBinding {

public static void main(String[] args) {

Animal obj = new Dog();

obj.sound(); // Dynamic binding

}

}

1. How does Java determine whether to use static or dynamic binding for a method call? Provide an example to clarify.

Java uses static binding for static, private, and final methods, and dynamic binding for overridden methods.

class Parent {

static void staticMethod() {

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

}

void instanceMethod() {

System.out.println("Instance method in Parent");

}

}

class Child extends Parent {

static void staticMethod() {

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

}

@Override

void instanceMethod() {

System.out.println("Instance method in Child");

}

}

public class BindingExample {

public static void main(String[] args) {

Parent obj = new Child();

obj.staticMethod(); // Static binding

obj.instanceMethod(); // Dynamic binding

}

}

1. How do you open a file for reading using the BufferedReader class in Java? Write a program to read a file line by line.

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class FileReading {

public static void main(String[] args) {

try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {

String line;

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

System.out.println(line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

1. Write a Java code snippet to read a file line by line using BufferedReader. Handle any possible exceptions that might occur.

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ReadFileWithExceptionHandling {

public static void main(String[] args) {

BufferedReader reader = null;

try {

reader = new BufferedReader(new FileReader("example.txt"));

String line;

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

System.out.println(line);

}

} catch (IOException e) {

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

} finally {

try {

if (reader != null) {

reader.close();

}

} catch (IOException e) {

System.err.println("Error closing the reader: " + e.getMessage());

}

}

}

}

1. Explain how to handle exceptions when working with file I/O in Java. Write a program that reads a file and handles exceptions appropriately.

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class ExceptionHandlingFileIO {

public static void main(String[] args) {

try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {

String line;

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

System.out.println(line);

}

} catch (IOException e) {

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

}

}

}

1. What is method overloading in Java? How does it differ from method overriding? Write a class that demonstrates both concepts.

**Method Overloading**: It occurs when multiple methods in the same class have the same name but different parameters (number, type, or both).

**Method Overriding**: It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

class MethodOverloadingAndOverriding {

// Method overloading

public void display(int a) {

System.out.println("Method with int argument: " + a);

}

public void display(String a) {

System.out.println("Method with String argument: " + a);

}

// Method overriding

static class SuperClass {

public void show() {

System.out.println("Show method in SuperClass");

}

}

static class SubClass extends SuperClass {

@Override

public void show() {

System.out.println("Show method in SubClass");

}

}

public static void main(String[] args) {

// Demonstrating method overloading

MethodOverloadingAndOverriding obj = new MethodOverloadingAndOverriding();

obj.display(10);

obj.display("Hello");

// Demonstrating method overriding

SuperClass superClass = new SuperClass();

SuperClass subClass = new SubClass();

superClass.show();

subClass.show();

}

}

1. Provide an example of method overloading in Java. Write a program with multiple overloaded methods and explain their use cases.

Method overloading allows different ways to call a method based on different parameters.

class Calculator {

// Overloaded methods for addition

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

public int add(int a, int b, int c) {

return a + b + c;

}

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println("Sum of two integers: " + calc.add(1, 2));

System.out.println("Sum of two doubles: " + calc.add(1.1, 2.2));

System.out.println("Sum of three integers: " + calc.add(1, 2, 3));

}

}

1. What are the rules for method overriding in Java? Provide an example by creating a superclass and a subclass that overrides a method.

 The method in the subclass must have the same name, return type, and parameters.

 The access level cannot be more restrictive than the overridden method.

 The method cannot throw a new or broader checked exception.

class ParentClass {

void display() {

System.out.println("Display method in ParentClass");

}

}

class ChildClass extends ParentClass {

@Override

void display() {

System.out.println("Display method in ChildClass");

}

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.display();

}

}

1. What is the difference between a process and a thread in Java? Provide an example program to create and start a thread.

 **Process**: An independent executing instance of a program with its own memory space.

 **Thread**: A smaller unit of a process that shares the process's memory and resources.

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running");

}

public static void main(String[] args) {

MyThread thread = new MyThread();

thread.start(); // Starts the thread

}

}

1. How do you create and start a thread in Java? Provide a code example that demonstrates thread creation and starting.

class RunnableExample implements Runnable {

@Override

public void run() {

System.out.println("Thread is running using Runnable");

}

public static void main(String[] args) {

Thread thread = new Thread(new RunnableExample());

thread.start(); // Starts the thread

}

}

1. Explain the concept of thread synchronization in Java. Why is it important? Write a program that uses synchronized methods to manage thread access to shared resources.

Synchronization is crucial to prevent thread interference and consistency problems when multiple threads access shared resources.

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

class SyncDemo extends Thread {

Counter counter;

SyncDemo(Counter counter) {

this.counter = counter;

}

public void run() {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

}

public static void main(String[] args) throws InterruptedException {

Counter counter = new Counter();

SyncDemo thread1 = new SyncDemo(counter);

SyncDemo thread2 = new SyncDemo(counter);

thread1.start();

thread2.start();

thread1.join();

thread2.join();

System.out.println("Final count: " + counter.getCount());

}

}

1. What is an ArrayList in Java? How does it differ from an array? Write a program that demonstrates the use of an ArrayList.

An ArrayList is a resizable array, which is part of the java.util package.

Differences from an array:

* Dynamic size
* Provides more methods for manipulation

import java.util.ArrayList;

public class ArrayListDemo {

public static void main(String[] args) {

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

list.add("Apple");

list.add("Banana");

list.add("Cherry");

System.out.println("ArrayList: " + list);

}

}

1. How do you create an ArrayList in Java? Provide an example program that initializes an ArrayList and adds elements to it.

import java.util.ArrayList;

public class InitializeArrayList {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<>();

numbers.add(10);

numbers.add(20);

numbers.add(30);

System.out.println("Numbers: " + numbers);

}

}

1. Write a Java code snippet to add items to an ArrayList. Demonstrate adding different types of elements to the list.

import java.util.ArrayList;

public class AddItemsToArrayList {

public static void main(String[] args) {

ArrayList<Object> list = new ArrayList<>();

list.add("Hello");

list.add(123);

list.add(45.67);

System.out.println("ArrayList: " + list);

}

}

1. What is the difference between an abstract class and an interface in Java? Provide examples of each and explain their use cases.

**Abstract Class**: Can have both abstract and concrete methods, constructors, and fields. **Interface**: Can only have abstract methods (Java 8 onwards allows default and static methods) and constants.

abstract class Animal {

abstract void makeSound();

void sleep() {

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

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Bark");

}

public static void main(String[] args) {

Dog dog = new Dog();

dog.makeSound();

dog.sleep();

}

}

Example of an interface:

interface Animal {

void makeSound();

}

class Cat implements Animal {

@Override

public void makeSound() {

System.out.println("Meow");

}

public static void main(String[] args) {

Cat cat = new Cat();

cat.makeSound();

}

}

1. Can an abstract class have a constructor? Explain why or why not, and provide a program to demonstrate your explanation.

abstract class Vehicle {

String type;

Vehicle(String type) {

this.type = type;

}

abstract void start();

}

class Car extends Vehicle {

Car(String type) {

super(type);

}

@Override

void start() {

System.out.println("Starting a " + type);

}

public static void main(String[] args) {

Car car = new Car("Sedan");

car.start();

}

}

1. How do you implement an interface in a Java class? Provide an example program that implements multiple interfaces.

interface Printable {

void print();

}

interface Showable {

void show();

}

class TestClass implements Printable, Showable {

@Override

public void print() {

System.out.println("Print method");

}

@Override

public void show() {

System.out.println("Show method");

}

public static void main(String[] args) {

TestClass obj = new TestClass();

obj.print();

obj.show();

}

}

1. Explain the concept of inheritance in object-oriented programming with an example in Java. Write a program that demonstrates class inheritance.

class Animal {

void eat() {

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

}

}

class Dog extends Animal {

void bark() {

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

}

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat();

dog.bark();

}

}

1. What is the "has-a" relationship in Java? How does it differ from the "is-a" relationship? Provide an example to illustrate both.

**"Has-a"**: Composition relationship where a class contains another class.

**"Is-a"**: Inheritance relationship where a class is a subtype of another class.

class Engine {

void start() {

System.out.println("Engine started");

}

}

class Car {

private Engine engine;

Car() {

engine = new Engine();

}

void startCar() {

engine.start();

}

public static void main(String[] args) {

Car car = new Car();

car.startCar();

}

}

Example of "is-a":

class Vehicle {

void run() {

System.out.println("Vehicle is running");

}

}

class Bike extends Vehicle {

@Override

void run() {

System.out.println("Bike is running");

}

public static void main(String[] args) {

Bike bike = new Bike();

bike.run();

}

}

1. How does polymorphism support object relationships in Java? Provide an example program that demonstrates polymorphism.

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

class Animal {

void makeSound() {

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

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Bark");

}

}

class Cat extends Animal {

@Override

void makeSound() {

System.out.println("Meow");

}

}

public class PolymorphismDemo {

public static void main(String[] args) {

Animal animal1 = new Dog();

Animal animal2 = new Cat();

animal1.makeSound();

animal2.makeSound();

}

}

1. What are checked and unchecked exceptions in Java? Provide examples of each and explain how they differ.

**Checked Exceptions**: These are exceptions that are checked at compile-time. They must be either caught or declared in the method signature using the throws keyword. Examples include IOException, SQLException, etc.

**Unchecked Exceptions**: These are exceptions that are checked at runtime. They are not required to be caught or declared in the method signature. Examples include NullPointerException, ArithmeticException, etc.

import java.io.FileReader;

import java.io.IOException;

public class CheckedExceptionExample {

public static void main(String[] args) {

try {

FileReader file = new FileReader("nonexistentfile.txt");

file.read();

file.close();

} catch (IOException e) {

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

}

}

}

Example of Unchecked Exception:

public class UncheckedExceptionExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // This will cause ArithmeticException

} catch (ArithmeticException e) {

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

}

}

}

1. How does Java enforce exception handling for checked exceptions? Write a program that demonstrates handling a checked exception.

Java enforces handling of checked exceptions by requiring them to be either caught in a try-catch block or declared in the method signature with the throws keyword

import java.io.FileReader;

import java.io.IOException;

public class CheckedExceptionEnforcement {

public static void readFile() throws IOException {

FileReader file = new FileReader("nonexistentfile.txt");

file.read();

file.close();

}

public static void main(String[] args) {

try {

readFile();

} catch (IOException e) {

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

}

}

}

1. Explain how to create a custom checked exception in Java. Write a program that defines and uses a custom exception.

To create a custom checked exception, extend the Exception class.

class CustomCheckedException extends Exception {

public CustomCheckedException(String message) {

super(message);

}

}

public class CustomCheckedExceptionDemo {

public static void validate(int age) throws CustomCheckedException {

if (age < 18) {

throw new CustomCheckedException("Age is less than 18");

}

}

public static void main(String[] args) {

try {

validate(15);

} catch (CustomCheckedException e) {

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

}

}

}

1. What is an inner class in Java? How does it differ from a regular class? Provide an example program that uses an inner class.

An inner class is a class defined within another class. It has access to the members of the outer class.

public class OuterClass {

private String message = "Hello from OuterClass";

class InnerClass {

void display() {

System.out.println(message); // Accessing private member of OuterClass

}

}

public static void main(String[] args) {

OuterClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.new InnerClass();

inner.display();

}

}

1. How do you define an interface in Java? Provide an example program that defines and implements an interface.

interface Drawable {

void draw();

}

class Circle implements Drawable {

@Override

public void draw() {

System.out.println("Drawing Circle");

}

}

public class InterfaceExample {

public static void main(String[] args) {

Drawable d = new Circle();

d.draw();

}

}

1. Can an interface have inner classes? Explain with an example program.

interface Drawable {

void draw();

class Color {

public void fillColor() {

System.out.println("Filling color");

}

}

}

class Circle implements Drawable {

@Override

public void draw() {

System.out.println("Drawing Circle");

}

}

public class InterfaceWithInnerClassExample {

public static void main(String[] args) {

Drawable d = new Circle();

d.draw();

Drawable.Color color = new Drawable.Color();

color.fillColor();

}

}

1. What is modularity in Java? How does it benefit large-scale applications? Write a simple Java module and explain its components.

Modularity allows a program to be divided into separate, interchangeable components, making large-scale applications more manageable.

Module Definition (module-info.java):

module com.example.module {

exports com.example.module;

}

Module Class:

package com.example.module;

public class HelloWorld {

public void sayHello() {

System.out.println("Hello from the module!");

}

}

Main Class in a Different Module:

import com.example.module.HelloWorld;

public class Main {

public static void main(String[] args) {

HelloWorld helloWorld = new HelloWorld();

helloWorld.sayHello();

}

}

1. How do you create a module in Java? Provide an example program that includes multiple modules and explains how they interact.

**Module 1 (module-info.java):**

module com.example.module1 {

exports com.example.module1;

}

Module 1 Class:

package com.example.module1;

public class Module1Class {

public void display() {

System.out.println("Hello from Module1");

}

}

Module 2 (module-info.java):

module com.example.module2 {

requires com.example.module1;

}

Module 2 Class:

package com.example.module2;

import com.example.module1.Module1Class;

public class Module2Class {

public static void main(String[] args) {

Module1Class module1Class = new Module1Class();

module1Class.display();

}

}

1. Explain the role of the module-info.java file in Java modularity. Write an example module with a module-info.java file and explain its content.

The module-info.java file defines the module and its dependencies, exports packages, and specifies services.

module com.example.myModule {

requires java.base;

exports com.example.myModule;

}

 module com.example.myModule: Declares a module named com.example.myModule.

 requires java.base: Specifies that this module depends on the java.base module.

 exports com.example.myModule: Makes the package com.example.myModule accessible to other modules.

1. How do you sort an array of objects in Java? Provide an example using the Comparable interface to sort objects

Implement the Comparable interface in the class to define the natural ordering.

import java.util.Arrays;

class Student implements Comparable<Student> {

String name;

int age;

Student(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int compareTo(Student other) {

return this.age - other.age; // Sorting by age

}

@Override

public String toString() {

return name + ": " + age;

}

public static void main(String[] args) {

Student[] students = {

new Student("Alice", 23),

new Student("Bob", 20),

new Student("Charlie", 22)

};

Arrays.sort(students);

for (Student student : students) {

System.out.println(student);

}

}

}

1. Write a Java code snippet to delete an item from an ArrayList. Demonstrate removing elements by index and by value.

To delete an item from an ArrayList in Java, you can remove elements by their index or by their value.

Removing by Index:

import java.util.ArrayList;

public class RemoveByIndex {

public static void main(String[] args) {

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

list.add("Apple");

list.add("Banana");

list.add("Cherry");

System.out.println("Original list: " + list);

list.remove(1); // Remove element at index 1 ("Banana")

System.out.println("After removing element at index 1: " + list);

}

}

Removing by Value:

import java.util.ArrayList;

public class RemoveByValue {

public static void main(String[] args) {

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

list.add("Apple");

list.add("Banana");

list.add("Cherry");

System.out.println("Original list: " + list);

list.remove("Banana"); // Remove element by value

System.out.println("After removing 'Banana': " + list);

}

}

1. How do you pass an object as an argument to a method in Java? Provide an example program that demonstrates passing and modifying an object in a method.

When you pass an object to a method in Java, you are passing the reference to the object. This means that any changes made to the object inside the method will affect the original object.

class Person {

String name;

Person(String name) {

this.name = name;

}

void setName(String name) {

this.name = name;

}

@Override

public String toString() {

return name;

}

}

public class PassObjectExample {

public static void modifyPerson(Person p) {

p.setName("John Doe");

}

public static void main(String[] args) {

Person person = new Person("Jane Smith");

System.out.println("Before modification: " + person);

modifyPerson(person);

System.out.println("After modification: " + person);

}

}