GPCET & RECW UNIT – II Q&A

1.Explain the difference between public, private, protected, and default access in Java with an example.

ANS:

1. **public (Most Accessible)**
   - A member declared as public is accessible **from anywhere** in the program.
   - It can be accessed from **within the same class, same package, different packages, and subclasses**.
   - A public class must be saved with the same name as the file name.
2. **private (Most Restrictive)**
   - A member declared as private is **accessible only within the same class**.
   - It **cannot** be accessed from other classes, even within the same package.
   - Useful for **encapsulation** to hide internal implementation details
3. **protected (Limited Accessibility)**
   - The member is accessible **within the same package** and **in subclasses outside the package (only through inheritance)**.
   - Not accessible in unrelated classes in a different package.
   - Used to allow access in **subclasses while still restricting outside access**.
4. **default (Package-Private Access)**
   - If no access modifier is specified, the member has **default (package-private) access**.
   - The member is **accessible only within the same package** and not outside the package, even if inherited.

Access Modifier	**Same Class**	**Same Package**	**Subclass (Different Package)**	**Other Classes (Different Package)**
public	✓ Yes	✓ Yes	✓ Yes	✓ Yes
protected	✓ Yes	✓ Yes	✓ Yes (only through inheritance)	✗ No
default	✓ Yes	✓ Yes	✗ No	✗ No
private	✓ Yes	✗ No	✗ No	✗ No

2. Write a Java program to create a class Car with attributes like brand, model, and price. Create objects and display their details.

ANS: A **class** in Java is a **blueprint** for creating objects. It defines the **attributes (variables)** and **methods (functions)** that an object will have.

An **object** is an instance of a class. It has its own **state (attribute values)** and **behavior (methods)**.

```
// Car.java

class Car {

   // Attributes

   String brand;

   String model;

   double price;


   // Constructor

   Car(String brand, String model, double price) {

      this.brand = brand;

      this.model = model;

      this.price = price;

   }


   // Method to display car details

   void displayDetails() {

      System.out.println("Brand: " + brand);

      System.out.println("Model: " + model);

      System.out.println("Price: $" + price);

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

   }

}

// Main class to test Car objects

public class CarDemo {

   public static void main(String[] args) {
      // Creating car objects

      Car car1 = new Car("Toyota", "Camry", 30000);

      Car car2 = new Car("Honda", "Civic", 25000);

      Car car3 = new Car("BMW", "X5", 60000);


      // Displaying details of cars

      System.out.println("Car Details:");

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

      car1.displayDetails();

      car2.displayDetails();

      car3.displayDetails();

   }

}
```

3. What is the difference between static members and instance members in a class? Give an example.

ANS: In Java, **class members** (variables and methods) can be classified into **static members** and **instance members** based on how they are associated with the class and objects.

1. Static Members

- **Belong to the class** rather than individual objects.
- **Declared using the static keyword**.
- **Shared among all objects** of the class.
- Can be accessed using the **class name** (e.g., ClassName.member).
- **Memory is allocated only once** at the time of class loading.
- Cannot access **non-static (instance) members** directly.

2. Instance Members

- **Belong to an individual object**.
- **Created every time a new object is instantiated**.
- Can be accessed using an **object reference** (objectName.member).
- Each object **gets its own copy** of instance variables.
- Can access **both instance and static members**.

Key Differences:

Feature	**Static Members (static)**	**Instance Members**
Belongs To	Class (shared by all objects)	Individual objects
Memory Allocation	Once, at class loading	Each time an object is created
Accessed By	ClassName.member	objectName.member
Can Access	Other static members only	Both static and instance members
Example	static int totalCars;	String brand;

```
class Car {
   // Static member (shared by all objects)
   static int totalCars = 0;
   // Instance members (unique for each object)
   String brand;
   String model;
   // Constructor
   Car(String brand, String model) {
      this.brand = brand;
      this.model = model;
      totalCars++; // Increment static member
   }
   // Instance method (can access both static and instance members)
   void displayCarDetails() {
      System.out.println("Brand: " + brand + ", Model: " + model);
   }
   // Static method (can only access static members)
   static void displayTotalCars() {
      System.out.println("Total Cars: " + totalCars);
   }
}
// Main class
public class StaticVsInstanceDemo {
   public static void main(String[] args) {
      // Creating objects
      Car car1 = new Car("Toyota", "Camry");
      Car car2 = new Car("Honda", "Civic");
      // Accessing instance members
      car1.displayCarDetails();
      car2.displayCarDetails();

      // Accessing static member using class name
      Car.displayTotalCars(); // Static method called using class name
   }
}
```

4. Write a Java program to demonstrate **pass-by-reference** and **pass-by-value** using objects.

ANS: In Java, method arguments are always passed **by value**. This means that a copy of the actual parameter is passed to the method, and any modifications inside the method do not affect the original value in the caller.

However, Java handles objects differently, creating an illusion of **pass-by-reference** because references to objects are passed **by value**. This distinction is crucial to understanding how Java handles method parameters.

Passing Primitive Data Types (Pass by Value)

In Java, when a primitive data type (e.g., int, double, char, boolean) is passed to a method, a **copy** of the value is sent. The original variable remains unchanged.

```
Package RECWCSEEF
public class Demo {
        public static void modify(int a, int b) {
                a = 1;
                b = 2;
```

```
		System.out.println("this values in side method : "+a+" "+b);
	}
	public static void main(String[] args) {
		int a = 10;
		int b = 20;
		System.out.println("Before method call: "+a+" "+b);
		modify(a, b);
		System.out.println("After method call: " +a+" "+b);
	}
}
```

Output:

```
Before method call: 10 20
this values in side method : 1 2
After method call: 10 20
```

Pass by Reference (Objects)

- In Java, objects are passed by value, but the value is a reference (memory address).
- This means changes inside the method affect the original object.

```
Package RECWCSEEF
public class Box {
	int length;
	Box(int length){
		this.length = length;
	}
	public void modify(Box obj) {
		obj.length += 10;
	}
	public static void main(String[] args) {
		Box bx = new Box(1);
		System.out.println(bx.length);
		bx.modify(bx);
		System.out.println(bx.length);
```

```
        }
}
```

6. Write a Java program to demonstrate method overloading.

ANS:

Method Overloading in Java allows a class to have **multiple methods with the same name** but different **parameter lists** (different number of parameters, data types, or both).

Rules for Method Overloading

1. **Same Method Name**: The overloaded methods must have the **same name**.
2. **Different Parameters**: Methods must differ in the **number** or **type** of parameters.
3. **Return Type Doesn't Matter**: Changing only the return type **does not** differentiate overloaded methods.
4. **Methods in the Same Class**: All overloaded methods must be defined in the **same class**.

```
class Calculator {
  // Method 1: Adding two integers
  int add(int a, int b) {
    return a + b;
  }
  // Method 2: Adding three integers (Overloaded method)
  int add(int a, int b, int c) {
    return a + b + c;
  }

  // Method 3: Adding two double values (Overloaded method)
  double add(double a, double b) {
    return a + b;
  }

  // Method 4: Concatenating two strings (Overloaded method)
  String add(String a, String b) {
        return a + b;
    }
}

// Main class to test method overloading
public class MethodOverloadingExample {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling overloaded methods
        System.out.println("Sum of 5 and 10: " + calc.add(5, 10));               // Calls add(int, int)
        System.out.println("Sum of 5, 10, and 15: " + calc.add(5, 10, 15));       // Calls add(int, int, int)
        System.out.println("Sum of 4.5 and 2.5: " + calc.add(4.5, 2.5));          // Calls add(double, double)
        System.out.println("Concatenation of 'Hello' and 'World': " + calc.add("Hello", " World")); // Calls
add(String, String)
    }
}
```

7.Explain the difference between static methods and instance methods in Java with examples.

Difference Between Static Methods and Instance Methods in Java

In Java, methods can be categorized into **static methods** and **instance methods**, depending on whether they belong to the class itself or an instance of the class. Below are the key differences:

Feature	Static Method	Instance Method
Belongs To	Class	Object (Instance of Class)
Accessing Members	Can only access static variables and static methods	Can access both static and non-static variables and methods
Invocation	Called using the class name (ClassName.methodName()) or an instance	Called using an instance of the class (objectName.methodName())
Memory Allocation	Allocated in the **Method Area** at class loading time	Allocated in **Heap Memory** for each instance

Feature	Static Method	Instance Method
Use Case	Utility methods (e.g., Math.pow(), Collections.sort())	Methods that operate on instance variables (e.g., getName(), setAge())
Polymorphism (Overriding)	Cannot be overridden (but can be hidden using method hiding)	Can be overridden in subclasses
Keyword Used	static keyword is required	No special keyword needed

---

Example Demonstration

```
class Example {
    int instanceVar = 10;  // Instance variable
    static int staticVar = 20;  // Static variable


    // Static Method
    static void staticMethod() {
        System.out.println("Inside Static Method");
        System.out.println("Static Variable: " + staticVar); // Allowed
        // System.out.println("Instance Variable: " + instanceVar); // ERROR! Cannot access instance variables
    }


    // Instance Method
    void instanceMethod() {
        System.out.println("Inside Instance Method");
        System.out.println("Instance Variable: " + instanceVar); // Allowed
        System.out.println("Static Variable: " + staticVar); // Allowed
    }
}


public class StaticVsInstance {
    public static void main(String[] args) {
        // Calling Static Method
        Example.staticMethod(); // Called using class name


        // Calling Instance Method
        Example obj = new Example(); // Creating an object
        obj.instanceMethod(); // Called using object
    }
}
```

Output

```
Inside Static Method
Static Variable: 20
Inside Instance Method
Instance Variable: 10
Static Variable: 20
```

Key Takeaways

1. **Static methods** belong to the **class** and cannot access instance variables or methods directly.
2. **Instance methods** belong to an **object** and can access both instance and static members.
3. **Static methods** are useful for utility operations (e.g., Math.sqrt(), Arrays.sort()).
4. **Instance methods** are used when we need object-specific behavior (e.g., setting or retrieving instance values).
5. **Overriding is possible only for instance methods**, while static methods support **method hiding**.
6. **Memory Allocation**:
   - Static methods: **Method Area (class-level)**
   - Instance methods: **Heap Memory (object-level)**
7. **Accessing Methods**:
   - Static: ClassName.methodName()
   - Instance: objectName.methodName()

8.Write a program where method overloading is used to calculate the area of different shapes (circle, rectangle, and triangle).

demonstrating **method overloading** to calculate the area of different shapes: **circle, rectangle, and triangle**.

Program: Method Overloading for Calculating Area

```
class AreaCalculator {

  // Method to calculate the area of a circle
  static double calculateArea(double radius) {
    return Math.PI * radius * radius;
  }

  // Method to calculate the area of a rectangle
  static double calculateArea(double length, double width) {
    return length * width;
  }

  // Method to calculate the area of a triangle
  static double calculateArea(double base, double height, boolean isTriangle) {
    return 0.5 * base * height;
  }

  public static void main(String[] args) {
    // Calculating area of different shapes
    double circleArea = calculateArea(7.0);
    double rectangleArea = calculateArea(5.0, 10.0);
    double triangleArea = calculateArea(6.0, 8.0, true);

    // Displaying results
    System.out.println("Area of Circle: " + circleArea);
    System.out.println("Area of Rectangle: " + rectangleArea);
```

```
        System.out.println("Area of Triangle: " + triangleArea);
    }
}
```

Explanation:

1. **Method Overloading:**
    - The calculateArea() method is overloaded three times with different parameter lists.
    - **Circle**: Takes one parameter (radius) and returns π * r².
    - **Rectangle**: Takes two parameters (length, width) and returns length * width.
    - **Triangle**: Takes three parameters (base, height, and a boolean flag) and returns 0.5 * base * height.
2. **Calling Methods:**
    - The appropriate method is called based on the number and type of arguments.

Output:

```
Area of Circle: 153.93804002589985

Area of Rectangle: 50.0

Area of Triangle: 24.0
```

9. Java Program for Method Overloading in Calculator

This program demonstrates **method overloading** in a Calculator class, which performs **addition** for different data types and numbers.

```
class Calculator {


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


    // Method to add two floating-point numbers
    double add(double a, double b) {
        return a + b;
    }
    // Method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

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

        // Calling overloaded methods
        System.out.println("Addition of two integers: " + calc.add(5, 10));
        System.out.println("Addition of two floating numbers: " + calc.add(4.5, 3.2));
        System.out.println("Addition of three integers: " + calc.add(2, 3, 4));
    }
}
```

Output:

```
Addition of two integers: 15
Addition of two floating numbers: 7.7
Addition of three integers: 9
```

This example illustrates **method overloading**, where methods with the same name but different parameters perform different operations.

---

10. What is a Method in Java? Explain with an Example.

A **method** in Java is a block of code that performs a specific task. It helps in **code reuse** and makes programs **modular**.

Example of a Method in Java

```
class Example {
    // Method to display a message
    void displayMessage() {
        System.out.println("Hello, this is a method in Java!");
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.displayMessage();  // Calling the method
    }
}
```

Output:

```
Hello, this is a method in Java!
```

Key Points:

- Methods define **reusable behavior**.
- They improve **code organization and readability**.
- A method **must be called** to execute.

---

11. Two Advantages of Using Methods in Java

1. **Code Reusability**
    - Methods allow us to **reuse** the same block of code without rewriting it.
    - Example: The Math.pow(x, y) method can be used multiple times instead of writing power logic repeatedly.
2. **Modular and Readable Code**
    - Methods break the program into **smaller units**, making it easier to debug and understand.
    - Large programs become **structured and manageable**.

---

12. What is a Constructor in Java? How Does it Differ from a Method?

A **constructor** is a special method used to initialize objects in Java. It is **invoked automatically** when an object is created.

Example of a Constructor

```
class Student {
    String name;

    // Constructor
    Student(String studentName) {
        name = studentName;
    }
    void display() {
        System.out.println("Student Name: " + name);
    }

    public static void main(String[] args) {
        Student s1 = new Student("John");  // Constructor is called
        s1.display();
    }
}
```

Output:

```
Student Name: John
```

Differences Between a Constructor and a Method

Feature	**Constructor**	**Method**
Purpose	Initializes an object	Defines behavior (operations)
Invocation	Called automatically when an object is created	Must be called explicitly
Return Type	No return type (not even void)	Can return a value
Name	Must match the class name	Can have any valid name

---

13. Why Do We Need Constructors in Java?

In Java, a **constructor** is a special method used to **initialize objects** when they are created. Unlike regular methods, a constructor is **automatically called** when an instance of a class is created. The primary purpose of a constructor is to **set initial values** for instance variables.

Need for Constructors in Java

1. **Automatic Initialization**
   - Without a constructor, instance variables must be assigned manually after creating an object.
   - A constructor ensures that necessary initialization happens when the object is created.
2. **Encapsulation and Code Reusability**
   - Encapsulates object initialization logic in one place.
   - Avoids repetitive code by setting default values.

3. **Improves Code Readability and Maintainability**
   - Instead of calling multiple setter methods, constructors allow initialization in a **single step**.
4. **Prevents Uninitialized Objects**
   - Ensures that an object has a valid state immediately after creation.

Types of Constructors in Java

Java supports **three types of constructors**:

1. Default Constructor (No-Argument Constructor)

- A constructor that **does not take any parameters**.
- If no constructor is defined, Java **automatically provides** a default constructor.
- It initializes instance variables with **default values** (0, null, false).

Example of a Default Constructor

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

  // Default Constructor
  Student() {
    name = "Unknown";
    age = 18;
  }

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

  public static void main(String[] args) {
    Student s1 = new Student(); // Default constructor is called
    s1.display();
  }
}
```

Output:

```
Name: Unknown, Age: 18
```

---

2. Parameterized Constructor

- A constructor that **accepts arguments** to initialize object attributes with specific values.
- Provides flexibility for initializing objects with different values.

Example of a Parameterized Constructor

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

   // Parameterized Constructor
   Student(String studentName, int studentAge) {
      name = studentName;
      age = studentAge;
   }

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

   public static void main(String[] args) {
      Student s1 = new Student("Alice", 20); // Constructor with parameters
      s1.display();
   }
}
```

Output:

```
Name: Alice, Age: 20
```

---

3. Copy Constructor

- A constructor that **copies values from another object**.
- Used when a **duplicate object** is needed with the same values.

Example of a Copy Constructor

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

  // Parameterized Constructor
  Student(String studentName, int studentAge) {
    name = studentName;
    age = studentAge;
  }

  // Copy Constructor
  Student(Student s) {
    name = s.name;
    age = s.age;
  }

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

  public static void main(String[] args) {
    Student s1 = new Student("Bob", 21); // Original object
    Student s2 = new Student(s1);       // Copy constructor is used
    s2.display();
  }
}
```

Output:

```
Name: Bob, Age: 21
```

Key Takeaways

1. **Default Constructor:** Assigns default values.
2. **Parameterized Constructor:** Allows custom initialization.
3. **Copy Constructor:** Creates a duplicate object.

---

14. Can a Constructor Call Another Constructor of the Same Class?

Yes, a constructor **can call another constructor** of the same class using the this() keyword. This is known as **constructor chaining**, which helps in avoiding code duplication and improves readability.

Constructor Chaining in Java

- The this() call must be the **first statement** in the constructor.
- It **reduces redundancy** and ensures all constructors **reuse common initialization logic**.

Example of Constructor Chaining

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

  // Constructor 1: Default Constructor
  Employee() {
    this("Unknown", 0, "Not Assigned"); // Calls Constructor 3
  }

  // Constructor 2: Constructor with two parameters
  Employee(String name, int age) {
    this(name, age, "General"); // Calls Constructor 3
  }

  // Constructor 3: Constructor with all three parameters
  Employee(String name, int age, String department) {
    this.name = name;
    this.age = age;
    this.department = department;
  }

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

  public static void main(String[] args) {
    Employee e1 = new Employee(); // Calls default constructor
    Employee e2 = new Employee("John", 25); // Calls constructor with two parameters
    Employee e3 = new Employee("Alice", 30, "HR"); // Calls full constructor

    e1.display();
    e2.display();
    e3.display();
  }
}
```

Output:

```
Name: Unknown, Age: 0, Department: Not Assigned
Name: John, Age: 25, Department: General
Name: Alice, Age: 30, Department: HR
```

Key Points About Constructor Chaining

1. **Uses this() to call another constructor** within the same class.
2. **Avoids code duplication** and ensures all constructors use the same initialization logic.
3. **The this() call must be the first statement** inside a constructor.

Conclusion

- **Constructors** ensure proper initialization of objects.

- **Types of Constructors**: Default, Parameterized, and Copy Constructors.
- **Constructor Chaining**: Helps in better code reuse and avoids redundancy.

15. Role of super() in Constructors in Java

Introduction to super()

In Java, super() is a special keyword used inside a constructor to **call the constructor of its immediate parent class**. It is primarily used for **constructor chaining in inheritance** and helps in **reusing the initialization logic** of the parent class.

Why is super() Used in Java?

1. **To call the parent class constructor explicitly** from a subclass.
2. **To reuse parent class properties** and avoid duplicate initialization.
3. **To ensure proper initialization** when dealing with inheritance.

Example of super() in a Constructor

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

   // Constructor of Parent class
   Person(String name, int age) {
      this.name = name;
      this.age = age;
      System.out.println("Person Constructor Called");
   }
}

// Child class
class Student extends Person {
   int studentID;

   // Constructor of Child class using super()
   Student(String name, int age, int studentID) {
        super(name, age); // Calls the constructor of Person class
        this.studentID = studentID;
        System.out.println("Student Constructor Called");
    }

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

    public static void main(String[] args) {
        Student s1 = new Student("John", 20, 101);
        s1.display();
    }
}
```

Output:

```
Person Constructor Called
Student Constructor Called
Name: John, Age: 20, Student ID: 101
```

Explanation of super()

1. When a **Student object** is created, the Student constructor is called.
2. Inside the Student constructor, super(name, age); is used to **call the constructor of the parent class (Person)**.
3. The **Person constructor executes first**, setting name and age.
4. Then, the **Student constructor continues execution**, setting studentID.

Key Points About super()

- **Must be the first statement** inside a constructor.
- **If super() is not written explicitly**, Java calls the **default constructor** of the parent class automatically.
- **Ensures that superclass attributes are initialized before subclass attributes**.

---

16. Java Program for Constructor Overloading in Student Class

Constructor overloading allows a class to **have multiple constructors** with different parameters, enabling **flexible object creation**.

Java Program: Student Class with Constructor Overloading

```
class Student {
    int studentID;
    String name;
    double marks;

    // Default Constructor
    Student() {
        this.studentID = 0;
        this.name = "Unknown";
        this.marks = 0.0;
        System.out.println("Default Constructor Called");
    }

    // Constructor with Student ID and Name
    Student(int studentID, String name) {
        this.studentID = studentID;
        this.name = name;
        this.marks = 0.0; // Default marks
        System.out.println("Constructor with ID & Name Called");
    }

    // Constructor with Student ID, Name, and Marks
    Student(int studentID, String name, double marks) {
        this.studentID = studentID;
        this.name = name;
        this.marks = marks;
        System.out.println("Constructor with ID, Name & Marks Called");
    }
    void display() {
        System.out.println("Student ID: " + studentID + ", Name: " + name + ", Marks: " + marks);
    }

    public static void main(String[] args) {
        // Creating objects using different constructors
        Student s1 = new Student(); // Calls Default Constructor
        Student s2 = new Student(101, "Alice"); // Calls Constructor with ID & Name
        Student s3 = new Student(102, "Bob", 85.5); // Calls Constructor with ID, Name & Marks

        // Displaying student details
        s1.display();
        s2.display();
        s3.display();
    }
}
```

Output:

```
Default Constructor Called
Constructor with ID & Name Called
Constructor with ID, Name & Marks Called
Student ID: 0, Name: Unknown, Marks: 0.0
Student ID: 101, Name: Alice, Marks: 0.0
Student ID: 102, Name: Bob, Marks: 85.5
```

Explanation of Constructor Overloading

1. **Default Constructor (Student())**
   - Initializes **default values** (0, "Unknown", 0.0).
   - Used when no parameters are provided.
2. **Parameterized Constructor (Student(int, String))**
   - Initializes **Student ID and Name**.

  - Marks are set to 0.0 by default.

3. **Fully Parameterized Constructor (Student(int, String, double))**
   - Initializes **all attributes (Student ID, Name, and Marks)**.

Advantages of Constructor Overloading

- **Improves code flexibility** by allowing different ways to initialize an object.
- **Reduces redundant code**, as each constructor calls the appropriate initialization.
- **Enhances readability and maintainability**.

Key Takeaways

super() in Constructors

- Used to **call the constructor of the parent class**.
- Helps in **reusing initialization logic** in inheritance.
- Must be the **first statement** in a constructor.

Constructor Overloading

- Allows multiple constructors with **different parameter lists**.
- Enables **flexible object creation**.
- Improves **code reusability and clarity**.

17. Write a Java Program Demonstrating super() in a Constructor

In Java, when a child class extends a parent class, the child class can use super() to call the **constructor of the parent class**. This ensures that the **parent class is properly initialized before the child class** executes its constructor logic.

Example Program

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

    // Constructor of Parent class
    Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Person Constructor Called");
    }

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

// Child class
class Student extends Person {
    int studentID;

    // Constructor of Child class using super()
    Student(String name, int age, int studentID) {
        super(name, age); // Calls the constructor of Person class
        this.studentID = studentID;
        System.out.println("Student Constructor Called");
    }

    void display() {
        super.display(); // Calls the display method of Person class
        System.out.println("Student ID: " + studentID);
    }

    public static void main(String[] args) {
        Student s1 = new Student("Alice", 20, 101); // Object creation
        s1.display();
    }
}
```

Output:

```
Person Constructor Called
Student Constructor Called
```

```
Name: Alice, Age: 20

Student ID: 101
```

Explanation

1. **The Student constructor is called** when an object is created.
2. **Inside the Student constructor, super(name, age); is used** to call the Person constructor.
3. The **Person constructor initializes the name and age variables** and prints "Person Constructor Called".
4. After the parent class is initialized, the **child class sets studentID** and prints "Student Constructor Called".
5. The **display() method** in Student calls the parent class display() using super.display() before printing studentID.

---

18. Explain Constructor Overloading Example in a Person Class

Constructor overloading allows a class to **have multiple constructors** with different parameters. It improves **code flexibility** by enabling object creation in different ways.

Example Program

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

  // Constructor 1: Name only
  Person(String name) {
    this.name = name;
    this.age = 0; // Default age
    this.address = "Not Provided"; // Default address
    System.out.println("Constructor with Name Called");
  }

  // Constructor 2: Name and Age
  Person(String name, int age) {
    this.name = name;
```

```
        this.age = age;
        this.address = "Not Provided"; // Default address
        System.out.println("Constructor with Name and Age Called");
    }

    // Constructor 3: Name and Address
    Person(String name, String address) {
        this.name = name;
        this.age = 0; // Default age
        this.address = address;
        System.out.println("Constructor with Name and Address Called");
    }

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

    public static void main(String[] args) {
        // Creating objects using different constructors
        Person p1 = new Person("John");
        Person p2 = new Person("Alice", 25);
        Person p3 = new Person("Bob", "New York");

        // Displaying details
        p1.display();
        p2.display();
        p3.display();
    }
}
```

Output:

```
Constructor with Name Called
Constructor with Name and Age Called
Constructor with Name and Address Called
Name: John, Age: 0, Address: Not Provided
Name: Alice, Age: 25, Address: Not Provided
Name: Bob, Age: 0, Address: New York
```

Explanation

1. **Three constructors** are defined for different ways of initializing a Person object.
2. The constructor **overloaded with different parameter lists** allows object creation with:
    - **Only name**
    - **Name and age**
    - **Name and address**
3. When objects are created, the **appropriate constructor is called**, and **default values are assigned** where needed.

---

19. List and explain different Types of Nested Classes in Java

A **nested class** is a class defined inside another class. It is used to **group logically related classes together**, improve encapsulation, and make code more readable.

Types of Nested Classes

Java provides two main types of nested classes:

1. Static Nested Class

- Declared using the static keyword.
- **Does not have access to instance variables and methods** of the outer class.
- Can be instantiated **without creating an instance of the outer class**.

Example: Static Nested Class

```java
class OuterClass {
  static class StaticNested {
    void display() {
      System.out.println("Inside Static Nested Class");
    }
  }

  public static void main(String[] args) {
        OuterClass.StaticNested obj = new OuterClass.StaticNested();
        obj.display();
    }
}
```

Output:

```
Inside Static Nested Class
```

2. Non-Static (Inner) Class

- **Has access to all members** (including private members) of the outer class.
- Requires an instance of the outer class to be instantiated.

Example: Non-Static Inner Class

```
class OuterClass {
    private String message = "Hello from Outer Class";

    class InnerClass {
        void display() {
            System.out.println(message); // Accessing outer class private variable
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

Output:

```
Hello from Outer Class
```

3. Local Inner Class

- Defined **inside a method**.
- **Cannot have access modifiers** (like public, private, protected).
- **Can access local variables** of the method **if they are final or effectively final**.

Example: Local Inner Class

```
class OuterClass {
    void outerMethod() {
        class LocalInner {
            void display() {
                System.out.println("Inside Local Inner Class");
            }
        }
        LocalInner obj = new LocalInner();
        obj.display();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

Output:

```
Inside Local Inner Class
```

4. Anonymous Inner Class

- **Does not have a name**.
- **Used when a class needs to be instantiated only once**.
- **Created by extending a class or implementing an interface**.

Example: Anonymous Inner Class

```
abstract class Animal {
    abstract void makeSound();
}

public class Main {
    public static void main(String[] args) {
        // Anonymous Inner Class implementing makeSound() method
        Animal dog = new Animal() {
            void makeSound() {
                System.out.println("Woof! Woof!");
            }
        };
        dog.makeSound();
    }
}
```

Output:

```
Woof! Woof!
```

Summary

Nested Class Type	**Characteristics**
Static Nested Class	Independent of the outer class, can be accessed without an instance of the outer class.
Non-Static Inner Class	Needs an instance of the outer class, has access to all members of the outer class.
Local Inner Class	Defined inside a method, can access local variables if they are final or effectively final.
Anonymous Inner Class	No name, instantiated once, used for overriding methods of an abstract class or interface.

---

20. Can a nested class access private members of its enclosing class? Explain with an example.

Yes, a **nested class** can access the **private members** of its enclosing (outer) class. This is possible because a nested class is considered a part of the enclosing class, allowing it to access even **private variables and methods** of the outer class.

Example: Nested Class Accessing Private Members

```
class Outer {
    private String message = "Hello from Outer Class"; // Private member

    // Nested Inner Class
    class Inner {
        void display() {
            // Accessing private member of Outer class
            System.out.println("Message from Outer: " + message);
        }
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner(); // Creating an instance of Inner class
        inner.display(); // Calling method of Inner class
    }
}
```

Output:

```
Message from Outer: Hello from Outer Class
```

Why Can a Nested Class Access Private Members?

- **Nested classes** are part of the same **scope** as the outer class.
- The Java compiler allows them to access **private fields and methods** of the outer class as if they were part of it.

Types of Nested Classes That Can Access Private Members

1. **Member Inner Class** (as shown above)
2. **Static Nested Class** (via an outer class instance)
3. **Local Inner Class** (declared inside a method)
4. **Anonymous Inner Class** (declared inline)

Example: Static Nested Class Accessing Private Members

Even a **static nested class** can access private members, but only through an **instance of the outer class**.

```
class Outer {
    private String message = "Hello from Outer Class";
    // Static Nested Class
    static class StaticNested {
        void display() {
            Outer outer = new Outer(); // Creating an instance of Outer class
            System.out.println("Message: " + outer.message);
        }
    }

    public static void main(String[] args) {
        Outer.StaticNested nested = new Outer.StaticNested();
        nested.display();
    }
}
```

Output:

```
Message: Hello from Outer Class
```

---

21. What is the difference between a Local Inner Class and an Anonymous Inner Class?

Local Inner Class

- A **named** inner class declared inside a method.
- Can **extend a class or implement an interface**.
- Can **have multiple methods**.
- Requires **explicit instantiation** within the method.

Anonymous Inner Class

- A **class without a name**, declared and instantiated in a single step.
- **Always extends a class or implements an interface**.
- **Cannot define multiple methods explicitly** (only method overrides are allowed).
- **No explicit constructor** because it has no name.

Example for Local Inner Class

```
class Outer {
    void outerMethod() {
        class LocalInner {
            void display() {
                System.out.println("Inside Local Inner Class");
            }
        }
        LocalInner obj = new LocalInner();
        obj.display();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
}
```

Example for Anonymous Inner Class

```
interface Greeting {
    void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greeting obj = new Greeting() {
            public void sayHello() {
                System.out.println("Hello from Anonymous Inner Class");
            }
        };
        obj.sayHello();
    }
}
```

22. Write a Java program to demonstrate a Member Inner Class and access its methods.

A **member inner class** is a **non-static inner class** that is defined inside another class but outside any method.

Example Program

```
class OuterClass {
    private String message = "Hello from Outer Class";

    // Member Inner Class
    class InnerClass {
        void display() {
            System.out.println(message); // Accessing outer class private member
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

Output:

```
Hello from Outer Class
```

---

23. Explain the advantages and disadvantages of using nested classes in Java.

Advantages:

1. **Encapsulation:** Helps in **hiding implementation details** from the outer world.
2. **Logical Grouping:** If a class is useful only for one other class, **grouping them together improves readability**.
3. **Code Organization:** Nested classes help **keep the code structured and modular**.
4. **Access to Private Members:** Inner classes can **access private members of the outer class**, reducing the need for getters/setters.

Disadvantages:

1. **Reduced Readability:** Code can become **complex and harder to understand**.
2. **Increased Coupling:** The inner class **depends on the outer class**, reducing flexibility.
3. **Difficult Debugging:** Debugging nested classes can be **tricky due to multiple levels of code dependencies**.
4. **Memory Overhead:** If used **improperly**, nested classes **increase memory consumption**.

---

24. Explain Local Inner Classes with an example. How are they different from Member Inner Classes?

Local Inner Class

- A **class defined inside a method**.
- Can only be **accessed within the method** where it is defined.
- Cannot have **static members**.

Example of Local Inner Class

```
class Outer {
    void show() {
        class LocalInner {
            void display() {
                System.out.println("This is a Local Inner Class");
            }
        }
        LocalInner obj = new LocalInner();
        obj.display();
    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.show();
    }
}
```

Difference Between Local and Member Inner Class

Feature	Local Inner Class	Member Inner Class
Where Declared	Inside a method	Inside a class but outside methods
Access Scope	Only inside the method	Accessible from all methods of the outer class
Static Members	Not allowed	Not allowed
Instantiation	Created inside the method	Created inside or outside methods using an outer class object

---

25. What is an Anonymous Inner Class? Write a Java program that demonstrates its use in implementing an interface.

An **anonymous inner class** is a **class without a name**, used to extend a class or implement an interface **inline**.

Example: Implementing an Interface with Anonymous Inner Class

```
interface Vehicle {
    void start();
}


public class Main {
    public static void main(String[] args) {
        Vehicle car = new Vehicle() {
            public void start() {
                System.out.println("Car is starting...");
            }
        };
        car.start();
    }
}
```

Output:

```
Car is starting...
```

Key Characteristics of Anonymous Inner Class

1. **No explicit name**.
2. **Used when a class is needed only once**.

3. **Can extend a class or implement an interface**.
4. **Cannot have constructors**.
5. **Declared and instantiated in a single step**.

---

26. What is the purpose of an abstract method? Provide an example.

Purpose of an Abstract Method

An **abstract method** is a method that **does not have a body** and must be implemented by subclasses. It is used in **abstract classes** to enforce that subclasses **must provide their own implementation** for certain behaviors.

- Declared using the abstract keyword.
- Cannot have a body ({}).
- Must be implemented in a subclass.
- Helps achieve **runtime polymorphism**.

Example: Using an Abstract Method

```
// Abstract class
abstract class Animal {
    abstract void makeSound(); // Abstract method (no body)

    void sleep() { // Concrete method (has body)
        System.out.println("Sleeping...");
    }
}

// Subclass providing implementation
class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark! Bark!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound(); // Calls the overridden method
        myDog.sleep(); // Calls the inherited concrete method
    }
}
```

Output:

```
Bark! Bark!
Sleeping...
```

Key Points:

✓ An abstract method **must be overridden** in a subclass.
✓ Abstract classes **can have both abstract and concrete methods**.
✓ Cannot create an **instance** of an abstract class.

---

27. What is the difference between a final variable, final method, and final class?

Feature	Final Variable	Final Method	Final Class
Definition	A variable whose value **cannot be changed** after initialization.	A method that **cannot be overridden** in subclasses.	A class that **cannot be inherited**.
Keyword	final int x = 10;	final void display() {}	final class MyClass {}
Usage	Prevents **reassignment**.	Prevents **method overriding**.	Prevents **inheritance**.
Example	final int speed = 100;	final void show() {}	final class Vehicle {}
When to Use	When a value **must remain constant**.	When you want to **prevent subclasses from changing behavior**.	When you want to **prevent modification of a class structure**.

Example of Final Variable

```
class Car {
    final int speed = 100; // Cannot be modified

    void displaySpeed() {
        // speed = 120;  // ERROR! Cannot change final variable
        System.out.println("Speed: " + speed);
    }
}
```

Example of Final Method

```
class Vehicle {
    final void show() { // Cannot be overridden
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {
    // void show() { // ERROR! Cannot override final method
    //     System.out.println("Car is running");
    // }
}
```

Example of Final Class

```
final class Bike {
    void run() {
        System.out.println("Bike is running");
    }
}

// class SportsBike extends Bike { // ERROR! Cannot inherit final class
// }
```

Key Takeaways

✓ **Final Variable:** Prevents value changes after initialization.
✓ **Final Method:** Prevents method overriding in child classes.
✓ **Final Class:** Prevents inheritance, ensuring the class structure is not altered.

28. Can a final class be subclassed? Explain with an example.

No, a final class **cannot** be subclassed in Java. When a class is declared as final, it means that it cannot be extended by any other class, ensuring its implementation remains unchanged.

Example:

```
final class FinalClass {
    void showMessage() {
        System.out.println("This is a final class.");
    }
}

// This will cause a compilation error
class SubClass extends FinalClass {  // ERROR: Cannot inherit from final class
    void display() {
        System.out.println("Trying to extend a final class.");
    }
}
```

Explanation:
Attempting to extend FinalClass will result in a compilation error because final prevents subclassing.

---

29. Write a Java program demonstrating the use of an abstract class and abstract method.

An abstract class in Java is a class that cannot be instantiated and may contain abstract methods (methods without a body).

Example:

```
abstract class Animal {
    abstract void makeSound();  // Abstract method

    void sleep() {  // Concrete method
        System.out.println("Sleeping...");
    }
}

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

public class AbstractDemo {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound(); // Calls Dog's implementation
        myDog.sleep(); // Calls inherited method from Animal
    }
}
```

Output:

```
Dog barks
Sleeping...
```

---

30. Explain with examples how the final keyword is used for variables, methods, and classes.

The final keyword in Java can be used in three ways:

1. **Final Variable**: Once assigned, its value cannot be changed.
2. **Final Method**: Cannot be overridden by subclasses.
3. **Final Class**: Cannot be extended (inherited).

Example:

```
class Example {
    // Final variable
    final int MAX_VALUE = 100;

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

// Attempting to override a final method will cause a compilation error
class SubExample extends Example {
    // void display() { System.out.println("Overriding final method."); } // ERROR
}

// Final class cannot be subclassed
final class FinalClass {
    void show() {
        System.out.println("This is a final class.");
    }
}

// class SubFinalClass extends FinalClass { } // ERROR: Cannot extend final class

public class FinalKeywordDemo {
    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println("Final Variable: " + obj.MAX_VALUE);
        obj.display();
    }
}
```

Output:

```
Final Variable: 100

This is a final method.
```

---

31. Create an abstract class Shape with an abstract method area(). Implement subclasses Circle and Rectangle to calculate and display their respective areas

Example:

```
abstract class Shape {
    abstract void area(); // Abstract method
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    void area() {
        System.out.println("Circle Area: " + (Math.PI * radius * radius));
    }
}

class Rectangle extends Shape {
    double length, width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    void area() {
        System.out.println("Rectangle Area: " + (length * width));
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        Shape circle = new Circle(5);
        Shape rectangle = new Rectangle(4, 6);

        circle.area();
        rectangle.area();
    }
}
```

Output:

```
Circle Area: 78.53981633974483
Rectangle Area: 24.0
```

---

32. Write a Java program to create an abstract class Vehicle with an abstract method start(), and implement it in subclasses Car and Bike.

Example:

```
abstract class Vehicle {
    abstract void start(); // Abstract method
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with a key.");
    }
}

class Bike extends Vehicle {
    void start() {
        System.out.println("Bike starts with a kick.");
    }
}
```

```
public class VehicleDemo {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        myCar.start();
        myBike.start();
    }
}
```

Output:

```
Car starts with a key.
Bike starts with a kick.
```