2.28. Q5c: Write a java program that executes two threads. One thread displays “Java Programming” every 3 seconds, and the other displays “Semester - 4th IT” every 6 seconds.(Create the threads by extending the Thread class)

1. 4341602 - Java: Winter 2023 Paper Solution

1.1. Q1a: List out basic concepts of Java OOP. Explain any one in detail.

Basic Concepts of Java OOP (Object-Oriented Programming):

1. **Classes and Objects**: Classes are blueprints for objects. They define the properties (attributes) and behaviors (methods) that objects of that class will have. Objects are instances of classes.
2. **Encapsulation**: Encapsulation refers to the bundling of data (attributes) and methods that operate on the data into a single unit or class. It hides the internal state of an object from the outside world and only exposes the necessary functionalities.
3. **Inheritance**: Inheritance is a mechanism in which a new class inherits properties and behaviors from an existing class. The new class (subclass or derived class) can reuse the code of the existing class (superclass or base class) and can also add its own unique features.
4. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It allows methods to be called on objects of different classes through a common interface, often resulting in different behaviors depending on the type of object.
5. **Abstraction**: Abstraction is the process of hiding the implementation details and showing only the essential features of the object. It helps in reducing programming complexity and effort.
6. **Association**: Association represents a relationship between two or more classes where objects of one class are connected with objects of another class through a specific type of relationship. It can be one-to-one, one-to-many, or many-to-many.
7. **Composition**: Composition is a special form of association where one class contains objects of another class as part of its state. The composed objects cannot exist independently of the containing class.

One of the concepts I'll explain in detail is Inheritance:

Inheritance:

Inheritance is one of the fundamental concepts of object-oriented programming. It allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). This promotes code reusability and establishes a hierarchical relationship between classes.

Example:

```java
// Base class or superclass
class Animal {
    void eat() {
        System.out.println("Animal is eating...");
    }
}

// Derived class or subclass inheriting from Animal
```

```java
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // inherited from Animal
        dog.bark(); // unique to Dog
    }
}
```

In this example, `Animal` is the superclass, and `Dog` is the subclass. The `Dog` class inherits the `eat()` method from the `Animal` class. By using inheritance, we can avoid rewriting the `eat()` method in the `Dog` class, thus promoting code reuse.

Inheritance supports the concept of **code extensibility**, as the subclass can add its own unique features (such as the `bark()` method in this example) while retaining the features of the superclass.

Inheritance also facilitates **polymorphism**, as objects of the subclass can be treated as objects of the superclass, enabling more flexible and generic code.

1.2. Q1b: Explain JVM in detail.

The Java Virtual Machine (JVM) is a crucial component of the Java Runtime Environment (JRE). It plays a central role in executing Java bytecode, which is the compiled form of Java source code. Below, I'll explain the JVM in detail:

1. Execution Environment:

- The JVM provides a runtime environment for executing Java bytecode. It abstracts away the underlying hardware and operating system details, providing platform independence.
- JVM implementations are available for various platforms, including Windows, Linux, macOS, and others.

2. Just-In-Time (JIT) Compilation:

- The JVM employs a combination of interpretation and Just-In-Time (JIT) compilation techniques for bytecode execution.
- Initially, bytecode is interpreted, which involves executing the bytecode instructions one by one. This allows for quick startup and adaptive optimization.
- As the program runs, the JVM identifies frequently executed code segments (hot spots) and applies JIT compilation to translate these segments into native machine code for improved performance.

3. Memory Management:

- The JVM manages memory allocation and deallocation for Java objects through automatic memory management, known as garbage collection.
- It divides the memory into different areas such as the heap, method area (or permgen space), and stack.

- The heap is used for storing objects dynamically allocated during program execution. Garbage collection is responsible for reclaiming memory occupied by unreachable objects in the heap.
- The stack is used for storing method invocations and local variables.

4. Class Loading and Dynamic Class Loading:

- The JVM dynamically loads Java classes into memory as they are referenced during program execution.
- Class loading involves locating the binary representation of a class, reading it into memory, and then defining it within the JVM.
- JVM supports dynamic class loading, allowing classes to be loaded at runtime based on specific conditions or requirements, such as through the use of reflection or custom class loaders.

5. Security and Sandboxing:

- The JVM incorporates various security features to ensure safe execution of Java programs.
- Security Manager: It defines a security policy that specifies the permissions granted to Java code based on its origin and other factors.
- Bytecode Verification: Before executing bytecode, the JVM performs bytecode verification to ensure it adheres to the language specifications, preventing malicious code from being executed.

6. Performance Monitoring and Profiling:

- JVMs often include tools for performance monitoring and profiling, allowing developers to analyze the runtime behavior of Java applications.
- These tools provide insights into CPU utilization, memory usage, garbage collection activity, and other performance-related metrics, helping developers optimize their code.

In summary, the JVM provides a robust execution environment for Java programs, abstracting away hardware and operating system details while offering features such as memory management, dynamic class loading, security, and performance monitoring. Its ability to execute Java bytecode efficiently makes it a key component of the Java platform, enabling the development of portable and scalable applications.

1.3. Q1c: Write a program in java to print Fibonacci series for n terms.

Sure, here's a Java program to print the Fibonacci series for n terms:

```
import java.util.Scanner;

public class FibonacciSeries {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of terms in the Fibonacci series: ");
        int n = scanner.nextInt();
        scanner.close();

        System.out.println("Fibonacci series for " + n + " terms:");
        int firstTerm = 0, secondTerm = 1;
```

```java
        // Print the first two terms
        System.out.print(firstTerm + " " + secondTerm + " ");

        // Generate and print the rest of the terms
        for (int i = 3; i <= n; i++) {
            int nextTerm = firstTerm + secondTerm;
            System.out.print(nextTerm + " ");
            firstTerm = secondTerm;
            secondTerm = nextTerm;
        }
    }
}
```

This program prompts the user to enter the number of terms (n) they want in the Fibonacci series. It then calculates and prints the Fibonacci series for n terms. The Fibonacci series starts with 0 and 1, and each subsequent term is the sum of the previous two terms. The loop iterates from the third term onwards, calculating each term based on the previous two terms. Finally, it prints each term of the Fibonacci series.

1.4. Q1c: Write a program in java to find out minimum from any ten numbers using command line argument.

Sure, here's a Java program that finds the minimum from any ten numbers using command-line arguments:

```java
public class MinimumNumberFinder {
    public static void main(String[] args) {
        if (args.length != 10) {
            System.out.println("Please provide exactly 10 numbers as command line
arguments.");
            return;
        }

        // Parse the command line arguments and find the minimum
        int min = Integer.parseInt(args[0]); // Assume the first number as the
minimum initially

        for (int i = 1; i < args.length; i++) {
            int num = Integer.parseInt(args[i]);
            if (num < min) {
                min = num; // Update min if a smaller number is found
            }
        }

        System.out.println("The minimum number among the given ten numbers is: " +
min);
    }
}
```

To run this program, compile it using `javac MinimumNumberFinder.java` and then execute it with ten numbers as command-line arguments:

```
java MinimumNumberFinder 5 3 9 2 8 1 7 6 4 10
```

This will output:

```
The minimum number among the given ten numbers is: 1
```

Ensure that exactly ten numbers are provided as command-line arguments when running the program, otherwise, it will display an error message.

1.5. Q2a: What is Java wrapper class? Explain with example.

In Java, a wrapper class is a class that encapsulates (or "wraps") primitive data types into objects. While primitive data types represent simple values, wrapper classes provide a way to treat these values as objects. This is particularly useful when dealing with collections, as many collection classes in Java require objects, not primitives.

The Java platform provides a set of predefined wrapper classes for each primitive data type:

1. `Byte` for `byte`
2. `Short` for `short`
3. `Integer` for `int`
4. `Long` for `long`
5. `Float` for `float`
6. `Double` for `double`
7. `Character` for `char`
8. `Boolean` for `boolean`

Here's an example to illustrate the usage of wrapper classes:

```
public class WrapperExample {
    public static void main(String[] args) {
        // Using primitive data types
        int num1 = 10;
        double num2 = 3.14;
        char letter = 'A';
        boolean flag = true;

        // Using wrapper classes
        Integer numObj1 = Integer.valueOf(num1); // Wrapping int into Integer
        Double numObj2 = Double.valueOf(num2);   // Wrapping double into Double
        Character charObj = Character.valueOf(letter); // Wrapping char into
Character
        Boolean flagObj = Boolean.valueOf(flag); // Wrapping boolean into Boolean

        // Displaying values
        System.out.println("Wrapped Integer value: " + numObj1);
        System.out.println("Wrapped Double value: " + numObj2);
        System.out.println("Wrapped Character value: " + charObj);
        System.out.println("Wrapped Boolean value: " + flagObj);
    }
}
```

In this example, we have primitive variables (`num1`, `num2`, `letter`, `flag`) representing different data types. We then use the corresponding wrapper classes (`Integer`, `Double`, `Character`, `Boolean`) to wrap these primitive values into objects (`numObj1`, `numObj2`, `charObj`, `flagObj`). Finally, we print out the values of these wrapped objects.

Wrapper classes also provide utility methods to convert strings into primitive values and vice versa, and to perform various operations on the wrapped values. They also facilitate interoperability between primitive types and objects in Java.

1.6. Q2b: List out different features of java. Explain any two.

Java is a versatile programming language known for its rich set of features that contribute to its popularity and widespread use. Here are some key features of Java:

1. **Simple**: Java was designed to be easy to learn and use. It has a concise, readable syntax, automatic memory management (garbage collection), and eliminates complex features such as pointers and operator overloading found in languages like C++.
2. **Object-Oriented**: Java is an object-oriented programming language, which means it supports the creation of modular, reusable code through classes and objects. It embodies concepts like encapsulation, inheritance, polymorphism, and abstraction, promoting better code organization and maintenance.
3. **Platform-Independent**: Java programs are compiled into bytecode, which can be executed on any platform with a Java Virtual Machine (JVM). This "write once, run anywhere" capability makes Java platform-independent, enabling the development of cross-platform applications.
4. **Secure**: Java's security features help protect systems from malicious code and unauthorized access. It incorporates a robust security model with features like bytecode verification, class loaders, and a Security Manager that enforces access control policies.
5. **Multithreaded**: Java provides built-in support for multithreading, allowing concurrent execution of multiple threads within a single program. This enables developers to write efficient, responsive applications that can perform tasks concurrently, enhancing performance and responsiveness.
6. **Dynamic**: Java supports dynamic memory allocation and dynamic class loading, enabling applications to adapt to changing runtime conditions. Dynamic features like reflection allow Java programs to introspect and modify their own structure and behavior at runtime.
7. **High Performance**: While Java's interpreted nature might suggest slower performance compared to languages like C or C++, modern Java implementations use techniques like Just-In-Time (JIT) compilation and adaptive optimization to achieve high performance, often rivaling or surpassing native code performance.
8. **Distributed**: Java's built-in networking capabilities and Remote Method Invocation (RMI) framework facilitate the development of distributed applications. Java's networking APIs allow seamless communication between distributed components, making it suitable for building networked systems.

Let's delve into explanations for two of these features:

1. Platform-Independence:
Java achieves platform-independence through its bytecode compilation. When you compile a Java source file, it's translated into bytecode, which is a platform-independent intermediate representation of the program. This bytecode can then be executed on any device or platform that has a Java Virtual Machine (JVM). The JVM interprets the bytecode and translates it into machine code that is specific to the underlying hardware and operating system. This allows Java programs to run on diverse platforms without modification, making it an ideal choice for developing cross-platform applications.

2. Object-Oriented:
Java is a pure object-oriented programming language, which means it revolves around the concept of objects. Everything in Java is an object, which has attributes (fields or properties) and behaviors (methods). Object-oriented programming promotes modularity, reusability, and extensibility of code. Encapsulation ensures that the internal state of an object is hidden from the outside world, providing data security and abstraction. Inheritance allows classes to inherit properties and behaviors from other classes, facilitating code reuse and hierarchical organization. Polymorphism enables objects to exhibit different behaviors based on their types, enhancing flexibility and code maintainability. Java's object-oriented features make it well-suited for building large-scale, maintainable software systems.

1.7. Q2c: What is method overload in Java ? Explain with example.

Method overloading in Java refers to the ability to define multiple methods within the same class with the same name but different parameter lists. These methods can have different numbers or types of parameters. Java distinguishes between overloaded methods based on the number, type, and sequence of their parameters.

When a method is invoked, Java determines which overloaded method to call based on the arguments provided at the time of invocation. This process is known as compile-time polymorphism or static polymorphism because the decision on which method to call is made by the compiler at compile time, rather than at runtime.

Here's an example to illustrate method overloading in Java:

```java
public class Calculator {

    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }

    // Method to concatenate two strings
    public String add(String a, String b) {
        return a + b;
    }

    // Method to add an integer and a double
    public double add(int a, double b) {
        return a + b;
    }
}
```

In this example, the `Calculator` class contains multiple overloaded `add` methods:

1. `add(int a, int b)`: Adds two integers and returns the result.
2. `add(int a, int b, int c)`: Adds three integers and returns the result.
3. `add(double a, double b)`: Adds two doubles and returns the result.
4. `add(String a, String b)`: Concatenates two strings and returns the result.
5. `add(int a, double b)`: Adds an integer and a double and returns the result.

These methods have the same name (`add`) but different parameter lists. Depending on the arguments passed during the method invocation, Java determines which overloaded method to call. For example:

```java
public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        int sum1 = calculator.add(5, 3); // Calls add(int a, int b)
        int sum2 = calculator.add(5, 3, 2); // Calls add(int a, int b, int c)
        double sum3 = calculator.add(2.5, 3.7); // Calls add(double a, double b)
        String concatenatedString = calculator.add("Hello ", "world!"); // Calls
add(String a, String b)
        double sum4 = calculator.add(5, 3.7); // Calls add(int a, double b)

        System.out.println("Sum 1: " + sum1);
        System.out.println("Sum 2: " + sum2);
        System.out.println("Sum 3: " + sum3);
        System.out.println("Concatenated String: " + concatenatedString);
        System.out.println("Sum 4: " + sum4);
    }
}
```

Output:

```
Sum 1: 8
Sum 2: 10
Sum 3: 6.2
Concatenated String: Hello world!
Sum 4: 8.7
```

In this example, depending on the type and number of arguments provided, Java resolves the method calls to the appropriate overloaded `add` method during compilation.

1.8. Q2a: Explain Garbage collection in java.

Garbage collection in Java is the automatic process of reclaiming memory occupied by objects that are no longer in use or reachable by the application. It is a fundamental feature of the Java Virtual Machine (JVM) that helps manage memory efficiently, prevents memory leaks, and reduces the risk of memory-related errors such as segmentation faults.

Here's how garbage collection works in Java:

1. **Object Allocation**: When you create objects in Java using the `new` keyword, memory is allocated from the heap to store those objects. The JVM keeps track of all allocated memory.
2. **Reachability Analysis**: The JVM periodically performs reachability analysis starting from a set of root objects, typically references held by active threads, static variables, and local variables. It traverses the object graph, marking objects that are reachable as live objects. Objects that are not reachable from any root are considered garbage.
3. **Garbage Collection Process**: Once the reachability analysis identifies garbage objects, the garbage collector (GC) is invoked to reclaim the memory occupied by those objects. The garbage collector uses different algorithms to reclaim memory, such as the Mark-Sweep algorithm, Mark-Compact algorithm, or Generational Garbage Collection.
4. **Reclamation and Compaction**: During garbage collection, the memory occupied by garbage objects is reclaimed, and the memory space is compacted to reduce fragmentation. This involves moving live objects together to create contiguous free space.
5. **Finalization**: Before reclaiming the memory of objects, the JVM calls the `finalize()` method of those objects (if it's overridden) to perform any necessary cleanup operations. However, it's important to note that the `finalize()` method is deprecated and is not guaranteed to be called promptly or at all by the garbage collector.
6. **Performance Considerations**: Garbage collection can impact application performance, as it involves stopping application threads temporarily to perform garbage collection tasks. To minimize the impact on application responsiveness, modern JVMs use techniques like concurrent garbage collection, where garbage collection runs concurrently with the application, and incremental garbage collection, where garbage collection tasks are divided into smaller increments.

Here are some key benefits of garbage collection in Java:

- **Automatic Memory Management**: Developers do not need to manually allocate and deallocate memory, reducing the risk of memory leaks and memory-related bugs.
- **Simplified Memory Management**: Garbage collection eliminates the need for explicit memory management techniques like manual memory deallocation, reducing the complexity of programming.
- **Improved Application Reliability**: By preventing memory leaks and segmentation faults caused by dangling pointers, garbage collection enhances the reliability and stability of Java applications.

Overall, garbage collection is a critical feature of the Java platform that helps manage memory efficiently, allowing developers to focus on writing robust and reliable software.

1.9. Q2b: Explain final keyword in Java with example.

In Java, the `final` keyword is used to restrict the behavior of classes, methods, and variables. When applied to different elements, it signifies different meanings:

1. **Final Variables**: When applied to a variable, the `final` keyword indicates that the variable's value cannot be changed once initialized. It creates a constant.
2. **Final Methods**: When applied to a method, the `final` keyword indicates that the method cannot be overridden by subclasses. It effectively prevents method overriding.
3. **Final Classes**: When applied to a class, the `final` keyword indicates that the class cannot be subclassed. It prevents inheritance.

Here's how `final` keyword works with examples:

1. Final Variables:

```
public class FinalExample {
    // Declaring final variable
    final int constantValue = 10;

    public static void main(String[] args) {
        FinalExample obj = new FinalExample();
        // Trying to modify the final variable will result in a compilation error
        // obj.constantValue = 20; // Compilation error: The final field
FinalExample.constantValue cannot be assigned
        System.out.println("Constant value: " + obj.constantValue);
    }
}
```

In this example, `constantValue` is declared as a final variable. Attempting to modify its value after initialization will result in a compilation error.

2. Final Methods:

```
public class Parent {
    // Final method
    public final void display() {
        System.out.println("This is a final method.");
    }
}

public class Child extends Parent {
    // Trying to override the final method will result in a compilation error
    // @Override
    // public void display() {
    //     System.out.println("Attempting to override a final method.");
    // }
}
```

In this example, the `display()` method in the `Parent` class is declared as final. Attempting to override this method in the `Child` class will result in a compilation error.

3. Final Classes:

```java
final public class FinalClass {
    // Some code
}

// Trying to subclass a final class will result in a compilation error
// class SubClass extends FinalClass {
//     // Some code
// }
```

In this example, the `FinalClass` is declared as a final class. Attempting to subclass `FinalClass` will result in a compilation error.

In summary, the `final` keyword in Java is used to create constants, prevent method overriding, and prevent class inheritance, depending on where it's applied. It helps enforce immutability, security, and design constraints in Java programs.

1.10. Q2c: What is constructor in Java? Explain parameterized constructor with example.

In Java, a constructor is a special type of method that is automatically called when an instance (object) of a class is created. It is used to initialize the newly created object and perform any necessary setup operations. Constructors have the same name as the class and do not have a return type, not even `void`.

There are two types of constructors in Java:

1. **Default Constructor**: A constructor with no parameters is called a default constructor. If you do not explicitly define any constructors in a class, Java provides a default constructor automatically. Its purpose is to initialize instance variables to default values.
2. **Parameterized Constructor**: A constructor with parameters is called a parameterized constructor. It allows you to initialize instance variables with specified values when the object is created. Parameterized constructors give more flexibility and control over object initialization.

Here's an example of a parameterized constructor:

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

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

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

    public int getAge() {
        return age;
    }
    public static void main(String[] args) {
        // Creating objects using parameterized constructor
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);

        // Accessing object properties
        System.out.println("Person 1 - Name: " + person1.getName() + ", Age: " +
person1.getAge());
        System.out.println("Person 2 - Name: " + person2.getName() + ", Age: " +
person2.getAge());
    }
}
```

In this example:

- We have a `Person` class with private instance variables `name` and `age`.
- The `Person` class has a parameterized constructor that takes two parameters: `name` and `age`.
- Inside the constructor, the values of `name` and `age` parameters are assigned to the corresponding instance variables using the `this` keyword.
- We then create two `Person` objects (`person1` and `person2`) using the parameterized constructor, passing different values for `name` and `age`.
- Finally, we use getter methods (`getName()` and `getAge()`) to retrieve the values of `name` and `age` for each object and print them out.

1.11. Q3a: Explain super keyword in Java with example.

In Java, the `super` keyword is used to refer to the superclass (parent class) of the current object or to access members (fields or methods) of the superclass. It is often used in subclasses (child classes) to access superclass constructors, methods, or variables. The `super` keyword is particularly useful when there is a need to differentiate between superclass and subclass members with the same name.

Here are the main uses of the `super` keyword:

1. **Accessing Superclass Constructors**: The `super()` constructor call is used to invoke the constructor of the superclass from within the constructor of the subclass. It is typically used when the subclass constructor needs to perform additional initialization that is not handled by the superclass constructor.
2. **Accessing Superclass Methods and Variables**: The `super` keyword can also be used to access methods and variables of the superclass. This is useful when a subclass overrides a method from the superclass but still needs to call the superclass implementation of that method.

Here's an example to illustrate the use of the `super` keyword:

```
class Vehicle {
    int speed;
```

```java
    Vehicle(int speed) {
        this.speed = speed;
    }

    void display() {
        System.out.println("Vehicle speed: " + speed + " km/h");
    }
}

class Car extends Vehicle {
    int mileage;

    Car(int speed, int mileage) {
        super(speed); // invoking superclass constructor
        this.mileage = mileage;
    }

    // overriding superclass method
    void display() {
        super.display(); // calling superclass method
        System.out.println("Car mileage: " + mileage + " km/l");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car(100, 15);
        car.display(); // invoking overridden method
    }
}
```

In this example:

- We have a superclass `Vehicle` with a field `speed` and a constructor to initialize the `speed`.
- We then have a subclass `Car` that extends the `Vehicle` class. The `Car` class has an additional field `mileage` and a constructor to initialize both `speed` and `mileage`. Inside the `Car` constructor, we use `super(speed)` to call the constructor of the superclass `Vehicle`.
- The `Car` class also overrides the `display()` method of the superclass. Inside the overridden `display()` method, we use `super.display()` to call the `display()` method of the superclass before displaying the `mileage` of the car.
- In the `main()` method, we create an instance of the `Car` class and invoke its `display()` method. This will print both the vehicle speed and the car mileage.

1.12. Q3b: List out different types of inheritance in Java. Explain multilevel inheritance.

In Java, there are several types of inheritance, each representing different relationships between classes. These types include:

1. **Single Inheritance**: In single inheritance, a subclass inherits from only one superclass. Java supports single inheritance only, meaning a class can have only one direct superclass.

2. **Multilevel Inheritance**: In multilevel inheritance, a subclass inherits from a superclass, and then another subclass inherits from the first subclass, creating a chain of inheritance.
3. **Hierarchical Inheritance**: In hierarchical inheritance, multiple subclasses inherit from a single superclass, creating a tree-like structure.
4. **Multiple Inheritance** (not supported in Java): In multiple inheritance, a subclass inherits from multiple superclasses. Java does not support multiple inheritance of classes to avoid the diamond problem, where the same member can be inherited from multiple superclasses, leading to ambiguity.
5. **Hybrid Inheritance** (not supported in Java): Hybrid inheritance is a combination of multiple inheritance and hierarchical inheritance. It is also not supported in Java to avoid complications and ambiguity.

Let's focus on explaining **multilevel inheritance**:

Multilevel Inheritance:

In multilevel inheritance, a subclass extends a class that is itself a subclass of another class. This creates a chain of inheritance, where each subclass inherits the properties and behaviors of its immediate superclass, as well as all of its ancestor classes up the hierarchy chain.

Here's an example of multilevel inheritance:

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

// Subclass inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking...");
    }
}

// Subclass inheriting from Dog
class Labrador extends Dog {
    void color() {
        System.out.println("Labrador is black...");
    }
}

public class Main {
    public static void main(String[] args) {
        Labrador labrador = new Labrador();
        labrador.eat();   // inherited from Animal
        labrador.bark();  // inherited from Dog
        labrador.color(); // unique to Labrador
    }
}
```

In this example:

- `Animal` is the superclass, `Dog` is a subclass of `Animal`, and `Labrador` is a subclass of `Dog`.
- `Labrador` inherits the `eat()` method from `Animal`, the `bark()` method from `Dog`, and it adds its own method `color()` to represent the unique characteristic of a Labrador.
- When an instance of `Labrador` is created, it can access methods from all levels of the inheritance hierarchy, including methods from its superclass (`Animal`) and its immediate superclass (`Dog`). Additionally, it can access methods specific to the `Labrador` class.

Multilevel inheritance allows for the creation of a hierarchy of classes, where each subclass can inherit and extend the functionality of its parent classes, leading to better code organization and reuse. However, it's important to use multilevel inheritance judiciously to avoid creating overly complex class hierarchies.

1.13. Q3c: What is Java interface? Explain multiple inheritance with example.

In Java, an interface is a reference type similar to a class that defines a set of abstract methods and constants. An interface can also contain default methods, static methods, and nested types. It provides a way to achieve abstraction and multiple inheritance of type. Interfaces are used to specify a contract that classes must adhere to by implementing the methods declared in the interface.

Here's the syntax for declaring an interface in Java:

```
interface MyInterface {
    // Constant declarations
    int CONSTANT1 = 1;
    String CONSTANT2 = "Hello";

    // Abstract method declarations
    void method1();
    int method2(int x);
}
```

In the above example, `MyInterface` is an interface that declares a constant `CONSTANT1` and `CONSTANT2`, along with two abstract methods `method1()` and `method2(int x)`.

Now, let's discuss multiple inheritance with interfaces:

Multiple Inheritance with Interfaces:

Java supports multiple inheritance of type through interfaces, but it does not support multiple inheritance of implementation. This means a class can implement multiple interfaces, inheriting abstract method signatures from all of them, but it cannot extend multiple classes.

Here's an example to illustrate multiple inheritance with interfaces:

```
interface A {
    void methodA();
}

interface B {
    void methodB();
}
```

```java
class MyClass implements A, B {
    // Implementation of methodA from interface A
    public void methodA() {
        System.out.println("Method A implementation");
    }

    // Implementation of methodB from interface B
    public void methodB() {
        System.out.println("Method B implementation");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.methodA(); // Method A implementation
        obj.methodB(); // Method B implementation
    }
}
```

In this example:

- `interface A` declares an abstract method `methodA()`.
- `interface B` declares an abstract method `methodB()`.
- `MyClass` implements both interfaces `A` and `B` and provides implementations for both `methodA()` and `methodB()`.
- In the `main` method, we create an instance of `MyClass` and call both `methodA()` and `methodB()`, which will print their respective implementation messages.

By implementing multiple interfaces, `MyClass` inherits the abstract method signatures from both `A` and `B`, effectively achieving multiple inheritance of type. This allows for increased flexibility and code reuse while avoiding the complications associated with multiple inheritance of implementation.

1.14. Q3a: Explain Java static keyword with example.

In Java, the `static` keyword is used to define members (variables and methods) that belong to the class itself, rather than to instances of the class (objects). These members are shared among all instances of the class and can be accessed directly through the class name, without the need to create an object of the class.

Here's how `static` keyword works with examples:

1. **Static Variables (Class Variables)**: Static variables are shared among all instances of a class. They are declared using the `static` keyword and are initialized only once, at the start of the program execution.

```java
class MyClass {
    static int count = 0; // static variable

    MyClass() {
        count++; // increment count on each object creation
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass();
        MyClass obj2 = new MyClass();
        MyClass obj3 = new MyClass();

        System.out.println("Total objects created: " + MyClass.count); //
accessing static variable
    }
}
```

In this example, `count` is a static variable that keeps track of the total number of objects created from the `MyClass`. Since it's static, its value is shared among all instances of the class. The output will be `Total objects created: 3`.

2. **Static Methods (Class Methods)**: Static methods are associated with the class itself, rather than with instances of the class. They are declared using the `static` keyword and can be called directly through the class name, without the need to create an object of the class.

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

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.add(5, 3); // calling static method
        System.out.println("Result of addition: " + result);
    }
}
```

In this example, `add` is a static method of the `MathUtils` class. It can be called directly using the class name `MathUtils.add(5, 3)` without creating an object of `MathUtils`.

Key Points:

- Static members belong to the class, not to individual objects.
- They can be accessed using the class name directly.
- Static variables are initialized only once, at the start of the program execution.
- Static methods cannot access non-static members directly, as they are not associated with any specific instance of the class.
- Static members are commonly used for utility methods, constants, and for maintaining global state within a class.

1.15. Q3b: Explain different access controls in Java.

In Java, access controls are used to restrict the visibility and accessibility of classes, variables, methods, and constructors. This helps in encapsulating the implementation details, promoting code reusability, and enhancing security. Java provides four types of access controls, also known as access modifiers:

1. **Default (No Modifier)**:
    - Accessible within the same package only.
    - If no access modifier is specified, it is considered as default.
    - Members with default access are not accessible outside the package.

```java
package com.example;

class MyClass {
    void method() {
        // This method is accessible within the same package
    }
}
```

2. **Public**:
    - Accessible from anywhere, both within and outside the package.
    - Public members can be accessed by any other class.

```java
package com.example;

public class MyClass {
    public void method() {
        // This method is accessible from anywhere
    }
}
```

3. **Private**:
    - Accessible only within the same class.
    - Private members are not visible outside the class, including subclasses.

```java
package com.example;

public class MyClass {
    private int num;

    private void method() {
        // This method is accessible only within this class
    }
}
```

4. **Protected**:
    - Accessible within the same package and by subclasses, even if they are in different packages.

    - Protected members are not accessible by classes outside the package that are not subclasses.

```java
package com.example;

public class MyClass {
    protected int num;

    protected void method() {
        // This method is accessible within the same package and by subclasses
    }
}
```

These access controls provide a way to manage the visibility and accessibility of members in Java classes, allowing developers to design and implement classes with appropriate encapsulation and access restrictions based on their requirements. Proper use of access controls helps in creating more modular, maintainable, and secure Java applications.

1.16. Q3c: What is Java package? Write steps to create a package in Java and give example of it.

In Java, a package is a way to organize related classes and interfaces into a single namespace. It helps in avoiding naming conflicts, improving code organization, and providing access control. Packages can contain classes, interfaces, sub-packages, and other resources.

Here are the steps to create a package in Java:

1. **Choose a Package Name**: Determine a meaningful name for your package. Typically, package names are in reverse domain name notation to ensure uniqueness.
2. **Create a Directory Structure**: Create a directory structure that matches the package name. Each component of the package name corresponds to a directory in the file system.
3. **Place Java Files in the Directory**: Create Java files (`.java`) containing classes or interfaces within the directory structure. Each file should contain at most one public class or interface, and the file name should match the class or interface name.
4. **Define the Package Declaration**: At the top of each Java file, include a package declaration statement specifying the package name.
5. **Compile Java Files**: Compile the Java files using the `javac` compiler. Make sure the compiler is invoked from the root directory of the package structure.

Here's an example of creating and using a package in Java:

Suppose we want to create a package named `com.example.utils` containing a class named `StringUtils` with a method to capitalize a string.

Step 1: Choose a Package Name:

```
com.example.utils
```

Step 2: Create a Directory Structure:

```
- com
  - example
    - utils
```

Step 3: Place Java Files in the Directory:
Create a Java file named `StringUtils.java` containing the `StringUtils` class within the `com/example/utils` directory.

Step 4: Define the Package Declaration:
At the top of `StringUtils.java`, include the package declaration:

```java
package com.example.utils;
```

Step 5: Define the Class:
Define the `StringUtils` class with a method to capitalize a string:

```java
package com.example.utils;

public class StringUtils {
    public static String capitalize(String str) {
        if (str == null || str.isEmpty()) {
            return str;
        }
        return str.substring(0, 1).toUpperCase() + str.substring(1);
    }
}
```

Step 6: Compile Java Files:
Compile the `StringUtils.java` file. Make sure the current directory is the parent directory of `com`.

```
javac com/example/utils/StringUtils.java
```

After compiling, you can use the `StringUtils` class in other Java files by importing the package:

```java
import com.example.utils.StringUtils;

public class Main {
    public static void main(String[] args) {
        String str = "hello";
        String capitalized = StringUtils.capitalize(str);
        System.out.println(capitalized); // Output: Hello
    }
}
```

By following these steps, you've created and used a package in Java, demonstrating the organization and encapsulation benefits it provides.

1.17. Q4a: Explain Java thread priorities with suitable example.

In Java, thread priorities are used to indicate the importance or urgency of a thread's execution relative to other threads. Thread priorities are represented by integer values ranging from 1 to 10, where 1 is the lowest priority and 10 is the highest priority. The default priority for a thread is typically inherited from its parent thread, but it can be explicitly set using the `setPriority()` method.

Thread priorities are used by the Java Virtual Machine's thread scheduler to determine the order in which threads are scheduled for execution. However, thread priorities are merely hints to the scheduler, and the JVM's implementation of thread scheduling may vary across different platforms.

Here's an example to illustrate Java thread priorities:

```
public class PriorityDemo {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new Worker(), "Thread 1");
        Thread thread2 = new Thread(new Worker(), "Thread 2");
        Thread thread3 = new Thread(new Worker(), "Thread 3");

        // Set thread priorities
        thread1.setPriority(Thread.MIN_PRIORITY); // Lowest priority
        thread2.setPriority(Thread.NORM_PRIORITY); // Default priority
        thread3.setPriority(Thread.MAX_PRIORITY); // Highest priority

        // Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }

    static class Worker implements Runnable {
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + ": " + i);
                try {
                    Thread.sleep(1000); // Sleep for 1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

In this example:

- We create three threads (`thread1`, `thread2`, and `thread3`) and assign them instances of the `Worker` class, which implements the `Runnable` interface.

- We set different priorities for each thread using the `setPriority()` method. `thread1` is set to the lowest priority (`MIN_PRIORITY`), `thread2` is set to the default priority (`NORM_PRIORITY`), and `thread3` is set to the highest priority (`MAX_PRIORITY`).
- Each thread runs a simple loop printing numbers from 1 to 5 with a one-second delay between each iteration.
- When you run this program, the output may vary depending on the thread scheduler's behavior, but in general, you may observe that `thread3` (highest priority) gets more CPU time compared to the other threads, followed by `thread2` (default priority), and finally `thread1` (lowest priority). However, thread scheduling behavior is platform-dependent, and thread priorities should be used with caution as they may not always have the desired effect.

1.18. Q4b: What is Java Thread? Explain Thread life cycle.

In Java, a thread is the smallest unit of execution within a process. It represents an independent path of execution that can run concurrently with other threads in a Java program. Threads allow programs to perform multiple tasks simultaneously, making efficient use of CPU resources and enabling concurrent and parallel processing.

Thread Life Cycle:

The life cycle of a thread in Java consists of several states, and a thread transitions through these states during its lifetime. The states are typically represented by constants defined in the `Thread.State` enumeration. The thread life cycle states are as follows:

1. **New**: When a thread is created but not yet started, it is in the new state. The `Thread` object has been created, but the `start()` method has not been called.
2. **Runnable**: After the `start()` method is called, the thread becomes runnable. In this state, the thread is eligible to run, but it may or may not be executing, depending on the availability of CPU resources. Once the scheduler selects the thread for execution, it moves to the running state.
3. **Running**: When the thread is executing its code, it is in the running state. The thread scheduler has allocated CPU time for the thread, and the thread's `run()` method is being executed.
4. **Blocked/Waiting**: A thread can transition to a blocked or waiting state for various reasons, such as waiting for I/O operations to complete, waiting for locks, or waiting for other threads to complete. In these cases, the thread temporarily gives up the CPU and waits for the condition to be satisfied.
5. **Timed Waiting**: Similar to the blocked/waiting state, but with a specified timeout. Threads enter this state when they invoke methods such as `sleep()` or `join()` with a timeout parameter.
6. **Terminated**: When the `run()` method of the thread completes or when the thread is explicitly terminated using the `interrupt()` method, the thread enters the terminated state. Once terminated, a thread cannot be restarted or transitioned to any other state.

Here's a simple visual representation of the thread life cycle:

```
New       ->     Runnable    ->    Running   ->   Terminated
                          |        |     ^    ^
                          v        v     |    |
                        Blocked  Waiting  |    |
                                        |    |
                                  Timed Waiting
```

Understanding the thread life cycle is crucial for writing multithreaded Java applications efficiently, as it helps in managing and coordinating the execution of concurrent tasks. It allows developers to control thread behavior, handle synchronization, and avoid common concurrency issues such as race conditions and deadlocks.

1.19. Q4c: Write a program in java that create the multiple threads by implementing the Thread class.

Here's a simple Java program that creates multiple threads by implementing the `Thread` class:

```java
class MyThread extends Thread {
    private String threadName;

    public MyThread(String name) {
        this.threadName = name;
    }

    public void run() {
        System.out.println("Thread " + threadName + " is running.");
        try {
            // Simulating some work being done by the thread
            Thread.sleep(2000); // Sleep for 2 seconds
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }
}

public class MultiThreadExample {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread("Thread 1");
        MyThread thread2 = new MyThread("Thread 2");
        MyThread thread3 = new MyThread("Thread 3");

        // Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

In this program:

- We define a class `MyThread` that extends the `Thread` class. This class represents a simple thread that prints a message, does some simulated work (sleeps for 2 seconds), and then exits.
- The `run()` method is overridden to define the behavior of the thread.
- In the `MultiThreadExample` class, we create three instances of `MyThread` with different names (`Thread 1`, `Thread 2`, and `Thread 3`).
- We start each thread using the `start()` method. This method initiates the execution of the thread by invoking its `run()` method in a separate thread of control.
- As a result, all three threads are running concurrently, executing their tasks independently.
- The output may vary on each run, but you'll see messages indicating that each thread is running, then after a 2-second delay, it exits.

This example demonstrates how to create multiple threads by extending the `Thread` class and starting them concurrently to achieve parallel execution of tasks.

1.20. Q4a: List four different inbuilt exceptions of Java. Explain any one inbuilt exception.

In Java, there are many built-in exceptions provided by the Java API, which are organized in a hierarchy under the `java.lang.Exception` class. Here are four commonly encountered built-in exceptions:

1. **NullPointerException**: This exception occurs when you try to access or perform an operation on an object reference that is `null`.
2. **ArrayIndexOutOfBoundsException**: This exception occurs when you try to access an element of an array at an invalid index (i.e., an index that is less than 0 or greater than or equal to the length of the array).
3. **NumberFormatException**: This exception occurs when you try to convert a string to a numeric format (e.g., using `Integer.parseInt()` or `Double.parseDouble()`) but the string does not contain a valid numeric value.
4. **FileNotFoundException**: This exception occurs when an attempt to open a file or a file pathname specified by a string in the code fails because the file with the specified pathname does not exist or cannot be opened for reading.

Let's explain the `NullPointerException` in more detail:

NullPointerException:

A `NullPointerException` is one of the most common exceptions encountered by Java programmers. It occurs when you try to access or perform an operation on an object reference that is `null`, i.e., it does not refer to any object in memory.

Here's an example to illustrate a `NullPointerException`:

```java
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;
        System.out.println(str.length()); // This line will throw a
NullPointerException
    }
}
```

In this example, we have a `String` variable `str` that is initialized to `null`. When we try to access the `length()` method of `str`, a `NullPointerException` will be thrown at runtime because we are attempting to invoke a method on a `null` reference.

To handle a `NullPointerException`, you can either check if the reference is `null` before accessing it or use try-catch blocks to catch and handle the exception gracefully:

```java
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;

        // Using if statement to check for null reference
        if (str != null) {
            System.out.println(str.length());
        } else {
            System.out.println("String is null.");
        }

        // Using try-catch block to handle NullPointerException
        try {
            System.out.println(str.length());
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
        }
    }
}
```

It's important to handle `NullPointerExceptions` properly in your code to prevent unexpected crashes and ensure the robustness of your Java applications.

1.21. Q4b: Explain multiple catch with suitable example in Java.

In Java, you can use multiple `catch` blocks to handle different types of exceptions that may occur within a `try` block. This allows you to handle each type of exception differently, based on the specific error conditions that may arise during the execution of the code.

Here's an example to illustrate the usage of multiple `catch` blocks:

```java
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println("Element at index 3: " + numbers[3]); // This will
throw ArrayIndexOutOfBoundsException
```

```java
            String str = null;
            System.out.println("Length of string: " + str.length()); // This will throw NullPointerException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException caught: " + e.getMessage());
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Generic Exception caught: " + e.getMessage());
        }
    }
}
```

In this example:

- We have a `try` block containing two statements that may throw different types of exceptions:
    - Accessing an element at index 3 of an array (`numbers[3]`), which may throw an `ArrayIndexOutOfBoundsException`.
    - Attempting to get the length of a null string (`str.length()`), which may throw a `NullPointerException`.
- We have multiple `catch` blocks to handle each type of exception separately:
    - The first `catch` block catches `ArrayIndexOutOfBoundsException`, prints a message, and handles the exception.
    - The second `catch` block catches `NullPointerException`, prints a message, and handles the exception.
- We also have a generic `catch` block (`catch (Exception e)`) at the end to catch any other type of exception that may occur. This is optional but can be useful for handling unexpected exceptions or providing a fallback mechanism.

When you run this program, if an `ArrayIndexOutOfBoundsException` occurs, the first `catch` block will handle it and print a message. Similarly, if a `NullPointerException` occurs, the second `catch` block will handle it. If any other type of exception occurs, the generic `catch` block will handle it.

Using multiple `catch` blocks allows you to handle different exceptions gracefully and provide appropriate error messages or recovery mechanisms based on the specific type of exception encountered.

1.22. Q4c: What is Java Exception? Write a program that show the use of Arithmetic Exception in Java.

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exceptional condition arises, an object representing that condition is created and thrown in the method that caused the error. This object is an instance of a subclass of the `Throwable` class, which can be either an `Exception` or an `Error`.

An `ArithmeticException` is a subclass of `RuntimeException` and is thrown when an arithmetic operation fails due to certain conditions, such as division by zero or integer overflow.

Here's a program that demonstrates the use of `ArithmeticException` in Java:

```java
public class ArithmeticExceptionExample {
    public static void main(String[] args) {
        int dividend = 10;
        int divisor = 0;

        try {
            int quotient = dividend / divisor; // Division by zero will throw
ArithmeticException
            System.out.println("Quotient: " + quotient);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        }
    }
}
```

In this program:

- We have two integers, `dividend` and `divisor`, where `divisor` is initialized to 0.
- We attempt to perform a division operation (`dividend / divisor`), which will result in an `ArithmeticException` when `divisor` is 0.
- We have a `try-catch` block to handle the potential `ArithmeticException`. Inside the `try` block, the division operation is performed, and if an `ArithmeticException` occurs, it is caught by the `catch` block.
- Inside the `catch` block, we print a message indicating that an `ArithmeticException` was caught, along with the error message provided by the exception object (`e.getMessage()`).

When you run this program, it will output:

```
ArithmeticException caught: / by zero
```

This demonstrates how to use try-catch blocks to handle `ArithmeticException` and gracefully handle the error condition, preventing the program from crashing. It's important to handle exceptions appropriately in your code to ensure robustness and provide meaningful error messages to users.

1.23. Q5a: Explain ArrayIndexOutOfBound Exception in Java with example.

In Java, `ArrayIndexOutOfBoundsException` is a runtime exception that occurs when you try to access an element of an array at an index that is outside the valid range of indices for that array. This means you are trying to access an array element with an index that is either negative or greater than or equal to the length of the array.

Here's an example to illustrate `ArrayIndexOutOfBoundsException`:

```java
public class ArrayIndexOutOfBoundsExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};

        // Accessing an element at an invalid index
        System.out.println("Element at index 3: " + numbers[3]); // This will
throw ArrayIndexOutOfBoundsException
    }
}
```

In this example:

- We have an integer array `numbers` containing three elements: `1`, `2`, and `3`.
- We attempt to access the element at index `3` using `numbers[3]`.
- However, the valid indices for the array `numbers` are `0`, `1`, and `2`. Since we are trying to access an element at an index (`3`) that is beyond the valid range, it will result in an `ArrayIndexOutOfBoundsException` at runtime.

When you run this program, it will throw an `ArrayIndexOutOfBoundsException` with an error message similar to:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out
of bounds for length 3
    at
ArrayIndexOutOfBoundsExceptionExample.main(ArrayIndexOutOfBoundsExceptionExample.j
ava:7)
```

To prevent `ArrayIndexOutOfBoundsException`, you should always ensure that the index used to access an array element is within the valid range of indices (i.e., between `0` and `array.length - 1`). You can use conditional statements or loop constructs to check the validity of array indices before accessing elements to handle such exceptions gracefully in your code.

1.24. Q5b: Explain basics of Java stream classes.

In Java, stream classes are part of the Java I/O (Input/Output) API, which provides a way to efficiently read from and write to data sources and destinations, such as files, network connections, and memory buffers. Stream classes are used to handle input and output operations in Java programs, allowing data to be transferred between an application and external sources or sinks.

There are two main types of stream classes in Java:

1. **Byte Streams**:
    - Byte streams, represented by classes such as `InputStream` and `OutputStream`, are used for reading and writing raw bytes of data.
    - Byte streams are suitable for handling binary data or text data where character encoding is not a concern.
    - Examples of byte stream classes include `FileInputStream`, `FileOutputStream`, `BufferedInputStream`, `BufferedOutputStream`, etc.
2. **Character Streams**:

  - Character streams, represented by classes such as `Reader` and `Writer`, are used for reading and writing character data.
  - Character streams handle character encoding automatically, converting characters to and from bytes using the specified character encoding.
  - Character streams are suitable for reading and writing text data from/to external sources, ensuring proper character encoding and decoding.
  - Examples of character stream classes include `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, etc.

Basics of using Java stream classes:

- **Reading from Streams**: To read data from a stream, you typically create an appropriate input stream class object (e.g., `FileInputStream` or `BufferedReader`), and then use methods provided by the stream class to read data from the source. For example:

  ```
  BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
  String line = reader.readLine();
  ```

- **Writing to Streams**: To write data to a stream, you create an appropriate output stream class object (e.g., `FileOutputStream` or `BufferedWriter`), and then use methods provided by the stream class to write data to the destination. For example:

  ```
  BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));
  writer.write("Hello, World!");
  ```

- **Closing Streams**: It's important to close streams after using them to release system resources. You can use the `close()` method provided by stream classes to close the stream. Alternatively, you can use try-with-resources statement introduced in Java 7 to automatically close streams. For example:

  ```
  try (BufferedReader reader = new BufferedReader(new FileReader("file.txt")))
  {
      String line = reader.readLine();
      // Process the data
  } catch (IOException e) {
      // Handle exception
  }
  ```

Java stream classes provide a flexible and efficient way to perform input and output operations in Java programs, making it easy to interact with external data sources and sinks. Whether you're reading from files, network connections, or writing data to them, Java stream classes offer a consistent and convenient API for handling I/O operations.

1.25. Q5c: Write a java program to create a text file and perform read operation on the text file.

To create a text file and perform a read operation using `FileInputStream` in Java, you need to use `FileOutputStream` for writing to the file since `FileInputStream` is designed for reading bytes from a file. Below is a Java program that demonstrates how to create a text file using `FileOutputStream` and then reads it back using `FileInputStream`.

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileStreamExample {
    public static void main(String[] args) {
        String fileName = "sample.txt";
        String content = "Hello, World!\nThis is a sample text file.";

        // Write content to file
        try (FileOutputStream fos = new FileOutputStream(fileName)) {
            fos.write(content.getBytes());
            System.out.println("File has been written successfully.");
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }

        // Read content from file
        try (FileInputStream fis = new FileInputStream(fileName)) {
            int i;
            System.out.println("Reading from file: ");
            while ((i = fis.read()) != -1) {
                // i is a byte. Convert it to char and print it
                System.out.print((char) i);
            }
        } catch (IOException e) {
            System.err.println("Error reading from file: " + e.getMessage());
        }
    }
}
```

In this program:

1. The `main` method defines a `fileName` for the file to be created and a `String` named `content` that holds the text to be written to the file.
2. It uses a `FileOutputStream` to write the text content to the file. The `String` content is converted to bytes using the `getBytes()` method before writing, as `FileOutputStream` works with bytes.
3. After writing the content to the file, it uses a `FileInputStream` to read the bytes from the file. It reads the file byte by byte in the `while` loop until `read()` returns `-1`, indicating the end of the file.
4. Each byte read from the file is cast to a `char` and printed to the console, allowing the text content of the file to be displayed.

This program demonstrates the basic use of `FileInputStream` and `FileOutputStream` for reading and writing text files, though it's worth noting that these classes are primarily intended for binary data. For reading and writing character data, consider using `FileReader` and `FileWriter` or `BufferedReader` and `BufferedWriter` for efficiency and simplicity.

1.26. Q5a: Explain Divide by Zero Exception in Java with example.

In Java, a `DivideByZeroException` is not explicitly provided as a standard exception class. Instead, the exception that occurs when you attempt to divide by zero is called `ArithmeticException`. This exception is thrown when an arithmetic operation fails due to certain conditions, such as division by zero.

Here's an example to illustrate `ArithmeticException` (which commonly occurs due to divide by zero):

```java
public class DivideByZeroExceptionExample {
    public static void main(String[] args) {
        int dividend = 10;
        int divisor = 0;

        try {
            int quotient = dividend / divisor; // Division by zero will throw
ArithmeticException
            System.out.println("Quotient: " + quotient);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        }
    }
}
```

In this example:

- We have two integers, `dividend` and `divisor`, where `divisor` is initialized to `0`.
- We attempt to perform a division operation (`dividend / divisor`), which will result in an `ArithmeticException` when `divisor` is `0`.
- We have a `try-catch` block to handle the potential `ArithmeticException`. Inside the `try` block, the division operation is performed, and if an `ArithmeticException` occurs, it is caught by the `catch` block.
- Inside the `catch` block, we print a message indicating that an `ArithmeticException` was caught, along with the error message provided by the exception object (`e.getMessage()`).

When you run this program, it will output:

```
ArithmeticException caught: / by zero
```

This demonstrates how attempting to divide by zero results in an `ArithmeticException` being thrown at runtime in Java. To prevent such exceptions, it's important to ensure that you handle cases where division by zero may occur or validate input data to avoid such scenarios.

1.27. Q5b: Explain java I/O process.

In Java, Input/Output (I/O) operations involve the exchange of data between a Java program and external sources or destinations, such as files, network connections, or other programs. The Java I/O process encompasses several key concepts and classes provided by the Java API to facilitate reading from and writing to various data sources and sinks.

The Java I/O process typically involves the following steps:

1. **Selecting a Data Source or Destination**:
    - Determine the source or destination of the data you want to read from or write to. This could be a file, network socket, standard input/output streams (e.g., `System.in` and `System.out`), or any other data stream.
2. **Creating Stream Objects**:
    - Once you've identified the source or destination, you need to create appropriate stream objects to interact with it.
    - For reading data, you typically use input stream classes such as `InputStream` or `Reader`.
    - For writing data, you typically use output stream classes such as `OutputStream` or `Writer`.
    - Stream classes provide methods for reading/writing data in the form of bytes or characters, depending on the type of data source or destination.
3. **Reading from or Writing to Streams**:
    - Use the methods provided by the stream classes to read data from or write data to the associated data source or destination.
    - For example, you can use methods like `read()` or `write()` to read/write bytes, or `readLine()` or `writeLine()` to read/write characters.
4. **Closing Streams**:
    - After you've finished reading from or writing to streams, it's important to close them to release system resources and ensure proper cleanup.
    - You can use the `close()` method provided by stream classes to close the streams.
    - Alternatively, you can use the try-with-resources statement introduced in Java 7 to automatically close streams when they are no longer needed.
5. **Handling Exceptions**:
    - I/O operations can throw exceptions due to various reasons, such as file not found, network errors, or invalid data formats.
    - It's essential to handle these exceptions gracefully using try-catch blocks or propagate them to the calling code for proper error handling and recovery.
6. **Optional: Buffering and Efficiency**:
    - To improve performance and efficiency, you can use buffered stream classes such as `BufferedReader`, `BufferedWriter`, `BufferedInputStream`, or `BufferedOutputStream`.
    - Buffered stream classes reduce the number of actual I/O operations by reading/writing data in larger chunks, resulting in improved performance.

Overall, the Java I/O process involves selecting the appropriate stream classes, reading from or writing to streams, closing streams after use, handling exceptions, and optionally using buffering for improved efficiency. Understanding these concepts and using the provided Java I/O classes effectively is crucial for performing input/output operations in Java programs.

1.28. Q5c: Write a java program to display the content of a text file and perform append operation on the text file.

Below is a Java program that displays the content of a text file and performs an append operation on the text file using `FileInputStream` and `FileOutputStream`:

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileDisplayAndAppend {
    public static void main(String[] args) {
        String fileName = "sample.txt";

        // Display the content of the text file
        displayFileContent(fileName);

        // Perform append operation on the text file
        performAppendOperation(fileName);
    }

    // Method to display the content of the text file
    private static void displayFileContent(String fileName) {
        try (FileInputStream fis = new FileInputStream(fileName)) {
            int i;
            System.out.println("Contents of the text file:");
            while ((i = fis.read()) != -1) {
                System.out.print((char) i);
            }
            System.out.println("\n");
        } catch (IOException e) {
            System.err.println("Error reading from file: " + e.getMessage());
        }
    }

    // Method to perform append operation on the text file
    private static void performAppendOperation(String fileName) {
        String appendContent = "\nThis line is appended to the file.";

        try (FileOutputStream fos = new FileOutputStream(fileName, true)) {
            fos.write(appendContent.getBytes());
            System.out.println("Append operation completed successfully.");
        } catch (IOException e) {
            System.err.println("Error appending to file: " + e.getMessage());
        }
    }
}
```

In this program:

1. The `displayFileContent()` method reads and displays the content of the specified text file using `FileInputStream`.

2. The `performAppendOperation()` method appends a new line of content to the end of the text file using `FileOutputStream` with the `append` parameter set to `true`.
3. In the `main()` method, both methods are called sequentially to display the initial content of the file and then perform the append operation.
4. The content to be appended (`appendContent`) is specified as a `String` and converted to bytes using the `getBytes()` method before writing to the file.

When you run this program, it will display the initial content of the text file (if it exists) and then append a new line of content to the file. Make sure to replace `"sample.txt"` with the actual file name you want to read from and append to.

2. 4341602 - Java: Summer 2023 Paper Solution

2.1. Q1a: Differentiate between Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP).

Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP) are two distinct paradigms in software development. Here's a differentiation between the two:

1. **Fundamental Unit**:
   - POP: In POP, the fundamental unit of the program is a function or a procedure, which operates on data.
   - OOP: In OOP, the fundamental unit is an object, which combines data (attributes) and behaviors (methods) into a single entity.
2. **Data and Functionality**:
   - POP: Data and functionality are separate entities. Functions operate on data that is often stored in data structures.
   - OOP: Data and functionality are bundled together within objects. Objects encapsulate both data (attributes) and functionality (methods) related to that data.
3. **Data Encapsulation**:
   - POP: Encapsulation is not a primary concern. Data can be accessed and modified by any function that has access to it.
   - OOP: Encapsulation is a key principle. Data within objects is typically hidden from external access, and can only be manipulated through defined methods, providing better control and security.
4. **Inheritance**:
   - POP: Inheritance is not directly supported.
   - OOP: Inheritance allows objects to inherit attributes and methods from parent classes, promoting code reusability and establishing hierarchical relationships.
5. **Polymorphism**:
   - POP: Polymorphism is achieved through function overloading and procedure overriding.
   - OOP: Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and extensibility in code design.
6. **Modifiability and Scalability**:
   - POP: Modifying and scaling code can be more challenging as the program grows larger, due to the lack of modularization inherent in the procedural approach.
   - OOP: OOP promotes modularity and scalability through the use of classes and objects, making it easier to manage and extend code as requirements change.
7. **Example Languages**:
   - POP: Languages like C, Fortran, and Pascal primarily follow the procedural paradigm.

- OOP: Languages like Java, Python, and C++ are designed with OOP principles in mind, although many also support procedural programming.

In summary, while both paradigms aim to organize code and facilitate software development, they differ significantly in their approach to data organization, code structure, and principles of modularity and reusability.

2.2. Q1b: Explain Super keyword in inheritance with suitable example.

In Java, the `super` keyword is used to refer to the superclass (parent class) of a subclass (child class). It can be used to access superclass methods, constructor, and instance variables. This is particularly useful when the subclass overrides a method or hides an instance variable of the superclass and you want to access the superclass version.

Let's illustrate the usage of the `super` keyword with an example involving inheritance and method overriding:

```java
// Parent class
class Animal {
    String color = "White";

    void display() {
        System.out.println("Animal is " + color);
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    String color = "Black"; // hiding the color variable in parent class

    void display() {
        System.out.println("Dog is " + color);
        System.out.println("Superclass Animal is " + super.color); // accessing
superclass variable
        super.display(); // calling superclass method
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.display();
    }
}
```

In this example:

- The `Animal` class defines a variable `color` and a method `display()` which prints the color.
- The `Dog` class extends `Animal` and defines its own `color` variable, hiding the `color` variable of the superclass. It also overrides the `display()` method to print the dog's color and then calls `super.display()` to call the superclass's `display()` method.

- In the `main()` method, we create an instance of `Dog` and call its `display()` method.

Output:

```
Dog is Black
Superclass Animal is White
Animal is White
```

Here's what's happening:

- The `display()` method in the `Dog` class prints the color of the dog, then it uses `super.color` to access the `color` variable of the superclass (which is "White").
- `super.display()` invokes the `display()` method of the superclass, printing "Animal is White".

This demonstrates how `super` can be used to access superclass members from a subclass, allowing for controlled access to overridden methods and hidden variables.

2.3. Q1c: Define: Method Overriding. List out Rules for method overriding. Write a java program that implements method overriding.

Method overriding is a feature in object-oriented programming that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This means that a subclass can redefine the implementation of a method that it inherits from its superclass according to its own requirements.

Rules for method overriding in Java:

1. **Method Signature**:
    - The method in the subclass must have the same name, return type, and parameter list (including order and type of parameters) as the method in the superclass. Changing the return type or parameter list results in method overloading instead of overriding.
2. **Access Modifier**:
    - The access modifier of the overriding method in the subclass should not be more restrictive than the access modifier of the overridden method in the superclass. However, it can be less restrictive or the same.
    - Access levels in Java: `public`, `protected`, package-private (default), and `private`.
    - The order of access modifiers from least restrictive to most restrictive is: `public`, `protected`, package-private, and `private`.
3. **Exception Handling**:
    - The subclass method can only throw exceptions that are subclasses of the exceptions thrown by the superclass method, or it can choose not to throw any exceptions (this is also known as "covariant return types").
4. **Return Type**:
    - If the return type of the method in the subclass is a subclass of the return type of the method in the superclass, it's considered a valid overriding (covariant return types).

- In Java 5 and later versions, covariant return types allow the return type of the overriding method to be a subclass of the return type of the overridden method.

5. **Method Visibility**:
   - If a method in the superclass is declared as `final`, it cannot be overridden in any subclass.
   - If a method in the superclass is declared as `static`, it cannot be overridden because static methods belong to the class, not to the instance.
   - Constructors and private methods cannot be overridden because they are not inherited by subclasses.
6. **Super Keyword**:
   - Within the overriding method, you can use the `super` keyword to call the overridden method from the superclass.
   - This can be useful for extending the functionality of the superclass method while still utilizing its original implementation.

Method overriding allows for polymorphism in Java, enabling different behavior for objects of the same superclass type based on their actual runtime types.

Sure, here's a Java program that demonstrates method overriding:

```java
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Generic animal sound");
    }
}

// Subclass inheriting from Animal
class Dog extends Animal {
    // Override makeSound method
    @Override
    void makeSound() {
        System.out.println("Woof!");
    }
}

// Another subclass inheriting from Animal
class Cat extends Animal {
    // Override makeSound method
    @Override
    void makeSound() {
        System.out.println("Meow!");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog(); // Animal reference, Dog object
        Animal animal2 = new Cat(); // Animal reference, Cat object

        animal1.makeSound(); // Calls Dog's makeSound method
        animal2.makeSound(); // Calls Cat's makeSound method
    }
}
```

Output:

```
Woof!
Meow!
```

Explanation:

- We have a superclass `Animal` with a method `makeSound()`.
- The `Dog` class and `Cat` class both extend `Animal` and override the `makeSound()` method with their own implementations.
- In the `Main` class, we create instances of `Dog` and `Cat` but store them in `Animal` references.
- When we call the `makeSound()` method on these instances, Java dynamically dispatches the call to the appropriate overridden method based on the actual type of the object at runtime, demonstrating polymorphism through method overriding.

2.4. Q1cOR: Describe: Interface. Write a java program using interface to demonstrate multiple inheritance.

In Java, an interface is a reference type that defines a set of abstract methods along with constants (static final variables). Interfaces cannot have instance fields (non-static variables) or concrete methods (methods with a body) until Java 8, where default and static methods were introduced in interfaces.

Interfaces serve as a contract or blueprint for classes, specifying methods that implementing classes must provide. They facilitate abstraction, allowing for the separation of specification and implementation in software design. Here are key features and characteristics of interfaces in Java:

1. **Declaration**:
    - Interfaces are declared using the `interface` keyword.
    - Example: `interface MyInterface { ... }`
2. **Abstract Methods**:
    - An interface can contain abstract methods, which are method declarations without a body.
    - All methods in an interface are implicitly `public` and `abstract`.
    - Example:

    ```
    interface MyInterface {
        void method1();
        int method2();
    }
    ```
3. **Constants**:
    - Interfaces can declare constants, which are implicitly `public`, `static`, and `final`.

- Constants are typically used to define immutable values that are relevant to the interface.
- Example:

```java
interface MyInterface {
    int CONSTANT_VALUE = 10;
}
```

4. **Default Methods (Java 8+)**:
   - Java 8 introduced the concept of default methods in interfaces, allowing interfaces to have concrete methods with a default implementation.
   - Default methods are declared using the `default` keyword and can be overridden by implementing classes if needed.
   - Default methods were introduced to provide backward compatibility when introducing new methods to existing interfaces.
   - Example:

```java
interface MyInterface {
    default void defaultMethod() {
        System.out.println("Default method implementation");
    }
}
```

5. **Static Methods (Java 8+)**:
   - Java 8 also introduced static methods in interfaces, allowing interfaces to contain static utility methods.
   - Static methods are declared using the `static` keyword and can be invoked using the interface name.
   - Example:

```java
interface MyInterface {
    static void staticMethod() {
        System.out.println("Static method implementation");
    }
}
```

6. **Multiple Inheritance**:
   - Java allows interfaces to support multiple inheritance, meaning a class can implement multiple interfaces.
   - This enables a class to inherit behavior from multiple sources, promoting code reuse and flexibility.
   - Example:

```java
interface Interface1 {
    void method1();
}

interface Interface2 {
    void method2();
}

class MyClass implements Interface1, Interface2 {
    public void method1() {
        // Implementation
    }

    public void method2() {
        // Implementation
    }
}
```

7. **Implementation**:
   - Classes implement interfaces using the `implements` keyword.
   - Implementing classes must provide concrete implementations for all abstract methods declared in the interface.
   - Example:

```java
class MyClass implements MyInterface {
    public void method1() {
        // Implementation
    }

    public int method2() {
        // Implementation
    }
}
```

Interfaces play a crucial role in Java's abstraction mechanisms, enabling the definition of contracts and facilitating polymorphism and code reusability in object-oriented programming. They are widely used in Java APIs and frameworks to define specifications and promote interoperability between different components.

In Java, multiple inheritance is not directly supported for classes, meaning a class cannot extend multiple classes simultaneously. However, Java provides a way to achieve a form of multiple inheritance using interfaces. An interface in Java defines a contract for classes that implement it, specifying a set of methods that must be implemented by any class that claims to conform to the interface.

Here's a Java program demonstrating multiple inheritance using interfaces:

```java
// Interface 1
interface Animal {
    void eat();
}

// Interface 2
interface Mammal {
    void run();
}

// Class implementing Interface 1
class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats bones");
    }
}

// Class implementing Interface 2
class Horse implements Mammal {
    @Override
    public void run() {
        System.out.println("Horse runs at high speed");
    }
}

// Class implementing both Interface 1 and Interface 2
class DogHorseHybrid implements Animal, Mammal {
    @Override
    public void eat() {
        System.out.println("Dog-Horse Hybrid eats bones and hay");
    }

    @Override
    public void run() {
        System.out.println("Dog-Horse Hybrid runs");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Horse horse = new Horse();
        DogHorseHybrid hybrid = new DogHorseHybrid();

        dog.eat();
        horse.run();
        hybrid.eat();
        hybrid.run();
    }
}
```

Output:

```
Dog eats bones
Horse runs at high speed
Dog-Horse Hybrid eats bones and hay
Dog-Horse Hybrid runs
```

Explanation:

- We define two interfaces: `Animal` and `Mammal`, each with their own set of methods.
- We define two classes: `Dog` and `Horse`, each implementing one of the interfaces.
- We define a class `DogHorseHybrid` that implements both interfaces, thereby inheriting behavior from both `Animal` and `Mammal`.

- In the `Main` class, we create instances of `Dog`, `Horse`, and `DogHorseHybrid`, and call their respective methods to demonstrate multiple inheritance through interfaces.

2.5. Q2a: Explain the Java Program Structure with example.

In Java, a program is typically structured into classes, which are the fundamental building blocks of Java applications. Each class encapsulates data (attributes) and behaviors (methods) related to a specific entity or concept. The overall structure of a Java program involves one or more classes, with one class containing a special method called `main()` where the program execution begins.

Here's an example of a simple Java program structure:

```java
// Main class
public class HelloWorld {
    // Main method where the program execution begins
    public static void main(String[] args) {
        // Program logic
        System.out.println("Hello, world!");
    }
}
```

Let's break down the structure of this Java program:

1. **Class Declaration**:
    - The program starts with the declaration of a class using the `class` keyword. In this example, the class is named `HelloWorld`.
    - Class names in Java must start with an uppercase letter and follow camel case convention.
2. **Main Method**:
    - Inside the class, we define a special method called `main()`. This is the entry point of the program where the execution begins.
    - The `main()` method must be declared as `public`, `static`, and `void`.
    - It accepts a single parameter, an array of strings (`String[] args`), which allows command-line arguments to be passed to the program.
3. **Program Logic**:
    - Inside the `main()` method, we write the logic or instructions that we want the program to execute.
    - In this example, we have a single statement that prints "Hello, world!" to the console using the `System.out.println()` method.
4. **Comments**:
    - Comments in Java start with `//` for single-line comments or `/* */` for multi-line comments.
    - Comments are used to document and explain the code, making it more readable and understandable.
5. **Semicolons**:
    - Java statements are terminated by semicolons (`;`). They indicate the end of a statement.

Overall, this Java program structure demonstrates the basic elements required for a Java program: a class declaration, a main method, and program logic. This structure forms the foundation for writing Java applications of varying complexity.

2.6. Q2b: Explain static keyword with suitable example.

In Java, the `static` keyword is used to declare members (variables and methods) that belong to the class itself rather than to instances of the class. This means that `static` members are shared among all instances of the class and can be accessed directly through the class name without the need to create an object of the class.

Here's an explanation of the `static` keyword with a suitable example:

```java
class Counter {
    static int count = 0; // Static variable

    // Static method to increment the count
    static void increment() {
        count++;
    }

    // Static method to display the count
    static void displayCount() {
        System.out.println("Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        // Accessing static variable and method using class name
        Counter.increment();
        Counter.displayCount();

        // Creating multiple instances of Counter
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        // Accessing static variable and method using instances
        c1.increment();
        c2.increment();
        Counter.displayCount(); // Output: Count: 3
    }
}
```

Explanation:

- In the `Counter` class, `count` is declared as a static variable. This means that all instances of the `Counter` class share the same `count` variable.
- `increment()` and `displayCount()` are static methods. These methods can be called directly using the class name (`Counter.increment()`, `Counter.displayCount()`), without needing to create an object of the class.

- In the `Main` class, we demonstrate accessing and modifying the static variable and calling static methods both through the class name and through instances of the class.
- The output demonstrates that the static variable `count` is shared among all instances of the `Counter` class. When we increment `count` using one instance, it reflects the change when accessed through another instance or the class name itself.

In summary, the `static` keyword allows for the creation of class-level variables and methods that are shared among all instances of the class. It provides a way to manage and manipulate shared data and behavior within the context of a class.

2.7. Q2c: Define: Constructor. List out types of it. Explain Parameterized and copy constructor with suitable example.

A constructor in Java is a special type of method that is automatically called when an object of a class is created. It is used to initialize the newly created object. Constructors have the same name as the class and do not have a return type, not even `void`. Constructors can be used to set initial values for instance variables, allocate resources, or perform any other initialization tasks needed by the object.

Types of constructors in Java:

1. **Default Constructor**:
   - A default constructor is automatically created by Java if no other constructor is defined explicitly.
   - It has no parameters and typically initializes instance variables to their default values (e.g., `0` for numeric types, `null` for reference types).
2. **Parameterized Constructor**:
   - A parameterized constructor accepts parameters which are used to initialize instance variables with specific values.
   - It allows for custom initialization of objects based on the provided arguments.
3. **Copy Constructor**:
   - A copy constructor is a special type of constructor that takes an object of the same class as a parameter and creates a new object by copying the values of the instance variables from the passed object.
   - It is used to create a new object with the same state as an existing object.

Let's explain parameterized and copy constructors with suitable examples:

2.7.1. Parameterized Constructor Example:

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

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

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

public class Main {
    public static void main(String[] args) {
        // Creating objects using parameterized constructor
        Student student1 = new Student("Alice", 20);
        Student student2 = new Student("Bob", 22);

        // Displaying student details
        student1.display();
        student2.display();
    }
}
```

In this example:

- We define a `Student` class with instance variables `name` and `age`.
- The `Student` class has a parameterized constructor that initializes the `name` and `age` instance variables with the values passed as arguments.
- We create two `Student` objects (`student1` and `student2`) using the parameterized constructor and display their details.

2.7.2. Copy Constructor Example:

```java
class Employee {
    String name;
    int age;

    // Copy Constructor
    public Employee(Employee emp) {
        this.name = emp.name;
        this.age = emp.age;
    }

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

public class Main {
    public static void main(String[] args) {
        // Creating an object
        Employee emp1 = new Employee();
        emp1.name = "John";
        emp1.age = 30;

        // Creating another object using copy constructor
        Employee emp2 = new Employee(emp1);

        // Displaying employee details
        emp1.display();
        emp2.display();
    }
}
```

In this example:

- We define an `Employee` class with instance variables `name` and `age`.
- The `Employee` class has a copy constructor that takes an `Employee` object as a parameter and initializes the instance variables of the new object with the values from the passed object.
- We create an `Employee` object `emp1`, set its `name` and `age`, and then create another `Employee` object `emp2` using the copy constructor with `emp1` as an argument.
- Both `emp1` and `emp2` have the same state, demonstrating the use of the copy constructor to create a new object with the same state as an existing object.

2.8. Q2a: Explain the Primitive Data Types and User Defined DataTypes in java.

In Java, data types specify the type of data that a variable can hold. There are two main categories of data types: primitive data types and user-defined data types.

2.8.1. Primitive Data Types:

Primitive data types are the basic building blocks of data manipulation in Java. They are predefined by the language and represent simple values. Java provides eight primitive data types:

1. **byte**: 8-bit signed integer.
2. **short**: 16-bit signed integer.
3. **int**: 32-bit signed integer.
4. **long**: 64-bit signed integer.
5. **float**: 32-bit floating-point number.
6. **double**: 64-bit floating-point number.
7. **char**: 16-bit Unicode character.
8. **boolean**: Represents true or false.

Example:

```
int number = 10;
double pi = 3.14;
char letter = 'A';
boolean isJavaFun = true;
```

2.8.2. User-Defined Data Types:

User-defined data types are created by the programmer to meet specific requirements. They are derived from primitive data types and/or other user-defined data types. In Java, user-defined data types include classes, interfaces, arrays, and enumerated types.

1. **Classes**: Classes are user-defined data types that encapsulate data for a specific object and provide methods to operate on that data.

```
class Car {
    String brand;
    String model;
    int year;
}
```

2. **Interfaces**: Interfaces define a contract for classes that implement them, specifying a set of methods that must be implemented.

```
interface Shape {
    double area();
    double perimeter();
}
```

3. **Arrays**: Arrays are collections of elements of the same type that are stored in contiguous memory locations.

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

4. **Enumerated Types (Enums)**: Enums define a set of named constants representing a fixed set of values.

```
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

User-defined data types allow programmers to organize and manipulate complex data structures and represent real-world entities in their programs. They contribute to the modularity, maintainability, and extensibility of Java code.

2.9. Q2b: Explain this keyword with suitable example.

In Java, the `this` keyword is a reference to the current object within a method or constructor. It can be used to access instance variables and methods of the current object, differentiate between instance variables and local variables with the same name, and to pass the current object as a parameter to other methods.

Here's an explanation of the `this` keyword with a suitable example:

```
class Student {
    String name;
    int age;
    // Parameterized Constructor
    public Student(String name, int age) {
        // Use 'this' to distinguish between instance variables and constructor
parameters
        this.name = name;
        this.age = age;
    }

    // Method to display student details
    void display() {
        // Access instance variables using 'this'
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
    }

    // Method to compare two Student objects
    public boolean isOlder(Student otherStudent) {
        // Use 'this' to refer to the current object
        return this.age > otherStudent.age;
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a Student object
        Student student1 = new Student("Alice", 20);

        // Call display method
        student1.display();

        // Create another Student object
        Student student2 = new Student("Bob", 22);

        // Compare ages using isOlder method
        if (student1.isOlder(student2)) {
            System.out.println(student1.name + " is older than " + student2.name);
        } else {
            System.out.println(student2.name + " is older than " + student1.name);
        }
    }
}
```

Explanation:

- In the `Student` class constructor, `this.name` and `this.age` are used to refer to the instance variables of the current object (`Student`).
- In the `display()` method, `this.name` and `this.age` are used to access the instance variables of the current object.
- In the `isOlder()` method, `this.age` is used to access the age of the current object (`this`) and compare it with the age of another `Student` object passed as a parameter.
- In the `Main` class, we create two `Student` objects (`student1` and `student2`) and call methods using the `this` keyword to demonstrate its usage.

2.10. Q2c: Define Inheritance. List out types of it. Explain multilevel and hierarchical inheritance with suitable example.

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit attributes and behaviors from an existing class (superclass or base class). This enables code reuse and promotes the creation of a hierarchy of classes, where classes at higher levels in the hierarchy share common characteristics, and subclasses can specialize or extend those characteristics.

Types of Inheritance:

1. **Single Inheritance**:
    - A subclass inherits from only one superclass.
2. **Multiple Inheritance**:
    - A subclass inherits from more than one superclass. This is not directly supported in Java due to the potential ambiguity and complexity it introduces.
3. **Multilevel Inheritance**:
    - A subclass inherits from a superclass, and another subclass inherits from the first subclass, forming a chain of inheritance.
4. **Hierarchical Inheritance**:
    - Multiple subclasses inherit from a single superclass, forming a tree-like structure.

2.10.1. Multilevel Inheritance Example:

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

// Subclass inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Subclass inheriting from Dog
class Labrador extends Dog {
    void color() {
        System.out.println("Labrador is brown");
    }
}

public class Main {
    public static void main(String[] args) {
        Labrador labrador = new Labrador();
        labrador.eat(); // inherited from Animal
        labrador.bark(); // inherited from Dog
        labrador.color(); // own method
    }
}
```

Explanation:

- In this example, `Animal` is the superclass, `Dog` is a subclass inheriting from `Animal`, and `Labrador` is a subclass inheriting from `Dog`.
- `Dog` inherits the `eat()` method from `Animal` and adds its own method `bark()`.
- `Labrador` inherits both `eat()` and `bark()` methods from `Dog` and adds its own method `color()`.
- The `main()` method demonstrates calling methods from different levels of the inheritance hierarchy using an object of the `Labrador` class.

2.10.2. Hierarchical Inheritance Example:

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

// Subclass 1 inheriting from Animal
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Subclass 2 inheriting from Animal
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // inherited from Animal
        dog.bark(); // own method

        Cat cat = new Cat();
        cat.eat(); // inherited from Animal
        cat.meow(); // own method
    }
}
```

Explanation:

- In this example, both `Dog` and `Cat` classes inherit the `eat()` method from the `Animal` superclass.

- `Dog` adds its own method `bark()`, while `Cat` adds its own method `meow()`.
- The `main()` method demonstrates creating objects of both `Dog` and `Cat` classes and calling their respective methods.

2.11. Q3a: Explain Type Conversion and Casting in java.

In Java, type conversion refers to the process of converting one data type into another. This can occur implicitly, where the conversion is done automatically by the compiler, or explicitly, where the programmer explicitly specifies the conversion using casting.

2.11.1. Implicit Type Conversion (Widening Conversion):

- Implicit type conversion occurs when a data type with a smaller range or precision is converted into a data type with a larger range or precision.
- This conversion is performed by the compiler automatically and does not require any explicit casting.
- It's also known as widening conversion because the range of the data type is widened.
- For example, converting an integer to a floating-point number.

Example:

```java
int numInt = 10;
double numDouble = numInt; // Implicit conversion from int to double
```

2.11.2. Explicit Type Conversion (Narrowing Conversion):

- Explicit type conversion, also known as casting, occurs when a data type with a larger range or precision is converted into a data type with a smaller range or precision.
- Casting requires explicit syntax where the programmer specifies the desired type in parentheses before the value to be converted.
- This conversion may result in loss of data if the target type cannot represent the entire range of the source type.
- It's also known as narrowing conversion because the range of the data type is narrowed.
- For example, converting a floating-point number to an integer.

Example:

```java
double numDouble = 10.5;
int numInt = (int) numDouble; // Explicit conversion (casting) from double to int
```

2.11.3. Type Casting:

- Type casting is the process of converting a variable from one data type to another.
- It's done by explicitly specifying the target data type in parentheses before the variable.
- There are two types of casting: primitive type casting and object casting.
- Primitive type casting is used for converting between primitive data types, while object casting is used for converting between reference types.

Example of Primitive Type Casting:

```java
double numDouble = 10.5;
int numInt = (int) numDouble; // Primitive type casting from double to int
```

Example of Object Casting:

```java
class Animal {}
class Dog extends Animal {}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        Dog dog = (Dog) animal; // Object casting from Animal to Dog
    }
}
```

In summary, type conversion in Java involves converting one data type to another either implicitly or explicitly through casting. Implicit conversion occurs automatically by the compiler, while explicit conversion requires the programmer to specify the desired type using casting syntax.

2.12. Q3b: Explain different visibility controls used in Java.

In Java, visibility controls, also known as access modifiers, are keywords that determine the accessibility or visibility of classes, methods, and variables within Java programs. They specify the level of access that other classes or components have to the members of a class. Java provides four visibility controls:

1. **public**:
   - Members marked as `public` are accessible from any other class.
   - They can be accessed by classes in the same package as well as by classes in different packages.
   - Public members form the interface of the class, providing access to its functionality.
2. **protected**:
   - Members marked as `protected` are accessible within the same package and by subclasses (even if they are in a different package).
   - Protected members are useful when you want to provide access to subclasses while still restricting access from other classes.
3. **default (no modifier)**:
   - If no access modifier is specified, the default visibility is applied.
   - Members with default visibility are accessible only within the same package.
   - They are not accessible by classes outside the package, even if they are subclasses.
4. **private**:
   - Members marked as `private` are accessible only within the same class.
   - They are not visible to any other class, including subclasses and classes in the same package.

- Private members are used to encapsulate the internal state of a class and hide implementation details.

By using these visibility controls, you can control the access to your classes, methods, and variables, which helps in enforcing encapsulation, promoting code maintainability, and reducing coupling between classes.

Example:

```java
package com.example;

public class MyClass {
    public int publicVar;
    protected int protectedVar;
    int defaultVar; // Default visibility
    private int privateVar;
}
```

In this example:

- `publicVar` is accessible from any class, regardless of its location.
- `protectedVar` is accessible within the same package and by subclasses.
- `defaultVar` is accessible only within the same package.
- `privateVar` is accessible only within the same class.

2.13. Q3c: Define: Thread. List different methods used to create Thread. Explain Thread life cycle in detail.

2.13.1. Definition of Thread:

In Java, a thread refers to a single sequential flow of control within a program. It is the smallest unit of execution and represents an independent path of execution in a program. Multiple threads can run concurrently within a single Java program, allowing for parallel execution of tasks.

2.13.2. Methods to Create Thread:

In Java, there are several ways to create a thread:

1. **Extending the Thread Class**:
    - Create a new class that extends the `Thread` class.
    - Override the `run()` method to specify the task to be performed by the thread.
    - Create an instance of the subclass and call its `start()` method to start the execution of the thread.
2. **Implementing the Runnable Interface**:
    - Create a class that implements the `Runnable` interface.
    - Implement the `run()` method to specify the task to be performed by the thread.
    - Create an instance of the class and pass it as a parameter to a `Thread` object.
    - Call the `start()` method of the `Thread` object to start the execution of the thread.

3. **Using Lambda Expressions** (Java 8 and later):
    - Define the task to be performed by the thread using a lambda expression.
    - Create a `Thread` object and pass the lambda expression as a parameter to its constructor.
    - Call the `start()` method of the `Thread` object to start the execution of the thread.

2.13.3. Thread Life Cycle:

The life cycle of a thread in Java consists of several states, and the thread can transition between these states during its execution. The states of a thread in Java are as follows:

1. **New**:
    - The thread is in the new state if it has been created but has not yet started.
    - This state is characterized by the creation of a `Thread` object using the `new` keyword.
2. **Runnable**:
    - The thread is in the runnable state if it is ready to run but the scheduler has not yet selected it to be the running thread.
    - A runnable thread may be executing or waiting for its turn to be executed by the scheduler.
3. **Running**:
    - The thread is in the running state if it has been selected by the scheduler for execution.
    - In this state, the thread is actively executing its task.
4. **Blocked/Waiting**:
    - The thread is in the blocked or waiting state if it is waiting for a specific condition to occur or for another thread to release a lock.
    - A blocked thread cannot proceed until the condition is satisfied or the lock is released.
5. **Timed Waiting**:
    - The thread is in the timed waiting state if it is waiting for a specified period of time.
    - This state occurs when a thread calls a method that results in it waiting for a specified amount of time.
6. **Terminated**:
    - The thread is in the terminated state if it has completed its task or if it has been explicitly terminated by calling the `stop()` method.

2.13.4. Detailed Explanation of Thread Life Cycle:

1. **New**:
    - The thread is created using the `new` keyword, but the `start()` method has not yet been called.
2. **Runnable**:
    - The `start()` method is called, and the thread becomes ready to run.
    - The thread may be selected by the scheduler to run, or it may wait for its turn if other threads are currently running.
3. **Running**:

- The scheduler selects the thread to run, and it begins executing its task.
- In this state, the thread is actively executing its code.

4. **Blocked/Waiting**:
    - The thread may enter the blocked or waiting state if it encounters a blocking operation, such as waiting for I/O or waiting for a lock to be released.
    - While in this state, the thread is not executing, but it is not terminated either.
5. **Timed Waiting**:
    - Similar to the blocked or waiting state, but the thread waits for a specified period of time before resuming execution.
6. **Terminated**:
    - The thread completes its task or is explicitly terminated by calling the `stop()` method.
    - Once terminated, the thread cannot be restarted and its resources are released.

Understanding the life cycle of a thread is important for proper thread management and synchronization in Java programs. It allows developers to control the execution of threads and handle concurrency-related issues effectively.

2.14. Q3a: Explain the purpose of JVM in java.

The Java Virtual Machine (JVM) is a critical component of the Java Runtime Environment (JRE), serving as the engine that executes Java bytecode. It is the cornerstone of Java's "write once, run anywhere" (WORA) philosophy, allowing Java applications to run on any device or operating system that has a compatible JVM. The purpose and functionalities of the JVM are multifaceted:

2.14.1. Platform Independence:

- **Code Portability**: JVM enables Java applications to be platform-independent. Java programs are compiled into bytecode, which can be executed on any JVM, regardless of the underlying hardware and operating system. This means developers can write the code once and run it anywhere, without needing to modify it for different platforms.

2.14.2. Security:

- **Safe Execution Environment**: The JVM provides a secure execution environment by sandboxing the execution of bytecode. It enforces access controls and provides various security checks, preventing unauthorized access to system resources and ensuring that Java applications cannot harm the host system.
- **Bytecode Verification**: Before executing bytecode, the JVM verifies the code to ensure it adheres to Java's safety rules, further enhancing security.

2.14.3. Performance:

- **Just-In-Time (JIT) Compilation**: While the JVM interprets bytecode, it also employs Just-In-Time (JIT) compilation to improve the performance of Java applications. The JIT compiler translates bytecode into native machine code just before execution, which allows for faster execution compared to interpretation alone.

- **Garbage Collection**: JVM manages memory through garbage collection, automatically freeing memory allocated to objects that are no longer needed. This not only helps in managing resources efficiently but also reduces the likelihood of memory leaks and other memory-related issues.

2.14.4. Multithreading and Synchronization:

- **Thread Management**: The JVM supports multithreaded execution, allowing multiple threads to run concurrently within a single process. It manages synchronization between threads, ensuring that resources are properly shared and accessed in a thread-safe manner.

2.14.5. Load and Execution of Code:

- **Dynamic Loading**: JVM dynamically loads, links, and initializes classes and interfaces. This means classes are loaded as needed at runtime, making the execution process more modular and efficient.

2.14.6. Platform-Specific Features:

- **Native Interface and Libraries**: While JVM abstracts the details of the underlying platform, it also provides mechanisms (such as the Java Native Interface - JNI) for Java applications to interact with native libraries and call platform-specific functions when necessary.

2.14.7. Tooling and Debugging:

- **Support for Development Tools**: The JVM ecosystem includes a vast array of development and debugging tools that leverage JVM capabilities for profiling, debugging, and monitoring Java applications.

In summary, the JVM is a pivotal technology that not only ensures the portability, security, and performance of Java applications but also provides a robust platform for developing and executing high-performance, scalable, and secure applications across diverse computing environments.

2.15. Q3b: Define: Package. Write the steps to create a Package with suitable example.

2.15.1. Definition of Java Package:

In Java, a package is a way of organizing classes and interfaces into namespaces to prevent naming conflicts and provide a hierarchical structure to the Java codebase. It allows for better organization, management, and modularization of Java code. Packages also facilitate access control and provide a mechanism for code reuse.

2.15.2. Steps to Create a Java Package:

Creating a Java package involves the following steps:

1. **Choose a Package Name**:
   - Select a unique name for your package that reflects its purpose and functionality.
   - Package names typically follow the reverse domain naming convention, such as `com.example.package`.
2. **Create Package Directory Structure**:

- Create a directory structure corresponding to the package name.
- Each level of the package name corresponds to a directory in the file system.
- For example, if the package name is `com.example.package`, create the directory structure `com/example/package`.

3. **Place Java Files in the Package Directory**:
   - Create Java source files (`.java` files) containing classes or interfaces that belong to the package.
   - Place these Java files in the directory corresponding to the package name.
   - Ensure that the package declaration in each Java file matches the package name and directory structure.
4. **Compile Java Files**:
   - Compile the Java source files using the `javac` command.
   - Specify the directory containing the package structure as the source path using the `-d` option to ensure that compiled class files are placed in the appropriate package directory.
5. **Use the Package**:
   - Once the package is created and compiled, you can use it in other Java classes by importing it using the `import` statement.
   - Import the package or specific classes/interfaces from the package into your Java code to access its functionality.

2.15.3. Example of Creating a Java Package:

Suppose we want to create a package named `com.example.util` containing utility classes for string manipulation. Here are the steps to create and use this package:

1. **Create Package Directory Structure**:
   - Create a directory named `com` within your project directory.
   - Inside the `com` directory, create a subdirectory named `example`.
   - Inside the `example` directory, create another subdirectory named `util`.
2. **Place Java Files in the Package Directory**:
   - Create a Java source file named `StringUtils.java` containing utility methods for string manipulation.
   - Place this Java file in the `util` directory.
   - Add the package declaration `package com.example.util;` at the beginning of the `StringUtils.java` file.
3. **Compile Java Files**:
   - Open a terminal or command prompt.
   - Navigate to the directory containing the `com` directory.
   - Compile the `StringUtils.java` file using the following command:

     ```
     javac com/example/util/StringUtils.java -d .
     ```

   - The `-d .` option specifies that the compiled class file should be placed in the current directory (`.`), maintaining the package structure.
4. **Use the Package**:
   - In other Java classes where you want to use the `StringUtils` class, import it using the `import` statement:

     ```
     import com.example.util.StringUtils;
     ```

   - You can then use the methods provided by the `StringUtils` class in your Java code.

By following these steps, you can create and use Java packages to organize and manage your codebase effectively, promoting modularity, reusability, and maintainability.

Here's a code example demonstrating the creation and usage of a Java package named `com.example.util` containing a `StringUtils` class with utility methods for string manipulation:

1. **StringUtils.java** (inside `com/example/util` directory):

```
package com.example.util;

public class StringUtils {
    // Method to reverse a string
    public static String reverseString(String str) {
        return new StringBuilder(str).reverse().toString();
    }

    // Method to check if a string is palindrome
    public static boolean isPalindrome(String str) {
        String reversed = reverseString(str);
        return str.equals(reversed);
    }
}
```

2. **Main.java** (outside the `com.example.util` package):

```
import com.example.util.StringUtils;

public class Main {
    public static void main(String[] args) {
        String str = "radar";

        // Using StringUtils methods
        String reversed = StringUtils.reverseString(str);
        boolean isPalindrome = StringUtils.isPalindrome(str);

        System.out.println("Original string: " + str);
        System.out.println("Reversed string: " + reversed);
        System.out.println("Is palindrome? " + isPalindrome);
    }
}
```

2.15.4. Explanation:

- In the `StringUtils.java` file, we define a `StringUtils` class inside the `com.example.util` package.
- This class contains two static methods: `reverseString()` to reverse a given string and `isPalindrome()` to check if a string is a palindrome.
- In the `Main.java` file, we import the `StringUtils` class from the `com.example.util` package using the `import` statement.
- We then use the utility methods provided by the `StringUtils` class (`reverseString()` and `isPalindrome()`) in the `main()` method to demonstrate their functionality.

After compiling both files and running the `Main` class, the output will display the original string, its reversed form, and whether it is a palindrome or not based on the utility methods provided by the `StringUtils` class.

2.16. Q3c: Explain Synchronization in Thread with suitable example.

In Java, synchronization refers to the coordination of multiple threads to ensure proper and orderly access to shared resources, thereby preventing data corruption and race conditions. When multiple threads access shared data concurrently, synchronization ensures that only one thread can access the shared resource at a time, maintaining data integrity and consistency. Java provides several mechanisms for synchronization, including synchronized blocks and methods, locks, and atomic variables. Let's explore synchronization in Java in detail with a suitable example.

2.16.1. Synchronization with `synchronized` Keyword:

1. **Synchronized Blocks**:
   - In Java, synchronized blocks allow you to specify a block of code that can be executed by only one thread at a time.
   - You can synchronize on any object, typically using the `this` keyword to lock the current object.
   - Syntax: `synchronized (object) { ... }`
2. **Synchronized Methods**:
   - You can also declare entire methods as synchronized, ensuring that only one thread can execute the method at a time for a particular instance of the class.
   - Syntax: `public synchronized void methodName() { ... }`

2.16.2. Example: Bank Account Simulation with Synchronization:

Suppose we have a bank account class `BankAccount` that allows multiple threads to deposit and withdraw money. Without synchronization, concurrent access to the account balance could lead to inconsistencies. Let's see how synchronization can be applied to ensure thread safety:

```java
public class BankAccount {
    private double balance;
    public BankAccount(double balance) {
        this.balance = balance;
    }

    // Synchronized method to deposit money
    public synchronized void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount);
    }

    // Synchronized method to withdraw money
    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
        } else {
            System.out.println("Insufficient balance");
        }
    }

    // Method to get current balance
    public synchronized double getBalance() {
        return balance;
    }
}
```

In this example:

- Both the `deposit()` and `withdraw()` methods are declared as synchronized, ensuring that only one thread can execute them at a time for a particular `BankAccount` instance.
- The `getBalance()` method is also synchronized to prevent race conditions while accessing the balance.
- By using synchronized methods, we ensure that concurrent threads cannot access the `BankAccount` methods simultaneously, maintaining data consistency and integrity.

2.16.3. Usage of Bank Account Class in Multiple Threads:

```java
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);

        // Thread to deposit money
        Thread depositThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.deposit(100);
                System.out.println("Current balance (deposit thread): " +
account.getBalance());
            }
        });

        // Thread to withdraw money
        Thread withdrawThread = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                account.withdraw(200);
                System.out.println("Current balance (withdraw thread): " +
account.getBalance());
            }
        });

        // Start deposit and withdraw threads
        depositThread.start();
        withdrawThread.start();
    }
}
```

In this example:

- We create two threads, `depositThread` and `withdrawThread`, each performing deposit and withdrawal operations on the `BankAccount` instance concurrently.
- The synchronized methods in the `BankAccount` class ensure that deposit and withdrawal operations are performed atomically, preventing inconsistencies due to concurrent access.

By synchronizing critical sections of code, we ensure thread safety and prevent data corruption in multithreaded environments, maintaining the integrity and consistency of shared resources.

2.17. Q4a: Differentiate between String class and StringBuffer class.

In Java, both the `String` class and `StringBuffer` class are used for handling strings, but they have different characteristics and behaviors. Here's a comparison between the `String` class and the `StringBuffer` class:

2.17.1. String Class:

1. **Immutable**:
   - Objects of the `String` class are immutable, meaning once a `String` object is created, its value cannot be changed.
   - Any operation that appears to modify a `String` object actually creates a new `String` object with the modified value.
   - Example: `String str = "Hello"; str = str + " World";` creates a new `String` object with the value "Hello World".
2. **Thread-Safe**:
   - Since `String` objects are immutable, they are inherently thread-safe.
   - Multiple threads can safely share and access `String` objects without the risk of data corruption or race conditions.
3. **Performance Implications**:
   - Immutable nature leads to frequent object creation, which can impact memory usage and performance, especially in scenarios involving string concatenation or manipulation.

2.17.2. StringBuffer Class:

1. **Mutable**:
   - Objects of the `StringBuffer` class are mutable, meaning their value can be modified after creation.
   - `StringBuffer` provides methods for appending, inserting, deleting, and modifying characters within the string.
2. **Not Thread-Safe**:
   - Unlike `String`, `StringBuffer` is not inherently thread-safe. Multiple threads accessing a `StringBuffer` object concurrently without proper synchronization can lead to data corruption or inconsistencies.
3. **Better Performance for String Manipulation**:
   - `StringBuffer` is optimized for string manipulation operations such as concatenation, appending, and inserting.
   - It avoids frequent object creation by modifying the contents of the existing buffer, resulting in better performance compared to `String` for such operations.

2.17.3. Example:

```
String str = "Hello";
str = str + " World"; // New String object is created
```

```
StringBuffer buffer = new StringBuffer("Hello");
buffer.append(" World"); // Modifies existing StringBuffer object
```

In summary, the main differences between the `String` class and the `StringBuffer` class lie in their mutability, thread safety, and performance characteristics. Use `String` when dealing with immutable strings or when thread safety is a concern, and use `StringBuffer` when performing extensive string manipulation operations or when mutability is required.

2.18. Q4b: Write a Java Program to find sum and average of 10 numbers of an array.

Here's a Java program to find the sum and average of 10 numbers in an array:

```
public class SumAndAverage {
    public static void main(String[] args) {
        // Define an array of 10 numbers
        int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

        // Calculate sum of numbers
        int sum = 0;
        for (int number : numbers) {
            sum += number;
        }

        // Calculate average of numbers
        double average = (double) sum / numbers.length;
        // Display sum and average
        System.out.println("Sum of numbers: " + sum);
        System.out.println("Average of numbers: " + average);
    }
}
```

This program defines an array of 10 numbers and then iterates through the array to calculate the sum of all numbers. It then calculates the average by dividing the sum by the total number of elements in the array. Finally, it prints the sum and average of the numbers.

2.19. Q4c: Explain abstract class with suitable example. Explain final class with suitable example.

An abstract class in Java is a class that cannot be instantiated, meaning you cannot create objects of an abstract class. However, it can be subclassed. Abstract classes are used to provide a base for other classes to extend and implement abstract methods, alongside providing full implementations of other methods. Abstract classes allow you to define a template for a group of subclasses.

An abstract class may contain abstract methods, which are methods declared without an implementation. The subclasses of an abstract class must provide implementations for the abstract methods unless the subclass is also abstract.

2.19.1. Key Points:

- If a class includes at least one abstract method, the class itself must be declared abstract.
- Abstract classes can include both abstract methods (without a body) and regular methods (with a body).
- You cannot create instances of an abstract class directly.
- Abstract classes are useful for defining common templates for a family of subclasses.

2.19.2. Example:

Let's consider an example with a simple hierarchy for shapes where we define an abstract class `Shape` and concrete classes `Circle` and `Rectangle` that extend `Shape`.

```
abstract class Shape {
    String color;

    // Constructor
    public Shape(String color) {
        this.color = color;
    }

    // Abstract method
    abstract double area();

    // Concrete method
    public String getColor() {
        return color;
    }
```

```java
}

class Circle extends Shape {
    double radius;

    public Circle(String color, double radius) {
        super(color); // calling Shape constructor
        this.radius = radius;
    }

    // Implementing the abstract method
    @Override
    double area() {
        return Math.PI * Math.pow(radius, 2);
    }
}

class Rectangle extends Shape {
    double width;
    double height;

    public Rectangle(String color, double width, double height) {
        super(color); // calling Shape constructor
        this.width = width;
        this.height = height;
    }

    // Implementing the abstract method
    @Override
    double area() {
        return width * height;
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle("Red", 2.5);
        Shape rectangle = new Rectangle("Blue", 4.0, 5.0);

        System.out.println("Circle color: " + circle.getColor() + " and area: " +
circle.area());
        System.out.println("Rectangle color: " + rectangle.getColor() + " and
area: " + rectangle.area());
    }
}
```

2.19.3. Explanation:

- The `Shape` class is abstract and contains one abstract method `area()` and a concrete method `getColor()`.
- The `Circle` and `Rectangle` classes extend `Shape` and provide concrete implementations for the `area()` method.
- The `Shape` class cannot be instantiated directly due to its abstract nature, but we can reference `Circle` and `Rectangle` objects using a `Shape` reference.

- This design allows for flexibility and reusability, as other types of shapes can be easily added to the hierarchy by extending the `Shape` class and providing an implementation for the `area()` method.

In Java, a final class is a class that cannot be subclassed or extended. When a class is declared as final, it means that no other class can inherit from it. This is useful when you want to prevent the class from being modified or extended further, ensuring that its behavior remains unchanged.

2.19.4. Key Points:

- A final class cannot have any subclasses.
- All methods in a final class are implicitly final, meaning they cannot be overridden by subclasses.
- Final classes are typically used for utility classes, immutable classes, or classes with a fixed implementation that should not be extended.

2.19.5. Example:

```java
final class FinalClass {
    private final int value;

    // Constructor
    public FinalClass(int value) {
        this.value = value;
    }

    // Getter method
    public int getValue() {
        return value;
    }

    // This method cannot be overridden in subclasses
    public final void display() {
        System.out.println("Value: " + value);
    }
}
```

In this example:

- The `FinalClass` is declared as final, indicating that it cannot be subclassed.
- It contains a private field `value` and a constructor to initialize it.
- The `getValue()` method provides read-only access to the `value` field.
- The `display()` method is declared as final, meaning it cannot be overridden by subclasses.

Attempting to subclass a final class will result in a compilation error:

```java
// Compilation error: cannot inherit from final FinalClass
class SubClass extends FinalClass {
    // Attempting to extend a final class
}
```

By making a class final, you ensure that its behavior remains consistent and cannot be altered by subclasses, enhancing code stability and predictability. Final classes are particularly useful for creating utility classes, such as helper methods or constants, where you want to prevent unintended subclassing or modification of the class's behavior.

2.20. Q4a: Explain Garbage Collection in Java.

Garbage Collection (GC) in Java is a process by which the JVM automatically manages memory by reclaiming memory occupied by objects that are no longer referenced or needed by the program. The main goal of garbage collection is to free up memory resources by identifying and reclaiming objects that are no longer in use, thereby preventing memory leaks and allowing for efficient memory management.

2.20.1. Key Concepts:

1. **Automatic Memory Management**:
    - Unlike languages such as C or C++, where developers manually allocate and deallocate memory using `malloc()` and `free()` functions, Java employs automatic memory management through garbage collection.
    - Developers do not need to explicitly free memory occupied by objects. Instead, the JVM handles memory allocation and deallocation automatically.
2. **Garbage Collector**:
    - The Garbage Collector (GC) is a component of the JVM responsible for reclaiming memory occupied by objects that are no longer reachable or referenced by the program.
    - The GC periodically scans the heap (the region of memory where objects are allocated) to identify and mark objects that are still in use and reachable from the program.
    - Objects that are not reachable, either directly or indirectly, from any live threads are considered garbage and can be safely reclaimed.
3. **Heap Memory Management**:
    - In Java, objects are allocated memory on the heap using the `new` keyword. The heap is divided into generations (Young Generation, Old Generation, and Permanent Generation in older JVM versions).
    - The garbage collection process typically focuses on reclaiming memory from objects in the Young Generation, as they are short-lived and often become garbage quickly.
    - Older objects in the Old Generation undergo less frequent garbage collection cycles.

2.20.2. Garbage Collection Process:

1. **Mark Phase**:
    - The garbage collector traverses the object graph starting from the root objects (such as global variables, local variables, and method call stacks).
    - It marks objects that are reachable and in use as live objects, typically using a technique like Depth-First Search (DFS) or Tracing.
2. **Sweep Phase**:
    - After marking live objects, the garbage collector identifies and reclaims memory occupied by objects that are not marked (i.e., unreachable objects).

   - Reclaimed memory is returned to the heap for future allocations.
3. **Compact Phase (Optional)**:
   - Some garbage collectors perform memory compaction after reclaiming memory to reduce fragmentation and optimize memory usage.
   - Memory compaction involves moving live objects closer together to reduce fragmentation and improve memory access performance.

2.20.3. Advantages of Garbage Collection:

- **Automatic Memory Management**: Developers do not need to manually manage memory, reducing the risk of memory leaks and segmentation faults.
- **Improved Developer Productivity**: Developers can focus on application logic rather than memory management, leading to faster development cycles and fewer bugs related to memory management.
- **Dynamic Memory Allocation**: Garbage collection enables dynamic memory allocation and resizing of objects, allowing for flexible memory usage without the need for manual memory management.

In summary, garbage collection in Java is a crucial mechanism for automatic memory management, ensuring efficient use of memory resources and preventing memory-related issues such as memory leaks and segmentation faults. By automatically reclaiming memory occupied by unreachable objects, garbage collection allows Java applications to run reliably and efficiently.

2.21. Q4b: Write a Java program to handle user defined exception for 'DividebyZero' error.

To handle a user-defined exception for a "DivideByZero" error in Java, you can create a custom exception class that extends the `Exception` class. Then, you can throw this custom exception when encountering a divide-by-zero situation. Below is an example Java program demonstrating this:

```java
// Custom exception class for DivideByZero error
class DivideByZeroException extends Exception {
    public DivideByZeroException(String message) {
        super(message);
    }
}

// Class that performs division and throws DivideByZeroException
class Divider {
    public static double divide(int numerator, int denominator) throws
DivideByZeroException {
        if (denominator == 0) {
            throw new DivideByZeroException("Error: Division by zero is not
allowed.");
        }
        return (double) numerator / denominator;
    }
}

// Main class to demonstrate handling of DivideByZeroException
public class Main {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;

        try {
            double result = Divider.divide(numerator, denominator);
            System.out.println("Result of division: " + result);
        } catch (DivideByZeroException e) {
            System.out.println("Error: " + e.getMessage());
            // Additional handling can be done here, such as logging or informing
the user
        }
    }
}
```

In this example:

- We define a custom exception class `DivideByZeroException` that extends `Exception`.
- The `Divider` class provides a `divide` method that takes a numerator and a denominator as parameters and performs division. If the denominator is zero, it throws a `DivideByZeroException`.
- In the `Main` class, we attempt to divide by zero within a try-catch block. If a `DivideByZeroException` is thrown during the division operation, it is caught, and an appropriate error message is displayed.

This program demonstrates how to handle user-defined exceptions for divide-by-zero errors in Java. Custom exception classes provide flexibility in handling different types of errors and allow for more meaningful error messages and error handling strategies.

2.22. Q4c: Write a java program to demonstrate multiple try block and multiple catch block exception.

Certainly! Below is a Java program demonstrating the use of multiple `try` blocks and multiple `catch` blocks to handle different types of exceptions:

```
public class MultipleTryCatchDemo {
    public static void main(String[] args) {
        try {
            // Division by zero exception
            int result = divideByZero(10, 0);
            System.out.println("Result of division: " + result);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException caught: " + e.getMessage());
        }

        try {
            // Array index out of bounds exception
            int[] numbers = {1, 2, 3};
            int index = 4;
            int value = accessArrayElement(numbers, index);
            System.out.println("Value at index " + index + ": " + value);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBoundsException caught: " +
e.getMessage());
        }

        try {
            // NullPointerException
            String str = null;
            int length = str.length();
            System.out.println("Length of string: " + length);
        } catch (NullPointerException e) {
            System.out.println("NullPointerException caught: " + e.getMessage());
        }
    }

    // Method to perform division and throw ArithmeticException
    public static int divideByZero(int numerator, int denominator) {
        return numerator / denominator;
    }

    // Method to access array element and throw ArrayIndexOutOfBoundsException
    public static int accessArrayElement(int[] array, int index) {
        return array[index];
    }
}
```

In this program:

- We have three different `try` blocks, each attempting an operation that can potentially throw a different type of exception.
- Each `try` block is followed by a corresponding `catch` block that handles the specific type of exception thrown within that `try` block.
- The first `try` block attempts division by zero, and the `catch` block catches the `ArithmeticException`.
- The second `try` block attempts to access an element beyond the bounds of an array, and the `catch` block catches the `ArrayIndexOutOfBoundsException`.
- The third `try` block attempts to invoke a method on a `null` object reference, leading to a `NullPointerException`, which is caught by the corresponding `catch` block.

This program demonstrates how multiple `try` blocks and multiple `catch` blocks can be used to handle different types of exceptions separately, allowing for more precise error handling in Java programs.

2.23. Q5a: Write a program in Java to create a file and perform write operation on this file.

Below is a Java program that demonstrates how to create a file and perform write operations on it using the `File` and `FileOutputStream` classes:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
```

```java
public class FileWriteDemo {
    public static void main(String[] args) {
        // Specify the file name and content
        String fileName = "example.txt";
        String content = "Hello, world! This is a sample text file.";

        // Create a File object
        File file = new File(fileName);

        try {
            // Create a FileOutputStream to write to the file
            FileOutputStream fos = new FileOutputStream(file);

            // Convert the content string to bytes and write to the file
            fos.write(content.getBytes());

            // Close the FileOutputStream
            fos.close();

            System.out.println("File '" + fileName + "' has been created and
written successfully.");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

In this program:

- We specify the file name (`example.txt`) and the content that we want to write to the file (`Hello, world! This is a sample text file.`).
- We create a `File` object named `file` with the specified file name.
- We create a `FileOutputStream` named `fos` to write to the file.
- We convert the content string to bytes using the `getBytes()` method and write these bytes to the file using the `write()` method of `FileOutputStream`.
- We close the `FileOutputStream` after writing to the file.
- If an `IOException` occurs during file creation or writing, we handle it and print an error message.

After running this program, a file named `example.txt` will be created in the same directory as the Java program, and the specified content will be written to it.

2.24. Q5b: Explain throw and finally in Exception Handling with example.

In Java, exception handling is a powerful mechanism that allows you to manage runtime errors, ensuring the program's flow can be maintained even when unexpected events occur. Two key components of this mechanism are the `throw` keyword and the `finally` block.

2.24.1. The `throw` Keyword

The `throw` keyword in Java is used to explicitly throw an exception from a method or any block of code. You can throw either checked or unchecked exceptions. The thrown exception must be either caught by a `catch` block surrounding the `throw` statement or declared to be thrown by the method using the `throws` keyword.

Example of `throw` keyword:

```
public class ThrowExample {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least 18 
years old.");
        } else {
            System.out.println("Access granted - You are old enough!");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

In this example, the `checkAge` method throws an `ArithmeticException` if the `age` parameter is less than 18. The exception is caught in the `main` method's `catch` block.

2.24.2. The `finally` Block

The `finally` block is used to execute a block of code after a try-catch block has completed, regardless of whether an exception was thrown or caught. It is the ideal place to put cleanup code, such as closing file streams or releasing resources, ensuring that these operations are carried out regardless of what happens within the `try` block.

Example of `finally` block:

```
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 25 / 5;
            System.out.println(data);
        } catch (NullPointerException e) {
            System.out.println(e);
        } finally {
            System.out.println("Finally block is always executed");
        }
        System.out.println("Rest of the code...");
    }
}
```

In this example, the `try` block executes successfully, so the `catch` block is skipped. However, the `finally` block is executed regardless, ensuring the message "Finally block is always executed" is printed to the console.

Key Points:

- The `throw` keyword allows for manually throwing exceptions, providing control over error reporting.
- The `finally` block ensures certain code is executed after a try-catch block, regardless of the outcome, making it ideal for cleanup operations.

2.25. Q5c: Describe: Polymorphism. Explain run time polymorphism with suitable example in java.

2.25.1. Polymorphism:

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent multiple underlying forms. There are two types of polymorphism in Java: compile-time polymorphism (method overloading) and runtime polymorphism (method overriding).

2.25.2. Runtime Polymorphism:

Runtime polymorphism, also known as dynamic polymorphism, occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. It allows a method to be overridden in a subclass with a different implementation, and the decision of which method to execute is made at runtime based on the actual type of the object.

2.25.3. Example of Runtime Polymorphism in Java:

```java
// Superclass
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass 1
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Subclass 2
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Creating objects of different subclasses
        Animal animal1 = new Dog(); // Upcasting
        Animal animal2 = new Cat(); // Upcasting

        // Calling overridden methods
        animal1.sound(); // Calls Dog's sound method
        animal2.sound(); // Calls Cat's sound method
    }
}
```

In this example:

- We have a superclass `Animal` with a method `sound()`.
- We have two subclasses `Dog` and `Cat`, each overriding the `sound()` method with specific implementations.
- In the `main()` method, we create objects of the subclasses and assign them to references of the superclass (`Animal`). This is called upcasting.
- When we call the `sound()` method on these objects, Java determines which implementation to execute based on the actual type of the object at runtime. This is runtime polymorphism.
- As a result, the output of the program is:

  ```
  Dog barks
  Cat meows
  ```

2.25.4. Benefits of Runtime Polymorphism:

- It allows for flexibility and extensibility in code, enabling subclasses to provide their own implementations of methods.
- It promotes code reusability by allowing common interfaces to be shared across multiple classes.

Runtime polymorphism is a powerful mechanism in Java that facilitates code organization, maintenance, and flexibility by enabling dynamic method invocation based on the actual type of the object at runtime.

2.26. Q5a: Write a program in Java that read the content of a file byte by byte and copy it into another file.

Below is a Java program that reads the content of a file byte by byte and copies it into another file:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopyByteByByte {
    public static void main(String[] args) {
        String sourceFileName = "source.txt";
        String destinationFileName = "destination.txt";
```

```
        try (FileInputStream fis = new FileInputStream(sourceFileName);
             FileOutputStream fos = new FileOutputStream(destinationFileName)) {

            int byteRead;
            while ((byteRead = fis.read()) != -1) {
                fos.write(byteRead);
            }

            System.out.println("File copied successfully.");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

In this program:

- We specify the name of the source file (`source.txt`) and the destination file (`destination.txt`).
- We use `FileInputStream` to read bytes from the source file and `FileOutputStream` to write bytes to the destination file.
- Inside the try-with-resources block, we create instances of `FileInputStream` and `FileOutputStream`.
- We use a `while` loop to read bytes from the source file until the `read()` method returns `-1`, indicating the end of the file.
- Within the loop, each byte read from the source file is written to the destination file using the `write()` method.
- Any `IOException` that occurs during file operations is caught and handled, displaying an error message.

After running this program, the content of the source file (`source.txt`) will be copied byte by byte into the destination file (`destination.txt`).

2.27. Q5b: Explain the different I/O Classes available with Java.

In Java, the I/O (Input/Output) classes are used to perform input and output operations, such as reading from or writing to files, streams, consoles, and network connections. These classes are part of the `java.io` package and provide various functionalities for handling different types of I/O operations. Here are some of the commonly used I/O classes available in Java:

1. **InputStream and OutputStream**:
   - `InputStream` and `OutputStream` are abstract classes representing input and output streams of bytes, respectively.
   - They serve as the base classes for all byte-oriented I/O classes in Java.
2. **Reader and Writer**:
   - `Reader` and `Writer` are abstract classes representing input and output streams of characters, respectively.

- They serve as the base classes for all character-oriented I/O classes in Java.
- `InputStreamReader` and `OutputStreamWriter` are bridge classes that convert byte streams to character streams and vice versa.

3. **FileInputStream and FileOutputStream**:
    - `FileInputStream` and `FileOutputStream` are used to read from and write to files, respectively, as streams of bytes.
    - They are commonly used for file I/O operations.
4. **FileReader and FileWriter**:
    - `FileReader` and `FileWriter` are used to read from and write to files, respectively, as streams of characters.
    - They are commonly used for text file I/O operations.
5. **BufferedInputStream and BufferedOutputStream**:
    - `BufferedInputStream` and `BufferedOutputStream` are used for buffered input and output operations, respectively.
    - They improve I/O performance by reducing the number of physical I/O operations.
6. **BufferedReader and BufferedWriter**:
    - `BufferedReader` and `BufferedWriter` are used for buffered character input and output operations, respectively.
    - They provide efficient reading and writing of characters by buffering input and output streams.
7. **DataInputStream and DataOutputStream**:
    - `DataInputStream` and `DataOutputStream` are used for reading and writing primitive data types as binary data, respectively.
    - They provide methods for reading and writing Java primitive data types (e.g., int, double, boolean) from and to streams.
8. **ObjectInputStream and ObjectOutputStream**:
    - `ObjectInputStream` and `ObjectOutputStream` are used for reading and writing Java objects, respectively.
    - They allow objects to be serialized (converted into a stream of bytes) and deserialized (reconstructed from the stream of bytes).

These are some of the commonly used I/O classes available in Java. They provide a wide range of functionalities for performing input and output operations in Java programs, facilitating interactions with files, streams, consoles, and other I/O sources.

2.28. Q5c: Write a java program that executes two threads. One thread displays “Java Programming” every 3 seconds, and the other displays “Semester - 4th IT” every 6 seconds.(Create the threads by extending the Thread class)

Below is a Java program that creates two threads by extending the `Thread` class. One thread displays "Java Programming" every 3 seconds, and the other thread displays "Semester - 4th IT" every 6 seconds:

```java
class DisplayThread extends Thread {
    private String message;
    private int interval;

    public DisplayThread(String message, int interval) {
        this.message = message;
        this.interval = interval;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(message);
            try {
                Thread.sleep(interval * 1000); // Convert seconds to milliseconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        DisplayThread thread1 = new DisplayThread("Java Programming", 3);
        DisplayThread thread2 = new DisplayThread("Semester - 4th IT", 6);

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

In this program:

- We create a `DisplayThread` class that extends the `Thread` class. This class takes a message and an interval as parameters in its constructor.
- In the `run()` method of `DisplayThread`, the thread continuously prints the message and then sleeps for the specified interval.
- In the `main()` method, we create two instances of `DisplayThread`, one for each message with their respective intervals.
- We start both threads using the `start()` method, which causes the `run()` method of each thread to be executed concurrently.

As a result, the program will continuously display "Java Programming" every 3 seconds and "Semester - 4th IT" every 6 seconds in separate threads.