Q.1 What are the different programming paradigms?

- Structured Programming
- Procedural Programming
- Object Oriented Programming

Q.2 What is structured programming?

- Structured programming is a programming technique that emphasizes breaking down a program into smaller, manageable **modules or blocks**.
- Each module is designed to perform a specific task and follows a clear hierarchy, making the code easier to understand, maintain, and debug.

Key characteristics of structured programming:

1. **Hierarchy of Modules**: Programs are divided into a hierarchy of modules, each with a single entry and exit point.
2. **Single Entry/Exit**: Each module or function has a clear starting and ending point, ensuring a predictable flow of control.
3. **Controlled Flow**: Control is passed in a downward direction within the structure without jumping between levels (i.e., avoiding "goto" statements).
4. **Three Control Flows**:
    - **Sequential**: Executes instructions one after the other.
    - **Test/Selection**: Uses conditional statements (e.g., `if-else`) to choose between alternative paths.
    - **Iteration**: Repeats a set of instructions using loops (e.g., `for`, `while`).

This approach improves code readability, reduces errors, and encourages modular design.

Q.2 What is Procedural Programming?

- Procedural programming is a programming paradigm that follows a step-by-step approach to solving problems.
- It is derived from structured programming and relies on the **concept of procedure calls**, where the program is broken down into a series of computational steps, also known as procedures, routines, subroutines, or functions.
- In procedural programming, the programmer writes code that specifies the exact steps the program must take to reach a desired outcome.
- Each procedure can be called at any point in the program, sometimes even by other procedures or recursively by itself, allowing for efficient code reuse and organization.

Eg. C, C++, pascal, fortran

Q.3 Issues with Procedural Programming

Procedural programming faces several issues, especially as programs grow in size and complexity:

1. **Increased Complexity**: As the length of a program increases, managing the logic becomes harder, making it difficult to maintain and debug.
2. **Modularity**: Large programs need to be divided into smaller, manageable functions or modules to maintain clarity, which requires extra planning.
3. **Global Data Access**: Functions have unrestricted access to global data, which can lead to unintended side effects and errors in larger programs.
4. **Poor Mapping to Real World**: Procedural programming often struggles to model real-world entities and their relationships, making it less intuitive for certain types of problems.
5. **Lack of Extensibility**: Procedural languages are less flexible when it comes to extending functionality without modifying existing code.

Examples of procedural languages include Fortran, C, and Pascal.

Q.4 What is Modular Programming

Modular programming is a programming technique that involves separating a program's functionality into independent, interchangeable modules. Each module is designed to execute one specific aspect or task of the program, making it easier to manage and understand.

Key characteristics of modular programming:

1. **Independent Modules**: Each module operates independently, containing everything necessary to perform a specific function.
2. **Encapsulation**: Each module encapsulates its data and functions, which prevents interference between different parts of the program.
3. **Sub-programs/Functions**: Modules are often implemented as sub-programs or functions that perform distinct tasks, improving code organization.

Modular programming enhances code readability, reusability, and maintainability, making it easier to develop and test complex software.

Q.5 Differentiate between structured and modular programming

Modular programming and **structured programming** both aim to improve code readability, organization, and maintainability, but they focus on different aspects of program design. Here's how they differ:

1. Focus:

- **Modular Programming**: Focuses on dividing a program into independent, interchangeable modules. Each module is responsible for a specific functionality and is self-contained.
- **Structured Programming**: Focuses on controlling the flow of a program through a hierarchy of control structures like sequence, selection (if-else), and iteration (loops), avoiding unstructured "goto" statements.

2. Key Concept:

- **Modular Programming**: Centers on **modularity**, meaning breaking the program into smaller, functional units (modules), each capable of handling a specific task independently.
- **Structured Programming**: Emphasizes **structured flow**

Q.6 What is Object Oriented Programming? Also State the 4 basic properties of OOP

Object-Oriented Programming (OOP) is a programming paradigm that is centered around the concept of "objects" to design software. **These objects represent real-world entities, each having attributes (data) and behavior (functions/methods).** OOP allows programs to model real-world problems more naturally.

Key Features of OOP:

1. **Objects**: The basic units of OOP, objects have unique identities, attributes (data), and behaviors (methods/functions).
2. **Real-World Modeling**: OOP mimics the real world by creating objects that represent real-world entities.

Four Fundamental Properties of OOP:

1. **Inheritance**:

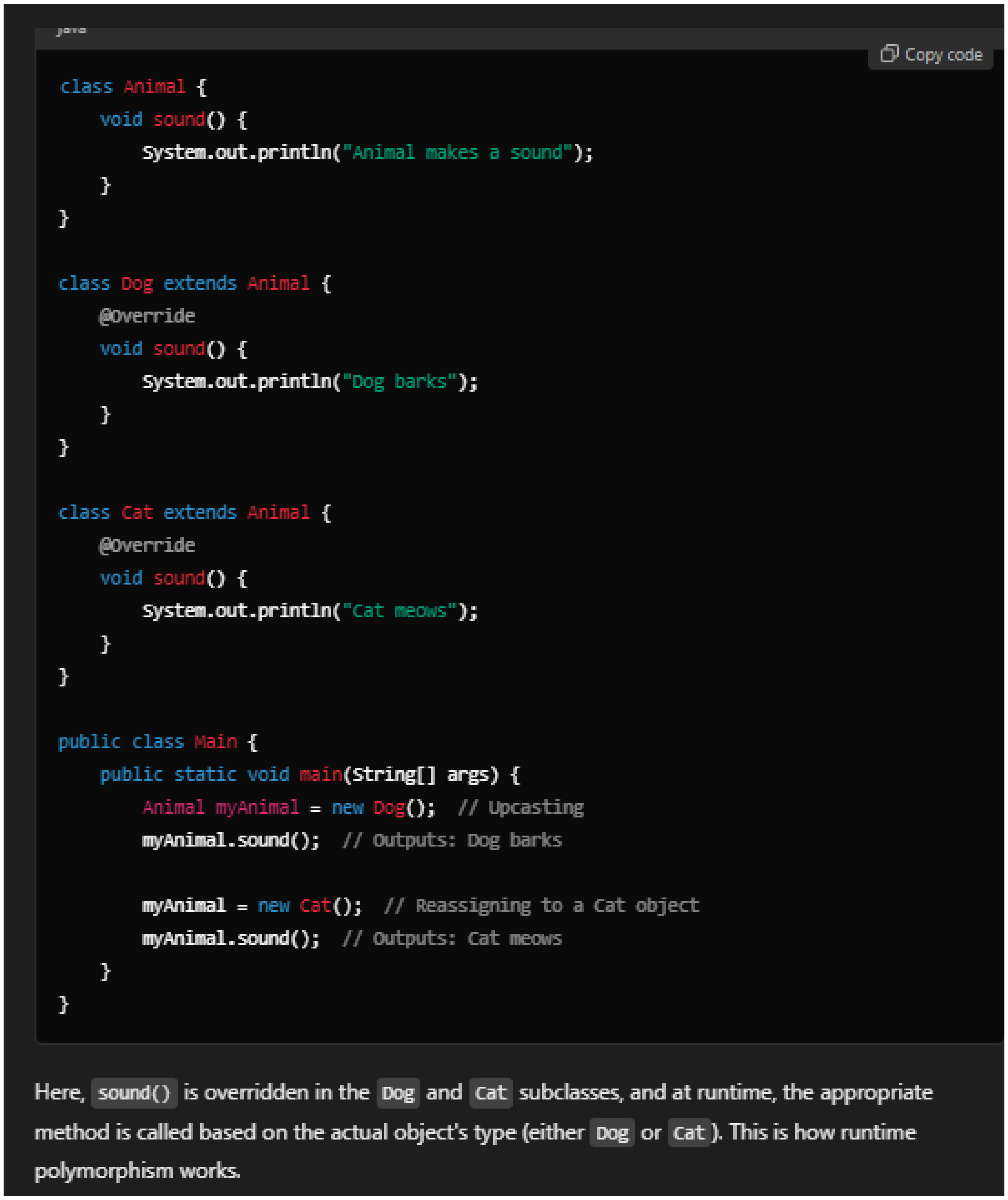
- A mechanism where a child class (subclass) inherits properties and behaviors (methods and fields) from a parent class (superclass).
- It promotes **code reusability**, allowing the child class to reuse the methods of the parent class, while also adding new methods and fields.
- Inheritance represents an **IS-A** relationship (e.g., a Dog **IS-A** Animal).

2. **Polymorphism**:
   - The ability of a single function or method to behave differently based on the context.
   - **Compile-time Polymorphism**: Achieved through method overloading, where multiple methods have the same name but different parameter lists within the same class.

Example:

```
void hello(int a)
void hello(float a)
void hello(int a, int b)
```

   - **Runtime Polymorphism**: Achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.


java

Copy code

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Upcasting
        myAnimal.sound();  // Outputs: Dog barks

        myAnimal = new Cat();  // Reassigning to a Cat object
        myAnimal.sound();  // Outputs: Cat meows
    }
}
```

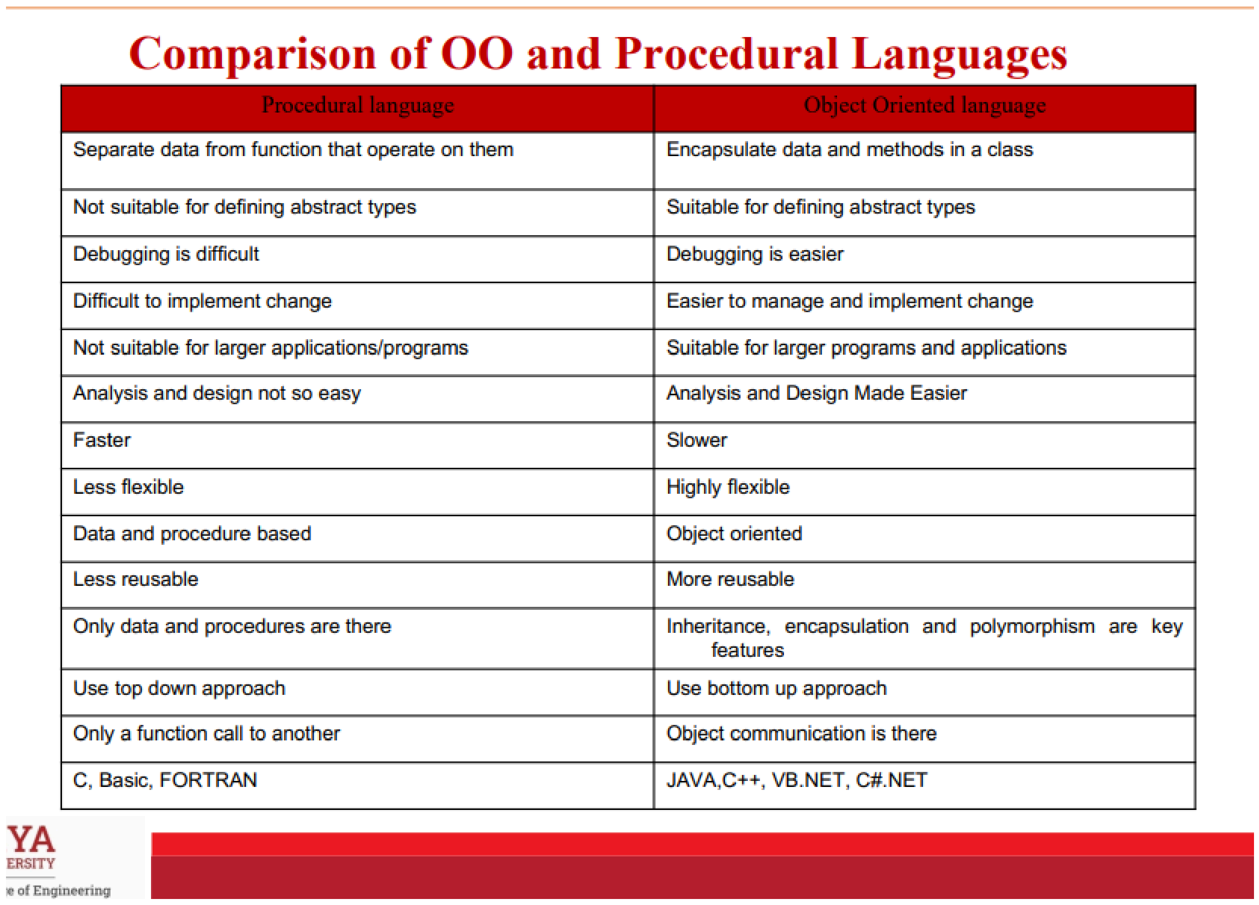
Here, `sound()` is overridden in the `Dog` and `Cat` subclasses, and at runtime, the appropriate method is called based on the actual object's type (either `Dog` or `Cat`). This is how runtime polymorphism works.

3. **Abstraction**:
    - The process of hiding the complex implementation details of an object and exposing only the essential features or interfaces.
    - It simplifies interaction with objects by focusing on **what** an object does, rather than **how** it does it.
4. **Encapsulation**:
    - The process of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, usually a class.

- It restricts direct access to some of an object's components, which helps maintain control over the object's data.

Together, these principles make OOP a powerful paradigm for building complex, scalable, and maintainable software systems.

Q.7 Differentiate between Procedural language and object oriented language



Comparison of OO and Procedural Languages

Procedural language	Object Oriented language
Separate data from function that operate on them	Encapsulate data and methods in a class
Not suitable for defining abstract types	Suitable for defining abstract types
Debugging is difficult	Debugging is easier
Difficult to implement change	Easier to manage and implement change
Not suitable for larger applications/programs	Suitable for larger programs and applications
Analysis and design not so easy	Analysis and Design Made Easier
Faster	Slower
Less flexible	Highly flexible
Data and procedure based	Object oriented
Less reusable	More reusable
Only data and procedures are there	Inheritance, encapsulation and polymorphism are key features
Use top down approach	Use bottom up approach
Only a function call to another	Object communication is there
C, Basic, FORTRAN	JAVA,C++, VB.NET, C#.NET

Q. 8 What is an object?

An **object** in Object-Oriented Programming (OOP) is an instance of a class, which serves as a blueprint for creating objects. It is both a **physical** and **logical entity** that encapsulates data and behavior.

Characteristics of an Object:

1. **State**: Represents the data or attributes of the object, which are defined by the class variables.
2. **Behavior**: Represents the functionality of the object, defined by the methods or functions of the class.
3. **Identity**: A unique identifier for the object, typically used internally by systems like the JVM (Java Virtual Machine) to differentiate between different objects. This identity is not exposed to the user but is used to manage objects during program execution.

Objects are the building blocks in OOP, allowing for the representation of real-world entities in software systems.

Q.9 What is a class?

A **class** in Object-Oriented Programming (OOP) is a blueprint or template used to create objects that share common properties and behaviors. It defines the structure and behavior of objects but is itself a logical entity without physical existence. Classes can contain fields, methods, constructors, blocks, nested class and interface.

Q.10 What is Java?

Key Features of Java:

1. **Platform Independence**:
   - Java is designed to be platform-independent, meaning that code written in Java can run on any device or operating system that has a Java Virtual Machine (JVM). This is achieved through the compilation of Java code into bytecode, which is interpreted by the JVM.
2. **Object-Oriented**:
   - Java is an object-oriented programming language, which means it uses objects and classes to structure code. This approach helps in organizing and managing complex software systems by modeling real-world entities.
3. **Compiled and Interpreted**:
   - Java code is first compiled into bytecode by the Java compiler. This bytecode is then interpreted or executed by the JVM. This combination of compilation and interpretation provides both performance and portability.
4. **Robust**:
   - Java is considered robust due to its strong type-checking, exception handling, and memory management. These features help prevent many common programming errors and ensure reliability.
5. **Security**:

    - Java provides a secure environment through its built-in security features such as the Java security manager and bytecode verification. This helps in protecting the system from malicious code.
6. **Strictly Typed Language**:
    - Java is a strongly typed language, which means that each variable and expression type is explicitly defined, and type errors are caught at compile-time.
7. **Lack of Pointers**:
    - Java does not support pointers, which helps avoid certain types of programming errors and security vulnerabilities associated with direct memory access.
8. **Garbage Collection**:
    - Java automatically manages memory through garbage collection, which reclaims memory used by objects that are no longer in use, helping to prevent memory leaks.
9. **Strict Compile-Time Checking**:
    - Java performs rigorous compile-time checking to ensure code correctness, reducing the likelihood of runtime errors.
10. **Sandbox Security**:
    - Java's sandboxing feature allows code to be executed in a restricted environment with limited access to system resources, enhancing security, particularly in web applications.

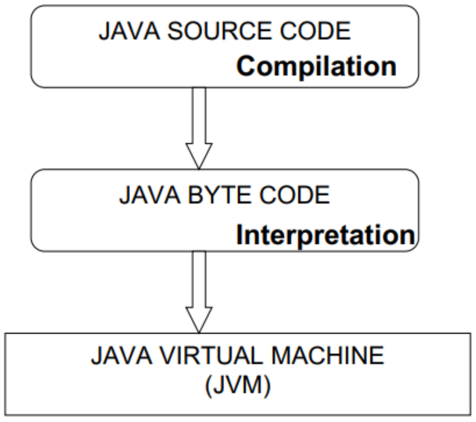
These features make Java a versatile, reliable, and secure language suitable for a wide range of applications, from web development to mobile apps and enterprise systems.

Q.11 How is java platform independent?

Java is often referred to as platform-independent because of its "Write Once, Run Anywhere" (WORA) capability.
Java Code is compiled into an intermediate form called bytecode. The bytecode is executed by the **Java Virtual Machine (JVM)**, which is available for various operating systems (Windows, Linux, macOS, etc.), and it interprets or compiles the bytecode into native machine code specific to the underlying hardware and OS.

Java Runtime

Java Runtime Environment includes JVM, class libraries and other supporting files



Q.12 Features of Java

Multithreaded:

- Java provides built-in support for multithreading, allowing multiple threads to run concurrently within a single program. This capability enables efficient use of CPU resources and improves the performance of applications that require concurrent operations.

Dynamic Binding:

- Java supports dynamic binding (or late binding), which means method calls are resolved at runtime rather than compile-time. This allows for more flexible and dynamic method invocation, especially useful in scenarios involving polymorphism and inheritance.

Performance:

- Java is designed to be performant with features like Just-In-Time (JIT) compilation, which compiles bytecode into native machine code at runtime, optimizing performance. Additionally, Java's garbage collector helps manage memory efficiently, contributing to overall performance.

Networking:

- Java includes a comprehensive set of APIs for network programming, enabling developers to create networked applications easily. The `java.net` package provides classes for working with sockets, URLs, and other network-related functions.

No Pointers:

- Java eliminates the use of pointers, which helps avoid common errors and security issues related to direct memory access. Instead, Java uses references to objects, simplifying memory management and improving safety.

No Global Variables:

- Java does not support global variables. Instead, it uses classes and objects to encapsulate data, promoting better organization and reducing the risk of unintended side effects from variable modifications.

Automatic Garbage Collection:

- Java includes an automatic garbage collection mechanism that periodically reclaims memory from objects that are no longer in use. This helps prevent memory leaks and reduces the burden of manual memory management for developers.

Q. 13 What is multithreading in java?

Multithreading in Java refers to the capability of a program to execute multiple threads concurrently within a single process. A thread is a lightweight sub-process that allows multiple tasks to be executed simultaneously, improving the efficiency and performance of programs. Threads share the same memory and resources within a process, which allows them to communicate with each other without the need for separate memory allocation for each thread.

Q 14. Why is java robust?

Key Features Contributing to Java’s Robustness:

1. **Strong Memory Management**:

    - Java uses automatic **garbage collection**, which automatically deallocates memory that is no longer in use. This helps prevent memory leaks and ensures efficient memory usage, reducing the risk of program crashes.
2. **No Pointers**:
    - Java does not support explicit pointers, which are a common source of security vulnerabilities and memory corruption issues in languages like C/C++. By eliminating direct access to memory addresses, Java ensures safer memory management.
3. **Exception Handling**:
    - Java provides a strong **exception handling mechanism** to manage runtime errors. By catching and handling exceptions, Java programs can recover from unexpected errors gracefully without crashing. This feature improves the program's stability and reliability.
4. **Type Checking at Compile-Time**:
    - Java has strict compile-time type checking. Errors such as mismatched data types are detected during compilation, reducing the chances of errors occurring at runtime.
5. **Automatic Garbage Collection**:
    - Java's garbage collector automatically manages memory, reclaiming memory from objects that are no longer referenced. This reduces the chances of memory-related bugs like memory leaks or excessive memory usage.
6. **Multithreading Support**:
    - Java's built-in support for multithreading is designed to avoid concurrency-related issues, providing synchronized blocks to handle resource-sharing between threads safely.

These features together make Java robust, ensuring that programs are reliable, secure, and less prone to crashes or memory issues.

Q.15 JAVA VS C++

Here are the key **differences between C++ and Java** based on various features:

1. Operator Overloading:

- **C++**: Supports operator overloading, allowing developers to define custom behaviors for operators.
- **Java**: Does not support operator overloading, except for the + operator, which can be used for string concatenation.

2. Boolean Type:

- **C++**: Booleans can be compared with integers (`0` and `1`), as booleans are sometimes treated as `int`.
- **Java**: Boolean is a separate type with two values: `true` and `false`. It cannot be compared with integers like in C++.

3. Array Length:

- **C++**: No direct method to determine the length of an array.
- **Java**: Every array object has a `length` field that provides the number of elements in the array.

4. Goto Statement:

- **C++**: Supports the `goto` statement, though its use is generally discouraged.
- **Java**: Does not support `goto`. Instead, Java provides `break` and `continue` for controlling loops.

5. Pointers:

- **C++**: Fully supports pointers for direct memory access.
- **Java**: Does not support pointers, which helps prevent common memory errors like segmentation faults.

6. Null Pointers:

- **C++**: Dereferencing a null pointer leads to undefined behavior and may cause a crash.
- **Java**: Null pointers are caught by a built-in `NullPointerException`, making errors easier to debug.

7. Memory Management:

- **C++**: Requires explicit memory management with destructors and manual deallocation (e.g., `delete`).
- **Java**: Automatic memory management via garbage collection, preventing memory leaks and freeing memory automatically.

8. Variable Initialization:

- **C++**: Variables are not automatically initialized and may contain garbage values unless initialized by the programmer.
- **Java**: All variables, except local variables, are automatically initialized to default values.

9. Runtime Bounds Checking:

- **C++**: Does not perform automatic bounds checking on arrays.
- **Java**: Performs runtime bounds checking on arrays and throws an `IndexOutOfBoundsException` if necessary.

10. Access Modifiers:

- **C++**: By default, members of a class are private.
- **Java**: By default, members have package-level access (default access), which is different from private.

11. Integer Type Sizes:

- **C++**: The sizes of integer types are platform-dependent.
- **Java**: The sizes of integer types are well-defined across platforms:
    - `byte`: 1 byte
    - `short`: 2 bytes
    - `int`: 4 bytes
    - `long`: 8 bytes

12. Unicode Support:

- **C++**: Does not inherently support Unicode.
- **Java**: Supports Unicode, allowing for representation of characters from different languages like Japanese, Latin, etc.

13. String Handling:

- **C++**: Strings are managed using `std::string` from the Standard Library.
- **Java**: Provides a predefined `String` class along with `StringBuffer` and `StringBuilder` for efficient string manipulation.

14. Utility Libraries:

- **C++**: Standard library provides utility classes like `std::vector`, `std::map`, etc.
- **Java**: Extensive utility libraries are provided in the `java.util` package, including classes like `Enumeration`, `Hashtable`, `Vector`, etc.

15. Multithreading:

- **C++**: Multithreading support is available but requires additional libraries like `pthread` or C++11 threading features.
- **Java**: Provides built-in support for multithreading with synchronization, simplifying concurrent programming.

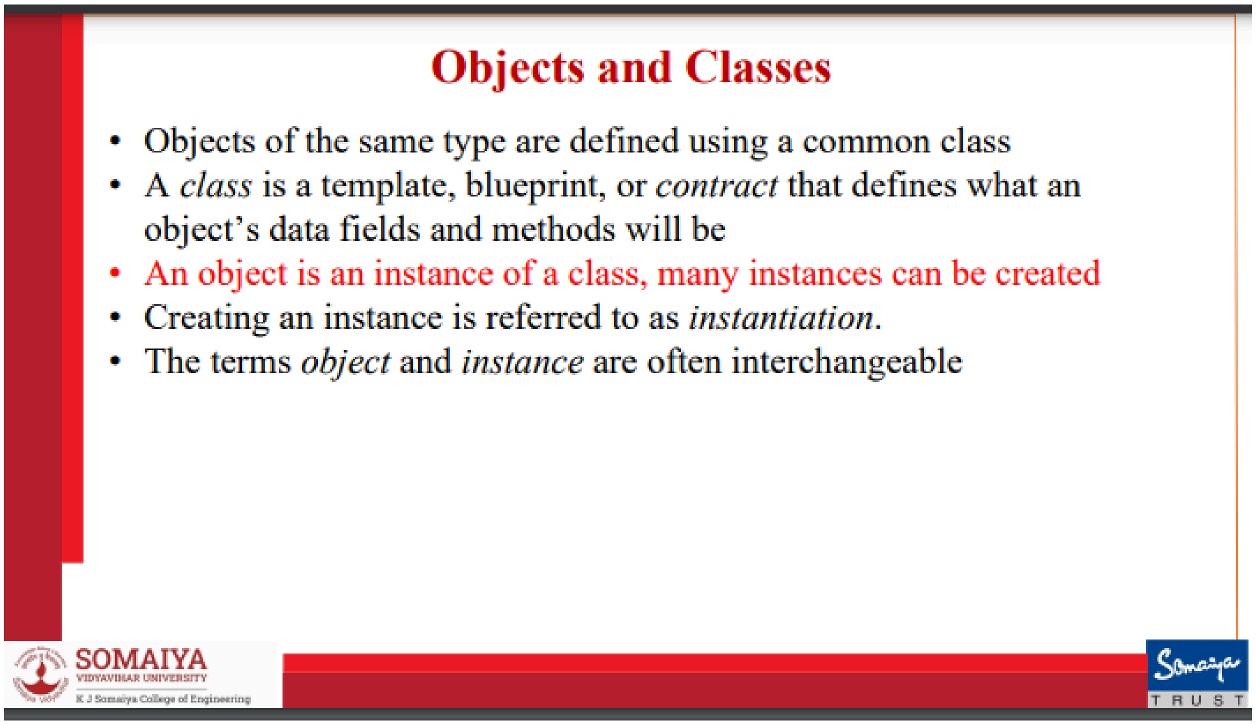
16. Default Access Specifier:

- **C++**: The default access specifier for class members is `private`.
- **Java**: The default access specifier is package-private (default access), which allows access to members by other classes in the same package.

These differences highlight the contrast in design philosophy between C++ and Java, with Java focusing more on security, platform independence, and ease of use, while C++ offers more control over memory and system resources

Q. 16 That is why it is a good idea to give the Java source files the same name as that of the class they contain?

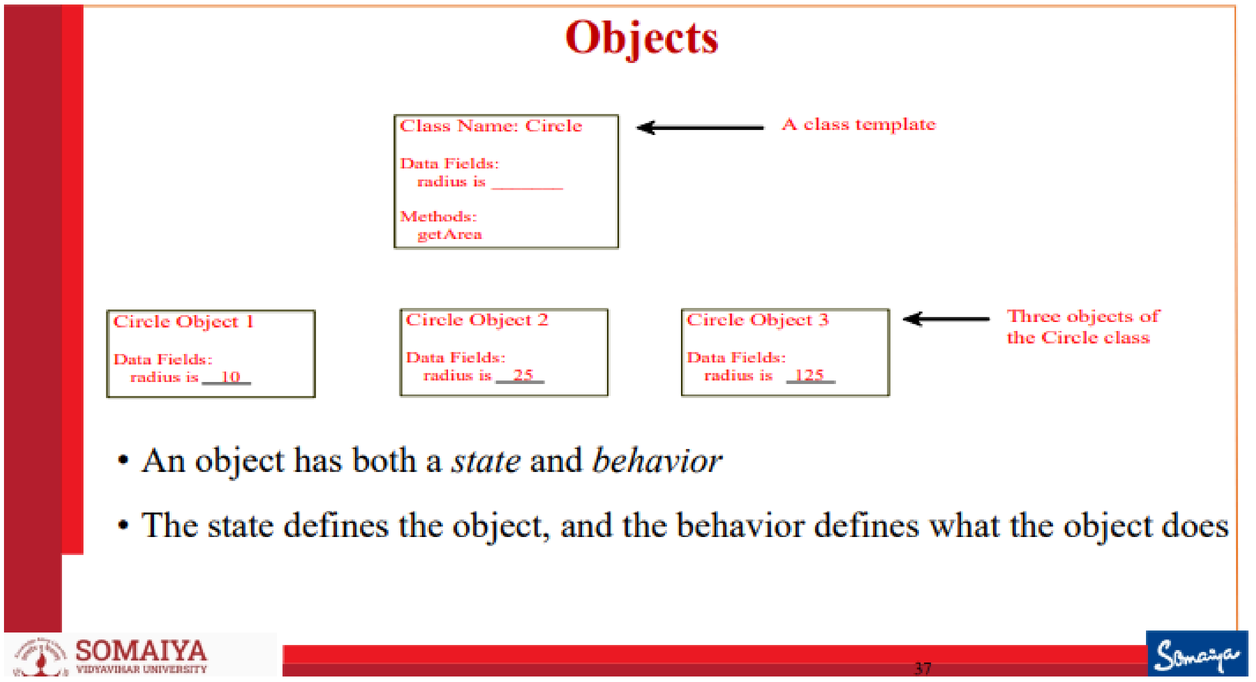
In Java, the name of the `.class` file, which is created after compiling your source code, must exactly match the name of the public class in your source file. This means if you have a class called `MyClass` in your `MyClass.java` file, the compiler will generate a `MyClass.class` file. This is important because **it keeps things organized** and helps the Java Virtual Machine (JVM) **easily find and run the right class**. So, it's a good practice to name your Java source files the same as the public class they contain to avoid confusion and errors.



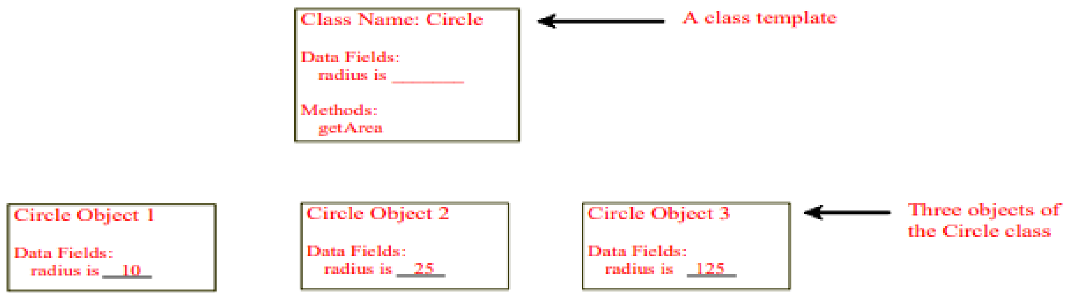
Objects and Classes

- Objects of the same type are defined using a common class
- A *class* is a template, blueprint, or *contract* that defines what an object's data fields and methods will be
- An object is an instance of a class, many instances can be created
- Creating an instance is referred to as *instantiation*.
- The terms *object* and *instance* are often interchangeable

SOMAIYA VIDYAVIHAR UNIVERSITY K J Somaiya College of Engineering



Objects



- An object has both a *state* and *behavior*
- The state defines the object, and the behavior defines what the object does

SOMAIYA VIDYAVIHAR UNIVERSITY

37

Q.17) INSTANTIATION

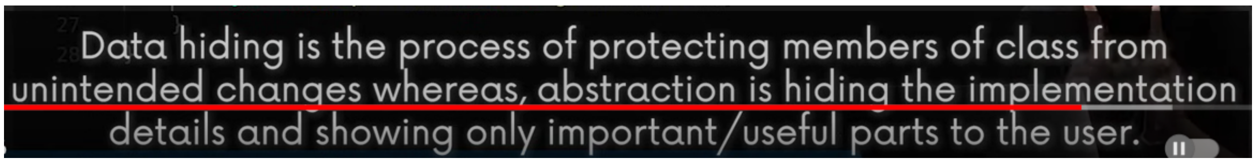
Instantiation in Java refers to the process of creating an object from a class. When you create an object, memory is allocated to store the object's data, which includes its fields (variables). The object is initialized using a **constructor**, which is a special method that gets called automatically when the object is created. The constructor helps set up the initial state of the object by assigning values to its fields.

Q.18) ABSTRACTION

In Object-Oriented Programming (OOP), **abstraction** means focusing on the essential features of an object while hiding the unnecessary details. Just like in real life, humans simplify complex things by ignoring irrelevant details and concentrating only on what's important. In programming, abstraction allows developers to manage complexity by defining what an object does without worrying about how it does it.

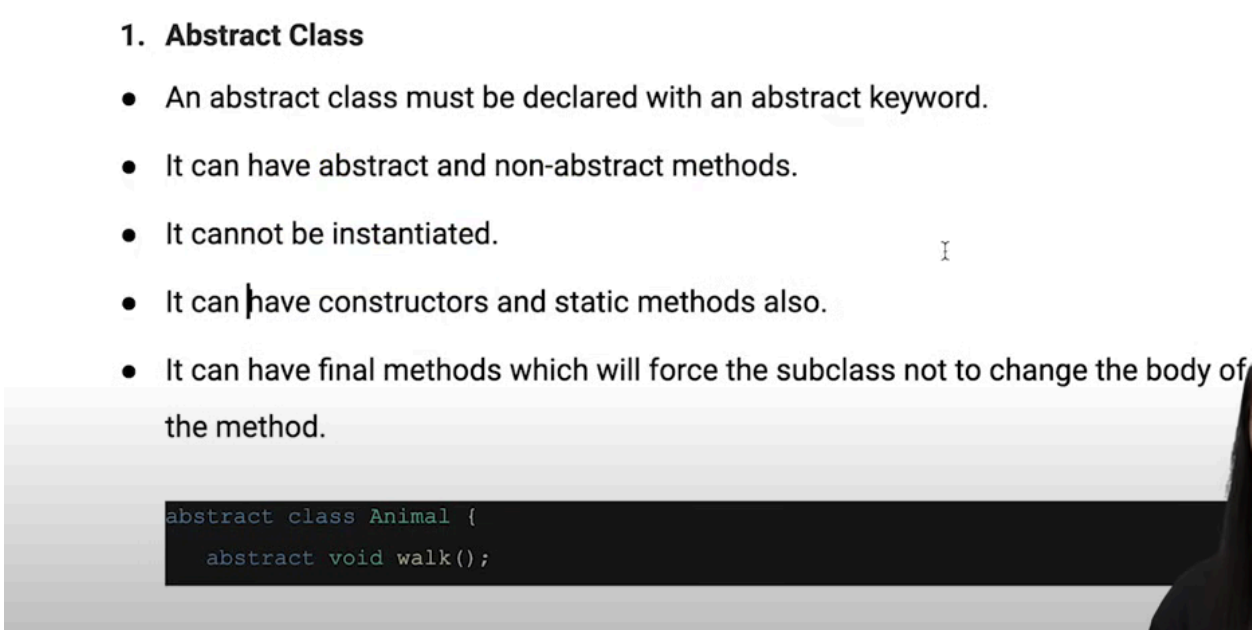
Encapsulation and **inheritance** are key ways to achieve abstraction.

To "abstract" something means to **simplify** or **hide the complexities** of that thing and focus only on what's important.



Data hiding is the process of protecting members of class from unintended changes whereas, abstraction is hiding the implementation details and showing only important/useful parts to the user.

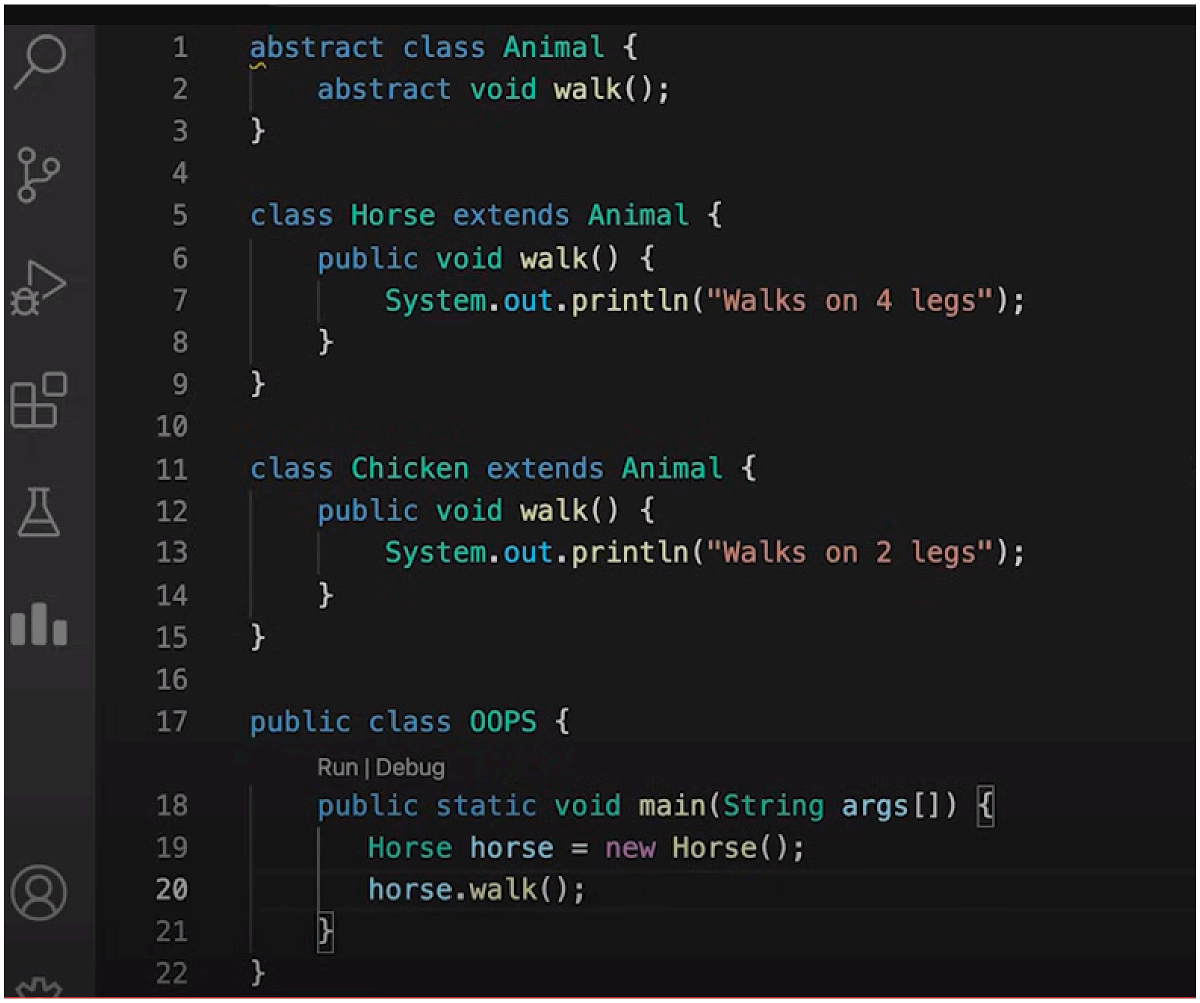
Run time error



1. **Abstract Class**

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

```
abstract class Animal {
   abstract void walk();
```



```java
abstract class Animal {
    abstract void walk();
}

class Horse extends Animal {
    public void walk() {
        System.out.println("Walks on 4 legs");
    }
}

class Chicken extends Animal {
    public void walk() {
        System.out.println("Walks on 2 legs");
    }
}

public class OOPS {
    Run | Debug
    public static void main(String args[]) {
        Horse horse = new Horse();
        horse.walk();
    }
}
```

Q.19) ENCAPSULATION

Encapsulation is a core concept in Object-Oriented Programming (OOP) that focuses on "information hiding." It means that when you use an object, you should only be able to access the methods or data that are necessary, while the unnecessary details and implementation should remain hidden. Encapsulation protects the internal workings of an object, so the user interacts with it only through well-defined methods.

In languages like **Java** and **C++**, this is achieved using **classes** and **access modifiers** like `public`, `private`, and `protected`. These modifiers control what can be accessed from outside the class, ensuring the data is safely encapsulated within.

Q.20) Inheritance

Inheritance is a feature in Object-Oriented Programming that allows a new class (called a **subclass**) to automatically acquire the properties and behaviors (variables and methods) of an existing class (called a **superclass**). This means that the subclass doesn't need to redefine everything from scratch; it can inherit and reuse the code from the superclass.

For example, if **Class B** is a subclass of **Class A**, then **Class B** inherits all the variables and methods defined in **Class A**. This helps with code reuse and establishing a natural **parent-child** relationship between classes.

Inheritance encourages **software reuse** by allowing developers to use existing code without rewriting it. When a new class is created, it can inherit the properties and methods from an existing class, saving time and effort. However, for this reuse to be effective, it requires careful planning and design. When defining classes, developers should think ahead about potential future changes or extensions to ensure that the code remains flexible and easy to maintain. This approach helps in building scalable and adaptable systems.

Q.21) Polymorphism

Polymorphism in Object-Oriented Programming refers to the ability of a single method or function to take on **many forms**, meaning it can have different behaviors depending on the object calling it. For example, a method named `move` can be defined in both a **person** class and a **car** class, but each will have its own unique implementation.

Polymorphism is made powerful through **dynamic binding**, where the program determines at runtime which version of the method to execute based on the type of object. This flexibility allows for more adaptable and reusable code.

Q.22) Static and Dynamic Binding

In Java, **binding** refers to the process of associating a method call or variable reference with its actual code. There are two types of binding:

1. **Static Binding**:
    - Occurs at **compile time**.
    - Used for **private**, **final**, and **static** methods or variables, as they can't be overridden. Since the compiler knows the exact method or variable to use, the decision is made early, making execution faster.
    - **Overloaded methods** are also resolved through static binding.
2. **Dynamic Binding**:
    - Happens at **runtime**.
    - This is used for **overridden methods**, **where the method to be executed depends on the actual object type at runtime,** not the reference type.
    - **Dynamic binding allows polymorphism**, enabling Java to choose the right method when multiple classes in a hierarchy have the same method name.

In summary, static binding is decided by the compiler, while dynamic binding depends on the actual object at runtime, making polymorphism and method overriding possible.

Q.23) Coupling

Coupling refers to the degree of dependency between classes, or how much one class knows about another. It impacts how easily code can be maintained or modified.

- **Tight Coupling**: When a class heavily depends on another class, it creates a tight bond. This is considered bad design because changes in one class can heavily affect the other, making the system less flexible and harder to maintain.
- **Loose Coupling**: This is when classes have minimal dependency on each other, which is considered a good design. With loose coupling, changes in one class have little impact on others, making the system more modular, flexible, and easier to modify or maintain.

Loose coupling is ideal for scalable, maintainable software.

Tight Coupling occurs when one class directly accesses the internal data of another class, which often leads to problems with maintaining and evolving the code.

Example of Tight Coupling:

1. **Class A**:
   - Has a public data member `name` and provides public getter and setter methods (`getName()` and `setName()`) to access and modify `name` with validity checks.
2. **Class B**:
   - Creates an object of `Class A` and directly accesses and modifies the `name` field since it's public.

Problems with Tight Coupling:

- **Bypassing Validations**: Class B can directly set `name` to `null` or access it without using the getter and setter methods, bypassing the validity checks that `Class A` intended to enforce. This undermines the data integrity and intended constraints of `Class A`.
- **Reduced Flexibility**: If `Class A` needs to change its internal structure or validation logic, `Class B` might also need modifications because it relies on the public fields of `Class A`. This tight interdependence makes the code less flexible and harder to maintain.

Good Design Practice (Loose Coupling): To avoid tight coupling, the data members of `Class A` should be private, and access should be controlled through getter and setter methods. This way, `Class B` can only interact with `Class A` through its public interface, ensuring that all validation and encapsulation rules are preserved.

Loose Coupling in application design refers to the practice of minimizing dependencies between classes. This is achieved by **encapsulation**, which involves hiding a class's internal data and exposing only necessary methods to interact with that data.

Example of Loose Coupling:

1. **Class A**:
   - Contains a private data member `name`.
   - Provides public getter and setter methods (`getName()` and `setName()`) to access and modify `name`, while performing necessary validation checks.
2. **Class B**:
   - Creates an instance of `Class A`.
   - Uses the public getter and setter methods to interact with `name`, ensuring that all validity checks are respected.

Benefits of Loose Coupling:

- **Data Integrity**: Since `name` is private, direct access is not possible from outside `Class A`. All interactions with `name` must go through the getter and setter methods, which enforce the validity checks.
- **Flexibility**: Changes to `Class A`'s internal implementation or validation logic won't affect `Class B`, as `Class B` only interacts through the public methods provided by `Class A`. This makes the code more modular and easier to maintain.
- **Improved Maintainability**: With loose coupling, modifying one class does not require changes in other classes, as long as the public interface remains consistent.

By ensuring that classes are loosely coupled, you create a more robust, maintainable, and adaptable application.

Q.24) Cohesion

Cohesion refers to how closely the operations and responsibilities of a class are related to each other. It measures how well the class performs a specific, focused task.

Types of Cohesion:

1. **High Cohesion**:

    - **Definition**: A class with high cohesion is designed to perform a single, well-defined task or purpose. All its methods and properties are closely related to this purpose.
    - **Benefits**:
        - **Simplicity**: The class is easier to understand and maintain because it has a clear focus.
        - **Reusability**: High cohesion often makes a class more reusable in different contexts because it encapsulates a specific functionality.
        - **Flexibility**: Changes to the class are less likely to affect other parts of the system, as it has a single responsibility.
2. **Low Cohesion**:
    - **Definition**: A class with low cohesion tries to handle multiple unrelated tasks or functionalities. Its methods and properties are not strongly related to a single purpose.
    - **Drawbacks**:
        - **Complexity**: The class becomes harder to understand and maintain because it has multiple, possibly unrelated responsibilities.
        - **Difficulty in Reuse**: It's harder to reuse a class with low cohesion because it does not have a clear, single purpose.
        - **Increased Risk of Errors**: Changes or bugs in one part of the class might affect other unrelated parts, leading to higher maintenance costs.

Good Design Practice: Aim for high cohesion when designing classes. This means that each class should focus on a single responsibility or closely related set of functionalities. This approach leads to more maintainable, understandable, and reusable code.

Low Cohesion refers to a class that tries to perform multiple unrelated tasks rather than focusing on a single, well-defined responsibility. This is considered a bad design because it makes the class harder to create, maintain, and update.

Example of Low Cohesion:

```
class PlayerDatabase {

   public void connectDatabase();

   public void printAllPlayersInfo();

   public void printSinglePlayerInfo();

   public void printRankings();
```

```
    public void printEvents();

    public void closeDatabase();

}
```

Program Analysis:

- The `PlayerDatabase` class handles too many unrelated responsibilities: connecting to a database, printing player information, printing rankings, events, and closing the database.
- Such a design is difficult to manage because any changes to one task (like printing player info) might affect other tasks (like printing events or rankings).
- **Result**: It becomes complex to maintain and update since it performs too many different tasks, violating the principle of high cohesion.

A better design would split these responsibilities into different classes, each focusing on one task, which would increase maintainability and clarity.

High Cohesion refers to designing classes that focus on a specific, well-defined responsibility. This is considered a good design because it makes the classes easier to create, maintain, and update.

Example of High Cohesion:

```
class PlayerDatabase {

    ConnectDatabase connectD = new ConnectDatabase();

    PrintAllPlayersInfo allPlayer = new PrintAllPlayersInfo();

    PrintRankings rankings = new PrintRankings();

    CloseDatabase closeD = new CloseDatabase();

    PrintSinglePlayerInfo singlePlayer = new PrintSinglePlayerInfo();

}
```

```
class ConnectDatabase {

   // Logic for connecting to the database

}


class CloseDatabase {

   // Logic for closing the database connection

}


class PrintRankings {

   // Logic for printing players' rankings

}


class PrintAllPlayersInfo {

   // Logic for printing all players' information

}


class PrintSinglePlayerInfo {

   // Logic for printing a single player's information

}
```

Program Analysis:

- Each class in the above example has a specific role: `ConnectDatabase` handles the database connection, `PrintRankings` focuses on printing player rankings, and so on.

- By dividing responsibilities in this manner, the design achieves **high cohesion**, where each class is easy to maintain, modify, and test independently.
- **Result**: Such classes are simpler to manage and update, and changes made to one part of the system do not affect other unrelated components.

This design approach promotes modularity and makes the application scalable and easier to work with.

Module 2 : Class, Object, Method and Constructor

Q.1 What is class?

A **class** in Object-Oriented Programming (OOP) is a blueprint or template used to create objects that share common properties and behaviors. It defines the structure and behavior of objects but is itself a logical entity without physical existence. Classes can contain fields, methods, constructors, blocks, nested class and interface.

Q.2 What is an object?

An **object** in Object-Oriented Programming (OOP) is an instance of a class, which serves as a blueprint for creating objects. It is both a **physical** and **logical entity** that encapsulates data and behavior.

Characteristics of an Object:

4. **State**: Represents the data or attributes of the object, which are defined by the class variables.
5. **Behavior**: Represents the functionality of the object, defined by the methods or functions of the class.
6. **Identity**: A unique identifier for the object, typically used internally by systems like the JVM (Java Virtual Machine) to differentiate between different objects. This identity is not exposed to the user but is used to manage objects during program execution.

Objects are the building blocks in OOP, allowing for the representation of real-world entities in software systems.

Q.3 Parts of class

In Java, a class is a blueprint that defines both the **state** (through variables) and the **behavior** (through methods) of an object.

A class is structured as follows:

java

Copy code

```
class ClassName {

    // Variable declarations: Define the state of the object

    type variableName;



    // Method declarations: Define the behavior of the object

    returnType methodName(parameters) {

        // Method body

    }

}
```

Components:

Variable Declarations: These define the data or properties of the object, representing its state. For example:
java
Copy code
```
String name;  // State of the object

int age;      // State of the object
```

1.

Method Declarations: These define the functionality or actions that the object can perform. For example:
java
Copy code
```
void setName(String newName) {  // Behavior of the object

    name = newName;
}
```

2.

Example:

java

Copy code

```
class Person {
    // Variable declarations: Define state
    String name;
    int age;

    // Method declarations: Define behavior
    void setName(String newName) {
        name = newName;
    }

    void setAge(int newAge) {
        age = newAge;
    }

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

In this example, `Person` has two variables (`name`, `age`) representing its state and three methods (`setName()`, `setAge()`, `displayInfo()`) defining its behavior.

Q.3 Parts of class

We should use **classes** and **objects** in programming because they offer several key benefits:

1. **Modularity**: Classes allow us to break down large programs into smaller, manageable pieces (modules). Each class can be developed and maintained independently, which makes the software more organized.
2. **Information Hiding/Encapsulation**: Classes enable **data encapsulation**, which means that the internal workings of a class can be hidden from the outside world. This ensures that users of a class interact with it only through its public methods, preventing direct access to sensitive data and reducing the chance of unintended interference.
3. **Reusability**: Classes are like blueprints, which means once a class is defined, it can be reused multiple times to create objects. This promotes code reuse, saving time and effort in development.
4. **Maintainability**: By using classes, the software becomes easier to maintain and extend. If you need to modify or update a class, changes can be made in one place without affecting other parts of the program.
5. **Abstraction**: Classes help in abstracting complex systems by allowing programmers to work with high-level concepts (objects) rather than focusing on lower-level details.

In summary, classes and objects promote a structured, efficient, and maintainable way to design software systems.

Q.4 Declaring an Object

Object Declaration is similar to **Variable Declaration**:

- Example: `SalesTaxCalculator obj1;`
- Syntax: `type name;`
  - **Type** is the class name.
  - **Name** is the reference variable used to refer to the object.

Difference Between Variables and Objects:

- **Variables** hold a single type of literal (like an integer or string).
- **Objects** are instances of a class with their own set of instance variables and methods that perform specific tasks based on the methods defined in the class.

Q.5 Initializing an object

Object creation involves three main steps: **declaration**, **instantiation**, and **initialization**.

- **Declaration**: Defining the object type and its reference variable, e.g., `SalesTaxCalculator obj1;`
- **Instantiation**: Creating the object in memory using the `new` keyword.
- **Initialization**: Assigning values to the object's instance variables, which is done using a **constructor**.

In one statement, all these three operations can be combined as:

java

Copy code

```
SalesTaxCalculator obj1 = new SalesTaxCalculator();
```

This single line declares, instantiates, and initializes the object.

Q.6 Accessing Instance Variables

To access and assign values to the instance variables of an object in Java, there are three common methods:

Direct Access: Using the dot operator to directly access or modify the instance variables:

```
obj1.amount = 100.0f;
```

Setter Method: Assign values using a setter method that encapsulates the logic of setting a variable:

```
obj1.setAmount(100.0f);
```

Constructor: Initialize values when the object is created by passing them as arguments to the constructor:

```
SalesTaxCalculator obj1 = new SalesTaxCalculator(100.0f);
```

Each method offers different levels of encapsulation and control over how the values are set.

Q.6 Instance variable (contd.)

In Java, **instance variables** are the variables declared inside a class but outside of any methods. These variables are unique to each instance (object) of the class, meaning that each object will have its own copy of the instance variables. For example:

```
class SalesTaxCalculator {

    float amount;

    float taxRate;

}
```

If two objects `obj1` and `obj2` are created, each will have its own `amount` and `taxRate`:

- **obj1** will have its own values for `amount` and `taxRate`.
- **obj2** will have its own separate set of values for `amount` and `taxRate`.

Static variables, on the other hand, are shared among all instances of a class and are not tied to any specific object. The initialization of instance variables is typically done via **constructors**, which ensure that each object has its own set of initialized values.

Here's an example of how you can create two objects, `obj1` and `obj2`, of the `SalesTaxCalculator` class, each with its own set of instance variables `amount` and `taxRate`:

```
class SalesTaxCalculator {
    float amount;

    float taxRate;


    // Constructor to initialize amount and taxRate

    SalesTaxCalculator(float amount, float taxRate) {

        this.amount = amount;

        this.taxRate = taxRate;

    }


    // Method to display the values

    void display() {

        System.out.println("Amount: " + amount + ", Tax Rate: " +
taxRate);

    }

}


public class Main {

    public static void main(String[] args) {

        // Creating obj1 with specific values

        SalesTaxCalculator obj1 = new SalesTaxCalculator(100.0f,
5.0f);


        // Creating obj2 with different values
        SalesTaxCalculator obj2 = new SalesTaxCalculator(200.0f,
7.5f);


        // Displaying values for obj1

        System.out.println("Object 1:");

        obj1.display();


        // Displaying values for obj2

        System.out.println("Object 2:");

        obj2.display();

    }

}
```

Explanation:

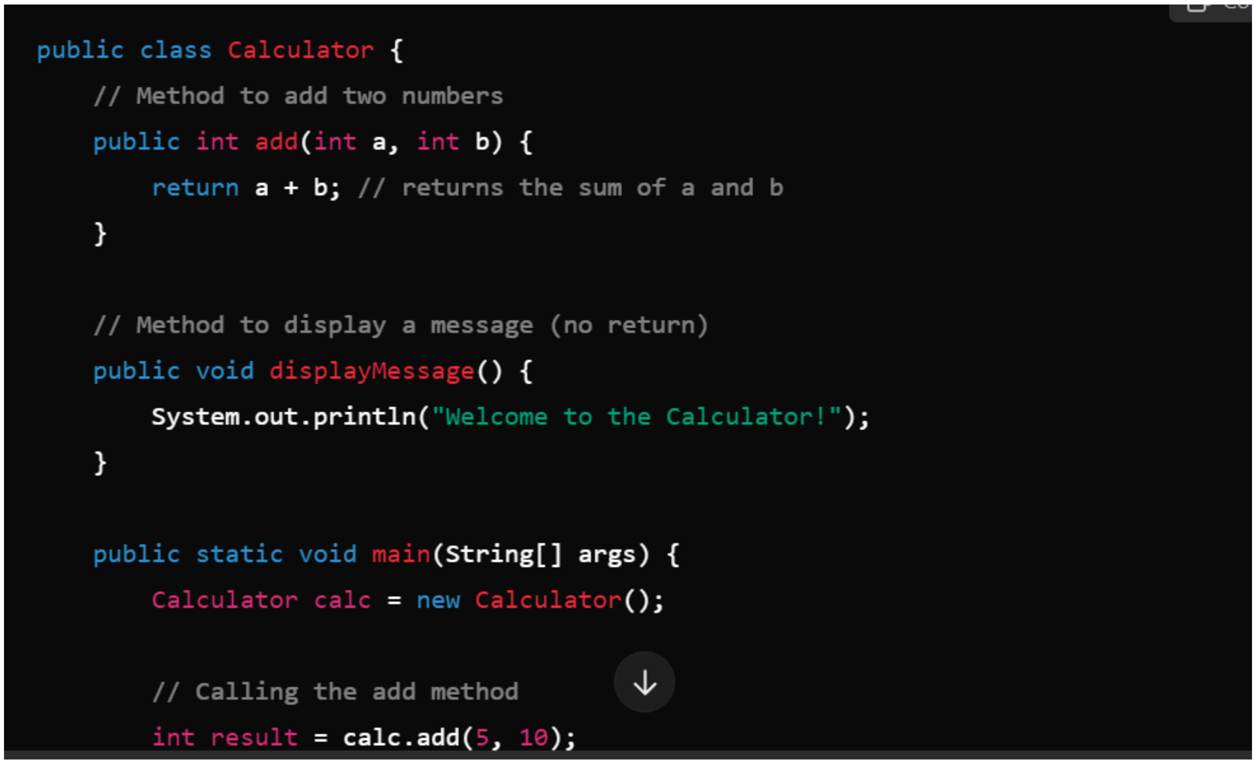
- `SalesTaxCalculator` class has two instance variables: `amount` and `taxRate`.
- A constructor is used to initialize these variables.
- Two objects `obj1` and `obj2` are created, each with different values for `amount` and `taxRate`.
- Each object has its own copy of the instance variables, which is demonstrated by the `display()` method.

Q.7 Methods

In Java, **methods** are similar to functions in other programming languages but are always defined within a class. Here's an overview of why and how to use methods, along with their general structure:

Why Use Methods?

- **Code Reusability**: You can write a method once and reuse it multiple times in different parts of the program.
- **Parameterization**: Methods allow you to pass parameters (data) and customize their behavior based on the input.
- **Top-Down Programming**: Methods help break down complex tasks into smaller, more manageable sub-tasks.
- **Code Simplification**: They help in organizing code and improving readability.



```java
public class Calculator {
    // Method to add two numbers
    public int add(int a, int b) {
        return a + b; // returns the sum of a and b
    }

    // Method to display a message (no return)
    public void displayMessage() {
        System.out.println("Welcome to the Calculator!");
    }

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

        // Calling the add method
        int result = calc.add(5, 10);
```

Q.8 Method invocation

In Java, **methods** must be invoked by an object or a class (in the case of static methods) to run. Here's a breakdown of how this works:

Key Points on Method Invocation:

- **Method Invocation**: Methods can't operate independently. They need to be called (invoked) by an object of the class in which they're defined.
- **Passing Values**: When an object invokes a method, it can pass specific values (called **parameters**) into the method.
- **Return Values**: Methods may return values if defined to do so.

Parameters:

- **Formal Parameters**: These are variables declared in the method definition that act as placeholders for the values passed when the method is called. For example, in `add(int a, int b)`, `a` and `b` are formal parameters.
- **Actual Parameters**: These are the real values provided when calling the method. For instance, in `add(5, 10)`, `5` and `10` are actual parameters.

Example:

```
public class Calculator {

    // Method with formal parameters a and b

    public int add(int a, int b) {

        return a + b;

    }



    public static void main(String[] args) {

        Calculator calc = new Calculator();



        // Method invocation with actual parameters 5 and 10

        int result = calc.add(5, 10); // result holds the value 15

        System.out.println("Sum: " + result); // Output: Sum: 15

    }

}
```

Important Notes:

- **Consistency**: The number and types of **actual parameters** passed during the method call must match the **formal parameters** in the method declaration.
- **Pass by Value**: Java passes arguments by value, meaning a copy of the value is passed to the method. Java doesn't support passing by reference (i.e., passing memory

addresses like pointers in some other programming languages). This means changes made to the parameters inside the method do not affect the original values outside the method.

Q.9 Constructors

Constructors in Java are special methods used to initialize objects when they are created. Here's a simplified overview:

Key Points About Constructors:

1. **Purpose**:
    - Constructors are used to set up initial values for an object's fields when it is created.
    - They ensure that the object is properly initialized before it is used.
2. **Naming and Syntax**:
    - A constructor must have the same name as the class in which it is defined.
    - Constructors have no return type, not even `void`. The constructor itself is implicitly of the class type.
3. **Automatic Invocation**:
    - Constructors are automatically called when a new object is created using the `new` operator. For example, `new MyClass()` invokes the constructor of `MyClass`.
4. **Types of Constructors**:
    - **No-Arg Constructor (Default Constructor)**:
        - This constructor does not take any parameters and initializes object fields with default values.

```
public class MyClass {

    public MyClass() {

        // Constructor code here

    }

}
```

    - **Parameterized Constructor**:
        - This constructor takes arguments to initialize the object with specific values.

```
public class MyClass {

    int value;

    public MyClass(int value) {

        this.value = value;

    }

}
```

- **Explicit Constructor**:
  - Any constructor defined by the programmer that is not the default constructor.

java
Copy code
```
public class MyClass {

    int value;

    public MyClass() {

        // Default constructor

    }

    public MyClass(int value) {

        this.value = value; // Parameterized constructor

    }

}
```

Example:

```
public class Car {
    String color;

    int year;



    // No-arg Constructor

    public Car() {

        color = "Unknown";

        year = 0;

    }


    // Parameterized Constructor

    public Car(String color, int year) {

        this.color = color;

        this.year = year;

    }


    public static void main(String[] args) {

        // Creating objects

        Car myCar1 = new Car(); // Calls no-arg constructor

        Car myCar2 = new Car("Red", 2022); // Calls parameterized
constructor
```

```
        System.out.println("Car1 - Color: " + myCar1.color + ", Year: " + myCar1.year);

        System.out.println("Car2 - Color: " + myCar2.color + ", Year: " + myCar2.year);

    }

}
```

In this example:

- `Car myCar1 = new Car();` initializes `myCar1` with default values.
- `Car myCar2 = new Car("Red", 2022);` initializes `myCar2` with specified values.

Here's a simplified summary of constructors in Java:

- **No-Arg Constructor**: A constructor that does not take any parameters. It initializes an object with default values.
- **Naming**: Constructors must have the same name as the class they belong to.
- **Return Type**: Constructors do not have a return type, not even `void`. They implicitly return the class type itself.
- **Invocation**: Constructors are called automatically using the `new` operator when an object is created.
- **Role**: The primary role of constructors is to initialize the fields of an object when it is created.

Example

```
public class MyClass {

    int value;



    // No-arg constructor

    public MyClass() {

        value = 0; // Initialize value with default
    }


    // Parameterized constructor

    public MyClass(int value) {

        this.value = value; // Initialize value with provided argument

    }


    public static void main(String[] args) {

        MyClass obj1 = new MyClass(); // Calls no-arg constructor

        MyClass obj2 = new MyClass(10); // Calls parameterized
constructor


        System.out.println("obj1 value: " + obj1.value); // Output:
obj1 value: 0

        System.out.println("obj2 value: " + obj2.value); // Output:
obj2 value: 10

    }

}
```

In this example:

- The `MyClass()` constructor initializes the `value` field to `0` by default.
- The `MyClass(int value)` constructor allows setting the `value` field to a specific integer when creating the object.

Q. Default Constructors:

In Java, constructors are essential for initializing objects, and there are two main types:

1. Default Constructor

- **Definition**: A default constructor is a no-argument constructor automatically provided by Java if no constructors are explicitly defined in the class.
- **Purpose**: It initializes objects with default values (e.g., `0` for integers, `null` for objects).
- **Characteristics**:
    - **No Parameters**: It does not take any parameters.
    - **Automatic Provision**: Only provided if no other constructors are defined.

Example:

```
public class Example {

    int number;



    // Default constructor provided by Java

    // Initializes 'number' to 0 by default

}



public class Test {

    public static void main(String[] args) {

        Example obj = new Example(); // Uses default constructor

        System.out.println(obj.number); // Output: 0

    }

}
```

2. Parameterized Constructor

- **Definition**: A parameterized constructor allows you to initialize objects with specific values passed as arguments.
- **Purpose**: It provides the ability to initialize objects with different values, making it flexible and useful for more complex initialization.
- **Characteristics**:
    - **Has Parameters**: It takes one or more parameters.
    - **Custom Initialization**: Allows setting initial values based on parameters passed.

Example:

```
public class Example {
    int number;


    // Parameterized constructor
    Example(int num) {
        number = num;
    }
}


public class Test {
    public static void main(String[] args) {
        Example obj1 = new Example(10); // Initializes 'number' with
10
        Example obj2 = new Example(20); // Initializes 'number' with
20
        System.out.println(obj1.number); // Output: 10
        System.out.println(obj2.number); // Output: 20
    }
}
```

In summary:

- **Default Constructor**: Provides default values and is automatically provided if no other constructor is defined.
- **Parameterized Constructor**: Allows specifying initial values for an object, offering more control over object initialization.

Q. Static Keyword

The `static` keyword in Java is used to declare class-level variables and methods that are shared among all instances of the class. Here's a breakdown of how `static` works and its effects:

Static Keyword

Static Variables

- **Definition**: Static variables are variables that are shared among all instances of a class. They belong to the class rather than to any individual object.
- **Characteristics**:
  - **Single Copy**: There is only one copy of a static variable per class, regardless of how many objects are created.
  - **Shared Across Objects**: All instances of the class share the same static variable, meaning changes to it are reflected across all instances.
- **Declaration**: Use the `static` keyword when declaring the variable.

Example:

```
public class Example {

    static int count = 0; // Static variable



    Example() {

        count++; // Increment static variable for each new instance

    }
    public static void displayCount() {

        System.out.println("Count: " + count); // Access static
variable

    }

}


public class Test {

    public static void main(String[] args) {

        Example obj1 = new Example();

        Example obj2 = new Example();

        Example obj3 = new Example();


        Example.displayCount(); // Output: Count: 3

    }

}
```

Static Methods

- **Definition**: Static methods belong to the class rather than any instance of the class. They can be called without creating an object of the class.
- **Characteristics**:
    - **No Access to Instance Variables**: Static methods can only directly access static variables and other static methods. They cannot access instance variables or methods.
    - **Called on the Class**: They are invoked using the class name.

Example:

```java
public class Example {
    static int count = 0;

    static void increment() {
        count++; // Modify static variable
    }

    static void displayCount() {
        System.out.println("Count: " + count);
    }
}

public class Test {
    public static void main(String[] args) {
        Example.increment();
        Example.increment();
        Example.displayCount(); // Output: Count: 2
    }
}
```

Static Blocks

- **Definition**: Static blocks are used for static initializations of a class. They run once when the class is first loaded into memory.
- **Characteristics**:

    - **Initialization**: Useful for initializing static variables or performing actions that need to be done only once.

Example:

```
public class Example {
    static int count;


    // Static block
    static {
        count = 10; // Initialization of static variable
        System.out.println("Static block executed");
    }


    public static void main(String[] args) {
        System.out.println("Count: " + count);
    }
}
```

Summary:

- **Static Variables**: Shared across all instances; only one copy exists.
- **Static Methods**: Belong to the class, not to objects; can be called on the class itself.
- **Static Blocks**: Used for static initializations and run once when the class is loade

Q. What is Constructor overloading in Java

Constructor overloading in Java refers to the ability to have multiple constructors in the same class, each with a different parameter list. This allows for different ways of initializing objects of that class based on different inputs. Here's a breakdown of constructor overloading:

Constructor Overloading

Concept

- **Multiple Constructors**: You can define more than one constructor in a class, each with a unique combination of parameters.
- **Differentiation**: The Java compiler distinguishes between these constructors by the number and types of parameters, allowing you to initialize objects in various ways.

Key Points

1. **Different Parameter Lists**: Each constructor must have a different parameter list, meaning they must differ in the number, type, or both types of parameters.
2. **Initialization**: Each overloaded constructor can perform different tasks or initialize the object in different ways based on the parameters provided.
3. **No Return Type**: Like all constructors, overloaded constructors do not have a return type, not even `void`.

Example

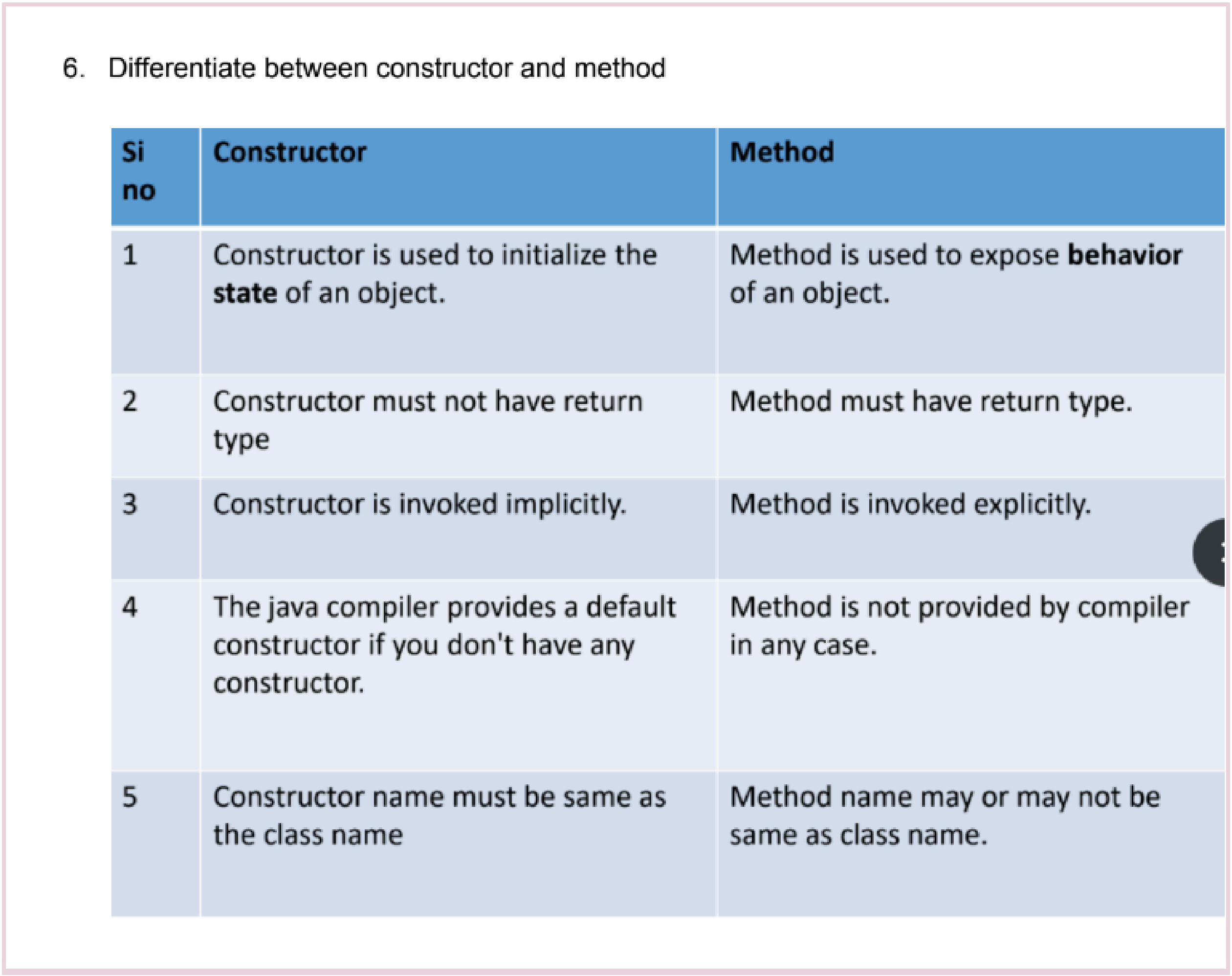
```
public class Rectangle {

    int length;

    int width;



    // No-argument constructor

    Rectangle() {

        length = 0;

        width = 0;

    }



    // Parameterized constructor with two parameters

    Rectangle(int l, int w) {

        length = l;
        width = w;

    }


    // Parameterized constructor with one parameter

    Rectangle(int side) {

        length = side;

        width = side;

    }


    void display() {

        System.out.println("Length: " + length + ", Width: " + width);

    }


    public static void main(String[] args) {

        Rectangle rect1 = new Rectangle(); // Calls no-argument
constructor

        Rectangle rect2 = new Rectangle(10, 20); // Calls constructor
with two parameters

        Rectangle rect3 = new Rectangle(15); // Calls constructor with
one parameter


        rect1.display(); // Output: Length: 0, Width: 0

        rect2.display(); // Output: Length: 10, Width: 20

        rect3.display(); // Output: Length: 15, Width: 15
    }
}
```

Explanation

- **No-Argument Constructor**: Initializes `length` and `width` to `0`.
- **Two-Parameter Constructor**: Allows initialization with specific `length` and `width`.
- **One-Parameter Constructor**: Initializes a square with the given side length.

Summary: Constructor overloading allows a class to have multiple constructors with different parameter lists, enabling various ways to initialize objects. The Java compiler selects the appropriate constructor based on the arguments provided during object creation.

Q.



6. Differentiate between constructor and method

Si no	Constructor	Method
1	Constructor is used to initialize the **state** of an object.	Method is used to expose **behavior** of an object.
2	Constructor must not have return type	Method must have return type.
3	Constructor is invoked implicitly.	Method is invoked explicitly.
4	The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
5	Constructor name must be same as the class name	Method name may or may not be same as class name.

Q. What is a destructor in java?

A destructor is a special member function of a class in object-oriented programming languages like C++ that is automatically called when an object is destroyed. This happens when the object goes out of scope or is explicitly deleted using the `delete` operator. The destructor is responsible for cleaning up resources allocated to the object, such as memory or file handles, before the object is removed from memory.

Key characteristics of destructors:

- It has the same name as the class but is preceded by a tilde (`~`).
- It takes no parameters and does not return any value.
- It's invoked automatically and ensures that the object is properly cleaned up to avoid memory leaks.

In Java, destructors don't exist, but a similar concept is handled by the `finalize()` method. However, the actual process of deallocating memory is managed by the **garbage collector** in Java, making explicit memory management unnecessary in most cases.

Example in C++:

```
class MyClass {

public:

    ~MyClass() {

        // cleanup code here

    }

};
```

Q. What are access specifiers/modifiers in java?

The access modifiers in Java specify the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

1. Public: Class, Method, Constructor or Interface which are public can be accessed from any other class in the program, within the package and even outside the package.

2. Protected: The access level of methods, variables and constructors declared protected is all classes within the package and child classes outside the package. It cannot be accessed from outside the package without making a child class. Cannot be applied to class and interface.

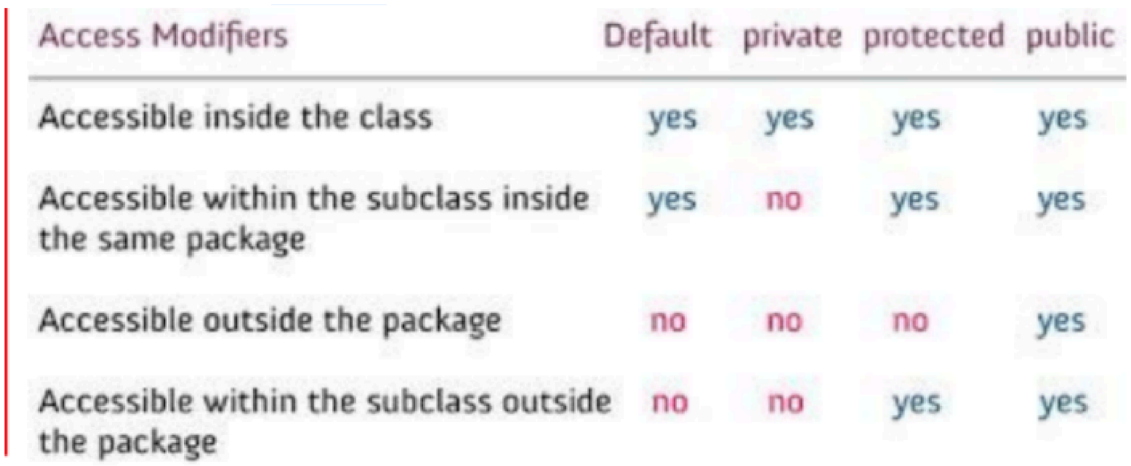
3. Private: Methods, Variables and Constructors which are private can only be accessed from within the class, classes and interfaces cannot be private.

4. Default: Default access modifier is applied when we do not explicitly

declare an access modifier. The variable or method which has default

access level can be accessed from any other class within the same

Package.



Access Modifiers	Default	private	protected	public
Accessible inside the class	yes	yes	yes	yes
Accessible within the subclass inside the same package	yes	no	yes	yes
Accessible outside the package	no	no	no	yes
Accessible within the subclass outside the package	no	no	yes	yes

Q.9. What are the ways to read input in Java?

In Java, there are several ways to read input from the user. Here are some commonly used methods:

1. **Command Line Arguments (Command Line Interpreter)**:
    - Input can be passed to the `main` method through command-line arguments when the program is executed.
    - The `main` method's signature contains a `String[] args` parameter, which stores the arguments passed via the command line.

Example:
```
public class CommandLineExample {

    public static void main(String[] args) {

        if (args.length > 0) {

            System.out.println("First argument: " + args[0]);

        } else {
            System.out.println("No arguments provided.");

        }

    }

}
```

2. **BufferedReader Class**:
   - This is part of the `java.io` package and is used for reading text from an input stream efficiently. It's typically wrapped around `InputStreamReader`.

Example:

```
import java.io.*;



public class BufferedReaderExample {

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

        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        System.out.println("Enter your name:");

        String name = reader.readLine();

        System.out.println("Your name is: " + name);

    }

}
```

3. **Scanner Class**:
   - The `Scanner` class (from the `java.util` package) is one of the most commonly used ways to read input. It supports reading various data types like strings, integers, and floats.

Example:

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

```
public class ScannerExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your age:");

        int age = scanner.nextInt();

        System.out.println("Your age is: " + age);

    }

}
```

Each method has its own use case based on the scenario and the complexity of input processing needed.

Q. 10. What is a static variable in Java?

When a variable is declared with keyword static it is called a class variable.

All instances of the class share the same variable, a class variable can be accessed without the

need of creating an object which makes it memory efficient.

Q.11 What are the ways to read input in Java?

In Java, there are several ways to read input from the user. Here are some commonly used methods:

1. **Command Line Arguments (Command Line Interpreter)**:
   - Input can be passed to the `main` method through command-line arguments when the program is executed.
   - The `main` method's signature contains a `String[] args` parameter, which stores the arguments passed via the command line.

Example:

```
public class CommandLineExample {

    public static void main(String[] args) {

        if (args.length > 0) {

            System.out.println("First argument: " + args[0]);

        } else {

            System.out.println("No arguments provided.");

        }

    }

}
```

2. **BufferedReader Class**:
   - This is part of the `java.io` package and is used for reading text from an input stream efficiently. It's typically wrapped around `InputStreamReader`.

Example:

```
import java.io.*;



public class BufferedReaderExample {

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

        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));

        System.out.println("Enter your name:");

        String name = reader.readLine();

        System.out.println("Your name is: " + name);
    }

}
```

3. **Scanner Class**:
   - The `Scanner` class (from the `java.util` package) is one of the most commonly used ways to read input. It supports reading various data types like strings, integers, and floats.

Example:

```
import java.util.Scanner;


public class ScannerExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your age:");

        int age = scanner.nextInt();

        System.out.println("Your age is: " + age);

    }

}
```

Each method has its own use case based on the scenario and the complexity of input processing needed.

Q 10. What is a static variable in Java?

A **static variable** in Java is a class-level variable that is shared among all instances of a class. When a variable is declared with the `static` keyword, it is associated with the class itself rather than with individual objects of the class. This means that all instances of the class share the same copy of the static variable, and any changes to the variable will be reflected across all objects.

Key points about static variables:

- **Shared among all instances**: All objects of the class use the same memory for the static variable, rather than each object having its own copy.
- **Accessible without an object**: You can access a static variable directly through the class name, without needing to create an instance.
- **Memory efficiency**: Since only one copy of the variable exists, it helps save memory, especially when many instances of the class are created.

Example:

```
class MyClass {

    static int counter = 0;  // static variable



    MyClass() {

        counter++;  // incrementing the static variable

    }

}



public class StaticExample {

    public static void main(String[] args) {

        MyClass obj1 = new MyClass();

        MyClass obj2 = new MyClass();

        System.out.println(MyClass.counter);  // Output will be 2

    }

}
```

In this example, the static variable `counter` is shared across all objects of `MyClass`, and its value increases every time a new object is created.

Q.11 What is a static method in java?

A **static method** in Java is a method that belongs to the class rather than an instance of the class. It can be invoked without creating an object of the class and is generally used to perform operations that do not depend on instance-specific data.

Key points about static methods:

- **Belongs to the class**: Static methods are called on the class itself, not on an object.
- **No object creation needed**: You can call static methods directly using the class name.
- **Can access static variables**: Static methods can access and modify static variables, as both belong to the class and are independent of any object.

Example:

```
class MyClass {

    static int counter = 0;  // static variable



    static void incrementCounter() {

        counter++;  // static method can modify the static variable

    }

}



public class StaticMethodExample {

    public static void main(String[] args) {

        MyClass.incrementCounter();  // no need to create an object

        System.out.println(MyClass.counter);  // Output: 1

    }

}
```

Disadvantages of static methods:

1. **Cannot access non-static data**: Static methods cannot directly access non-static (instance) variables or call non-static methods because non-static members belong to an instance, not to the class.
   - Example: You cannot do something like `this.instanceVariable` inside a static method.
2. **Cannot use `this` or `super`**: Since static methods do not belong to any specific object, they cannot use the `this` keyword (which refers to the current object) or `super` (used to refer to superclass members in inheritance).

For instance:

```
class MyClass {

    int instanceVariable = 10;



    static void staticMethod() {

        // instanceVariable++; // This would cause a compile-time
error

    }

}
```

In this example, the `staticMethod` cannot access or modify `instanceVariable` because it's non-static, and the method cannot use `this` to reference an object of the class.

Q. 13) Automatic Garbage Collection

Garbage Collection (GC) in Java is an automatic process that helps manage memory by reclaiming and removing objects that are no longer referenced or in use. It frees up memory by destroying unreferenced objects, making the program more memory-efficient and allowing space for new objects. The Java Garbage Collector operates in the background to ensure that memory is managed without explicit intervention from the developer.

Key Points:

- **Automatic memory management**: Java developers do not need to manually allocate or deallocate memory.
- **Garbage collection is efficient**: By removing unreferenced objects from heap memory, it prevents memory leaks and optimizes resource use.
- **Operates on heap memory**: The GC works within the heap area where Java objects are dynamically allocated.

Phases of Garbage Collection:

1. **Mark Phase**:
    - The Garbage Collector scans the memory to identify which objects are still in use (referenced) and which are no longer referenced.
    - During this step, it marks the objects that are reachable (still in use) and those that are unreachable (unreferenced).
2. **Sweep Phase**:
    - After identifying the unreferenced objects in the mark phase, the Garbage Collector proceeds to remove (or sweep away) these unreferenced objects.
    - This frees up space in the heap for future object allocations.

Example:

java

Copy code

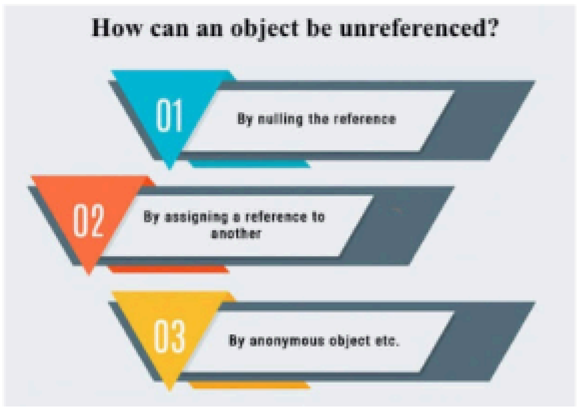
```java
public class GarbageCollectionExample {

    public static void main(String[] args) {

        String str = new String("Hello");

        str = null;  // Now the object is unreferenced



        // Suggesting GC to run

        System.gc();  // This is a request, GC may or may not run
immediately
```

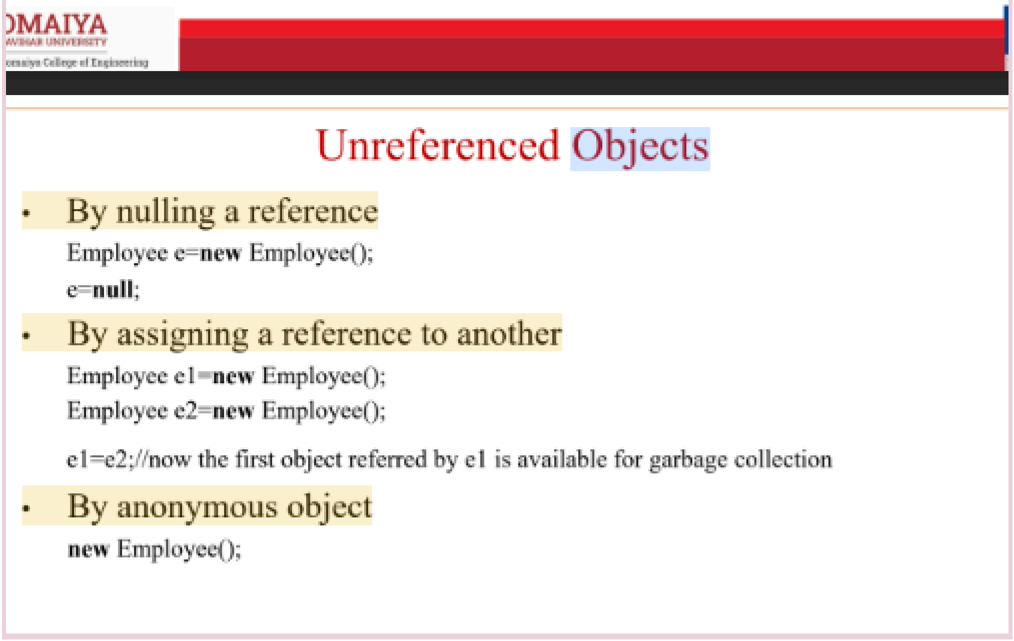
```
    }


    @Override

    protected void finalize() {

        System.out.println("Garbage collected.");

    }

}
```

In this example, the object originally referenced by `str` becomes unreferenced when `str` is set to `null`. Java’s Garbage Collector will eventually identify this object as no longer needed and will reclaim the memory during the sweep phase.

Though `System.gc()` is a way to request garbage collection, it's important to note that the Java Virtual Machine (JVM) decides when the garbage collection actually occurs.

Q.14 Unreferenced objects


How can an object be unreferenced?
01
By nulling the reference
02
By assigning a reference to another
03
By anonymous object etc.



Unreferenced Objects

- By nulling a reference

```
Employee e=new Employee();
e=null;
```

- By assigning a reference to another

```
Employee e1=new Employee();
Employee e2=new Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection
```

- By anonymous object

```
new Employee();
```

MODULE 3: ARRAYS

Q.1 What are arrays in Java?

In Java, an **array** is a collection of elements that are of the same type, stored in contiguous memory locations. Java arrays are objects, meaning they have a fixed size once created, and they store elements of a specific data type. Arrays in Java can hold primitive data types (like `int`, `char`, etc.) or objects (like `String`, `Integer`, etc.).

Key Points:

- **Arrays are objects**: In Java, arrays are considered objects, which means they are created on the heap and accessed via references.
- **Fixed size**: Once an array is created, its size cannot be changed. You must specify the size when creating an array.
- **Zero-based index**: Array indexing starts from 0, meaning the first element is accessed using index 0.

Types of Arrays in Java:

1. **Single Dimensional Array**:
    - This is the most common type of array, where elements are stored in a single row (or a single list of elements).

Example:
```
int[] arr = new int[5];  // Declaring an array of size 5

arr[0] = 10;             // Assigning values

arr[1] = 20;

System.out.println(arr[0]);  // Output: 10
```

2. **Multidimensional Array**:
    - An array of arrays, where each element in the main array is itself an array.
    - The most common example is the **two-dimensional array**, which can be visualized as a table with rows and columns.

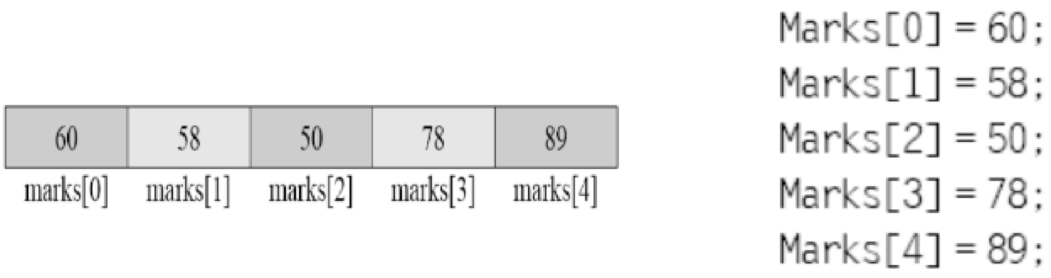
Example:

```
int[][] matrix = new int[2][3];  // 2 rows and 3 columns
```

```
matrix[0][0] = 1;

matrix[0][1] = 2;

matrix[1][2] = 3;

System.out.println(matrix[1][2]);  // Output: 3
```

In a multidimensional array, elements are accessed by specifying more than one index, like `matrix[row][column]`.

Q.2 1D Arrays



```
Marks[0] = 60;
Marks[1] = 58;
Marks[2] = 50;
Marks[3] = 78;
Marks[4] = 89;
```

In a **one-dimensional (1D) array**, a single subscript or index is used to access individual elements. Each element in the array is referred to by its index, which starts from 0 and goes up to `n-1`, where `n` is the number of elements in the array.

Key Points:

- **Indexing starts from 0**: The first element of the array has an index of 0, and the last element has an index of `n-1`.
- **Contiguous memory**: The array elements are stored in contiguous memory locations.

When creating an array in Java, the process typically involves three main steps: **declaring the array**, **allocating memory for the array**, and **initializing or assigning values** to the array elements. The array behaves like an object, and the size must be specified when allocating memory.

Steps for Creating an Array:

1. **Declaring an Array**:
    - There are two ways to declare an array:
        - `type arrayname[];`
        - `type[] arrayname;`

Examples:
java
Copy code
```
int marks[];   // Declaration using the first method

int[] marks;   // Declaration using the second method
```

2. **Creating Memory Locations (Allocating Space)**:
    - After declaring an array, you must allocate memory by specifying the size of the array.
    - This is done using the `new` keyword followed by the data type and the size of the array.

Example:

```
marks = new int[5];  // Allocating space for 5 integer elements
```

3. **Initializing/Assigning Values to an Array**:
    - Once the array is declared and memory is allocated, you can initialize or assign values to its elements using the index.
    - Array elements are accessed using zero-based indexing.

Example:

```
marks[0] = 85;  // Assigning value 85 to the first element

marks[1] = 90;  // Assigning value 90 to the second element
```

```
}
```

Alternatively, you can declare, allocate, and initialize an array all in one step:

```
int[] marks = {85, 90, 78, 92, 88};  // Declaration, allocation, and
initialization in one line
```

Example:

```
public class ArrayExample {

    public static void main(String[] args) {

        int[] marks;             // Step 1: Declaring an array

        marks = new int[5];      // Step 2: Allocating memory for 5
elements

        marks[0] = 85;           // Step 3: Initializing array
elements

        marks[1] = 90;

        marks[2] = 78;

        marks[3] = 92;

        marks[4] = 88;


        // Displaying the array elements

        for (int i = 0; i < marks.length; i++) {

            System.out.println("Marks " + (i + 1) + ": " + marks[i]);

        }

    }

}
```

Output:

```
Marks 1: 85

Marks 2: 90

Marks 3: 78

Marks 4: 92

Marks 5: 88
```

This example demonstrates the entire process of declaring, allocating memory, initializing, and accessing the elements of an array.

For example, if we declare an array `marks` to store the marks of 5 students:

```
int[] marks = new int[5];  // declaring an array of 5 integers
```

The computer reserves 5 contiguous memory locations for this array, and we can access the individual marks using the index.

Example of array with 5 elements:

Index	Value (marks[i])
0	marks[0]
1	marks[1]
2	marks[2]

3	marks[3]
4	marks[4]

If the array is initialized with values, it could look like:

```
int[] marks = {85, 90, 78, 92, 88};  // array initialization

System.out.println(marks[0]);  // Output: 85

System.out.println(marks[4]);  // Output: 88
```

Accessing and modifying elements:

- To access an element, use the array name followed by the index in square brackets (e.g., `marks[2]`).

You can also modify elements in the array by assigning new values using their index:
java
Copy code

```
marks[2] = 80;  // changing the value of the third element
```

-

In summary, a one-dimensional array allows you to store and manipulate a sequence of elements using a single index to access or update each element.

Q.2 **Java Collection Interface**

The Java Collection Framework provides a standardized architecture to store, manage, and manipulate groups of objects. It simplifies the tasks of searching, sorting, inserting, manipulating, and deleting data. The framework consists of various interfaces and classes to handle different types of collections, like lists, sets, queues, and maps, in an efficient and flexible manner.

Key Features of the Java Collection Framework:

- Unified architecture: Provides a consistent way to work with different types of data structures.
- Operations: Supports essential data operations such as searching, sorting, insertion, updating, and deletion.
- Flexibility: Offers multiple data structures to handle various use cases.

Key Interfaces:

1. Set: A collection that does not allow duplicate elements.
   - Implementing classes: `HashSet`, `LinkedHashSet`, `TreeSet`.
2. List: An ordered collection that allows duplicate elements and maintains the insertion order.
   - Implementing classes: `ArrayList`, `LinkedList`, `Vector`.
3. Queue: A collection used to hold elements prior to processing, typically following a FIFO (First-In-First-Out) order.
   - Implementing classes: `PriorityQueue`, `LinkedList`.
4. Deque: A double-ended queue that allows elements to be added or removed from both ends.
   - Implementing classes: `ArrayDeque`, `LinkedList`.

Q. 3 List Interface

The List Interface in Java is a child interface of the `Collection` interface and represents an ordered collection of objects. It allows duplicate elements and provides positional access, meaning you can access elements based on their index.

Key Features of the List Interface:

- Ordered collection: Elements are stored in the order they are inserted, and they can be accessed by their index.
- Allows duplicates: A `List` can contain multiple elements with the same value.
- Zero-based indexing: The first element is at index 0, and the last element is at index `size()-1`.

Commonly Used Classes that Implement List Interface:

1. ArrayList: A resizable array implementation that offers fast access to elements.
2. LinkedList: A doubly linked list implementation, ideal for frequent insertions and deletions.
3. Vector: A synchronized version of `ArrayList`, which is thread-safe.
4. Stack: A subclass of `Vector` that follows the Last-In-First-Out (LIFO) principle.

Q. 3 List Interface

The List Interface in Java is a child interface of the `Collection` interface and represents an ordered collection of objects. It allows duplicate elements and provides positional access, meaning you can access elements based on their index.

Key Features of the List Interface:

- Ordered collection: Elements are stored in the order they are inserted, and they can be accessed by their index.
- Allows duplicates: A `List` can contain multiple elements with the same value.
- Zero-based indexing: The first element is at index 0, and the last element is at index `size()-1`.

Commonly Used Classes that Implement List Interface:

1. ArrayList: A resizable array implementation that offers fast access to elements.
2. LinkedList: A doubly linked list implementation, ideal for frequent insertions and deletions.
3. Vector: A synchronized version of `ArrayList`, which is thread-safe.
4. Stack: A subclass of `Vector` that follows the Last-In-First-Out (LIFO) principle.

Q.7 ArrayList

• Java ArrayList class uses a dynamic array for storing the elements(there is no size

limit)

• We can add or remove elements anytime, much more flexible than the traditional array

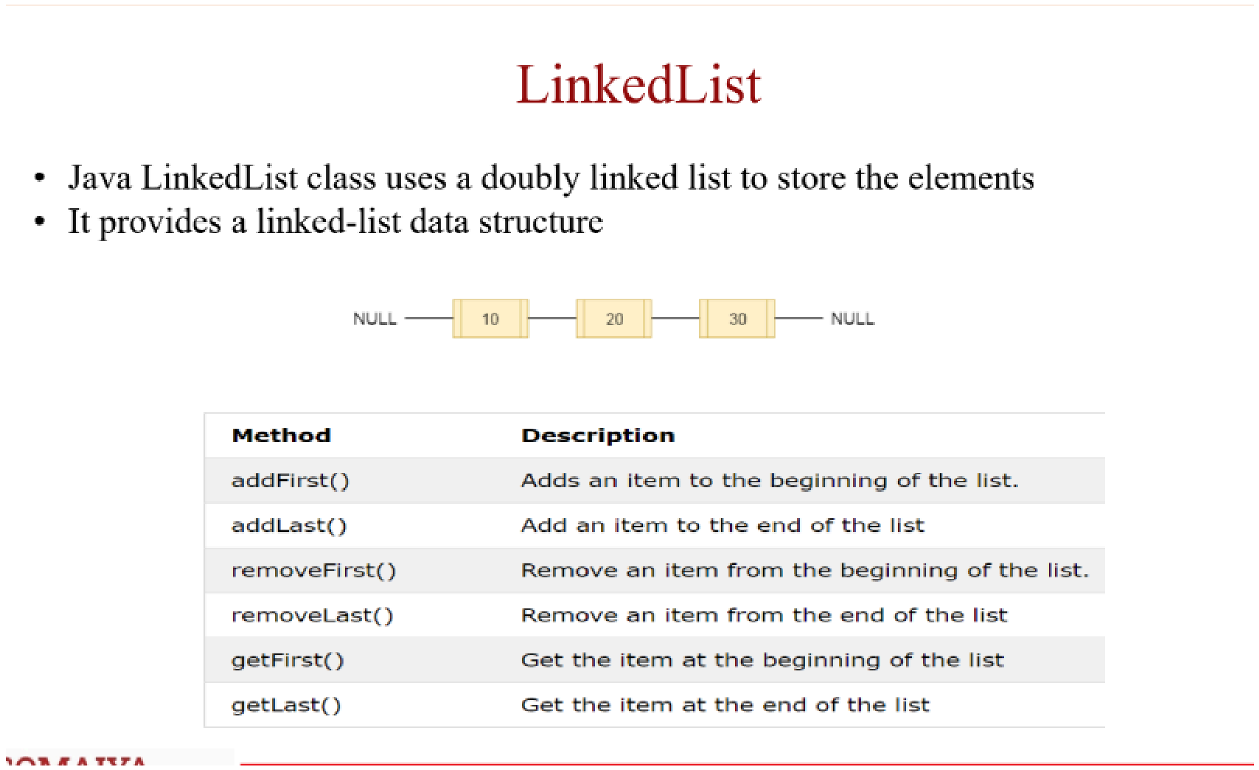
• It is found in the java.util package

• The ArrayList in Java can have the duplicate elements also

• It implements the List interface so we can use all the methods of List interface here.

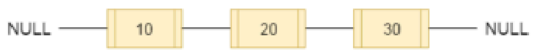
• The ArrayList maintains the insertion order internally

Q.7 LinkedList



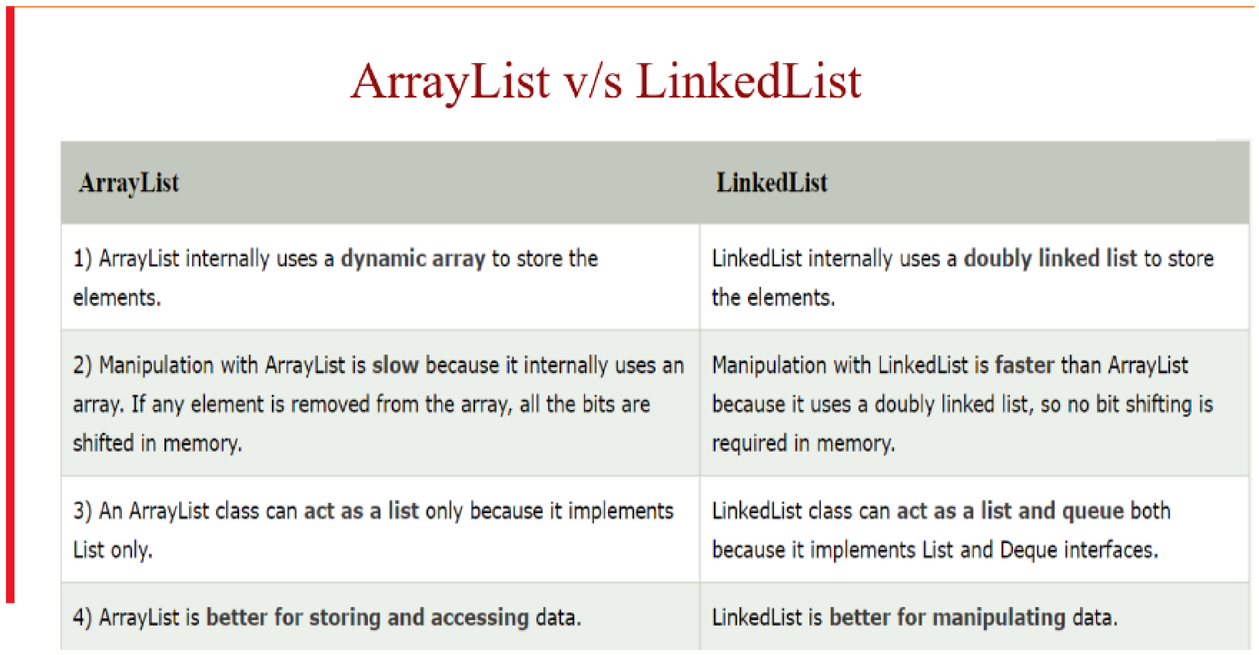
LinkedList

- Java LinkedList class uses a doubly linked list to store the elements
- It provides a linked-list data structure



Method	Description
addFirst()	Adds an item to the beginning of the list.
addLast()	Add an item to the end of the list
removeFirst()	Remove an item from the beginning of the list.
removeLast()	Remove an item from the end of the list
getFirst()	Get the item at the beginning of the list
getLast()	Get the item at the end of the list

Q. DIFFERENCE



ArrayList v/s LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Q.9 Vectors

• Vector implements a dynamic array of objects

• Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program

• Vector can contain heterogeneous objects

• We cannot store elements of primitive data type; first it need to be converted to objects. A vector can store any objects

• Its defined in java.util package and class member of the Java Collections Framework

• Vector implements List Interface

• A vector has an initial capacity, if this capacity is reached then size of vector automatically increases

• This default initial capacity of vectors are 10

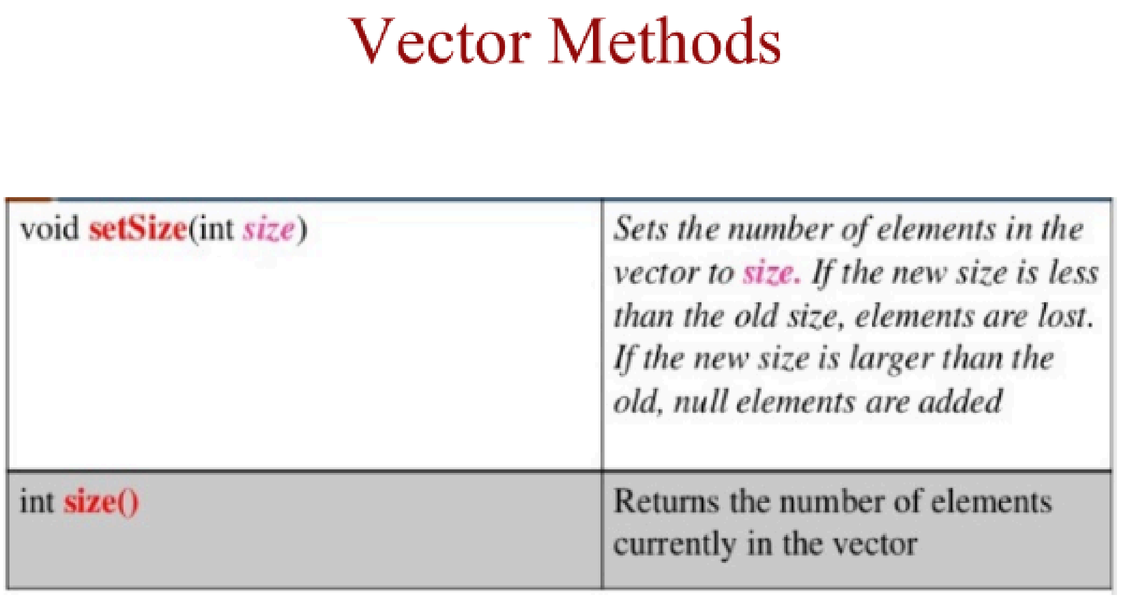
• Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement arguments

• To traverse elements of a vector class we use Enumeration interface

Vector Methods

void **addElement**(Object *element*)	The object specified by *element* is added to the vector
int **capacity()**	Returns the capacity of the vector
boolean **contains(**Object *element*)	Returns **true** if **element** is contained by the vector, else **false**
void **copyInto**(Object *array[]*)	The elements contained in the invoking vector are copied into the array specified by **array[]**
elementAt(int *index*)	Returns the element at the location specified by *index*
Object **firstElement().**	Returns the first element in the vector

void **insertElementAt**(Object *element*, int *index*)	*Adds* *element* to the vector at the location specified by *index*
boolean **isEmpty()**	Returns **true** if Vector is empty, **else false**
Object **lastElement()**	Returns the last element in the vector
void **removeAllElements()**	Empties the vector. After this method executes, the size of vector is zero.
void **removeElementAt**(int *index*)	Removes element at the location specified by *index*
void **setElementAt**(Object *element*, int *index*)	The location specified by *index* is assigned *element*



Vector Methods

void setSize(int *size*)	*Sets the number of elements in the vector to size. If the new size is less than the old size, elements are lost. If the new size is larger than the old, null elements are added*
int size()	Returns the number of elements currently in the vector

Q. 10 Access Specifiers

In Java, **Access Specifiers** define the visibility and accessibility of classes, methods, and variables within a program. They determine where a class or its members can be accessed from. Java provides four types of access specifiers:

1. public:

- **Visibility**: Accessible from **anywhere** in the program, across all classes and packages.
- **Usage**: It is commonly used for methods, constructors, and classes that need to be accessed by other parts of the program or external libraries.

Example:
java
Copy code
```
public class MyClass {

    public int data;

    public void display() {

        System.out.println("Public method");
    }
}
```

-

2. private:

- **Visibility**: Accessible only within the **same class**. It is not visible to subclasses or any other class, even if they belong to the same package.
- **Usage**: Commonly used to hide internal data or methods that should not be accessed directly from outside the class.

Example:
java
Copy code

```
class MyClass {

    private int data;

    private void display() {

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

    }

}
```

-

3. default (no specifier):

- **Visibility**: Also known as **package-private**, it is accessible only within the **same package**. It is the default access level if no specifier is mentioned.
- **Usage**: Used for methods or classes that should only be accessed by classes in the same package.

Example:
java
Copy code

```
class MyClass {  // default class, only accessible within the same package

    int data;    // default variable
    void display() {  // default method

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

    }

}
```

-

4. protected:

- **Visibility**: Accessible within the **same package** and also **accessible in subclasses** even if they are in a different package.
- **Usage**: Commonly used in inheritance scenarios, where you want to allow access to methods or variables by subclasses while keeping them hidden from non-subclass classes.

Example:
java
Copy code
```
class MyClass {

    protected int data;

    protected void display() {

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

    }

}
```

-

Summary Table:

Access Specifier	Same Class	Same Package	Subclass (Different Package)	Other Packages
public	Yes	Yes	Yes	Yes

private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes (if subclass)	No

These access specifiers allow Java developers to control the scope and visibility of data members and methods in classes, making the code more secure and modular.

The **`toString()`** **method** in Java is used to represent an object as a string. It is a method from the `Object` class, which is the superclass of all Java classes. By default, the `toString()` method returns a string that contains the **class name** followed by the **"@" symbol** and the object's **hashcode**. However, it is often overridden to provide a more meaningful string representation of an object.

Key Points:

- **Purpose**: To return a string representation of an object.
- **Default Behavior**: Without overriding, it returns a string that includes the class name and the hashcode.
- **Overriding**: By overriding the `toString()` method, you can return custom values (usually the object's data) instead of the default representation.

Example of Default `toString()`:

java

Copy code

```
class MyClass {

    int x = 10;

}
```

```
public class Main {

    public static void main(String[] args) {

        MyClass obj = new MyClass();

        System.out.println(obj.toString());  // Output: 
MyClass@1b6d3586 (class name + hashcode)

    }

}
```

Overriding the `toString()` Method:

To make the string representation more meaningful, you can override the `toString()` method in your class.

java

Copy code

```
class MyClass {

    int x = 10;

    int y = 20;


    // Overriding toString() method

    @Override

    public String toString() {

        return "MyClass [x=" + x + ", y=" + y + "]";

    }

}
```

```
public class Main {

    public static void main(String[] args) {

        MyClass obj = new MyClass();

        System.out.println(obj.toString());  // Output: MyClass [x=10,
y=20]

    }

}
```

Q. 11 ToString Method

Benefits of Overriding `toString()`:

- **Improves readability**: Makes it easier to display an object's state.
- **Less boilerplate code**: You don't need to manually write the object details every time you print the object.
- **Useful for debugging**: Provides quick access to an object's data when printing it.

Example in Real Use Case:

In most Java applications, overriding `toString()` is particularly useful when working with logging frameworks or when printing objects in a user-friendly format.

java

Copy code

```
class Employee {

    String name;

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

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

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("John", 30);
        System.out.println(emp);  // Output: Employee{name='John', 
age=30}
    }
}
```

In this example, `toString()` provides a clear representation of the `Employee` object, making it much easier to interpret than the default string format.

Q. Java Wrapper Classes

Wrapper classes in Java provide a way to use primitive data types as objects. Each primitive type in Java has a corresponding wrapper class that encapsulates the primitive value and provides useful methods for manipulating the data.

Key Points:

- **Purpose**: To allow primitive data types to be treated as objects, making them usable in scenarios where objects are required, such as in generic collections (`ArrayList`, `HashMap`).
- **Immutability**: Like `String`, wrapper objects are immutable, meaning their values cannot be changed once they are created.
- **Common Uses**: Converting between primitive types and strings, working with collections that only accept objects, and using methods like `parseInt()` and `valueOf()`.

Primitive Types and Their Corresponding Wrapper Classes:

Primitive Type	**Wrapper Class**
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`

`boolean`	`Boolean`

• Converts primitive to wrapper

o double a = 4.3;

o Double wrp = new Double(a);

• Each wrapper provides a method to return the primitive value.

o double r = wrp.doubleValue();

Autoboxing and Unboxing

• Autoboxing: Automatic conversion of primitive types to the object of

their corresponding wrapper classes is known as autoboxing

• Unboxing: It Automatically converting an object of a wrapper class to

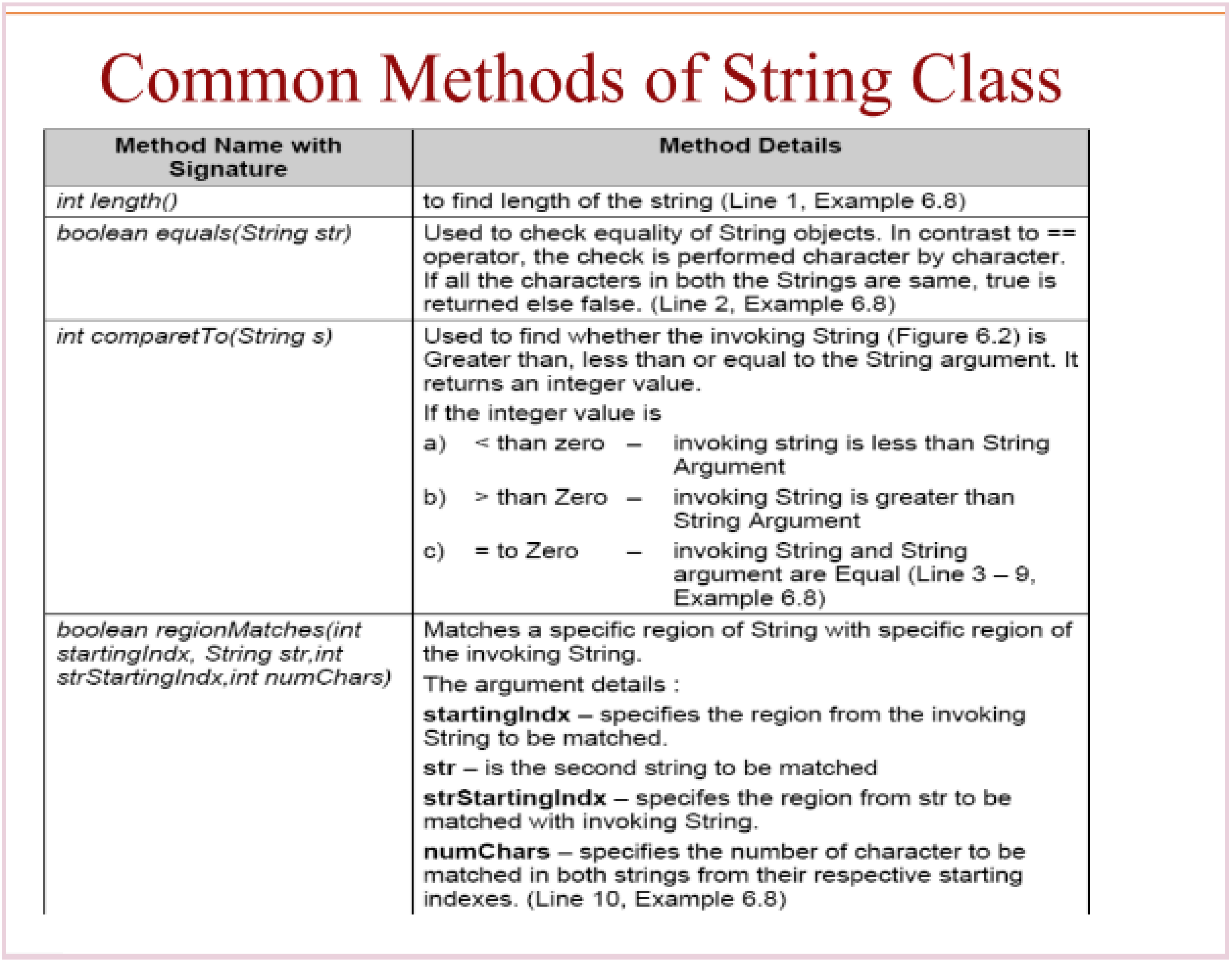
its corresponding primitive type is known as unboxing

Q. String Class

The String class in Java represents a sequence of characters and provides various methods to work with string data. Strings in Java are immutable, meaning their content cannot be changed once they are created.

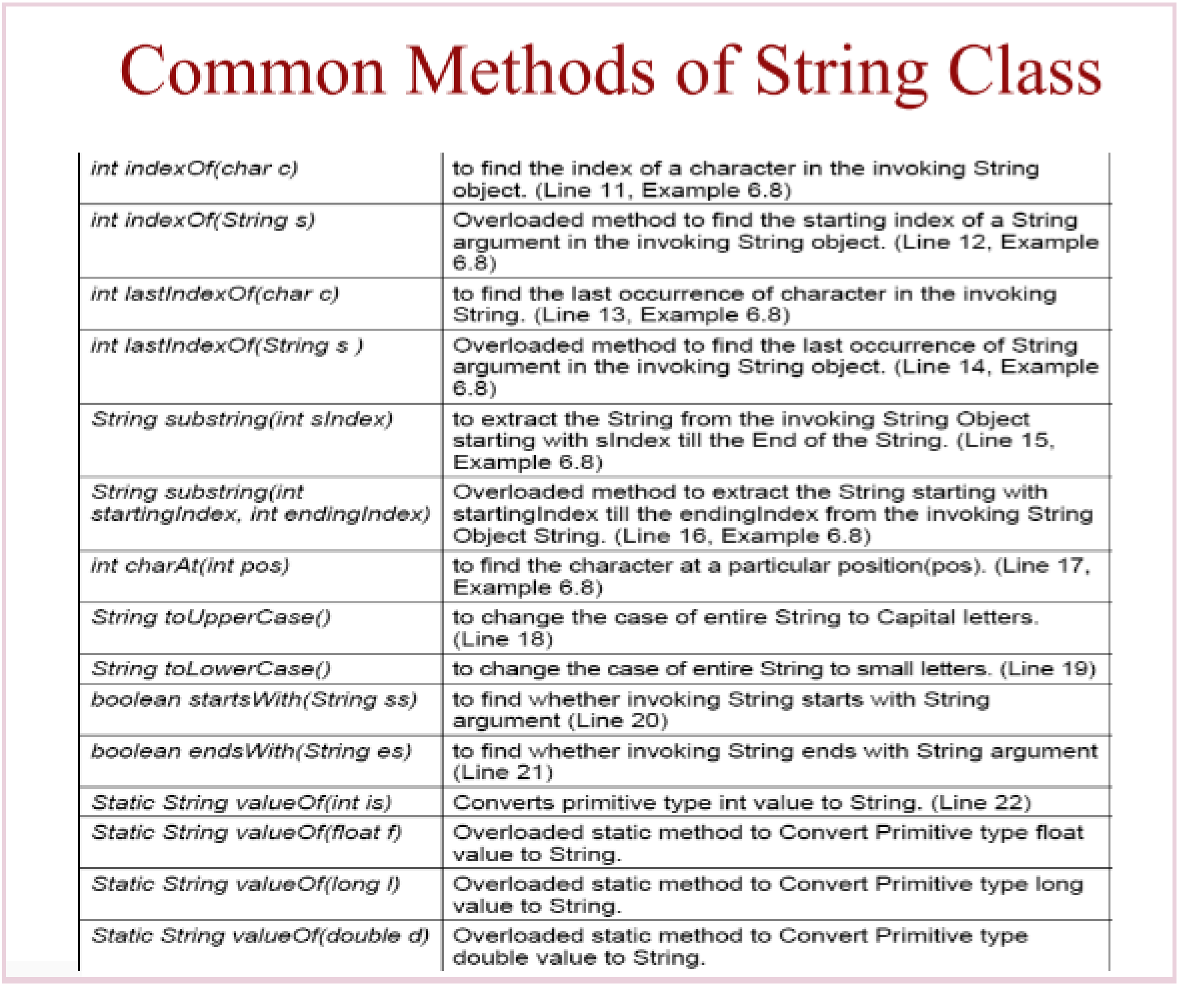
Key Points:

- Immutable: Once a String object is created, it cannot be modified. Any operation that appears to modify a string will actually create a new string object.
- String Pool: Java uses a special memory area called the string pool to store string literals. If a new string literal is created and it already exists in the pool, Java will return a reference to the existing string to save memory.



Common Methods of String Class

Method Name with Signature	Method Details
int length()	to find length of the string (Line 1, Example 6.8)
boolean equals(String str)	Used to check equality of String objects. In contrast to == operator, the check is performed character by character. If all the characters in both the Strings are same, true is returned else false. (Line 2, Example 6.8)
int comparetTo(String s)	Used to find whether the invoking String (Figure 6.2) is Greater than, less than or equal to the String argument. It returns an integer value. If the integer value is a) < than zero – invoking string is less than String Argument b) > than Zero – invoking String is greater than String Argument c) = to Zero – invoking String and String argument are Equal (Line 3 – 9, Example 6.8)
boolean regionMatches(int startingIndx, String str,int strStartingIndx,int numChars)	Matches a specific region of String with specific region of the invoking String. The argument details : **startingIndx** – specifies the region from the invoking String to be matched. **str** – is the second string to be matched **strStartingIndx** – specifes the region from str to be matched with invoking String. **numChars** – specifies the number of character to be matched in both strings from their respective starting indexes. (Line 10, Example 6.8)



Common Methods of String Class

int indexOf(char c)	to find the index of a character in the invoking String object. (Line 11, Example 6.8)
int indexOf(String s)	Overloaded method to find the starting index of a String argument in the invoking String object. (Line 12, Example 6.8)
int lastIndexOf(char c)	to find the last occurrence of character in the invoking String. (Line 13, Example 6.8)
int lastIndexOf(String s)	Overloaded method to find the last occurrence of String argument in the invoking String object. (Line 14, Example 6.8)
String substring(int sIndex)	to extract the String from the invoking String Object starting with sIndex till the End of the String. (Line 15, Example 6.8)
String substring(int startingIndex, int endingIndex)	Overloaded method to extract the String starting with startingIndex till the endingIndex from the invoking String Object String. (Line 16, Example 6.8)
int charAt(int pos)	to find the character at a particular position(pos). (Line 17, Example 6.8)
String toUpperCase()	to change the case of entire String to Capital letters. (Line 18)
String toLowerCase()	to change the case of entire String to small letters. (Line 19)
boolean startsWith(String ss)	to find whether invoking String starts with String argument (Line 20)
boolean endsWith(String es)	to find whether invoking String ends with String argument (Line 21)
Static String valueOf(int is)	Converts primitive type int value to String. (Line 22)
Static String valueOf(float f)	Overloaded static method to Convert Primitive type float value to String.
Static String valueOf(long l)	Overloaded static method to Convert Primitive type long value to String.
Static String valueOf(double d)	Overloaded static method to Convert Primitive type double value to String.

Q. StringBuffer Class in Java

The StringBuffer class in Java is used to create mutable strings, meaning strings that can be modified after they are created. Unlike the String class, which creates a new object each time its value is changed, **StringBuffer allows direct modification of the string's contents, making it more efficient for cases where frequent string manipulation is needed.**

Key Points:

- Mutable: Strings created using StringBuffer can be modified without creating a new object.
- **Thread-Safe**: `StringBuffer` is synchronized, meaning it is thread-safe and can be used in multi-threaded environments where multiple threads might modify the same string.
- Performance: StringBuffer offers better performance in scenarios where strings are being frequently changed, such as in loops or complex concatenation operations.

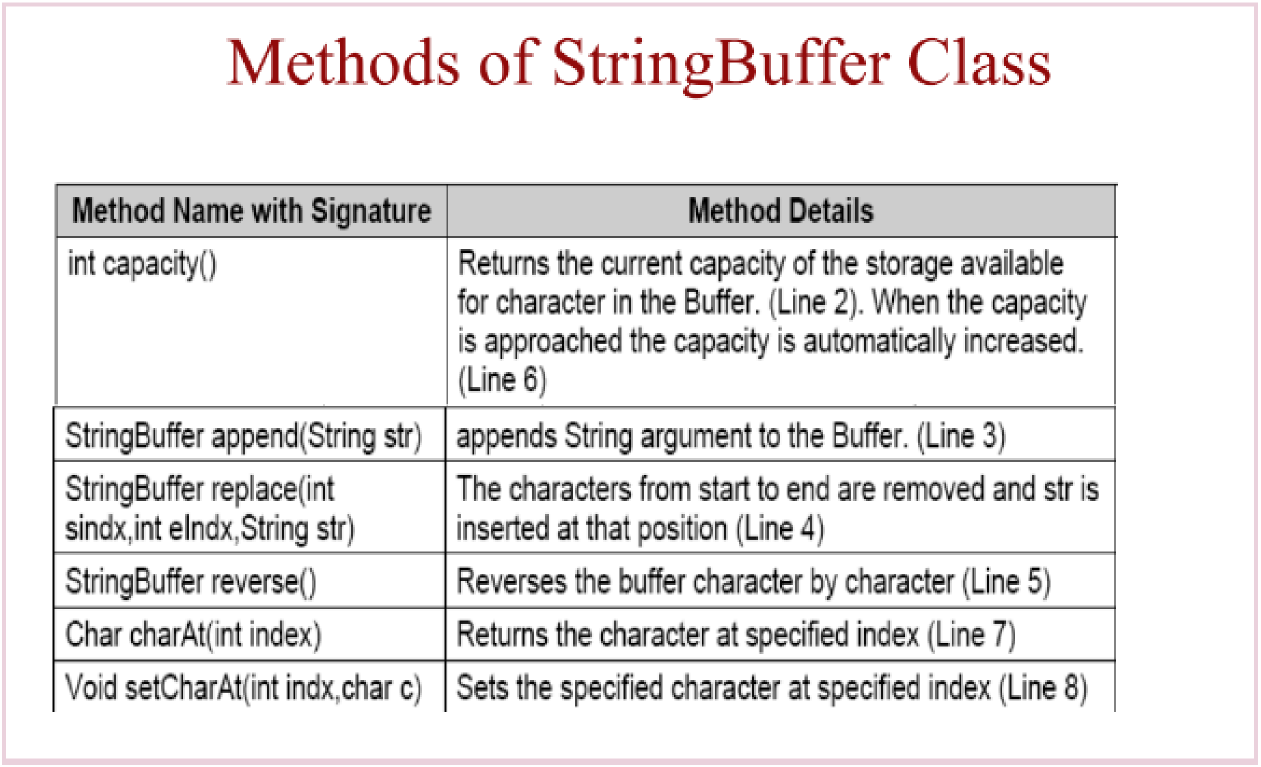
- Capacity: A StringBuffer has an initial capacity that can grow automatically when needed. The capacity() method returns the current capacity of the buffer.

Example of Creating a StringBuffer:

java

Copy code

```
public class StringBufferExample {

    public static void main(String[] args) {

        // Creating a StringBuffer with initial content

        StringBuffer sb = new StringBuffer("Hello");



        // Modifying the string (mutable)

        sb.append(" World");



        // Printing the modified string

        System.out.println(sb);  // Output: "Hello World"



        // Capacity of the StringBuffer

        System.out.println("Capacity: " + sb.capacity());  // Output: 21

    }

}
```



Methods of StringBuffer Class

Method Name with Signature	Method Details
int capacity()	Returns the current capacity of the storage available for character in the Buffer. (Line 2). When the capacity is approached the capacity is automatically increased. (Line 6)
StringBuffer append(String str)	appends String argument to the Buffer. (Line 3)
StringBuffer replace(int sindx,int eIndx,String str)	The characters from start to end are removed and str is inserted at that position (Line 4)
StringBuffer reverse()	Reverses the buffer character by character (Line 5)
Char charAt(int index)	Returns the character at specified index (Line 7)
Void setCharAt(int indx,char c)	Sets the specified character at specified index (Line 8)

In a multi-threaded environment, multiple threads can run concurrently, allowing different tasks to be performed simultaneously.

A single-threaded application or environment means that tasks are processed sequentially, one after another, without parallel execution.

Q. StringBuilder Class in Java

The StringBuilder class is similar to StringBuffer, but with **a key difference: it is not synchronized, making it faster for use in single-threaded environments**. Like StringBuffer, StringBuilder is used to create mutable strings, allowing for efficient modification of string content.

Key Points:

- Mutable: Like StringBuffer, StringBuilder allows strings to be modified in place without creating new objects.
- Faster: Since StringBuilder is not synchronized, it is faster than StringBuffer but should only be used when thread safety is not a concern.

- Methods: The StringBuilder class provides methods such as append(), insert(), delete(), replace(), and reverse()—similar to StringBuffer.
- Initial Capacity: Like StringBuffer, StringBuilder starts with an initial capacity of 16 characters, which expands dynamically as needed.

Example of Creating a StringBuilder:

```
public class StringBuilderExample {

   public static void main(String[] args) {

      // Creating a StringBuilder with initial content

      StringBuilder sb = new StringBuilder("Hello");



      // Modifying the string (mutable)

      sb.append(" World");



      // Printing the modified string

      System.out.println(sb);  // Output: "Hello World"

   }

}
```

Important Methods of StringBuilder:

append(String str): Adds a string to the end of the current string.
java
Copy code
```
sb.append(" World");  // Appends " World" to "Hello"
```

-

insert(int offset, String str): Inserts a string at the specified index.
java
Copy code
```
sb.insert(5, " Beautiful");  // Inserts " Beautiful" at index 5
```

- delete(int start, int end): Removes the characters from the specified start index to the end index.
  sb.delete(0, 5); // Deletes the first 5 characters

- replace(int start, int end, String str): Replaces the characters in the specified range with the provided string.
  sb.replace(6, 11, "Earth"); // Replaces "World" with "Earth"

- reverse(): Reverses the content of the StringBuilder.
  sb.reverse(); // Reverses the entire string buffer content

- capacity(): Returns the current capacity of the StringBuilder. The default capacity is 16 characters plus the length of the string.

  int cap = sb.capacity(); // Returns the current capacity of the buffer

- ensureCapacity(int minCapacity): Ensures that the capacity is at least equal to the specified minimum.

  sb.ensureCapacity(30); // Ensures the capacity is at least 30 characters

Key Differences Between `String`, `StringBuffer`, and `StringBuilder`:

- `String`: Immutable, once created, cannot be modified.
- `StringBuffer`: Mutable, **thread-safe (synchronized)**, slower due to synchronization.
- `StringBuilder`: Mutable, not thread-safe, faster in **single-threaded environments**

Q. What are ArrayLists?

Java ArrayList class uses a dynamic array for storing the elements(there is no size limit)

• We can add or remove elements anytime, much more flexible than the traditional array.

• It is found in the java.util package.

• The ArrayList in Java can have the duplicate elements also.

• It implements the List interface so we can use all the methods of List interface.

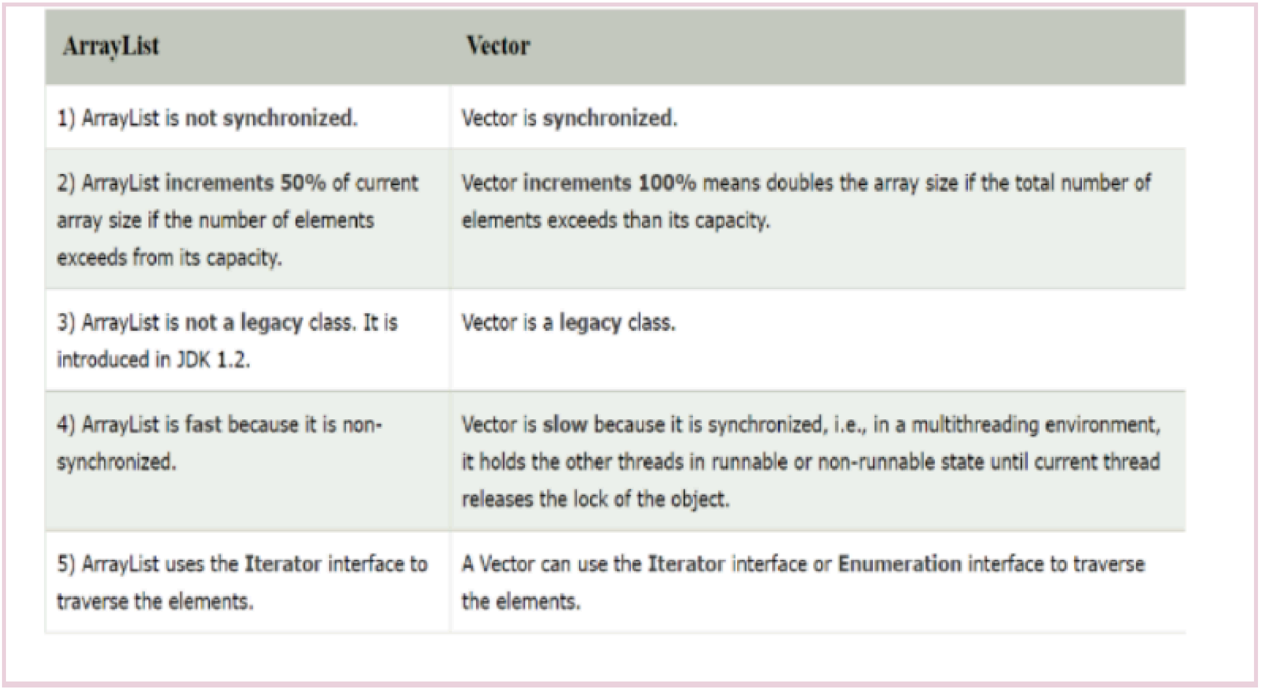
4. What are vectors in c++?

Vectors are a dynamic array of objects which can store heterogenous objects.

They are useful when you dont know the exact size of array in advance as they can grow and shrink as necessary.

We cannot store primitive data types in array they must be converted to objects, as vector can only store objects.

5. Differentiate between ArrayLists and vectors.



ArrayList	Vector
1) ArrayList is not synchronized.	Vector is synchronized.
2) ArrayList increments 50% of current array size if the number of elements exceeds from its capacity.	Vector increments 100% means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is not a legacy class. It is introduced in JDK 1.2.	Vector is a legacy class.
4) ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.
5) ArrayList uses the Iterator interface to traverse the elements.	A Vector can use the Iterator interface or Enumeration interface to traverse the elements.

6. What are wrapper classes?

A Wrapper class in Java is a class whose object wraps or contains primitive data types.

They are used to convert primitive data types to objects. It is useful because classes in the collection framework only allow objects as data type and not primitive data types, so Wrapper classes help us overcome this problem.

Implementation:

int k = 100; //The int data type k is converted into an object, it1 using Integer class.

Integer it1 = new Integer(k); **(Autoboxing /wrapping)**

int m = it1.intValue(); **(Unboxing/Unwrapping)**

System.out.println(m*m); **// prints 10000**

7. What is autoboxing and unboxing?

The process of automatically converting a primitive data type into the object of its corresponding wrapper class is called autoboxing.

The process of automatically converting a Wrapper Class Object into its corresponding primitive data type is called unboxing.

8. What is a String class?

String class provides many operations for string manipulations like constructors, utility, comparisons, conversions, etc.

String objects are read-only, i.e. immutable.

Declaration:

String stringName;

stringName = new String (“string value”);

9. What is a StringBuffer class?

All the operations of String class can be performed (Refer above).

Unlike the String class, StringBuffer class is mutable.

StringBuffer class is used in operations where the string has to be modified.

StringBuffer is synchronized i.e. thread safe.

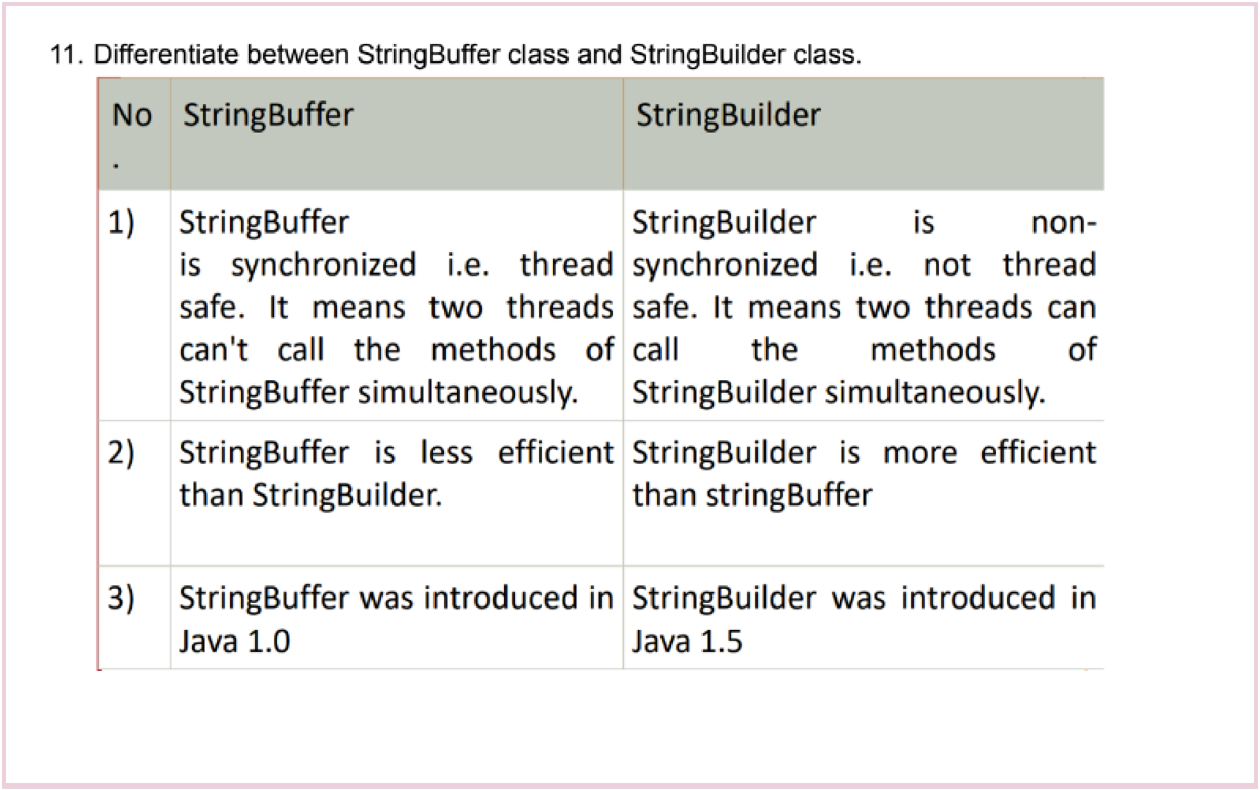
10. What is a StringBuilder class?

The StringBuilder in Java represents a mutable sequence of characters and provides an alternative to String Class, as it creates a mutable sequence of characters.

The function of StringBuilder is very much similar to the StringBuffer class, However the StringBuilder class differs from the StringBuffer class on the basis of synchronization.

It is non - synchronized.

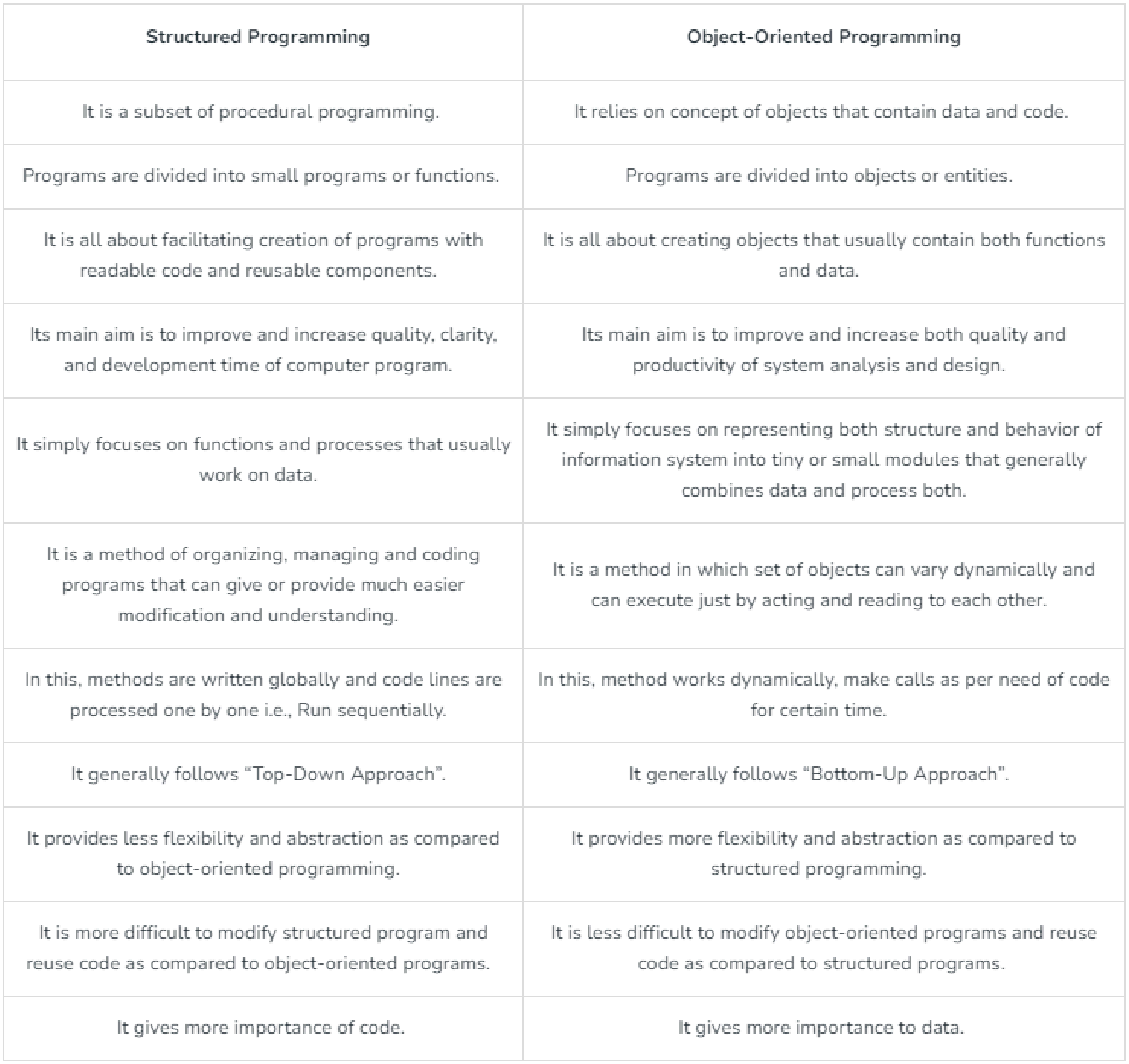
11. Differentiate between StringBuffer class and StringBuilder class.


11. Differentiate between StringBuffer class and StringBuilder class.

No.	StringBuffer	StringBuilder
1)	StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than stringBuffer
3)	StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

IMPORTANT QUESTIONS:

Q.1 Differentiate between Structured Programming and Object-Oriented Programming



Structured Programming	Object-Oriented Programming
It is a subset of procedural programming.	It relies on concept of objects that contain data and code.
Programs are divided into small programs or functions.	Programs are divided into objects or entities.
It is all about facilitating creation of programs with readable code and reusable components.	It is all about creating objects that usually contain both functions and data.
Its main aim is to improve and increase quality, clarity, and development time of computer program.	Its main aim is to improve and increase both quality and productivity of system analysis and design.
It simply focuses on functions and processes that usually work on data.	It simply focuses on representing both structure and behavior of information system into tiny or small modules that generally combines data and process both.
It is a method of organizing, managing and coding programs that can give or provide much easier modification and understanding.	It is a method in which set of objects can vary dynamically and can execute just by acting and reading to each other.
In this, methods are written globally and code lines are processed one by one i.e., Run sequentially.	In this, method works dynamically, make calls as per need of code for certain time.
It generally follows "Top-Down Approach".	It generally follows "Bottom-Up Approach".
It provides less flexibility and abstraction as compared to object-oriented programming.	It provides more flexibility and abstraction as compared to structured programming.
It is more difficult to modify structured program and reuse code as compared to object-oriented programs.	It is less difficult to modify object-oriented programs and reuse code as compared to structured programs.
It gives more importance of code.	It gives more importance to data.

Q.2 Method overloading & overriding

Method Overloading:

- Definition: Having multiple methods in the same class with the same name but different parameter lists (different type, number, or both).
- Compile-time Polymorphism: Resolved during compile-time.
- Purpose: Increases readability and flexibility of code.

Example:

```
void add(int a, int b) { }
void add(double a, double b) { }
```

Method Overriding:

- Definition: Redefining a method in the subclass that is already defined in the parent class.
- Run-time Polymorphism: Resolved at runtime.
- Purpose: Allows different implementations of a method in child classes.

Example:

```
class Parent {
   void show() { System.out.println("Parent class"); }
}
class Child extends Parent {
   void show() { System.out.println("Child class"); }

         }
```

Q.3 Explain static and dynamic runtime polymorphism

Static Polymorphism (Compile-time Polymorphism):

- Definition: Method overloading is an example of static polymorphism, where the method to be called is resolved at compile-time.

Example:

```
class StaticPolymorphismExample {
   void display(int a) {
      System.out.println(a);
```

```
    }

    void display(String a) {
        System.out.println(a);
    }
}
```

Dynamic Polymorphism (Runtime Polymorphism):

- Definition: Method overriding is an example of dynamic polymorphism, where the method to be called is resolved at runtime based on the object.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Cat();
        myAnimal.sound(); // Output: Cat meows
    }

            }
```

Q.4 Explain ‘instance of’ operator

Definition: The instanceof operator in Java is used to check whether an object is an instance of a specific class or interface. It returns a boolean value.

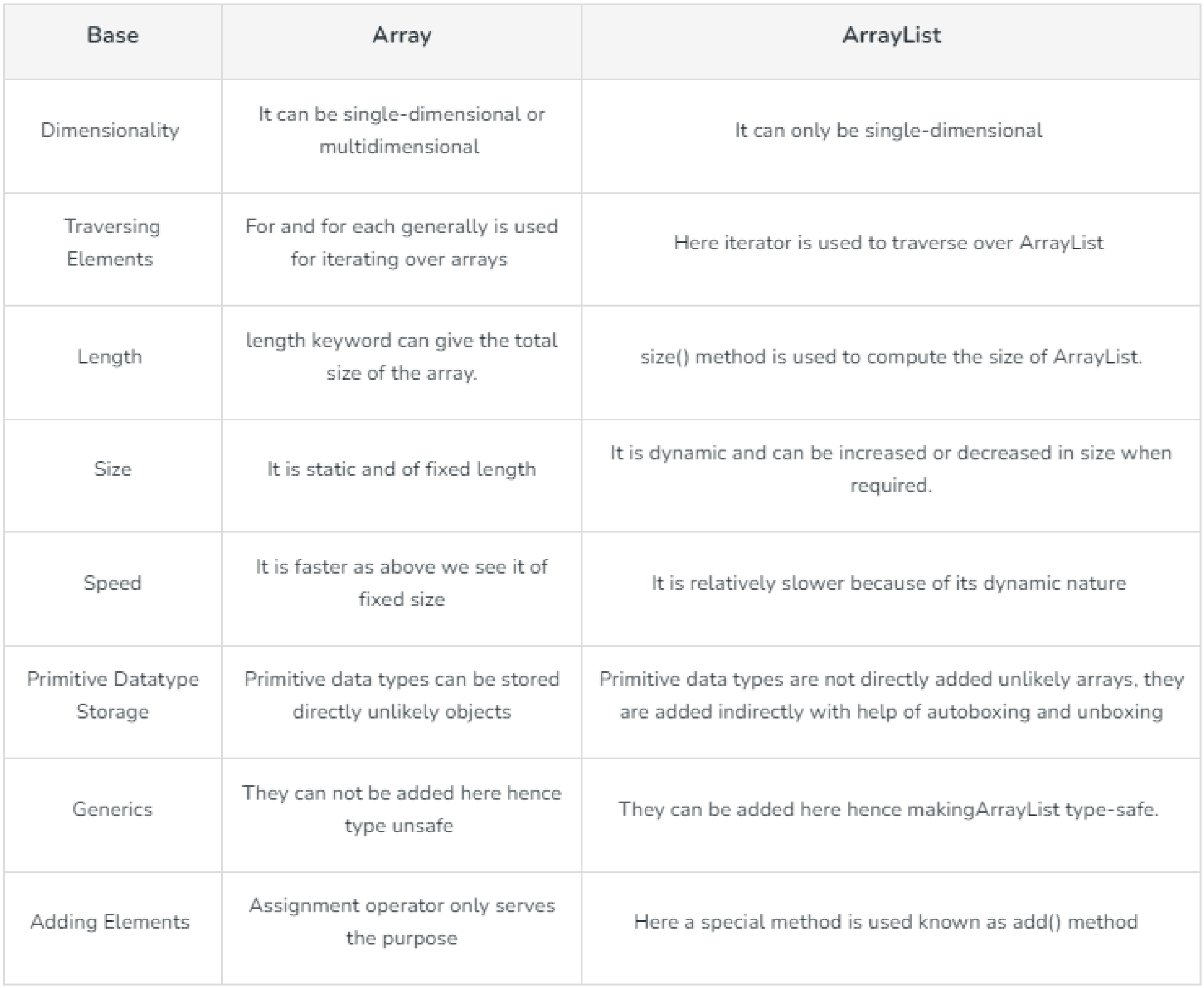
Example:

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

public class Test {
    public static void main(String[] args) {
        Dog d = new Dog();
        System.out.println(d instanceof Animal);  // true
    }
}
```

Use: Useful in runtime type checking to prevent ClassCastException when downcasting objects.

Q.5 Array and arraylist difference - 6 points

1. Size:
    - Array: Fixed size; cannot grow or shrink once created.
    - ArrayList: Dynamic size; can grow and shrink as needed.
2. Type of Data:
    - Array: Can store primitive types (e.g., `int`, `char`).
    - ArrayList: Only stores objects (wrapper classes for primitives).
3. Performance:
    - Array: Faster for fixed-size data due to direct memory allocation.
    - ArrayList: Slightly slower due to dynamic resizing overhead.
4. Memory:
    - Array: Uses less memory since it doesn't require overhead for resizing.
    - ArrayList: Requires extra memory for resizing and growth factor.
5. Traversal:
    - Array: Can be traversed using index-based loops.
    - ArrayList: Can be traversed using index-based loops or iterators.
6. Methods:
    - Array: No built-in methods like add, remove, etc.
    - ArrayList: Provides methods like `add()`, `remove()`, `get()`, etc.



Base	Array	ArrayList
Dimensionality	It can be single-dimensional or multidimensional	It can only be single-dimensional
Traversing Elements	For and for each generally is used for iterating over arrays	Here iterator is used to traverse over ArrayList
Length	length keyword can give the total size of the array.	size() method is used to compute the size of ArrayList.
Size	It is static and of fixed length	It is dynamic and can be increased or decreased in size when required.
Speed	It is faster as above we see it of fixed size	It is relatively slower because of its dynamic nature
Primitive Datatype Storage	Primitive data types can be stored directly unlikely objects	Primitive data types are not directly added unlikely arrays, they are added indirectly with help of autoboxing and unboxing
Generics	They can not be added here hence type unsafe	They can be added here hence makingArrayList type-safe.
Adding Elements	Assignment operator only serves the purpose	Here a special method is used known as add() method

Can read more here: https://www.geeksforgeeks.org/array-vs-arraylist-in-java/

Q.6 Explain 'this' keyword, 'final' keyword, 'super' keyword. State its significance and use

1. this Keyword:
    - Refers to the current object instance.
    - Significance: Used to eliminate ambiguity between instance variables and parameters, and to call another constructor in the same class.

Example:

```
class Example {
  int x;
```

```
    Example(int x) {
        this.x = x;  // 'this' refers to the instance variable x
    }
}
```

2. final Keyword:
   - Used to define constants, prevent method overriding, or prevent class inheritance.

Example:

```
final int MAX_VALUE = 100;  // constant
final class FinalClass {}  // class cannot be inherited
```

3. super Keyword:
   - Refers to the parent class object.
   - Significance: Used to call parent class methods or constructors.

Example:

```
class Parent {
    void display() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    void display() {
        super.display();  // calling Parent's display()
        System.out.println("Child method");
    }
}
```

Q.7 Types of constructor

- **Default Constructor: Provided by Java if no constructor is defined, initializes objects with default values.**

- Parameterized Constructor: Used to initialize objects with specific values provided as arguments.

Q.8 Constructor overloading

Definition: Defining multiple constructors in the same class with different parameters.
Example:
java
Copy code

```
class Student {
    Student() {
        System.out.println("Default constructor");
    }

    Student(String name) {
        System.out.println("Parameterized constructor with name: "
+ name);
    }
}
```

Q.9 Difference for & for-each

Here are the key differences between the `for` loop and `for-each` loop in Java:

1. Syntax

`for` loop:
Traditional loop with initialization, condition, and increment/decrement.

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

`for-each` loop:
Simpler loop used for traversing arrays or collections.

```
for (int element : array) {
    System.out.println(element);
}
```

2. Control over Index

- `for` loop:
  Gives you complete control over the index, allowing manipulation of the loop counter.
- `for-each` loop:
  You don't have access to the index; it iterates through all elements one by one without an index variable.

3. Flexibility

- `for` loop:
  Flexible for various conditions and allows you to break, continue, or use complex logic during iteration.
- `for-each` loop:
  Mainly used for simple traversal; you cannot modify the iteration process like skipping elements.

4. Modification of Collection

- `for` loop:
  Allows modification of the collection, including adding/removing elements while iterating (with caution).
- `for-each` loop:
  Modifying the collection while iterating may lead to issues like `ConcurrentModificationException`.

5. Use Cases

- `for` loop:
  Used when you need precise control over the loop (like iterating only through specific indices or reverse iteration).
- `for-each` loop:
  Used when you simply want to iterate over all elements of a collection or array in sequence.

6. Readability

- `for` loop:
  Can be more verbose, especially for simple traversals.
- `for-each` loop:
  More concise and readable for simple iteration tasks, making code cleaner.

7. Applicable To

- `for` loop:
  Can be used with arrays, collections, and even custom iteration logic.
- `for-each` loop:
  Limited to arrays and objects that implement the `Iterable` interface (e.g., `ArrayList`, `Set`).

Example

```java
// Using for loop
int[] array = {1, 2, 3, 4};
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}

// Using for-each loop
for (int element : array) {
    System.out.println(element);
}
```

The `for-each` loop is generally preferred when you want simplicity and readability, while the `for` loop is more suitable for scenarios where control over the loop's flow is needed.