1. What is Java? Developed by Whom?

- **What is Java?**
  - Java is a **high-level**, **object-oriented**, **platform-independent**, and **general-purpose** programming language.
  - It is designed to have as few implementation dependencies as possible, making it **write once, run anywhere (WORA)**.
  - Java is widely used for building **web applications**, **mobile applications (Android)**, **enterprise applications**, and **big data technologies**.
- **Developed by Whom?**
  - Java was developed by **James Gosling** and his team at **Sun Microsystems** in 1991.
  - It was officially released in **1995**.
  - Sun Microsystems was later acquired by **Oracle Corporation** in 2010, and Oracle now maintains Java.

---

2. What is oops?

OOPs (Object-Oriented Programming) is a programming paradigm that organizes software design around objects rather than functions and logic.

It is based on four key principles:

1. **Encapsulation:** Bundling data and methods that operate on the data into a single unit (class).
2. **Inheritance:** Creating new classes (subclasses) from existing classes (superclasses) to reuse code.
3. **Polymorphism:** Allowing objects to take on multiple forms (e.g., method overriding and overloading).
4. **Abstraction:** Hiding implementation details and exposing only essential features.

---

3. What is the need of oops?

Need of OOPs:

1. **Modularity:** Breaks down complex problems into smaller, manageable objects.
2. **Reusability:** Promotes code reuse through inheritance and polymorphism.
3. **Maintainability:** Makes code easier to maintain and extend.
4. **Scalability:** Allows for easier scaling of applications.
5. **Real-World Modeling:** Represents real-world entities as objects, making it easier to model complex systems.

---

4. Advantages of object oriented language over procedure oriented language.

Aspect	Object-Oriented Language	Procedural-Oriented Language
Approach	Focuses on objects and their interactions.	Focuses on functions and procedures.
Code Reusability	High (through inheritance and polymorphism).	Low (functions are not reusable across programs).
Maintainability	Easier to maintain and extend.	Harder to maintain as the program grows.
Real-World Modeling	Better at modeling real-world entities.	Less effective for modeling real-world systems.
Data Security	Provides encapsulation to secure data.	No built-in mechanism for data security.
Complexity Management	Easier to manage complex systems.	Harder to manage as complexity increases.

---

5. Why the name java? What is the previous name of this programming language?

- **Why the Name Java?**
  - The name Java was chosen during a brainstorming session by the development team.
  - It is named after Java coffee, a type of coffee from Indonesia, reflecting the team's love for coffee.
  - The name was chosen because it was simple, unique, and easy to spell.
- **Previous Name:**
  - Java was originally called Oak.
  - The name Oak was chosen because of an oak tree that stood outside James Gosling's office.
  - However, the name was later changed to Java because Oak was already trademarked by another company.

---

6. What are tokens? (Identifiers, Keywords, DataTypes, Operators)

- **Tokens** are the smallest individual units in a Java program. They are the building blocks of a Java program.
- **Tokens in Java include:**
    1. **Identifiers:**
        - Names given to variables, methods, classes, etc.
        - **Rules for identifiers:**
            - Must start with a letter, _, or `$`.
            - Cannot start with a digit.
            - Cannot use Java keywords.
            - **Example:** `int count;` (here, `count` is an identifier).
    2. **Keywords:**
        - Reserved words in Java that have a specific meaning and cannot be used as identifiers.
        - **Examples:** `class`, `public`, `static`, `void`, `int`, etc.
    3. **Data Types:**
        - Define the type of data a variable can hold.
        - **Examples:** `int`, `double`, `char`, `boolean`, etc.
    4. **Operators:**
        - Symbols used to perform operations on variables and values.
        - **Examples:** `+`, `-`, `*`, `/`, `==`, `!=`, etc.

---

7. What is method?

- Method is a set of statements written for a specific business task.
- Methods are created within a class and executed when they are called and methods can be called from other methods.

---

8. What are the categories of methods?

- **Methods in Java can be categorized into:**
- **Static Methods:**
    - Methods that belong to the class rather than an instance of the class.
    - Called using the class name.
- **Non-Static Methods:**
    - Methods that belong to an instance of a class.
    - Called using an object of the class.

---

9. What are the features of oops?

- **The features of OOPs are the four pillars of object-oriented programming:**
    1. **Encapsulation:**
        - Bundling data (attributes) and methods (behavior) into a single unit (class).
        - **Example:** A `BankAccount` class with attributes (`balance`) and methods (`deposit()`, `withdraw()`).
    2. **Inheritance:**
        - Creating new classes (subclasses) from existing classes (superclasses) to reuse code.
        - **Example:** A `Dog` class inheriting from an `Animal` class.
    3. **Polymorphism:**
        - Allowing objects to take on multiple forms (e.g., method overriding and overloading).
        - **Example:** A `Shape` class with a `draw()` method that behaves differently for `Circle` and `Rectangle` objects.
    4. **Abstraction:**
        - Hiding implementation details and exposing only essential features.
        - **Example:** An abstract class `Shape` with an abstract method `draw()`.

---

10. What is an object?

- Any real world entity involved in the business is known as Object.
- For every real world entity, a java object created in JVM.
- A real world entity has state and behavior, a java object has variables and methods, variables represents state and methods represents behavior of a real world entity.
- Before creating an object, a design is required in java class which acts as a design for an object.

---

11. What is class?

- A **class** is a blueprint or template for creating objects.
- It defines the **state (attributes)** and **behavior (methods)** that the objects of the class will have.
- A class is declared using the `class` keyword.

---

12. What are the types of variables in oops?

In Java, variables in OOPs can be categorized into three types:

1. Instance Variables:
    - Variables declared inside a class but outside any method.
    - They belong to an instance (object) of the class.
    - Each object has its own copy of instance variables.
2. Static Variables:
    - Variables declared with the `static` keyword.
    - They belong to the class rather than any specific instance.
    - Shared across all instances of the class.
3. Local Variables:
    - Variables declared inside a method, constructor, or block.
    - They are accessible only within the scope in which they are declared.

---

13. Explain static and non-static?

Static:

- Belongs to the **class** rather than an instance of the class.
- Can be accessed using the class name.
- Shared across all instances of the class.

Non-Static:

- Belongs to an **instance** of the class.
- Can be accessed using an object of the class.
- Each object has its own copy of non-static variables and methods.

---

14. Can we have more than one reference for an object?

- **Yes**, we can have more than one reference pointing to the same object.
- This is useful when you want to access the same object from different parts of the program.

```
Dog dog1 = new Dog();
Dog dog2 = dog1; // Both dog1 and dog2 refer to the same object
dog1.name = "Buddy";
System.out.println(dog2.name); // Output: Buddy
```

---

15. Can we have more than one reference pointing to more than one object?

- **Yes**, we can have multiple references, each pointing to different objects.
- Each reference will point to a separate instance of the class.

```
Dog dog1 = new Dog();
Dog dog2 = new Dog(); // dog1 and dog2 refer to different objects
dog1.name = "Buddy";
dog2.name = "Max";
System.out.println(dog1.name); // Output: Buddy
System.out.println(dog2.name); // Output: Max
```

---

16. What is anonymous object or abandoned objects?

- An object that is created without assigning it to any reference variable.
- It is used for **one-time use** and cannot be reused later in the program.
- It becomes eligible for **garbage collection** (memory cleanup) by the JVM.

---

17. Explain JVM?

- **JVM** is the **runtime engine** that executes Java bytecode.
- It is a part of the **Java Runtime Environment (JRE)**.
- **Functions of JVM**:
    - **Loads Code**: Reads the `.class` file (bytecode).
    - **Verifies Code**: Ensures the bytecode is valid and secure.
    - **Executes Code**: Converts bytecode into machine-specific instructions.
    - **Provides Runtime Environment**: Manages memory, garbage collection, and exception handling.
- **Key Features**:
    - **Platform Independence**: Java programs can run on any device with a JVM.
    - **Memory Management**: Automatically allocates and deallocates memory.

---

18. Explain JDK?

- **JDK** is a software development environment used to develop Java applications.
- It includes:
    - **Compiler (`javac`)**: Converts Java source code (`*.java`) into bytecode (`*.class`).
    - **JRE (Java Runtime Environment)**: Provides the runtime environment to execute Java programs.

    - **Development Tools**: Includes tools like `javadoc`, `jar`, `jdb`, etc.
- **Purpose**:
    - Used by developers to write, compile, and debug Java programs.

---

19. Explain JRE?

- **JRE** is the runtime environment that provides the libraries, JVM, and other components required to **run** Java programs.
- It does **not** include development tools like the compiler (`javac`).
- **Components of JRE**:
    - **JVM**: Executes the bytecode.
    - **Libraries**: Provides standard Java libraries (e.g., `java.lang`, `java.util`).
    - **Other Files**: Support files like property files and resource files.
- **Purpose**:
    - Used by end-users to **run** Java applications.

---

20. What is byte code?

- **Bytecode** is the intermediate code generated by the Java compiler (`javac`) from the source code (`*.java`).
- It is a set of instructions that the **JVM** can understand and execute.
- **Characteristics**:
    - **Platform-Independent**: Bytecode can run on any platform with a JVM.
    - **Efficient**: Optimized for execution by the JVM.
    - **File Extension**: Bytecode is stored in `.class` files.
- **Example**:
    - When you compile `HelloWorld.java`, it generates `HelloWorld.class` (bytecode).

---

21. What is native code?

- **Native code** is machine-specific code that is directly executed by the CPU.
- It is **platform-dependent** and is generated by compiling high-level languages like C or C++.
- In Java, native code is used in the **Java Native Interface (JNI)** to interact with platform-specific libraries or hardware.
- Example: Methods marked with the `native` keyword in Java are implemented in native code (e.g., C/C++).

---

22. Who are Platform Dependent and who are Platform Independent?

- **Platform Dependent**:
  - Programs or code that can run only on a specific platform (operating system or hardware).
  - Example: Native code (C/C++ binaries), as they are compiled for a specific platform.
- **Platform Independent**:
  - Programs or code that can run on any platform without modification.
  - Example: Java bytecode, as it is executed by the JVM, which is platform-independent.

---

23. What is recursion?

- **Recursion** is a programming technique where a method calls itself to solve a problem.
- It is used to break down complex problems into smaller, more manageable subproblems.
- **Key Components**:
  1. **Base Case**: The condition that stops the recursion.
  2. **Recursive Case**: The part where the method calls itself.

---

24. In which memory is JVM executing?

- The **JVM** executes in the **RAM (Random Access Memory)** of the system.
- It allocates memory for:
  1. **Method Area**: Stores class-level data (e.g., bytecode, static variables).
  2. **Heap**: Stores objects and instance variables.
  3. **Stack**: Stores method calls, local variables, and partial results.
  4. **PC Register**: Stores the address of the currently executing instruction.
  5. **Native Method Stack**: Used for native methods (e.g., C/C++ code).

---

25. In which memory is the main method executing?

- The **main method** executes in the **stack memory** of the JVM.
- Each method call (including `main`) creates a **stack frame** in the stack memory, which stores:
  - Local variables, Method parameters, Return address.

---

26. Objects are created in which memory?

- Objects are created in the **heap memory** of the JVM.
- The heap is shared across all threads and is used for dynamic memory allocation.

---

27. Explain method overloading?

- Creating multiple methods with the same name and different formal arguments is known as Method Overloading.
- In method overloading java matches method names and arguments not a return type.
- While Method Overloading formal argument or formal argument's length should be different.
- **When to use:**
    - When we have to perform different implementations for the same behavior, when arguments are different, we'll go for method overloading.
- **Advantages:**
    - By using method overloading we can achieve compile time polymorphism.
    - By using method overloading we can achieve readability.

---

28. Explain constructor?

- A **constructor** is a special method used to **initialize objects**.
- It is called automatically when an object is created.
- **Key Features**:
    1. It has the **same name** as the class.
    2. It does **not** have a return type (not even `void`).
    3. It can be **overloaded** (multiple constructors with different parameters).

---

29. What is the default constructor?

- The **default constructor** is a no-argument constructor that is automatically provided by Java if no constructor is defined in the class.
- It initializes the object with default values (e.g., `null` for objects, `0` for integers).

---

30. What are the types of constructors?

- There are two types of constructors in Java:
    1. **Default Constructor**:
        - A no-argument constructor provided by Java if no constructor is defined.

```
class Dog {
    Dog() { // Default constructor
    }
}
```

    2. **Parameterized Constructor**:
        - A constructor that takes parameters to initialize the object.

```
class Dog {
    String name;
    Dog(String name) { // Parameterized constructor
        this.name = name;
    }
}
```

---

31. Explain this keyword?

- `This` keyword is used to refer to non-static members of a current instance from non - static methods or constructors.
- `This` keyword can't be used in a static context (because the static method doesn't run for objects).
- If there isn't a local member then `This` keyword is not mandatory, as java will refer to the current instance only.

It is used to:

1. **Differentiate between instance variables and parameters** when they have the same name.
2. **Call one constructor from another constructor** in the same class (using `this()`).
3. **Pass the current object** as a parameter to another method.

---

32. Explain constructor overloading?

- **Constructor Overloading** is a feature in Java that allows a class to have multiple constructors with **different parameter lists**.
- It is used to initialize objects in different ways.

---

33. Explain this()?

- `This() - [Call To This]`
- It's a constructor calling statement.
- Used to call one constructor to another constructor within the same class.
- Should always be the first line of a constructor.

---

34. Explain copy constructor?

- Used to copy the values from one object to another object.
- A **copy constructor** is a constructor that creates an object by copying the state of another object of the same class.
- It is used to create a **deep copy** of an object.

---

35. Can we have class names as data types?

- **Yes**, class names can be used as data types in Java.
- When a class is defined, it becomes a **reference data type**.

---

36. Why is Java not a completely object-oriented Language?

- Java is **not a completely object-oriented language** because:
    1. It supports **primitive data types** (e.g., `int`, `char`, `boolean`), which are not objects.
    2. It allows **static methods and variables**, which belong to the class rather than an instance.
    3. It does not support **multiple inheritance** for classes (though it supports multiple inheritance through interfaces).

---

37. What is Association?

- **Association** is a relationship between two classes where one class is related to another class.
- It can be:
    1. **One-to-One**: One object of a class is associated with one object of another class.
    2. **One-to-Many**: One object of a class is associated with multiple objects of another class.
    3. **Many-to-Many**: Multiple objects of a class are associated with multiple objects of another class.

```java
class Teacher {
    String name;
}
class Student {
    String name;
    Teacher teacher; // Association
}
```

---

38. What is Aggregation and Composition?

Aggregation:

- A **weak relationship** where one class contains a reference to another class, but the contained object can exist independently.
- Aggregation is a form of Has-A relationship, we can achieve aggregation by placing one class reference into another class.
- In this, one object will exist without another object.
- It represents a weak dependency between two objects.
- Example:
    - A `Department` class contains a list of `Employee` objects.
    - Car and Music Player
    - Mobile and Sim Card

---

39. What is Inheritance?

- The process of acquiring properties from one class `(Super Class)` to another class `(Sub Class)` is known as Inheritance.
- Also known as `Is-A` relationship
- We can achieve Is-A relationship by using `extends` keyword.
- It promotes **code reusability** and **method overriding**.

---

40. What are the types of Inheritance?

- There are **five types of inheritance**:
    1. **Single Inheritance**: A subclass inherits from one superclass.
        - Example: `Dog` ⟶ `Animal`.
    2. **Multilevel Inheritance**: A subclass inherits from a superclass, which in turn inherits from another superclass.
        - Example: `Puppy` ⟶ `Dog` ⟶ `Animal`.
    3. **Hierarchical Inheritance**: Multiple subclasses inherit from a single superclass.
        - Example: `Dog` ⟶ `Animal`, `Cat` ⟶ `Animal`.

4. **Multiple Inheritance**: A subclass inherits from multiple superclasses (not supported in Java for classes, but supported through interfaces).
    - Example: `class C implements A, B`.
5. **Hybrid Inheritance**: A combination of two or more types of inheritance (not directly supported in Java due to multiple inheritance restrictions).

---

41. What are the advantages of Inheritance?

- **Advantages of Inheritance**:
    1. **Code Reusability**: Reuse the code of the superclass in the subclass.
    2. **Method Overriding**: Subclasses can provide specific implementations of methods defined in the superclass.
    3. **Extensibility**: Easily extend the functionality of existing classes.
    4. **Maintainability**: Reduces code duplication and improves code organization.
    5. **Polymorphism**: Enables objects of different classes to be treated as objects of a common superclass.

---

42. What is method overriding?

Method Overriding: The process of changing or providing new implementation for superclass non-static methods in subclass.

Rules for Overriding:

1. Is-A relationship is mandatory to achieve method overriding.
2. The method must have the **same name** and **parameters** as the method in the superclass.
3. The method in the subclass must have the **same or broader access modifier**.
4. The **return type** can be the same or co-variant type.
5. The `@Override` annotation is used to indicate that a method is being overridden.
6. In java, we can only override non-static, non-final, non-private methods.
7. In java, we can't override constructors, blocks, static, final, private methods.

---

43. Can we Override static, private and final methods?If not, why?

- **Static Methods**:
    - **Cannot be overridden** because they belong to the class, not the instance.
    - They can be **hidden** in the subclass by defining a method with the same signature.
- **Private Methods**:
    - **Cannot be overridden** because they are not accessible outside the class.

- **Final Methods**:
    - **Cannot be overridden** because the `final` keyword prevents method overriding.

---

44. What are covariant and contra-variant return types?

Covariant Return Types:

- In Java, a subclass can override a method and return a **subtype** of the return type declared in the superclass.

Contra-variant Return Types:

- Java does **not** support contra-variant return types (returning a supertype in the subclass).

---

45. Which is the supermost class of all the classes in the java hierarchy?

- The **`Object` class** is the supermost class of all classes in Java.
- Every class in Java directly or indirectly inherits from the `Object` class.

---

46. What are the methods of Object class?

- The `Object` class provides the following methods:
    1. `toString()`: Returns a string representation of the object.
    2. `equals(Object obj)`: Compares two objects for equality.
    3. `hashCode()`: Returns a hash code value for the object.
    4. `getClass()`: Returns the runtime class of the object.
    5. `clone()`: Creates and returns a copy of the object.
    6. `finalize()`: Called by the garbage collector before an object is reclaimed.
    7. `wait()`, `notify()`, `notifyAll()`: Used for thread synchronization.

---

47. List the final methods of Object class?

- The **final methods** of the `Object` class are:
    1. `getClass()`
    2. `wait()`
    3. `wait(long timeout)`
    4. `wait(long timeout, int nanos)`
    5. `notify()`
    6. `notifyAll()`

---

48. Overriding is used for ?

- **Overriding** is used to:
    1. Provide a **specific implementation** of a method in the subclass.
    2. Achieve **runtime polymorphism**.
    3. Modify or extend the behavior of the superclass method.

---

49. Overriding is binded at compile-time or run-time?

- **Overriding** is **binded at run-time** (dynamic binding).
- The method to be executed is determined by the **actual object type** at runtime, not the reference type.

---

50. What are Annotations?

- **Annotations** are metadata that provide information about the code to the compiler, runtime, or other tools.
- They start with the `@` symbol.
- **Common Annotations**:
    1. `@Override`: Indicates that a method is overriding a superclass method.
    2. `@FunctionalInterface`: Indicates that an interface is a functional interface (has exactly one abstract method).

---

51. Which class objects should be created in Inheritance?

- In inheritance, you can create objects of **both the superclass and the subclass**.
- However, the choice depends on the use case:
    1. **Superclass Object**: Used when you want to work with the general behavior defined in the superclass.
        - Example: `Animal animal = new Animal();`
    2. **Subclass Object**: Used when you want to work with the specific behavior defined in the subclass.
        - Example: `Dog dog = new Dog();`
    3. **Upcasting**: You can create a subclass object and assign it to a superclass reference.
        - Example: `Animal animal = new Dog();`

---

52. Can we create classes without Inheritance in java?

- **Yes**, you can create classes without inheritance in Java.
- A class that does not explicitly extend another class **implicitly extends the `Object` class** (the root class in Java).

---

53. What are Access modifiers?

- **Access modifiers** are keywords in Java that control the **visibility** and **accessibility** of classes, methods, and variables.
- They define the scope of a class, method, or variable.

---

54. List the access modifiers and their tasks?

Java has four access modifiers:

1. **`public`**:
    - Accessible from **anywhere**.
    - Example: `public int x;`
2. **`private`**:
    - Accessible only within the **same class**.
    - Example: `private int x;`
3. **`protected`**:
    - Accessible within the **same package** and by **subclasses** (even in different packages).
    - Example: `protected int x;`
4. **Default (no modifier)**:
    - Accessible only within the **same package**.
    - Example: `int x;`

---

55. What is a private constructor?

- A **private constructor** is a constructor that can only be accessed within the **same class**.
- It is used to:
    1. **Prevent object creation** from outside the class (e.g., in Singleton design pattern).
    2. **Restrict inheritance**.

```
class Singleton {
    private static Singleton instance;
    private Singleton() { // Private constructor
    }
```

```
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

---

56. Can I access private data members using objects of the same class?

- **Yes**, you can access private data members using objects of the **same class**.
- Private members are accessible only within the class where they are declared.

```
class MyClass {
    private int x = 10;

    void display(MyClass obj) {
        System.out.println(obj.x); // Accessing private member
    }
}
```

---

57. What is the factory method?

- A **factory method** is a design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.
- It is used to **decouple object creation** from the main logic of the application.

```
interface Vehicle {
    void drive();
}
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}
class VehicleFactory {
    public static Vehicle getVehicle(String type) {
        if (type.equalsIgnoreCase("car")) {
            return new Car();
        }
        return null;
    }
}
```

---

58. Explain super keyword?

- The **`super` keyword** is used to refer to the immediate **superclass** (parent class) of the current object from **subclass (**child class).
- It is used to:
    1. Access **superclass variables** and **methods**.
    2. Call the **superclass constructor** (using `super()`).

---

59. Explain super()?

- The **`super()`** keyword is used to call the immediate **superclass constructor** from the subclass constructor.
- It must be the **first statement** in the subclass constructor.
- It is used to **initialize the superclass part** of the object.

---

60. Difference between this/this() and super/super()?

Aspect	**`this/this()`**	**`super/super()`**
Purpose	Refers to the non-static members of the current **instance** of the class.	Refers to the properties of **superclass** of the current instance.
Usage	- `this`: Access current instance variables/methods. - `this()`: Call another constructor in the same class.	- `super`: Access superclass variables/methods. - `super()`: Call the superclass constructor.
Constructor Call	`this()` must be the **first statement** in the constructor.	`super()` must be the **first statement** in the constructor.
Example	`java class MyClass { MyClass() { this(10); } MyClass(int x) { } }`	`java class Dog extends Animal { Dog() { super(); } }`

---

61. What is upcasting and downcasting?

Upcasting:

- Casting a **subclass object** to a **superclass reference**.
- It is **implicit** (automatically done by the compiler).
- Example:

```
Animal animal = new Dog(); // Upcasting
```

Downcasting:

- Casting a **superclass reference** back to a **subclass reference**.
- It is **explicit** (requires manual casting).
- Example:

```
Dog dog = (Dog) animal; // Downcasting
```

---

62. Difference between upcasting and downcasting?

Aspect	Upcasting	Downcasting
Direction	Subclass → Superclass	Superclass → Subclass
Implicit/Explicit	Implicit (automatically done by compiler)	Explicit (requires manual casting)
Safety	Always safe (no runtime error)	May cause `ClassCastException` if invalid
Example	`Animal animal = new Dog();`	`Dog dog = (Dog) animal;`

---

63. Why is upcasting required?

- **Upcasting** is required to:
    1. Achieve **polymorphism**: Treat objects of different subclasses as objects of a common superclass.
    2. Write **generic code**: Code that works with the superclass can work with any subclass.
    3. Enable **dynamic method dispatch**: The method to be executed is determined at runtime based on the actual object type.

```
Animal animal = new Dog(); // Upcasting
animal.sound(); // Calls Dog's sound() method (runtime polymorphism)
```

---

64. What is Data Encapsulation?

- **Data Encapsulation** is the process of bundling data (attributes) and methods (behavior) into a single unit (class).
- It is achieved using **access modifiers** (e.g., `private`, `public`) to control access to the data.
- **Advantages**:
    1. **Data Hiding**: Protects data from unauthorized access.
    2. **Flexibility**: Allows changing the internal implementation without affecting the external interface.
    3. **Reusability**: Encapsulated classes can be reused in other programs.
- Example:

```
class BankAccount {
    private double balance; // Encapsulated data

    public void deposit(double amount) { // Encapsulated method
        balance += amount;
    }
}
```

---

65. What is Data Abstraction?

- **Data Abstraction** is the process of hiding the implementation details and showing only the essential features of an object.
- It is achieved using **abstract classes** and **interfaces**.
- **Advantages**:
    1. **Simplifies Complexity**: Hides unnecessary details.
    2. **Improves Maintainability**: Changes in implementation do not affect the external interface.
    3. **Promotes Reusability**: Abstract classes and interfaces can be reused across multiple classes.
- Example:

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

---

66. What is Recursion?

- **Recursion** is a programming technique where a method calls itself to solve a problem.
- It consists of:
    1. **Base Case**: The condition that stops the recursion.
    2. **Recursive Case**: The part where the method calls itself.
- Example:

```
int factorial(int n) {
    if (n == 0) { // Base case
        return 1;
    }
    return n * factorial(n - 1); // Recursive case
}
```

---

67. What are the advantages of recursion?

Advantages of Recursion:

1. **Simplifies Code**: Breaks down complex problems into smaller, manageable subproblems.
2. **Elegant Solutions**: Provides clean and concise solutions for problems like tree traversal, factorial, etc.
3. **Natural Fit**: Works well for problems that can be divided into similar subproblems (e.g., divide and conquer).

---

68. What are getter/accessor and setter/mutator?

Getter/Accessor:

- A method used to **read** the value of a private variable.
- Example:

```
public int getBalance() {
    return balance;
}
```

Setter/Mutator:

- A method used to **modify** the value of a private variable.
- Example:

```
public void setBalance(double balance) {
    this.balance = balance;
}
```

---

69. What is the SingleTon class?

- A **Singleton class** is a class that allows only **one instance** to be created and provides a global point of access to that instance.
- It is used to control object creation and ensure that only one instance exists in the application.
- Example:

```
class Singleton {
    private static Singleton instance;

    private Singleton() { // Private constructor
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

---

70. What are the steps to create a singleton class?

Private Constructor: Prevent instantiation from outside the class.

```
private Singleton() {}
```

Private Static Instance: Create a private static variable to hold the single instance.

```
private static Singleton instance;
```

Public Static Method: Provide a public static method to access the instance.

```
public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
```

---

71. What is Abstract class?

- An **abstract class** is a class declared with the `abstract` keyword.
- It cannot be instantiated (you cannot create objects of an abstract class).
- It can contain **abstract methods** (methods without a body) and **concrete methods** (methods with a body).
- Example:

```
abstract class Animal {
    abstract void sound(); // Abstract method
    void sleep() {         // Concrete method
        System.out.println("Animal is sleeping");
    }
}
```

---

72. What is the Abstract method?

- An **abstract method** is a method declared without a body (using the `abstract` keyword).
- It must be overridden in the subclass.
- Example:

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

---

73. What is a Concrete class?

- **A concrete class is a class that provides implementations for all its methods (no abstract methods).**
- **It can be instantiated (you can create objects of a concrete class).**
- **Example:**

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

---

74. Can we create objects of abstract class?

- **No**, you cannot create objects of an abstract class.
- Abstract classes are meant to be **extended** by subclasses, not instantiated directly.
- Example:

```
abstract class Animal { }
// Animal animal = new Animal(); // Error: Cannot instantiate abstract class
```

---

75. Can we have constructors in abstract class?

- **Yes**, you can have constructors in an abstract class.
- Constructors are used to initialize the state of the subclass objects.
- Example:

```
abstract class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }
}
```

---

76. Can we have static methods in abstract class?

- **Yes**, you can have static methods in an abstract class.
- Static methods belong to the class, not the instance, and can be called using the class name.
- Example:

```
abstract class Animal {
    static void display() {
        System.out.println("Static method in abstract class");
    }
}
```

---

77. Can we have static and non - static data members in abstract class?

- **Yes**, you can have both **static** and **non-static** data members in an abstract class.
- Example:

```
abstract class Animal {
    static int count; // Static data member
    String name;      // Non-static data member
}
```

---

78. Can we have final methods and private methods in abstract class?

Yes, you can have **final methods** and **private methods** in an abstract class.

1. **Final Methods**:
    - Cannot be overridden in subclasses.
    - Example:

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

2. **Private Methods**:
    - Cannot be accessed outside the class.
    - Example:

```
abstract class Animal {
    private void sleep() {
        System.out.println("Animal is sleeping");
    }
}
```

---

79. Why cannot multiple inheritance be achieved through classes in java?

- **Multiple inheritance** is not supported in Java for classes to avoid the **diamond problem**.
- The **diamond problem** occurs when a class inherits from two classes that have a common method, causing ambiguity in method resolution.
- Example:

```
class A {
  void display() {
    System.out.println("A");
  }
}
class B {
  void display() {
    System.out.println("B");
  }
}
class C extends A, B { } // Error: Multiple inheritance not allowed
```

---

80. What is the diamond problem?

- The **diamond problem** is an ambiguity that arises in multiple inheritance when a class inherits from two classes that have a common method.
- Java avoids this problem by not allowing multiple inheritance for classes.
- Example:

```
class A {
  void display() {
    System.out.println("A");
  }
}
class B {
  void display() {
    System.out.println("B");
  }
}
class C extends A, B { } // Error: Multiple inheritance not allowed
```

---

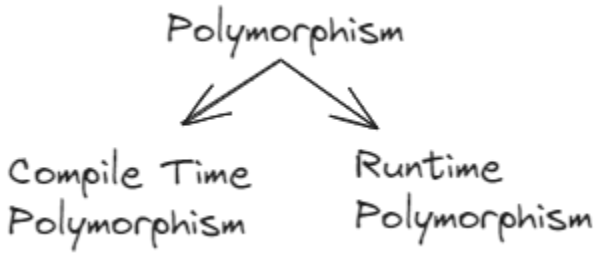
81. What is Early Binding and Late Binding?

- **Binding:** The process of connecting method calls with implementation is known as Binding.
- Binding has 2 types :
    - **Early Binding:**
        - Also known as **Compile-Time Binding**.
        - In Early Binding, method call is binded with implementation at compile time according to reference type.
        - Static, Private, Final methods are binded at compile time.
        - All data members are also binded at compile time.
    - **Late Binding:**
        - Also known as **Runtime Binding.**
        - In Late Binding, method call is binded at runtime according to instance type instead of reference.
        - If the method is non-static, non-private and non-final then late binding takes place.
        - Overridden method calls are binded at runtime.

---

82. What is Polymorphism?

- One entity showing behaviors of another entity in different situations is known as Polymorphism.
- In Java, there are 2 types of Polymorphism.



- **Compile Time Polymorphism:**
  - Call to overload method is resolved at compile time based on parameters type is known as **compile-time polymorphism**.
  - Overloading is used to achieve compile-time polymorphism.
- **Runtime Polymorphism:**
  - Call to overridden method is resolved at runtime based on instance type is known as **runtime polymorphism**.
  - Overriding is used to achieve runtime polymorphism.
- **Advantages of Polymorphism:**
  - Code Reusability
  - Abstraction
  - Loose Coupling

---

83. What is tight coupling and loose coupling?

Loose Coupling:

- **Definition**: Loose coupling occurs when service provider program and user program connected in such a way that any changes in service provider program does not affect user program.
- **Characteristics**:
  - Classes are **weakly connected**.
  - Code is **more flexible** and **easier to maintain**.
  - **Reusability** is increased because classes can be used independently.
- **Example:**

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

```
class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}
class Bike implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a bike");
    }
}
class Traveler {
    Vehicle vehicle; // Loose coupling: Traveler depends on the interface, not a
    Traveler(Vehicle vehicle) {                                    specific class.
        this.vehicle = vehicle;
    }
    void startJourney() {
        vehicle.drive();
    }
}
//The Traveler class is not tightly coupled to Car or Bike. It can work with any class that implements the Vehicle interface.
```

Tight Coupling:

- **Definition: Tight coupling occurs when two or more classes are highly dependent on each other. Changes in one class often require changes in the other class.**
- **Characteristics:**
  - **Classes are strongly connected.**
  - **Code is less flexible and harder to maintain.**
  - **Reusability is reduced because classes cannot be used independently.**
- **Example:**

```
class A {
    B b = new B(); // Tight coupling: A is directly dependent on B
    void display() {
        b.show();
    }
}
class B {
    void show() {
        System.out.println("Inside B");
    }
}
//If class B changes, class A may also need to change.
```

Key Differences Between Tight Coupling and Loose Coupling

Aspect	Tight Coupling	Loose Coupling
Dependency	Classes are **highly dependent** on each other.	Classes are **minimally dependent** on each other.
Flexibility	**Less flexible**; changes in one class affect others.	**More flexible**; changes in one class do not affect others.
Maintainability	**Harder to maintain** due to strong dependencies.	**Easier to maintain** due to weak dependencies.
Reusability	**Less reusable**; classes cannot be used independently.	**More reusable**; classes can be used independently.
Example	Directly creating objects of a specific class.	Using interfaces or abstraction to reduce dependency.

---

84. What is an interface?

- Interface is a non-primitive type in Java.
- Interface is used as an interface (mediator) between service provider and user program.
- Java Provides a keyword `'interface'` to create an interface.
- In interface by default all variables are **public**, **static**, and **final**.
- In interface by default all methods are **public** and **abstract.**
- From Java 8, interface allows a **non-static method** with implementation called **default method** and also **static method.**
- From Java 9, the interface allows **private methods.**
- Interface doesn't have a constructor.
- It can't be instantiated.
- Java provides a keyword `'implements'` to implement interface methods in a class.
- Any class which implements an interface is known as an **implementing class**.
- Implementing class should override all abstract methods of interface otherwise implementing class becomes abstract.
- A class can implement more than 1 interface.

---

85. Can we have constructors in the interface? Why?

- **No, interfaces cannot have constructors.**
- **Reason: Interfaces are meant to define a contract (what to do), not to provide implementation (how to do it). Constructors are used for object initialization, which is not applicable to interfaces.**

---

86. Can we have non static data members in the interface?

- **No**, interfaces cannot have non-static data members.
- All data members in an interface are **public**, **static**, and **final** by default.
- Example:

```
interface Constants {
    int MAX_VALUE = 100; // public, static, final
}
```

---

87. Can we have blocks in the interface?

- **No**, interfaces cannot have **instance initializer blocks** or **static blocks**.
- However, you can have **default methods** and **static methods** in interfaces (introduced in Java 8).

---

88. Explain extends and implements keywords?

`extends`:

- Used to create a subclass that **inherits** from a superclass.
- Example:

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

`implements`:

- Used to implement an **interface** in a class.
- Example:

```
interface Drawable {
    void draw();
}
class Circle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

---

89. Can an interface be extended by another interface?

- **Yes**, an interface can be extended by another interface using the `extends` keyword.
- Example:

```
interface A {
    void methodA();
}
interface B extends A {
    void methodB();
}
```

---

90. What is Functional Interface?

- A **functional interface** is an interface with **exactly one abstract method**.
- It can have any number of **default** or **static** methods.
- Functional interfaces are used in **lambda expressions** and **method references**.
- Example:

```
@FunctionalInterface
interface Calculator {
    int calculate(int a, int b);
}
```

---

91. What is Marker Interface?

- A **marker interface** is an interface with **no methods**.
- It is used to mark or tag a class with a specific behavior or capability.
- Examples in Java: `Serializable`, `Cloneable`, `Remote`.
- Example:

```
interface Deletable { } // Marker interface
class Document implements Deletable { }
```

---

92. Explain Java8 interface concrete methods?

In Java 8, interfaces can have **concrete methods** in the form of:

1. **Default Methods**:
    - Methods with a default implementation.
    - Example:

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle started");
    }
}
```

2. **Static Methods**:
    - Methods that belong to the interface and can be called using the interface name.
    - Example:

```
interface Vehicle {
    static void stop() {
        System.out.println("Vehicle stopped");
    }
}
```

---

93. What is the package? how to create it? When to use the import keyword?

- **Package**:
    - A package is a namespace that organizes related classes and interfaces.
    - It helps avoid naming conflicts and improves code maintainability.
- **How to Create a Package**:
    - Use the `package` keyword at the top of the Java file.
    - Example:

```
package com.example.mypackage;
public class MyClass { }
```

When to Use the `import` Keyword:

- Use the `import` keyword to include classes or interfaces from other packages.
- Example:

```
import java.util.Scanner;
```

---

94. How many times static blocks get executed and when?

- **Static blocks** are executed **once** when the class is loaded into memory.
- They are used to initialize **static variables** or perform one-time setup tasks.
- Example:

```
class MyClass {
    static {
        System.out.println("Static block executed");
    }
}
```

---

95. How many times non-static blocks gets executed and when?

- **Non-static blocks** are executed **every time an object is created**, before the constructor is called.
- They are used to initialize **instance variables** or perform setup tasks for each object.
- Example:

```
class MyClass {
    {
        System.out.println("Non-static block executed");
    }
}
```

---

96. Explain varargs?

- **Varargs (Variable Arguments)** is a feature that allows a method to accept **zero or more arguments** of a specified type.
- It is represented by three dots (. . .) after the data type.
- Example:

```
void display(String... values) {
    for (String value : values) {
        System.out.println(value);
    }
}
```

---

97. What is the first statement in a constructor?

The **first statement in a constructor** must be either:

1. A call to another constructor in the same class using `this()`.
2. A call to the superclass constructor using `super()`.

If neither is explicitly written, the compiler automatically inserts `super()` (calling the no-argument constructor of the superclass).

---

98. Why is the Object class not the final class? Explain ?

- The **`Object` class** is not a `final` class because it is meant to be **extended** by all other classes in Java.
- If it were `final`, no other class could extend it, which would break the inheritance hierarchy in Java.
- Example: `class MyClass { } // Implicitly extends Object`

---

99. Explain what happens when we instantiate subclass?

- When a subclass is instantiated:
    1. The **superclass constructor** is called first (implicitly or explicitly using `super()`).
    2. The **non-static blocks** of the superclass are executed.
    3. The **superclass constructor body** is executed.
    4. The **non-static blocks** of the subclass are executed.
    5. The **subclass constructor body** is executed.
- Example:

```
class Animal {
    Animal() {
        System.out.println("Animal constructor");
    }
}
class Dog extends Animal {
    Dog() {
        System.out.println("Dog constructor");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        // Output: // Animal constructor
    }              // Dog constructor
}
```