WEEK 3 MODULE: OBJECT ORIENTED PROGRAMMING

TOPICS

- Introduction to OOP
- Pillars of OOP
- SOLID principles
- Access Modifiers
- This and Super keyword
- Final and Static keyword
- Interface and Abstract Classes
- String Class

LEARNING OUTCOMES

- By the end of this module, students will be able to:
- Understand Basic OOP concepts
- Understand SOLID principles
- Differentiate use cases for interfaces and Abstract Classes
- Understand how to create classes, interfaces, abstract classes where they are to be used respectively.
- Understand how String, String Buffer and String Builder classes work and their differences.

DAY 1:

INTRODUCTION TO OOP

Object-oriented programming (OOP) in Java is a programming methodology or paradigm (model) to design a computer program using classes and objects. It is the most popular programming paradigm and widely used in the software industry today. It is an extension of procedural programming.

Procedural programming means writing code without objects. Whereas, object-oriented programming means writing code with objects. It contains data in the form of fields or variables (often known as attributes or properties in java) and functionality code in the form of methods (often known as behavior in java).

Java OOP concept uses variables and methods as procedural programs do, but it focuses on the objects that contain variables and methods. That's why this new approach is called object-oriented approach in which we write programs using classes and objects.

It has four pillars which are encapsulation, abstraction, inheritance, and polymorphism. When we write a program using these concepts, it is called Object-Oriented Programming System (OOPs). The main goal of the OOPs concept in java programming is that everything you want to do, do through objects.

FEATURES OF OOP IN JAVA

Some of the striking features of object-oriented programming system (OOPs model/design) are as follows:

- Higher priority is focused on Data rather than functions.
- Programs are divided into the number of entities known as objects.
- Objects communicate with each other through functions (methods).
- Methods that operate on Data of an object are closely bound together in the data structure.
- An object is a group of data and methods.
- Data is hidden in the OOP and cannot be accessed by external methods. Hence, it is safe from accidental modification.
- Data cannot move freely out of the object.
- New data and methods can be easily added whenever needs.
- A bottom-up approach is adopted in programming design.

Advantage/Benefit of OOPs Concepts in Java

OOPs concept in Java offers several advantages that are not available in procedural programming like C, Pascal, etc. Some of the major benefits of object-oriented programming in java are as follows:

- Security: In OOP, Data is encapsulated with methods in the class so that data is protected and secured from accidental modification by other external non-member methods.
- Reusability: Through inheritance, we can use the features of an existing class in a new class without repeating existing code that saves a lot of time for developers, and also increases productivity.
- Effective communication: In OOP, objects can communicate via message passing technique that makes interface descriptions with outside systems much simpler.
- Developing complex software: OOPs is the most suitable approach for developing complex software because it minimizes the complexity through the feature of inheritance.
- Easily upgraded: Object-oriented system can be easily upgraded from small to large systems because OOP uses bottom-up approach.
- Easy partition of work: It is easy to partition complicated work in a project based on objects.
- Maintenance: The maintenance of object-oriented code is easier.
- Efficiency: The concepts of OOP provide better efficiency and an easy development process.

PILLARS OF OOP

The pillars of Object-Oriented Programming (OOP) are a set of fundamental principles that guide the design and implementation of object-oriented systems. These principles contribute to the core concepts of OOP and help developers create code that is modular, reusable, and easier to maintain. The four pillars of OOP are:

ENCAPSULATION:

Encapsulation is the principle of bundling data (attributes or properties) and the methods (behavior) that operate on that data into a single unit called a class. It promotes data hiding and abstraction by controlling the access to the internal state of an object. Encapsulation helps in achieving data security, as the internal details of an object are hidden from external entities. It also allows for code modularity and flexibility, as changes made to the internal implementation of a class do not affect other parts of the program that use the class.

Importance: Encapsulation helps in bundling data and methods together within a class, promoting data hiding and abstraction. It ensures that the internal state of an object is protected and can be accessed and modified only through defined methods.

Benefits:

- Data security: Encapsulation protects the data from external modification, ensuring its integrity and consistency.
- Code modularity: By encapsulating related data and methods into classes, code becomes modular, making it easier to understand, maintain, and reuse.
- Flexibility: Encapsulation allows for changes in the internal implementation of a class without affecting other parts of the program that use the class.
- Information hiding: Encapsulation hides the internal details of an object, providing a clean and simplified interface for interacting with the object.

How to achieve or implement Encapsulation in Java

Declare instance variables as private: Encapsulation involves hiding the internal state of an object from external access. In Java, you can achieve this by declaring the instance variables of a class as private. Private variables can only be accessed within the same class.

```
public class Person {
   private String name;
   private int age;
   // Getter and setter methods
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
}
```

Provide public setter and getter methods: To control access to the private variables, you can provide public setter and getter methods. These methods allow other classes to interact with the private variables indirectly.

```
public class Person {
   private String name;
   private int age;
   // Getter and setter methods
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }
}
```

By encapsulating the variables and providing controlled access through setter and getter methods, you ensure that the internal state of an object is protected and can be modified or accessed only through defined methods.

INHERITANCE:

Inheritance is the principle that allows classes to inherit properties and behavior from other classes. It establishes a hierarchical relationship between classes, where a subclass (derived class) can inherit the characteristics of a superclass (base class). Inheritance promotes code reuse by allowing the reuse of existing code and extending it to add or modify functionality. It facilitates the creation of specialized classes that inherit common characteristics from a more general class, promoting code extensibility and reducing code duplication.

Importance: Inheritance establishes a hierarchical relationship between classes, allowing the subclass to inherit properties and behavior from the superclass. It promotes code reuse and modularity.

Benefits:

- Code reuse: Inheritance allows the subclass to inherit and reuse the attributes and methods of the superclass, reducing code duplication.
- Extensibility: Subclasses can extend the functionality of the superclass by adding new methods or overriding existing ones.
- Polymorphism: Inheritance facilitates polymorphism, allowing objects of different subclasses to be treated as objects of a common superclass, promoting code flexibility and extensibility.

Conceptual modeling: Inheritance helps in modeling real-world relationships and hierarchies, making the code more intuitive and easier to understand.

How is Inheritance implemented/achieved in Java?

Inheritance in Java can be implemented or achieved by using two keywords:

1. **extends:** extends is a keyword that is used for developing the inheritance between two classes and two interfaces. Note that a class always extends another class. An interface always extends another interface and can extend more than one interface.

2. **implements:** implements keyword is used for developing the inheritance between a class and interface. A class always implements the interface.

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

Polymorphism:

Polymorphism is the principle that allows objects of different types to be treated as objects of a common superclass. It enables the use of a single interface to represent

multiple types of objects. Polymorphism allows methods to be implemented in different ways in different classes, providing flexibility and code extensibility. It allows for method overriding, where a subclass can provide its own implementation of a method defined in the superclass. Polymorphism simplifies code maintenance and promotes code flexibility by enabling objects to be used interchangeably based on their common behavior.

Importance: Polymorphism allows objects of different types to be treated interchangeably based on their common behavior, providing flexibility and code extensibility.

Benefits:

- Code flexibility: Polymorphism enables the use of a single interface to represent multiple types of objects, promoting code flexibility and adaptability to different scenarios.
- Code extensibility: Polymorphism allows for method overriding, where subclasses can provide their own implementation of methods defined in the superclass, allowing for customization and specialization.
- Simplified code maintenance: Polymorphism simplifies code maintenance by allowing the addition of new subclasses without affecting existing code that uses the superclass interface.
- Loose coupling: Polymorphism promotes loose coupling between objects, as code can interact with objects based on their common interface rather than specific types.

To achieve or implement polymorphism in Java, you can use the following techniques:

Method Overriding: Polymorphism can be achieved through method overriding, where a subclass provides its own implementation of a method that is already defined in its superclass. The appropriate implementation is determined at runtime based on the actual type of the object.

```
public class Animal {
   public void makeSound() {
      System.out.println("Animal is making a sound.");
   }
}
public class Dog extends Animal {
   @Override
   public void makeSound() {
      System.out.println("Dog is barking.");
   }
}

public class Cat extends Animal {
   @Override
   public void makeSound() {
      System.out.println("Cat is meowing.");
   }
}

Animal dog = new Dog();
Animal cat = new Cat();
```

```
dog.makeSound(); // Output: Dog is barking.
cat.makeSound(); // Output: Cat is meowing.
```

Method Overloading: Polymorphism can also be achieved through method overloading, where multiple methods with the same name but different parameters are defined in a class. The appropriate method is selected at compile-time based on the number and types of arguments provided.

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

   public double add(double a, double b) {
      return a + b;
   }
}
Calculator calculator = new Calculator();
int result1 = calculator.add(5, 10); // Output: 15
double result2 = calculator.add(2.5, 3.5); // Output: 6.0
```

By using method overriding and method overloading, you can achieve polymorphism in Java, allowing objects to be used interchangeably based on their common behavior.

Abstraction:

Abstraction is the principle of simplifying complex systems by representing only the essential features and hiding unnecessary details. It focuses on defining the essential characteristics and behavior of an object, while hiding the implementation details. Abstraction allows developers to create abstract classes and interfaces, which provide a clear and simplified view of the system. It helps in managing complexity, improving code readability, and facilitating code maintenance. Abstraction allows for the creation of modular and flexible code by defining contracts and interactions between objects without exposing the internal implementation.

Importance: Abstraction simplifies complex systems by focusing on the essential features and hiding unnecessary details. It provides a clear and simplified view of the system.

Benefits:

- Managing complexity: Abstraction helps in managing complexity by breaking down a system into abstract classes and interfaces, focusing on high-level concepts rather than implementation details.
- Code readability: By hiding implementation details, abstraction improves code readability and understandability, making it easier for developers to work with the code.
- Modularity and maintainability: Abstraction allows for the creation of modular code, where changes made to the implementation of an abstract class or interface do not affect other parts of the program.

Contract-based programming: Abstraction defines contracts and interactions between objects, facilitating communication and collaboration between different components of a system.

To achieve or implement abstraction in Java, you can follow these steps:

Use abstract classes: An abstract class is a class that cannot be instantiated and is meant to be subclassed. It can have both abstract and non-abstract methods. Abstract methods are declared without an implementation and must be implemented in the subclasses. Abstract classes provide a way to define common methods and attributes that can be shared among multiple subclasses.

```
public abstract class Shape {
   public abstract void draw();
   // other common methods and attributes
}
public class Circle extends Shape {
   @Override
   public void draw() {
      // implementation specific to drawing a circle
   }
}

public class Square extends Shape {
   @Override
   public void draw() {
      // implementation specific to drawing a square
   }
}
```

Use interfaces: An interface is a collection of abstract methods that define a contract for a class to implement. It provides a way to achieve multiple inheritance in Java. Any class that implements an interface must provide an implementation for all the methods declared in the interface.

```
public interface Drawable {
   void draw();
   // other methods
}

public class Circle implements Drawable {
   @Override
   public void draw() {
      // implementation specific to drawing a circle
   }
}
public class Square implements Drawable {
   @Override
   public void draw() {
      // implementation specific to drawing a square
   }
}
```

DAY 2:

SOLID PRINCIPLES

The SOLID principles are a set of design principles that aim to guide developers in creating software systems that are robust, flexible, and maintainable. Each principle focuses on a specific aspect of software design and encourages practices that promote modularity, extensibility, and reusability. Let's explore each principle in detail:

Single Responsibility Principle (SRP): This principle states that a class or module should have only one reason to change. It means that a class should have a single responsibility or job, and it should be focused on performing that responsibility. By adhering to SRP, we ensure that a class is more cohesive, easier to understand, and less prone to changes when requirements change. It promotes code organization and reduces the risk of introducing bugs during maintenance.

- Implementing SRP in Java involves creating classes that have a single responsibility and are focused on performing that specific task.
- Each class should encapsulate related data and behavior, providing a clear and cohesive interface to interact with.
- By separating concerns into different classes, changes in one area of the system are less likely to impact other areas.

Open-Closed Principle (OCP): The OCP states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. It means that we should design our code in a way that allows us to add new functionality without modifying existing code. By using abstractions, interfaces, and inheritance, we can achieve this principle. The benefit is that we can introduce new features or behavior without affecting the existing codebase, making the system more maintainable and reducing the risk of introducing bugs.

- In Java, the OCP can be achieved through the use of abstraction, inheritance, and interfaces.
- Instead of modifying existing code, we can extend or override functionality by creating new classes that implement or extend existing interfaces or classes.
- This allows us to add new features or behaviors without modifying the existing codebase.
- By relying on abstraction, we can write code that depends on interfaces or abstract classes rather than concrete implementations, promoting flexibility and extensibility.
- Liskov Substitution Principle (LSP):

Liskov Substitution Principle (LSP): The LSP states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In other words, subclasses should be able to be used interchangeably with their base class. This principle ensures that inheritance relationships are properly designed, and it promotes code reuse and extensibility. By adhering to LSP, we ensure that our codebase remains robust and that changes in subclasses do not introduce unexpected behavior.

- To adhere to LSP in Java, subclassing should follow the "is-a" relationship and maintain the same contract as the superclass.
- Subclasses should be able to be used interchangeably with their superclass without altering the behavior of the program.
- In practice, this means that subclasses should override methods from the superclass without weakening the preconditions, strengthening the postconditions, or violating the invariants.
- Properly designed inheritance hierarchies in Java ensure that LSP is maintained and that polymorphism can be effectively utilized.

Interface Segregation Principle (ISP): The ISP states that clients should not be forced to depend on interfaces they do not use. It means that we should design interfaces that are specific to the needs of clients to avoid unnecessary dependencies. By segregating interfaces into smaller, more focused ones, we promote loose coupling and allow clients to depend only on the functionality they require. This principle leads to more maintainable and flexible systems, as changes in one part of the system do not affect unrelated parts.

- In Java, ISP can be implemented by defining fine-grained interfaces that are specific to the needs of clients.
- Instead of creating monolithic interfaces, we can break them down into smaller, focused interfaces.
- Clients then depend only on the interfaces they require, reducing unnecessary dependencies.
- This allows for greater flexibility, as changes in one interface do not affect unrelated parts of the system.

Dependency Inversion Principle (DIP): The DIP states that high-level modules should not depend on low-level modules. Both should depend on abstractions. It also states that abstractions should not depend on details; details should depend on abstractions. This principle promotes decoupling and dependency inversion through the use of interfaces or abstract classes. By adhering to DIP, we achieve loose coupling, dependency injection, and easier unit testing. It allows for flexible and interchangeable components, making the system more adaptable to changes.

- DIP can be achieved in Java through the use of dependency injection, inversion of control containers, and interfaces.
- High-level modules depend on abstractions (interfaces or abstract classes) rather than concrete implementations.
- Concrete implementations are injected into the high-level modules through constructors, setters, or method parameters.
- This allows for loose coupling and the ability to easily swap implementations, making the code more modular, testable, and maintainable.

By following the SOLID principles, developers can design software systems that are more modular, maintainable, and flexible. These principles provide guidelines for writing clean, reusable, and extensible code, reducing the impact of changes and making it easier to introduce new features. They contribute to the overall quality and longevity of the software system.

ACCESS MODIFIERS

Access modifiers in Java allow us to control the visibility and accessibility of class members, such as variables, methods, and constructors. There are four access modifiers in Java:

- **Public:** The public access modifier allows unrestricted access to a class member from any other class or package. A public member is visible to all classes and can be accessed globally.
- **Private:** The private access modifier restricts access to a class member within the same class. Private members are not accessible from outside the class, including subclasses and other packages.
- **Protected:** The protected access modifier allows access to a class member within the same class, subclasses, and other classes in the same package. Protected members are not accessible from classes in different packages unless there is a subclass relationship.
- **Default (No Modifier):** When no access modifier is specified, the default access modifier is applied. A default member is accessible within the same package but not accessible from classes in different packages.

The usage of access modifiers provides several benefits in Java:

- Encapsulation: Access modifiers allow us to enforce encapsulation by controlling the visibility of class members. By making certain members private, we ensure that they are only accessed and modified within the class itself.
- Information Hiding: By making members private or using the default access modifier, we can hide implementation details from other classes. This helps in maintaining the integrity of the class and prevents direct manipulation of sensitive data.
- Modularity: Access modifiers allow us to define clear boundaries between different parts of a program. By using the appropriate access modifier, we can specify which members should be accessible to other classes and which should be kept private.
- Security: By using access modifiers, we can control the accessibility of sensitive information or critical operations. Private members ensure that they can only be accessed by trusted code, enhancing the security of the system.
- Code Reusability: Access modifiers facilitate code reuse by allowing us to expose certain members through public or protected access. This allows other classes to utilize those members without having to rewrite the functionality.

In summary, access modifiers in Java play a crucial role in controlling the visibility and accessibility of class members. They help in achieving encapsulation, information hiding, modularity, security, and code reusability in Java programs.

DAY 3:

“This” AND “Super” KEYWORD

"this" keyword:

Usage: The "this" keyword is a reference to the current instance of a class. It is used to access or refer to the current object's members, such as variables and methods.

Accessing class members: When there is a naming conflict between a local variable and an instance variable, the "this" keyword is used to refer to the instance variable. It helps distinguish between the local variable and the instance variable with the same name.

Invoking constructors: The "this" keyword can be used to invoke another constructor within the same class. This is known as constructor chaining and allows constructors to call other constructors with different arguments.

Example: this.variableName or this.methodName()

"super" keyword:

Usage: The "super" keyword is a reference to the superclass or parent class of the current class. It is used to access or invoke the superclass's members, including variables, methods, and constructors.

Accessing class members: The "super" keyword is used to access the superclass's members when there is a method or variable with the same name in both the superclass and subclass. It helps differentiate between the superclass's member and the subclass's member.

Invoking constructors: The "super" keyword is used to invoke the superclass's constructor from the subclass. It is typically used in the subclass's constructor to initialize the inherited members defined in the superclass.

Handling method overriding: In an overridden method, the "super" keyword can be used to invoke the superclass's implementation of the method, even if it has been overridden in the subclass.

Example: super.variableName or super.methodName() or super(arguments)

Differences between "this" and "super" keywords:

- **"this"** is used to refer to the current instance of a class, while **"super"** is used to refer to the superclass of a class.
- "this" is used for accessing or invoking members of the current class, while "super" is used for accessing or invoking members of the superclass.
- "this" is used when there is a naming conflict between local variables and instance variables, while "super" is used when there is a naming conflict between subclass and superclass members.
- "this" is used to invoke constructors within the same class, while "super" is used to invoke the superclass's constructor from the subclass.
- "this" is not applicable in a static context (static methods or static variables), while "super" can be used in a static context to access static members of the superclass.

Overall, the "this" and "super" keywords play essential roles in Java, allowing for proper referencing and accessing of class members, handling constructors, and managing method overriding within a class hierarchy.

Final and Static keyword

"final" keyword:

Purpose: The "final" keyword is used to indicate that an entity (class, method, or variable) cannot be changed or overridden once it is defined.

Applied to classes: When applied to a class, the "final" keyword indicates that the class cannot be subclassed or extended. It ensures that the class's implementation and behavior remain unchanged.

Applied to methods: When applied to a method, the "final" keyword indicates that the method cannot be overridden in any subclass. It preserves the method's implementation from modification.

Applied to variables: When applied to a variable, the "final" keyword indicates that the variable's value cannot be changed once it is assigned. It creates a constant variable.

Example: final class ClassName, final void methodName(), final int variableName

"static" keyword:

Purpose: The "static" keyword is used to declare entities that belong to the class itself, rather than instances of the class. It allows accessing the entities without creating an instance of the class.

Applied to variables: When applied to a variable, the "static" keyword creates a single copy of the variable that is shared among all instances of the class. It is also known as a class variable or a static variable.

Applied to methods: When applied to a method, the "static" keyword creates a method that belongs to the class itself, rather than a specific instance of the class. It can be called using the class name without creating an object of the class.

Example: static int variableName, static void methodName()

Key points to remember:

- The "final" keyword ensures immutability and prevents modifications.
- The "static" keyword allows accessing variables and methods without creating instances of the class.
- "final" can be applied to classes, methods, and variables to indicate their unchangeable nature.
- "static" can be applied to variables and methods to indicate their association with the class itself, rather than instances.

The "final" and "static" keywords provide additional control and flexibility in Java programming. By using them appropriately, you can enforce immutability, prevent method overriding, create

class-level variables and methods, and optimize memory usage by sharing variables among instances.

DAY 4:

INTERFACE AND ABSTRACT CLASSES

Interfaces:

In Java, an interface is a reference type that defines a contract or a set of methods that a class implementing the interface must adhere to. It provides a way to achieve abstraction and specify behavior without specifying the implementation details. Here's an explanation of interfaces and their usage:

Role of Interfaces: Interfaces play a crucial role in defining contracts between different parts of a system. They establish a common set of methods that classes must implement, ensuring consistency and interoperability. Interfaces enable loose coupling between classes, allowing for flexibility, code reusability, and modular design.

Declaring Interfaces: In Java, an interface is declared using the interface keyword followed by the interface name. It consists of method declarations without implementations, constant variables, and default methods (Java 8 onwards). For example:

```
public interface MyInterface {
    void method1();
    int method2();
    // ...
}
```

Implementing Interfaces: To implement an interface, a class uses the implements keyword followed by the interface name. The class must provide implementations for all the methods declared in the interface. Multiple interfaces can be implemented by a single class, separated by commas. For example:

```
public class MyClass implements MyInterface {
    public void method1() {
        // Implementation for method1
    }
    public int method2() {
        // Implementation for method2
        return 0;
```

```
    }
}
```

Achieving Multiple Inheritance-like Behavior: Java doesn't support multiple inheritance of classes, but it allows implementing multiple interfaces. This enables classes to inherit behavior from multiple sources by implementing multiple interfaces. It allows for code reuse, promotes modularity, and provides flexibility in designing class hierarchies.

Abstract Classes:

Abstract classes in Java are classes that cannot be instantiated and serve as a blueprint or template for subclasses. They can contain both abstract and non-abstract methods, as well as instance variables. Here's an explanation of abstract classes and their usage:

Usage of Abstract Classes: Abstract classes are used to define common behavior and characteristics that are shared among multiple subclasses. They provide a way to establish a hierarchy and enforce a contract for subclasses to follow. Abstract classes can have method declarations without implementations (abstract methods), as well as concrete methods with implementations.

Declaring Abstract Classes: To declare an abstract class, the abstract keyword is used before the class declaration. It can have both abstract and non-abstract methods. Abstract classes may or may not have constructors. For example:

```
public abstract class MyBaseClass {
    public abstract void abstractMethod();
    public void concreteMethod() {
        // Implementation for concreteMethod
    }
}
```

Implementing Abstract Classes: Abstract classes are meant to be extended by concrete subclasses. Subclasses of an abstract class must provide implementations for all abstract methods inherited from the abstract class. Abstract methods are declared without implementations and are meant to be overridden by the subclasses.

Extending Abstract Classes: To create a concrete subclass of an abstract class, the extends keyword is used followed by the abstract class name. The subclass must provide implementations for all the abstract methods. For example:

```
public class MySubClass extends MyBaseClass {
    public void abstractMethod() {
        // Implementation for abstractMethod
    }
}
```

Key points to remember:

- Interfaces define contracts and specify behavior without implementation details.
- Classes implement interfaces using the implements keyword and provide implementations for all interface methods.
- Interfaces enable achieving multiple inheritance-like behavior in Java.
- Abstract classes cannot be instantiated and serve as blueprints for subclasses.
- Abstract classes can have both abstract and non-abstract methods.
- Subclasses of an abstract class must provide implementations for all abstract methods.

STRING CLASSES

In Java, the String, StringBuffer, and StringBuilder classes are used for working with textual data. Although they serve a similar purpose, there are differences in their behavior and usage. Here's an explanation of these string classes and their differences:

String Class:

- The String class represents an immutable sequence of characters. Once a String object is created, its value cannot be changed.
- String objects are stored in the string pool, which allows for efficient memory utilization and string sharing.
- String objects are thread-safe, meaning they can be safely shared among multiple threads.
- String concatenation using the + operator creates a new String object, which can be memory-intensive for frequent concatenations.

Example:

```
String str = "Hello";

str = str + " World";
```

StringBuffer Class:

- The StringBuffer class represents a mutable sequence of characters. It allows for dynamic modification of the character sequence.
- StringBuffer objects are synchronized, making them thread-safe for concurrent access.
- StringBuffer provides various methods for appending, inserting, and modifying the character sequence efficiently.
- StringBuffer is suitable when there is a need for frequent modifications to the string content.

Example:

```
StringBuffer sb = new StringBuffer("Hello");
```

sb.append(" World");

StringBuilder Class:

- The StringBuilder class is similar to StringBuffer and represents a mutable sequence of characters.
- StringBuilder objects are not synchronized, making them more efficient in single-threaded scenarios.
- StringBuilder provides methods for appending, inserting, and modifying the character sequence, similar to StringBuffer.
- StringBuilder is suitable when there is no need for thread safety and faster performance is desired.

Example:

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

Differences between StringBuffer and StringBuilder:

- Synchronization: StringBuffer is synchronized, meaning it is thread-safe for concurrent access. StringBuilder is not synchronized, providing better performance in single-threaded scenarios.
- Thread Safety: Due to synchronization, StringBuffer can be used in multi-threaded environments without causing data inconsistency. StringBuilder is not thread-safe and should be used in single-threaded scenarios or when synchronization is handled externally.
- Performance: StringBuilder generally performs better than StringBuffer in single-threaded scenarios due to its lack of synchronization overhead.

Key points to remember:

- String is immutable and thread-safe.
- StringBuffer is mutable, synchronized, and thread-safe.
- StringBuilder is mutable, not synchronized, and provides better performance in single-threaded scenarios.
- Choose StringBuffer for thread-safe string modifications.
- Choose StringBuilder for better performance in single-threaded scenarios.

EXTRA LEARNING MATERIALS

- https://www.javatpoint.com/java-oops-concepts
- https://www.youtube.com/watch?v=6T_HgnjoYwM
- https://www.youtube.com/watch?v=j0lBrYSlYaU
- https://www.guru99.com/interface-vs-abstract-class-java.html
- https://www.baeldung.com/java-method-overload-override
- https://javapapers.com/core-java/abstract-and-interface-core-java-2/difference-between-a-java-interface-and-a-java-abstract-class/

- https://medium.com/heuristics/interface-vs-abstract-class-vs-concrete-class-196f20c3af9a
- https://www.baeldung.com/java-method-overload-override
- https://www.javatpoint.com/access-modifiers
- https://www.javatpoint.com/this-keyword
- https://www.javatpoint.com/final-keyword
- https://www.javatpoint.com/java-inner-class
- https://www.baeldung.com/java-oop
- https://www.javatpoint.com/super-keyword