NAME: LUCY MWONGELI MUASA

REG NO: SCT121-0903/2022

JAVA PROGRAMMING ASSIGNMENT

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

1. Logical AND (&&):
    - The && operator returns true only if both conditions are true.
    - If either condition is false, the entire expression evaluates to false.
    - Example:

```
Java
int a = 10, b = 20, c = 20;
if (a < b && b == c) {
   // Both conditions are true
   int sum = a + b + c;
   System.out.println("The sum is: " + sum);
} else {
   System.out.println("False conditions");
}
```

2. Logical OR (||):
    - The || operator returns true if at least one of the conditions is true.
    - If both conditions are false, the entire expression evaluates to false.
    - Example:

```
Java
int a = 10, b = 20, c = 15;
System.out.println("Value of b: " + b);
if (a > c || ++b > c) {
   System.out.println("Inside if block");
}
System.out.println("Value of b: " + b);
```

3. Logical NOT (!):

- The ! operator negates the value of a condition.
- If the condition is true, ! makes it false, and vice versa.
- Example:

Java

```
int a = 10, b = 20;

boolean result = !(a < b);

System.out.println("Result: " + result); // Output: Result: false
```

2. 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 with boolean expressions.

Performs short-circuiting: If the first condition is false, the second condition is not evaluated because the result will always be false.

Bitwise AND (&):

Operates on both boolean and numeric (integral) types.

When used with booleans, it evaluates both operands regardless of the first operand's value (no short-circuiting).

When used with integers, it performs a bitwise AND operation, comparing each bit of the two operands.

```
public class AndOperatorExample {

  // Method to demonstrate the logical && and bitwise & with boolean
  public static boolean checkLogicalAnd(boolean a, boolean b) {
    System.out.println("Checking logical AND (&&)...");
    return a && b; // Short-circuits if 'a' is false
  }

  public static boolean checkBitwiseAnd(boolean a, boolean b) {
    System.out.println("Checking bitwise AND (&)...");
    return a & b; // No short-circuiting, evaluates both
  }

  // Method to demonstrate bitwise & with integers
```

```
    public static int bitwiseAndWithIntegers(int x, int y) {
        System.out.println("Performing bitwise AND with integers...");
        return x & y; // Bitwise AND on integers
    }

    public static void main(String[] args) {
        // Example with booleans
        boolean bool1 = true;
        boolean bool2 = false;

        // Logical AND (&&)
        System.out.println("Logical AND result: " + checkLogicalAnd(bool1, bool2));
        // Bitwise AND (&) with booleans
        System.out.println("Bitwise AND result: " + checkBitwiseAnd(bool1, bool2));

        // Example with integers
        int num1 = 6;  // Binary: 110
        int num2 = 3;  // Binary: 011

        // Bitwise AND (&) with integers
        System.out.println("Bitwise AND result with integers: " + bitwiseAndWithIntegers(num1, num2));
    }
}
```

3. 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):

If the first condition is false, the second condition is not evaluated because the entire expression will be false regardless of the second condition.

|| (Logical OR):

If the first condition is true, the second condition is not evaluated because the entire expression will be true regardless of the second condition.

```
public class ShortCircuitDemo {
    public static void main(String[] args) {
        int x = 5;
        int y = 0;

        // Example with Logical AND (&&) - short-circuiting
        if (x > 10 && divideByZero(y)) {
            System.out.println("This won't be printed because the first condition is false.");
        } else {
            System.out.println("Short-circuited: Second condition not evaluated.");
        }

        // Example with Logical OR (||) - short-circuiting
        if (x < 10 || divideByZero(y)) {
            System.out.println("Short-circuited: Second condition not evaluated, first condition is true.");
        }
    }

    // A method that will cause an exception if evaluated
    public static boolean divideByZero(int number) {
        System.out.println("Evaluating divideByZero...");
        return 10 / number > 1; // This will cause a divide by zero exception if evaluated
    }
}
```

4. What is the difference between the equals () method and the == operator in Java?

Write a program to compare two objects using both.

== Operator:

Used for reference comparison.

Compares whether two references point to the same object in memory.

For primitives (like int, char, double, etc.), == compares their values directly.

Equals () Method:

Used for value comparison.

Compares the content of two objects to determine if they are logically "equal."

The equals () method is defined in the Object class and can be overridden in custom classes to compare object values meaningfully.

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

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

    // Overriding equals() method to compare Person objects by name and age
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true; // Same reference
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false; // Null or different class
        }
        Person other = (Person) obj;
        return this.name.equals(other.name) && this.age == other.age; // Compare values
    }
}

public class EqualsVsDoubleEquals {
    public static void main(String[] args) {
        // Create two Person objects with the same values
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Alice", 30);

        // Compare using '==' operator (reference comparison)
        if (person1 == person2) {
            System.out.println("person1 and person2 are the same (== comparison).");
        } else {
            System.out.println("person1 and person2 are different (== comparison).");
        }

        // Compare using 'equals()' method (value comparison)
        if (person1.equals(person2)) {
            System.out.println("person1 and person2 are equal (equals() comparison).");
        } else {
            System.out.println("person1 and person2 are not equal (equals() comparison).");
        }
    }
}
```

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

Use the instanceof operator: Check if the given object is of the same type as the current object.

Self-check: Check if the current object and the passed object refer to the same instance using the == operator.

Check for null: Ensure the object being compared is not null.

Cast the object: After confirming the type, cast the object to the appropriate class.

Compare fields: Compare the relevant fields (usually using equals() for objects and == for primitives).

Ensure consistency with hashCode(): If you override equals(), you must also override hashCode() to ensure that equal objects have the same hash code.

```
import java.util.Objects;

class Employee {
```

```
private String name;
private int id;
private double salary;

// Constructor
public Employee(String name, int id, double salary) {
    this.name = name;
    this.id = id;
    this.salary = salary;
}

// Overriding equals() method
@Override
public boolean equals(Object obj) {
    // Self-check: if the two objects point to the same reference
    if (this == obj) {
        return true;
    }

    // Check if the obj is an instance of Employee and not null
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    // Type cast the object to Employee and compare relevant fields
    Employee other = (Employee) obj;
    return this.id == other.id &&
           this.salary == other.salary &&
           Objects.equals(this.name, other.name); // Using Objects.equals to handle nulls
}

// Overriding hashCode() method to ensure consistency with equals()
```

```
    @Override
    public int hashCode() {
        return Objects.hash(name, id, salary); // Combines fields into a hash code
    }

    // toString() method to display Employee details
    @Override
    public String toString() {
        return "Employee{name='" + name + "', id=" + id + ", salary=" + salary + "}";
    }
}

public class EqualsOverrideDemo {
    public static void main(String[] args) {
        Employee emp1 = new Employee("John Doe", 101, 50000.0);
        Employee emp2 = new Employee("John Doe", 101, 50000.0);
        Employee emp3 = new Employee("Jane Smith", 102, 60000.0);

        // Comparing emp1 and emp2 (same data)
        if (emp1.equals(emp2)) {
            System.out.println("emp1 and emp2 are equal.");
        } else {
            System.out.println("emp1 and emp2 are not equal.");
        }

        // Comparing emp1 and emp3 (different data)
        if (emp1.equals(emp3)) {
            System.out.println("emp1 and emp3 are equal.");
        } else {
            System.out.println("emp1 and emp3 are not equal.");
        }
        // Displaying hash codes for verification
        System.out.println("emp1 hashCode: " + emp1.hashCode());
        System.out.println("emp2 hashCode: " + emp2.hashCode());
        System.out.println("emp3 hashCode: " + emp3.hashCode());
    }
}
```

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

1. Contract Between equals() and hashCode():
   - If two objects are considered equal according to the equals() method, they must have the same hash code. This ensures that both objects are placed in the same bucket in hash-based collections.
   - If two objects are not equal, it's not required that they have different hash codes (although it's preferable to minimize collisions).
2. Hash-Based Collections:
   - Collections like HashMap and HashSet use the hashCode() method to determine the bucket where an object should be stored.
   - The equals() method is used to determine if two objects in the same bucket are equal.
   - If hashCode() is not overridden correctly, two objects that are considered equal by equals() might be placed in different hash buckets, making them impossible to retrieve or causing logical errors in sets and maps.

```
import java.util.HashSet;
import java.util.Objects;

class Employee {
    String name;
    int id;

    public Employee(String name, int id) {
        this.name = name;
        this.id = id;
    }

    // Overriding equals() method
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Employee other = (Employee) obj;
        return this.id == other.id && this.name.equals(other.name);
    }

    // Overriding hashCode() method to ensure consistency with equals()
    @Override
    public int hashCode() {
        return Objects.hash(name, id); // Combine fields to generate hashCode
    }
}

public class HashCodeWithOverride {
    public static void main(String[] args) {
        HashSet<Employee> employees = new HashSet<>();

        Employee emp1 = new Employee("John", 101);
        Employee emp2 = new Employee("John", 101);

        // Adding both employees to HashSet
        employees.add(emp1);
        employees.add(emp2);

        // Now the HashSet correctly identifies emp1 and emp2 as the same object
        System.out.println("Size of HashSet: " + employees.size());
    }
}
```

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

language?

Robustness and Reliability

Java is designed to minimize programming errors, which leads to the development of robust and reliable applications. Key features that contribute to this are:

Strong type checking at compile time.

Exception handling mechanisms to manage runtime errors gracefully.

Security

Java provides a high level of security through various mechanisms:

Bytecode verification ensures that code does not perform illegal operations.

Security manager restricts the access of untrusted code to system resources like the file system or network.

High Performance (Just-In-Time Compilation)

Although Java is not as fast as languages like C or C++, it is still relatively performant due to Just-In-Time (JIT) compilation. The JIT compiler converts bytecode into machine code at runtime, improving execution speed.

Scalability

Java is designed to scale easily, from small standalone applications to large enterprise systems. It supports distributed computing with technologies like Enterprise JavaBeans (EJB), Java RMI (Remote Method Invocation), and Spring frameworks.

Support for Mobile and Web Development

Java is the core language for developing Android applications, which are the majority of mobile apps worldwide.

Java is also used for web application development via technologies like Java Servlets, JSP (JavaServer Pages), and frameworks like Spring Boot.

8. 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.

a.) Java Bytecode:

When you write Java code, it’s transformed into special “bytecodes.”

These bytecodes are a non-executable form of the program, but they work everywhere, regardless of the computer or system12.

b.) The Java Virtual Machine (JVM):

The JVM reads and executes these bytecodes.

Acting as a universal translator, the JVM enables Java code to operate smoothly on any device equipped with a compatible JVM.

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

9. 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:

Garbage collection is an automatic process where the Java Virtual Machine (JVM) identifies and removes unused objects from memory.

Objective: It frees up memory by destroying unreachable objects, ensuring efficient memory utilization.

Automatic Process: The programmer doesn’t need to explicitly mark objects for deletion; the JVM handles it behind the scenes

How Garbage Collection Works:

The JVM periodically scans the heap memory to identify objects that are no longer referenced by any part of the program.

Unreferenced objects (those without any pointers) become eligible for garbage collection.

The memory occupied by these unused objects is reclaimed, making room for new allocations

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        // Create some objects
        String unused1 = new String("Unused Object 1");
        String unused2 = new String("Unused Object 2");

        // Nullify references to the objects
        unused1 = null;
        unused2 = null;
        // Explicitly trigger garbage collection
        System.gc();

        System.out.println("Garbage collection triggered!");
    }
}
```

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

- Static Methods:

A static method belongs to the class itself, not to any specific instance (object) of that class.

You can call a static method directly using the class name, without creating an object.

Static methods can only access other static members (variables and methods) within the same class.

Example of static Method

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

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.add(3, 6);
        System.out.println("Sum: " + result);
    }
}
```

- Non-Static Methods:

A non-static method is associated with instances of the class (objects).

You need to create an object to call a non-static method.

Non-static methods can access both static and non-static members within the same class.

```
class Calculator {
    public int multiply(int a, int b) {
        return a * b;
    }
}


public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.multiply(3, 6);
        System.out.println("Product: " + result);
    }
}
```

11. Can a static method access instance variables in Java? Why or why not? Write a

program to demonstrate this.

No, a static method in Java cannot access instance variables directly. This is because static methods belong to the class itself, rather than to any particular instance of the class. On the other hand, instance variables are tied to specific objects (instances) of the class.

```
public class StaticMethodDemo {


    // Instance variable
    private int instanceVariable = 42;


    // Static method
    public static void staticMethod() {
        // Try to access instance variable (This will cause a compilation error)
        // System.out.println("Instance Variable: " + instanceVariable);


        // Creating an instance of the class to access instance variables
        StaticMethodDemo obj = new StaticMethodDemo();
        System.out.println("Accessing instance variable via an object: " + obj.instanceVariable);
    }


    // Instance method
    public void instanceMethod() {
        // Instance methods can access both static and instance variables
        System.out.println("Instance Variable: " + instanceVariable);
    }

    public static void main(String[] args) {
        // Calling static method
        staticMethod();

        // Creating an object of the class and calling the instance method
        StaticMethodDemo obj = new StaticMethodDemo();
        obj.instanceMethod();
    }
}
```

12. 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.

Static Method:

- A static method is associated with the class itself and can be called without creating an instance of the class.

```
public static void methodName() {
    // Method body
}
```

Non-Static (Instance) Method:

- A non-static (instance) method is tied to an instance of the class and can access both static and instance members.

```
public void methodName() {
    // Method body
}
```

1. Differences:
    - Access: Static methods can be accessed directly using the class name, while non-static methods require an object reference.
    - Memory Allocation: Static methods have fixed memory allocation, whereas non-static methods depend on individual instances.

- Binding: Static methods use compile-time binding (early binding), while non-static methods use runtime binding (dynamic binding).
- Overriding: Static methods cannot be overridden, whereas non-static methods can be overridden in subclasses.

```
class MathOperations {
    // Static method: Adds two numbers
    public static int add(int a, int b) {
        return a + b;
    }

    // Non-static method: Multiplies two numbers
    public int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Using the static method
        int sum = MathOperations.add(3, 6);
        System.out.println("Sum: " + sum);

        // Creating an object to use the non-static method
        MathOperations mathObj = new MathOperations();
        int product = mathObj.multiply(3, 6);
        System.out.println("Product: " + product);
    }
}
```

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

Definition:

Instance Variables: These are variables that belong to a specific instance of a class. Each object of the class has its own copy of the instance variables. They can hold individual values of a given data type.

Arrays: Arrays are data structures that can hold multiple values of the same data type in a contiguous memory location. Unlike instance variables, arrays can store multiple elements, and they are indexed, which allows access to elements by position.

Size:

Instance Variables: Store a single value of a specific type (e.g., int, double, String).

Arrays: Can store multiple values of the same type. The size of the array is fixed once it's declared.

Access:

Instance Variables: Accessed directly by their name.

Arrays: Accessed using an index, where the first element starts at index 0.

Storage:

Instance Variables: Store a single value for each instance of a class.

Arrays: Can store multiple values, but the array itself is often considered a single variable (which contains references to multiple values).

```
public class Student {

    // Instance variables
    private String name;
    private int age;

    // Array to store marks of subjects
    private int[] marks;

    // Constructor to initialize instance variables and array
    public Student(String name, int age, int[] marks) {
        this.name = name;
        this.age = age;
        this.marks = marks; // Array assigned to the instance variable
    }
    // Method to display student details
    public void displayDetails() {
        // Access instance variables directly
        System.out.println("Student Name: " + name);
        System.out.println("Student Age: " + age);

        // Access array elements using indices
        System.out.println("Student Marks: ");
        for (int i = 0; i < marks.length; i++) {
            System.out.println("Subject " + (i + 1) + ": " + marks[i]);
        }
    }

    public static void main(String[] args) {
        // Create an array of marks
        int[] marksArray = {85, 90, 78, 88, 92};

        // Create a Student object, passing name, age, and the array of marks
        Student student = new Student("John Doe", 20, marksArray);

        // Call the method to display student details
        student.displayDetails();
    }
}
```

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

This means that all elements in a standard array must be of the same type, whether they are primitive types (like int, double, etc.) or reference types (like String, Object, etc.).

```
class Person {
    String name;
    int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an array of Person objects
        Person[] people = new Person[3];
        people[0] = new Person("John", 30);
        people[1] = new Person("Mary", 25);
        people[2] = new Person("David", 40);

        // Access and print the data
        for (Person person : people) {
            System.out.println("Name: " + person.name + ", Age: " + person.age);
        }
    }
}
```

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

Simplified Code:

Arrays allow you to manage multiple values of the same type more efficiently, avoiding the need to declare and initialize multiple individual variables.

Dynamic Data Handling:

Arrays provide a structured way to handle collections of data, making it easier to iterate over, sort, and manipulate the data.

Easier Data Access:

Arrays use indexing to access elements, which is more convenient than managing multiple separate variables.

Better Memory Management:

Arrays can be allocated dynamically, and their size can be adjusted if needed (using dynamic array structures like ArrayList).

```
public class ArrayAdvantagesDemo {
    public static void main(String[] args) {
        // Example: Storing student scores in an array
        int[] studentScores = { 85, 92, 78, 95, 88 };

        // Accessing and printing individual scores
        System.out.println("Third student's score: " + studentScores[2]);

        // Modifying a score
        studentScores[1] = 96;
        System.out.println("Updated second student's score: " + studentScores[1]);
    }
}
```

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

Type Parameter: Generic classes use type parameters (placeholders) that are replaced with actual types when the class is instantiated. This helps in creating reusable and type-safe code.

Type Safety: Generics ensure type safety by catching type errors at compile time rather than at runtime.

Code Reusability: They allow you to write more flexible and reusable code.

```
public class ClassName<T> {
    // Data members
    private T data;
    // Constructor
    public ClassName(T data) {
        this.data = data;
    }

    // Method to get data
    public T getData() {
        return data;
    }

    // Method to set data
    public void setData(T data) {
        this.data = data;
    }
}
```

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

Scope:

Generic Classes: Type parameters are declared at the class level. They apply to the entire class and its methods.

Generic Methods: Type parameters are declared at the method level. They apply only to the method where they are defined.

Use Case:

Generic Classes: Used when a class needs to operate on different types and the type is needed across multiple methods or instance variables.

Generic Methods: Used when a specific method needs to operate on different types, but the class itself does not need to be generic.

Declaration:

Generic Classes: Type parameters are specified in angle brackets (<T>) immediately after the class name.

Generic Methods: Type parameters are specified in angle brackets (<T>) before the return type of the method.

```
public class GenericMethodDemo {
```

```
// Generic method to swap elements in an array
public static <T> void swap(T[] array, int index1, int index2) {
    // Check if indices are within bounds
    if (index1 < 0 || index1 >= array.length || index2 < 0 || index2 >= array.length) {
        throw new IllegalArgumentException("Index out of bounds");
    }

    // Swap the elements
    T temp = array[index1];
    array[index1] = array[index2];
    array[index2] = temp;
}

public static void main(String[] args) {
    // Create an array of Integer
    Integer[] intArray = {1, 2, 3, 4, 5};
    System.out.println("Before swap: ");
    for (int i : intArray) {
        System.out.print(i + " ");
    }
    System.out.println();

    // Call the generic swap method
    swap(intArray, 1, 3);

    System.out.println("After swap: ");
    for (int i : intArray) {
        System.out.print(i + " ");
    }
    System.out.println();
```

```
        // Create an array of String
        String[] strArray = {"apple", "banana", "cherry"};
        System.out.println("Before swap: ");
        for (String s : strArray) {
            System.out.print(s + " ");
        }
        System.out.println();

        // Call the generic swap method
        swap(strArray, 0, 2);

        System.out.println("After swap: ");
        for (String s : strArray) {
            System.out.print(s + " ");
        }
        System.out.println();
    }
}
```

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

1. Type Safety:
   - Generics provide compile-time type checking, which helps catch errors related to type mismatches early in the development process. This reduces runtime errors and enhances code reliability.
2. Code Reusability:
   - Generics enable you to write classes, interfaces, and methods that can work with different data types while reusing the same code. This avoids duplication and simplifies maintenance.
3. Elimination of Type Casting:
   - With generics, you don't need to perform explicit type casting. This makes the code cleaner and easier to read.
4. Improved Code Readability:
   - Generics make the code more expressive and self-documenting by specifying the type of data a class or method operates on, making it clear what type of data is expected.
5. Enhanced Maintainability:

- By using generics, you can create flexible and reusable components that are easier to maintain and update.

```
// Define a generic class Container

public class Container<T> {

    private T item; // The container can hold an item of any type


    // Constructor to initialize the container with an item

    public Container(T item) {

        this.item = item;

    }


    // Method to get the item

    public T getItem() {

        return item;

    }


    // Method to set a new item

    public void setItem(T item) {

        this.item = item;

    }

}


// Main class to demonstrate the use of the generic Container class

public class GenericsDemo {


    public static void main(String[] args) {

        // Create a Container for Integer type

        Container<Integer> intContainer = new Container<>(123);

        System.out.println("Integer value: " + intContainer.getItem());


        // Create a Container for String type

        Container<String> stringContainer = new Container<>("Hello Generics");
        System.out.println("String value: " + stringContainer.getItem());

        // Create a Container for Double type
        Container<Double> doubleContainer = new Container<>(3.14);
        System.out.println("Double value: " + doubleContainer.getItem());

        // Set a new value to the intContainer
        intContainer.setItem(456);
        System.out.println("Updated Integer value: " + intContainer.getItem());
    }
}
```

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

static binding.

Static binding (also known as early binding) in Java occurs when the method to be invoked is determined at compile time. This typically happens with methods that are static, final, or private because their binding is resolved during compilation, not at runtime.

Static Methods: These are methods defined with the static keyword. Static methods are bound at compile time because they are associated with the class itself, not instances of the class.

Final Methods: Methods marked with the final keyword cannot be overridden in subclasses. Their binding is resolved at compile time.

Private Methods: Private methods are not visible to subclasses and thus their binding is resolved at compile time.

```
class Vehicle {
    static void start() {
        System.out.println("Vehicle started");
    }
}

class Car extends Vehicle {
    static void start() {
        System.out.println("Car started");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Car(); // Reference type: Vehicle, actual object: Car
        vehicle.start(); // Calls Vehicle's static method (static binding)
    }
}
```

20. Explain dynamic binding in Java with an example. How is it different from static

binding? Write a program to demonstrate dynamic binding.

Dynamic binding (also known as late binding) in Java refers to the process where the method to be invoked is determined at runtime based on the actual object type rather than the reference type. This is in contrast to static binding (early binding), where method resolution happens at compile time.

Static Binding:

Occurs at compile time.

Applies to static, final, and private methods.

Method resolution is based on the reference type.

Dynamic Binding:

Occurs at runtime.

Applies to instance methods that are overridden in subclasses.

Method resolution is based on the actual object type.

```
class Animal {
    void eat() {
        System.out.println("Animal is eating...");
    }
}

class Dog extends Animal {
    @Override
    void eat() {
        System.out.println("Dog is eating...");
    }
}

public class DynamicBindingDemo {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Reference type: Animal, actual object: Dog
        animal.eat(); // Calls Dog's overridden method (dynamic binding)
    }
}
```

21. How does Java determine whether to use static or dynamic binding for a method call?

Provide an example to clarify.

Static Binding

Static binding occurs when:

- Method is Static: Static methods are bound at compile time. The method call is resolved based on the reference type, not the actual object type.
- Method is Final: Final methods cannot be overridden, so their calls are resolved at compile time.
- Method is Private: Private methods are not accessible from subclasses and thus their calls are resolved at compile time based on the reference type.

Dynamic Binding

Dynamic binding occurs when:

- Method is Instance Method: Instance methods that are overridden in subclasses use dynamic binding. The actual method to be invoked is determined at runtime based on the actual object type.
- Polymorphism: When a method is overridden in a subclass, Java uses dynamic binding to determine the appropriate method to execute based on the actual object type.

```
class Base {
    // Static method
    public static void staticMethod() {
        System.out.println("Static method in Base");
    }

    // Instance method
    public void instanceMethod() {
        System.out.println("Instance method in Base");
    }
}

class Derived extends Base {
    // Static method with same name but different implementation
    public static void staticMethod() {
        System.out.println("Static method in Derived");
    }

    // Overriding instance method
    @Override
    public void instanceMethod() {
        System.out.println("Instance method in Derived");
    }
}

public class BindingDemo {
    public static void main(String[] args) {
        // Static binding
        Base baseRef = new Base();
        Base derivedRef = new Derived();
        Derived derived = new Derived();

        // Static methods are bound at compile time
        baseRef.staticMethod();  // Outputs: Static method in Base
        derivedRef.staticMethod(); // Outputs: Static method in Base (Base's static method is called)
        derived.staticMethod();  // Outputs: Static method in Derived

        // Dynamic binding for instance methods
        baseRef.instanceMethod(); // Outputs: Instance method in Base
        derivedRef.instanceMethod(); // Outputs: Instance method in Derived (Derived's instance method is called)
        derived.instanceMethod(); // Outputs: Instance method in Derived
    }
}
```

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

Create a FileReader object: This is used to open the file for reading.

Create a BufferedReader object: Wrap the FileReader with BufferedReader to read text efficiently.

Read lines from the file: Use the readLine() method of BufferedReader to read the file line by line.

Close the BufferedReader: After reading is complete, close the BufferedReader to release system resources.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        // Specify the path of the file to be read
        String filePath = "example.txt";

        // Initialize BufferedReader
        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;

            // Read the file line by line
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            // Handle exceptions
            System.err.println("An error occurred while reading the file: " + e.getMessage());
        }
    }
}
```

23. 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 ReadFileLineByLine {
    public static void main(String[] args) {
        String filename = "sample.txt"; // Replace with the actual file path

        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 the file: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

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

1. Handling Exceptions in File I/O:

- When reading or writing files, various exceptions can occur, such as FileNotFoundException, IOException, or SecurityException.
- To handle these exceptions:
    - Use try-catch blocks to catch specific exceptions.
    - Close resources (e.g., file readers or writers) in a finally block to release system resources.

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;


public class ReadFileLineByLine {

    public static void main(String[] args) {

        String filename = "sample.txt"; // Replace with the actual file path


        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 the file: " + e.getMessage());

            e.printStackTrace();

        }

    }

}
```

25. What is method overloading in Java? How does it differ from method overriding?

Write a class that demonstrates both concepts.

Method overloading deals with multiple methods of the same name but different parameters.

Method overriding involves providing a specific implementation for an inherited method.

```
// Parent class to demonstrate method overriding

class Animal {
    // Method to be overridden
    public void sound() {
        System.out.println("Animal makes a sound");
    }

    // Overloaded method (same method name but different parameters)
    public void eat(String food) {
        System.out.println("Animal eats " + food);
    }

    public void eat(String food, int quantity) {
        System.out.println("Animal eats " + quantity + " servings of " + food);
    }
}

// Subclass that demonstrates method overriding
class Dog extends Animal {
    // Overriding the sound() method in the parent class
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    // Overloading eat method with a different parameter list
    public void eat(int quantity) {
        System.out.println("Dog eats " + quantity + " servings of dog food");
    }
}

public class MethodDemo {
    public static void main(String[] args) {
        // Demonstrating method overloading
        Animal animal = new Animal();
        animal.eat("grass"); // Calls eat(String food)
        animal.eat("meat", 3); // Calls eat(String food, int quantity)

        // Demonstrating method overriding
        Dog dog = new Dog();
        dog.sound(); // Calls the overridden sound() method in Dog

        // Using method overloading in Dog class
        dog.eat(2); // Calls the overloaded eat(int quantity) method in Dog
    }
}
```

26. Provide an example of method overloading in Java. Write a program with multiple

overloaded methods and explain their use cases.

Handling Different Data Types: Overload a method to perform similar operations for different types of input (e.g., int, double).

Handling Different Number of Parameters: Overload a method to handle variable numbers of arguments for more flexible operations.

Optional Parameters: Instead of passing all parameters explicitly, provide overloaded methods with default values for certain parameters.

```
public 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;
    }

    // Overloaded method for concatenating strings
    public String concatenate(String str1, String str2) {
        return str1 + str2;
    }

    // Overloaded method for finding the maximum of three integers
    public int findMax(int a, int b, int c) {
        return Math.max(Math.max(a, b), c);
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Use case 1: Adding integers and doubles
        System.out.println("Sum (int): " + calc.add(3, 5));
        System.out.println("Sum (double): " + calc.add(3.5, 2.1));

        // Use case 2: Concatenating strings
        System.out.println("Concatenated string: " + calc.concatenate("Hello, ", "Java!"));

        // Use case 3: Finding the maximum of three integers
        System.out.println("Max value: " + calc.findMax(10, 25, 15));
    }
}
```

27. 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 must have the same name as in the parent class.
- The method must have the same parameter list (number and types of parameters) as in the parent class.
- There must be an IS-A relationship (inheritance) between the classes

```
class Animal {
    void makeSound() {
        System.out.println("The animal makes a sound.");
    }
}
```

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

public class OverridingDemo {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Reference type: Animal, actual object: Dog
        animal.makeSound(); // Calls Dog's overridden method
    }
}
```

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

1. Processes vs. Threads:
   - Process:
     - A process is an independent program that runs in its memory area.
     - Each process has its own memory space, executable code, and a unique process identifier (PID).
     - Processes do not share memory with other processes.
     - Inter-process communication (IPC) is essential for process communication.
     - Creating a new process is resource-intensive.
     - Processes can execute on different CPUs, allowing for real parallelism.
     - When one process fails, it does not affect other processes.
   - Thread:
     - A thread is a lightweight execution unit within a process.
     - Threads share the same memory area as the parent process.
     - Inter-thread communication (ITC) can be done easily using shared memory.
     - Creating a new thread is relatively inexpensive.
     - Threads can only execute on one CPU at a time (pseudo-parallelism).
     - If one thread fails, the entire process may suffer.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running!");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start(); // Start the thread
    }
}
```

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

By extending the Thread class.

By implementing the Runnable interface.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running!");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start(); // Start the thread
    }
}
```

30. 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.

Thread synchronization in Java is essential for coordinating the execution of multiple threads to ensure data consistency and prevent race conditions when they access shared resources concurrently.

1. Why Use Thread Synchronization?
    - Data Consistency: Synchronization ensures that shared data remains consistent during concurrent access.
    - Preventing Race Conditions: Without synchronization, race conditions can occur, leading to unpredictable behavior.
    - Avoiding Deadlocks: Proper synchronization prevents situations where threads wait indefinitely for each other.
2. Types of Synchronization in Java:
    - Process Synchronization: Coordinates execution among multiple processes (e.g., using semaphores or monitors).
    - Thread Synchronization: Coordinates and orders the execution of threads within a multi-threaded program.
3. Mutual Exclusion (Cooperation):
    - Ensures that threads do not interfere with each other while sharing data.

```
class SharedResource {
    synchronized void printMessage(String message) {
        for (int i = 0; i < 5; i++) {
            System.out.println(message);
        }
    }
}

class MyThread extends Thread {
    SharedResource resource;

    MyThread(SharedResource resource) {
        this.resource = resource;
    }

    public void run() {
        resource.printMessage("Hello from thread " + Thread.currentThread().getId());
    }
}

public class SynchronizationDemo {
    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();
        MyThread thread1 = new MyThread(sharedResource);
        MyThread thread2 = new MyThread(sharedResource);

        thread1.start();
        thread2.start();
    }
}
```

31. 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 in Java is a resizable array, part of the Java Collections Framework

Differences Between Array and ArrayList:

Array:

Fixed size (cannot be modified after creation).

Can store both primitives and objects.

Accessed using square brackets ([]).

No built-in methods for adding or removing elements.

ArrayList:

Dynamic size (can grow or shrink).

Can only store objects (not primitives).

Accessed using methods (e.g., add(), get(), remove()).

Provides flexibility for managing elements.

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Create an ArrayList of integers
        ArrayList<Integer> numbers = new ArrayList<>();
        // Add elements to the ArrayList
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Access and print elements
        System.out.println("First element: " + numbers.get(0));
        System.out.println("Size of ArrayList: " + numbers.size());

        // Remove an element
        numbers.remove(1);
        System.out.println("Updated size: " + numbers.size());

        // Loop through the ArrayList
        for (Integer num : numbers) {
            System.out.println("Element: " + num);
        }
    }
}
```

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

Syntax for Creating an ArrayList:

```
ArrayList<Type> arrayListName = new ArrayList<>();
```

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        // Create an ArrayList of String type
        ArrayList<String> languages = new ArrayList<>();

        // Add elements to the ArrayList
        languages.add("Java");
        languages.add("Python");
        languages.add("Swift");

        // Print the ArrayList
        System.out.println("ArrayList: " + languages);
    }
}
```

33. 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 ArrayListDemo {
    public static void main(String[] args) {
        // 1. Create an ArrayList of Strings and add elements
        ArrayList<String> stringList = new ArrayList<>();
        stringList.add("Apple");
        stringList.add("Banana");
        stringList.add("Cherry");

        // 2. Create an ArrayList of Integers and add elements
        ArrayList<Integer> integerList = new ArrayList<>();
        integerList.add(10);
        integerList.add(20);
        integerList.add(30);

        // 3. Create an ArrayList of Objects to store different types of elements
        ArrayList<Object> mixedList = new ArrayList<>();
        mixedList.add("Hello");    // Adding a String
        mixedList.add(100);        // Adding an Integer (Autoboxed)
        mixedList.add(45.67);      // Adding a Double (Autoboxed)
        mixedList.add(true);       // Adding a Boolean

        // Print the String ArrayList
        System.out.println("String ArrayList:");
        for (String item : stringList) {
            System.out.println(item);
        }

        // Print the Integer ArrayList
        System.out.println("\nInteger ArrayList:");
        for (Integer number : integerList) {
            System.out.println(number);
        }

        // Print the mixed ArrayList
        System.out.println("\nMixed ArrayList (Object types):");
        for (Object obj : mixedList) {
            System.out.println(obj);
        }
    }
}
```

34. What is the difference between an abstract class and an interface in Java? Provide

examples of each and explain their use cases.

Key Differences Between Abstract Classes and Interfaces:

Feature	Abstract Class	Interface
Definition	A class that can have abstract methods (without body) and non-abstract methods (with body).	A completely abstract type that can have only abstract methods (prior to Java 8), but can include default and static methods from Java 8 onwards.
Implementation	Can provide some method implementations (non-abstract methods).	Cannot provide any method implementations (until Java 8's default and static methods).
Multiple Inheritance	A class can extend only one abstract class (single inheritance).	A class can implement multiple interfaces (multiple inheritance of behavior).

Feature	Abstract Class	Interface
Constructors	Can have constructors.	Cannot have constructors.
Fields (Variables)	Can have both instance variables and constants.	Can only have static and final (constant) variables.

```
interface Vehicle {
    // Abstract method (must be implemented by classes that implement this interface)
    void start();


    void stop();
}


class Car implements Vehicle {
    public void start() {
        System.out.println("The car starts with a key.");
    }


    public void stop() {
        System.out.println("The car stops by pressing the brake.");
    }
}


class Bicycle implements Vehicle {
    public void start() {
        System.out.println("The bicycle starts by pedaling.");
    }


    public void stop() {
        System.out.println("The bicycle stops by applying the brake.");
    }
}


public class InterfaceDemo {
    public static void main(String[] args) {
        Vehicle car = new Car();
        Vehicle bicycle = new Bicycle();

        car.start(); // Output: The car starts with a key.
        car.stop();  // Output: The car stops by pressing the brake.

        bicycle.start(); // Output: The bicycle starts by pedaling.
        bicycle.stop();  // Output: The bicycle stops by applying the brake.
    }
}
```

35. Can an abstract class have a constructor? Explain why or why not, and provide a

program to demonstrate your explanation.

1. Abstract Class Constructors:
   - Yes, an abstract class can have a constructor. In fact, it’s common to define constructors in abstract classes.
   - When a subclass is instantiated, its constructor implicitly calls the constructor of the superclass (including abstract classes).
   - Abstract class constructors are used for initializing common fields or performing setup tasks that apply to all subclasses.
2. Why Abstract Classes Have Constructors:
   - Abstract classes often define shared behavior or state that subclasses inherit.
   - Constructors allow you to initialize fields specific to the abstract class.

```
abstract class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public abstract void makeSound(); // Abstract method

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

class Dog extends Animal {
    public Dog(String name) {
        super(name); // Call superclass constructor
    }

    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog("Buddy");
        myDog.printName();
        myDog.makeSound();
    }
}
```

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

Define the interface(s) with abstract methods.

Use the implements keyword in the class definition to indicate that the class implements the interface.

Provide concrete implementations of all abstract methods in the interface(s) within the class.

```
// Define the first interface
interface Flyable {
    // Abstract method
    void fly();
}
```

```
// Define the second interface
interface Swimmable {
    // Abstract method
    void swim();
}

// Class implementing both interfaces
class Duck implements Flyable, Swimmable {
    // Provide implementation for the fly() method from Flyable interface
    public void fly() {
        System.out.println("The duck is flying.");
    }

    // Provide implementation for the swim() method from Swimmable interface
    public void swim() {
        System.out.println("The duck is swimming.");
    }
}

// Main class to demonstrate the implementation
public class MultipleInterfacesDemo {
    public static void main(String[] args) {
        // Create an instance of Duck
        Duck duck = new Duck();

        // Call methods from both interfaces
        duck.fly();   // Output: The duck is flying.
        duck.swim();  // Output: The duck is swimming.
    }
}
```

37. Explain the concept of inheritance in object-oriented programming with an example

in Java. Write a program that demonstrates class inheritance.

Inheritance allows a subclass (also known as a child or derived class) to inherit properties (fields) and behaviors (methods) from a superclass (also known as a parent or base class).

The subclass can extend the functionality of the superclass by adding new methods or overriding existing ones.

```
class Animal {
    String name;

    public void eat() {
        System.out.println("I can eat");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("I can bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog labrador = new Dog();
        labrador.name = "Rohu"; // Accessing superclass field
        labrador.eat(); // Calling superclass method
        labrador.bark(); // Calling subclass method
    }
}
```

38. What is the "has-a" relationship in Java? How does it differ from the "is-a"

relationship? Provide an example to illustrate both.

Is-a Relationship (Inheritance)

The "is-a" relationship is established through inheritance (i.e., using the extends keyword in Java).

It indicates that a subclass is a type of its superclass. For example, a Dog is a(n) Animal.

This relationship is transitive: if class A extends class B, then A is a B.

2. Has-a Relationship (Composition/Aggregation)

The "has-a" relationship is established through composition or aggregation (i.e., using instance variables that reference other objects).

It indicates that a class contains another class as part of its state or behavior. For example, a Car has a Engine.

In this case, the contained object is typically a member of the class, and the class doesn’t inherit from it but holds a reference to it.

```
// Superclass representing an "is-a" relationship
class Animal {
    String name;

    // Constructor of Animal
    public Animal(String name) {
        this.name = name;
    }

    // Method of the Animal class
    public void makeSound() {
        System.out.println(name + " is making a sound.");
    }
}

// Subclass representing the "is-a" relationship
class Dog extends Animal {
    // Constructor of Dog
    public Dog(String name) {
        super(name); // Call the superclass constructor
    }

    // Overriding makeSound method
    @Override
    public void makeSound() {
        System.out.println(name + " says: Woof Woof!");
    }
}

// Class representing the "has-a" relationship
class Car {
    // Has-a relationship: A car has an engine
    private Engine engine;

    // Constructor to initialize Car with an Engine
    public Car() {
        engine = new Engine(); // Composition: Car owns an Engine
    }

    // Method to start the car (which starts the engine)
    public void start() {
        engine.start(); // Delegating the start task to the engine
    }
}

// Separate class for Engine
class Engine {
    // Engine class with a start method
    public void start() {
        System.out.println("Engine is starting...");
    }
}

// Main class to demonstrate both "is-a" and "has-a" relationships
public class HasAIsADemo {
    public static void main(String[] args) {
        // Demonstrating the "is-a" relationship
        Dog dog = new Dog("Buddy");
        dog.makeSound();  // Output: Buddy says: Woof Woof!


        // Demonstrating the "has-a" relationship
        Car car = new Car();
        car.start();  // Output: Engine is starting...
    }
}
```

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

- Polymorphism allows us to perform a single action (e.g., calling a method) in multiple ways.
- It ensures that the same method name can behave differently based on the actual object type.
- Polymorphism is achieved through method overriding and method overloading.

```
class Language {
    public void displayInfo() {
        System.out.println("Common English Language");
    }
}

class Java extends Language {
    @Override
    public void displayInfo() {
        System.out.println("Java Programming Language");
    }
}

public class Main {
    public static void main(String[] args) {
        Java javaLanguage = new Java();
        javaLanguage.displayInfo(); // Calls the overridden method

        Language commonLanguage = new Language();
        commonLanguage.displayInfo(); // Calls the superclass method
    }

}
```

40. What are checked and unchecked exceptions in Java? Provide examples of each and

explain how they differ.

Checked Exceptions:

Definition: Checked exceptions are exceptions that a method must handle in its body or explicitly declare using the throws keyword in its method signature.

Compile-Time Checking: The Java compiler enforces handling of checked exceptions during compilation.

```
try {

    Integer.parseInt("six"); // This line throws a NumberFormatException

} catch (NumberFormatException nfe) {

    // Handle the exception

}
```

Unchecked Exceptions (Runtime Exceptions):

- Definition: Unchecked exceptions (also known as runtime exceptions) do not require explicit handling. They are not checked by the compiler during compilation.
- Runtime Checking: These exceptions occur at runtime due to programming errors

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

int result = numbers[10]; // Throws ArrayIndexOutOfBoundsException
```

41. How does Java enforce exception handling for checked exceptions? Write a program

that demonstrates handling a checked exception.

checked exceptions are exceptions that must be handled during compile time. If a method can throw a checked exception, it must either handle it using a try-catch block or declare it using the throws keyword in its method signature.

```
import java.io.*;


public class CheckedExceptionExample {

    public static void main(String[] args) {

        try {

            FileInputStream fileInputStream = new FileInputStream("non_existing_file.txt");

            // Read from the file (not shown in this example)
            System.out.println("File opened successfully.");
        } catch (FileNotFoundException e) {
            System.out.println("File not found. Please check the file path.");
        }
    }
}
```

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

Steps to Create a Custom Checked Exception

1. Define the Exception Class:
    - Create a new class that extends Exception (or any subclass of Exception other than RuntimeException).
    - Provide constructors that initialize the exception with a message and/or cause.
2. Use the Custom Exception:
    - Throw the custom exception in your code when a specific condition occurs.
    - Handle the custom exception where it is thrown or declare it in the method signature if it needs to be handled by the caller.

```
// Define a custom checked exception
public class InvalidAgeException extends Exception {
    // Constructor that accepts a message
    public InvalidAgeException(String message) {
        super(message);
    }

    // Constructor that accepts a message and a cause
    public InvalidAgeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

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

Inner Class Basics:

- An inner class can access the members of its outer class, including private members.
- Inner classes are used to encapsulate related functionality within a single logical unit.

```
class Outer {
    class Inner {
        public void show() {
            System.out.println("In a nested class method");
        }
    }
}


public class Main {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.show();
    }
}
```

Differences from Regular Classes

- Scope: Inner classes are scoped within their outer class, meaning they can only be instantiated from within the outer class or context where they are defined.
- Access: Inner classes can access the outer class's members (fields and methods) directly, including private members. Regular classes do not have this capability unless through specific means.
- Instantiation: Inner classes can be instantiated only in the context of an instance of the outer class

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

implements an interface.

An interface in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types.

```
interface Language {
    void getType(); // Abstract method
    void getVersion(); // Abstract method
}
```

Implementing an Interface:

- Other classes can implement the interface by providing concrete implementations for its methods.
- Use the implements keyword to implement an interface.

```
class JavaLanguage implements Language {
    public void getType() {
        System.out.println("Type: Object-oriented");
    }

    public void getVersion() {
        System.out.println("Version: Java 17");
    }
}

public class Main {
    public static void main(String[] args) {
        JavaLanguage java = new JavaLanguage();
        java.getType();
        java.getVersion();
    }
}
```

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

1. Inner Classes in Interfaces:
    - An interface can contain inner classes just like any other class.
    - Inner classes within an interface are implicitly public and static.
    - These inner classes can be used to encapsulate related functionality within the interface.

```
interface Language {
    void getType(); // Abstract method

    class Version {
        private String versionNumber;

        public Version(String versionNumber) {
            this.versionNumber = versionNumber;
        }

        public void display() {
            System.out.println("Language version: " + versionNumber);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Language.Version java17 = new Language.Version("Java 17");
        java17.display();
    }
}
```

46. What is modularity in Java? How does it benefit large-scale applications? Write a

simple Java module and explain its components.

Modularity in Java refers to the concept of dividing a program into distinct, independent modules or units, each encapsulating a specific part of the functionality.

Benefits of Modularity in Large-Scale Applications

1. Improved Maintainability: Modularity allows different parts of the application to be developed, tested, and maintained independently. This makes it easier to manage changes and fix bugs without affecting other parts of the system.
2. Enhanced Reusability: Modules can be reused across different applications or parts of the same application. This reduces code duplication and fosters consistency.
3. Clear Separation of Concerns: Modularity ensures that different functionalities are separated into distinct units, making the application easier to understand and manage.
4. Better Scalability: As applications grow, modularity allows for better management and scaling of different components independently.

```
// File: com/example/math/module-info.java
module com.example.math {
    exports com.example.math.utils;
}
```

47. How do you create a module in Java? Provide an example program that includes

multiple modules and explains how they interact.

1: Create Module A (com.example.math)

- Directory Structure:

lua

Copy code

```
com.example.math/
├── src/
│   └── com/
│       └── example/
│           └── math/
│               ├── module-info.java
│               └── MathUtils.java
└── build/
```

- Define module-info.java:

java

Copy code

```
// File: com/example/math/module-info.java
module com.example.math {
    exports com.example.math.utils;
}
```

- Implement MathUtils Class:

java

Copy code

```
// File: com/example/math/utils/MathUtils.java
package com.example.math.utils;

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

    public static int subtract(int a, int b) {
        return a - b;
    }
}
```

Step 2: Create Module B (com.example.app)

- Directory Structure:

arduino

Copy code

```
com.example.app/
├── src/
│   └── com/
│       └── example/
│           └── app/
│               ├── module-info.java
│               └── MainApp.java
└── build/
```

- Define module-info.java:

java

Copy code

```
// File: com/example/app/module-info.java
module com.example.app {
    requires com.example.math;
}
```

- Implement MainApp Class:

java

Copy code

```
// File: com/example/app/MainApp.java
package com.example.app;

import com.example.math.utils.MathUtils;

public class MainApp {
    public static void main(String[] args) {
        int sum = MathUtils.add(5, 3);
        int difference = MathUtils.subtract(5, 3);

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
    }
}
```

Step 3: Compile and Run Modules

1. Compile Module A (com.example.math):

bash

Copy code

```
javac -d build/modulepath/com.example.math src/com/example/math/module-info.java
src/com/example/math/utils/MathUtils.java
```

2. Compile Module B (com.example.app):

bash

Copy code

```
javac --module-path build/modulepath -d build/modulepath/com.example.app src/com/example/app/module-
info.java src/com/example/app/MainApp.java
```

3. Run the Application:

bash

Copy code

```
java --module-path build/modulepath -m com.example.app/com.example.app.MainApp
```

48. 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.

1. Role of module-info.java:
    - The module-info.java file provides essential information about a module, including its name, dependencies, exported packages, and services.
    - It ensures strong encapsulation by explicitly specifying which parts of the module are accessible from outside.
    - The module descriptor allows the Java runtime to enforce module boundaries and resolve dependencies during both compile time and runtime.
2. Creating an Example Module: Let’s create a simple example module called my.module with a module-info.java file:

Java

```
// my.module/module-info.java
module my.module {
    exports com.example.mymodule; // Expose this package
}
```

AI-generated code. Review and use carefully.

In this example:

- We define a module named my.module.
- The com.example.mymodule package is made public using exports.

3. Explanation:
   - module my.module: Declares the name of the module.
   - exports com.example.mymodule: Makes the com.example.mymodule package accessible outside the module.
   - You can add more details (dependencies, services, etc.) to the module descriptor as needed.
4. Benefits:
   - Explicitly defines module boundaries.
   - Helps manage dependencies and encapsulation.
   - Enables better modularization of code.

49. How do you sort an array of objects in Java? Provide an example using

the Comparable interface to sort objects.

Steps to Sort an Array of Objects Using Comparable

1. Implement the Comparable<T> interface: This is done by the class of the objects that need to be sorted.
2. Override the compareTo method: Define the logic for comparison inside this method.
3. Use Arrays.sort to sort the array: After implementing the Comparable interface, you can call Arrays.sort on the array.

```
// File: Person.java
public class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    // Getter for name
    public String getName() {
        return name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Implementing compareTo method
    @Override
    public int compareTo(Person other) {
        // Sort by age (ascending)
        return Integer.compare(this.age, other.age);
    }

    // Overriding toString to display person details
    @Override
    public String toString() {
        return name + " (Age: " + age + ")";
    }
}
```

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

```
import java.util.ArrayList;
import java.util.List;
```

```
public class Main {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");
        fruits.add("Mango");

        // Removing by index
        int indexToRemove = 1; // Remove the second element (Banana)
        fruits.remove(indexToRemove);

        // Removing by value
        String valueToRemove = "Orange";
        fruits.remove(valueToRemove);

        // Print the updated ArrayList
        System.out.println(fruits); // [Apple, Mango]
    }
}
```

51. 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.

Pass by reference (actually pass-by-value of the reference): Java passes the reference (memory address) of the object, so the method can modify the object's fields.

The object itself is not copied, only the reference is passed.

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

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

    public void setName(String newName) {
        this.name = newName;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        System.out.println("Before modification: " + person);

        modifyPersonName(person, "Eve");

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

    public static void modifyPersonName(Person p, String newName) {
        p.setName(newName);
    }
```

```
}
```