

**Academic Year:****Semester:****Course Code:**

S.NO	NAME OF THE EXPERIMENT	Day-to-Day evaluation					Total	Faculty sign
		P1	P2	P3	P4	P5		
1								
2								
3								
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
Day-to-Day evaluation Total								

**1.Aim:** Write Java program to generate Fibonacci sequence for n numbers.

**Description:** The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, starting from 0 and 1.

A sequence of n numbers is an ordered list of n elements, where each element is a number. The order of the numbers is important, as it defines the sequence.

### **Applications of Fibonacci Series:**

Fibonacci numbers have numerous applications in various fields, including:

- **Mathematics:**
  - Golden ratio
  - Number theory
- **Computer Science:**
  - Algorithm analysis
  - Data structures
- **Nature:**
  - Plant growth patterns
  - Animal populations
  - Fractal geometry

Understanding the Fibonacci series and its properties is essential for various mathematical and computational tasks.

## Program:

```
class FibseriesPrint {  
  
    public static void main(String[] args) {  
  
        int n = 10, f1 = 0, f2 = 1;  
  
        System.out.println("Fibonacci Series till " + n + " terms:");  
  
        for (int i = 1; i <= n; i++) {  
            System.out.println(f1 + " ");  
  
            int f3 = f1 + f2;  
  
            f1 = f2;  
            f2 = f3;  
        }  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac FibseriesPrint.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java FibseriesPrint
```

Fibonacci Series till 10 terms:

0

1

1

2

3

5

8

13

21

34

**2. Aim:** Write a Java Program that prompts the user for an integer and then prints out all the prime numbers up to that Integer.

**Description:** Prime numbers are natural numbers greater than 1 that have only two distinct natural number divisors: 1 and the number itself.

we can determine whether a number is prime using a simple algorithm:

1. Iterate from 2 to the square root of the number: This is because if a number  $n$  is not prime, it must have a divisor less than or equal to its square root.
2. Check for divisibility: For each number in the range, check if the given number is divisible by it.
3. If divisible, it's not prime: If the number is divisible by any number in the range, it's not prime.

#### **Applications of Prime Numbers:**

- **Cryptography:** Prime numbers are fundamental to many cryptographic algorithms, such as RSA.
- **Number Theory:** Prime numbers are central to various number-theoretic concepts.
- **Computer Science:** Prime numbers are used in algorithms and data structures.

## Program:

```
import java.util.Scanner;

public class PrimeNumbers {

    public static void main(String[] args) {

        int num, count;

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter a number: ");

        num = sc.nextInt();

        for (int i = 2; i <= num; i++) {

            count = 0;

            for (int j = 2; j < i; j++) {

                if (i % j == 0) {

                    count++;

                }

            }

            if (count == 0) {

                System.out.println(i);

            }

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac PrimeNumbers.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java PrimeNumbers
```

```
Enter a number
```

```
9
```

```
2
```

```
3
```

```
5
```

```
7
```

### 3. Write a java program to demonstrate String and StringBuffer class methods.

**a) Aim:** Write a program to demonstrate the use of various string handling functions in Java.

**Description:** String handling functions are a set of methods provided in programming languages to manipulate and process strings. These functions allow you to perform various operations on strings.

#### Common String Handling Functions:

##### 1. Concatenation:

- Combines two or more strings into a single string.
- Example: `result = str1 + str2`

##### 2. Length:

- Determines the number of characters in a string.
- Example: `length = str.length()`

##### 3. Substrings:

- Extracts a portion of a string.
- Example: `substring = str.substring(start, end)`

##### 4. Searching:

- Finds the index of a substring within a string.
- Example: `index = str.indexOf("substring")`

##### 5. Replacing:

- Replaces occurrences of one substring with another.
- Example: `newStr = str.replace("old", "new")`



## Program:

```
class StringHandlingFunctionsDemo {  
    public static void main(String[] args) {  
        String s1 = "Adithya";  
        String s2 = "Adithya";  
        String s3 = "Adithya";  
        boolean x = s1.equals(s2);  
        System.out.println("Compare s1 and s2:" + x);  
        System.out.println("Character at given position is:" + s1.charAt(5));  
        System.out.println(s1.concat(" the author"));  
        System.out.println(s1.length());  
        System.out.println(s1.toLowerCase());  
        System.out.println(s1.toUpperCase());  
        System.out.println(s1.indexOf('a'));  
        System.out.println(s1.substring(0, 4));  
        System.out.println(s1.substring(4));  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac StringHandlingFunctionsDemo.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java StringHandlingFunctionsDemo
```

```
Compare s1 and s2:true
```

```
Character at given position is:y
```

```
Adithya the author
```

```
7
```

```
adithy
```

```
ADITHYA
```

```
6
```

```
Adit
```

```
hya
```

**b. Aim:** Demonstrate the use of various StringBuffer operations to manipulate and modify strings.

**Description:** StringBuffer class represents a mutable sequence of characters. Unlike String objects, which are immutable, StringBuffer objects can be modified after they are created. This makes them efficient for operations that involve frequent modifications to strings.

### **Key Features and Methods:**

- **Mutability:**
  - **append():** Appends characters or strings to the end of the buffer.
  - **insert():** Inserts characters or strings at a specific index.
  - **delete():** Deletes characters within a specified range.
  - **deleteCharAt():** Deletes a character at a specific index.
  - **replace():** Replaces characters within a specified range with a new string.
  - **reverse():** Reverses the order of characters in the buffer.

### **Why Use StringBuffer?**

- **Efficiency:** When performing multiple modifications to a string, using StringBuffer is more efficient than repeatedly creating new String objects.
- **Thread Safety:** StringBuffer is thread-safe, making it suitable for multi-threaded environments.

## Program:

```
class StringBufferDemo {  
    public static void main(String... args) {  
        StringBuffer sb = new StringBuffer("aditya");  
        int len = sb.length();  
        System.out.println("length of string is" + len);  
        int cap = sb.capacity();  
        System.out.println("default capacity of string is" + cap);  
        sb.append("university");  
        System.out.println("the new string is " + sb);  
        sb.insert(16, "surampalem");  
        System.out.println(sb);  
        sb.reverse();  
        System.out.println(sb);  
        sb.reverse();  
        System.out.println(sb);  
        sb.delete(16, 26);  
        System.out.println(sb);  
        sb.insert(6, " ");  
        System.out.println(sb);  
        sb.deleteCharAt(6);  
        System.out.println(sb);  
        sb.replace(6, 16, " engineering college");  
        System.out.println(sb);  
        String s1 = sb.substring(0, 6);  
        System.out.println(s1);  
        int x = s1.lastIndexOf("a");  
        System.out.println(x);  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac StringBufferDemo.java
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java StringBufferDemo
length of string is 6
default capacity of string is 22
the new string is adityauniversity
adityauniversitysurampalem
melapmarusytisrevinuaytida
adityauniversitysurampalem
adityauniversity
aditya university
adityauniversity
aditya engineering college
aditya
5
```

**4. Aim:** Implement the concept of constructor overloading.

**Description:** Constructor overloading is a technique in object-oriented programming where a class can have multiple constructors with the same name but different parameter lists. This allows for flexible object initialization, providing various ways to create objects with different initial values.

Key Points:

- **Same Name:** All overloaded constructors must share the same name as the class itself.
- **Different Parameter Lists:** Each constructor must have a unique signature, determined by the number and types of its parameters.
- **Automatic Invocation:** When an object is instantiated using the `new` keyword, the appropriate constructor is automatically selected based on the arguments provided during object creation.
- **Flexibility:** Constructor overloading enhances code adaptability and reusability by providing various ways to initialize objects.

Common Use Cases:

- **Default Values:** Providing default values for parameters when specific values are not provided during object creation.
- **Partial Initialization:** Allowing the initialization of only certain object properties, while others remain at their default values.
- **Complete Initialization:** Ensuring that all necessary object properties are initialized upon object creation.

## Program:

```
class ConstructorOverloadingDemo {  
  
    int a, b, c, d, sum;  
  
    ConstructorOverloadingDemo(int p, int q) {  
  
        a = p;  
  
        d = q;  
  
        sum = a + b + c + d;  
  
        System.out.println(sum);  
    }  
  
    ConstructorOverloadingDemo(int p, int q, int r) {  
  
        a = p;  
  
        b = q;  
  
        c = r;  
  
        sum = a + b + c + d;  
  
        System.out.println(sum);  
    }  
  
    public static void main(String... args) {  
  
        ConstructorOverloadingDemo c1 = new ConstructorOverloadingDemo(12, 13);  
  
        ConstructorOverloadingDemo c2 = new ConstructorOverloadingDemo(16, 13, 1);  
  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac ConstructorOverloadingDemo.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java ConstructorOverloadingDemo
```

```
25
```

```
30
```



## 5. Implement the concept of Inheritance,super,abstract and final keywords.

**I. Inheritance:** It is a mechanism that one can acquire the properties of another class. It is a process of deriving a new class from an existing class. A class that is inherited is called as superclass and the class that does the inheriting is called as subclass.

**a) Aim:** Write a java program to demonstrate Single Inheritance.

### Description:

#### Single Inheritance in Java

Single inheritance is a fundamental concept in object-oriented programming where a class inherits properties and behaviors from a single parent class. This creates a hierarchical relationship between classes, promoting code reusability, modularity, and polymorphism.

#### Key Concepts:

- **Parent Class (Superclass):** The class from which properties and methods are inherited.
- **Child Class (Subclass):** The class that inherits from the parent class.
- **Inheritance Relationship:** A "is-a" relationship exists between the child class and the parent class. For example, a "Dog" is an "Animal".

#### Benefits of Single Inheritance:

- **Code Reusability:** Common methods and attributes can be defined in the parent class and reused by child classes.
- **Modularity:** Code is organized into well-defined hierarchies, making it easier to understand and maintain.
- **Polymorphism:** Objects of a subclass can be treated as objects of the parent class, enabling flexible code design and dynamic method dispatch.

## Program:

```
class Animal {  
    void eat() {  
        System.out.println("eating...");  
    }  
}  
  
class Dog extends Animal {  
    void bark() {  
        System.out.println("barking...");  
    }  
}  
  
class Cat extends Animal {  
    void meow() {  
        System.out.println("meowing...");  
    }  
}  
  
class TestInheritance3 {  
    public static void main(String args[]) {  
        Cat c = new Cat();  
        c.meow();  
        c.eat();  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac TestInheritance3.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java TestInheritance3
```

```
meowing...
```

```
eating...
```

**b) Aim:** Write a java program to demonstrate Multilevel Inheritance.

## **Description:**

### **Multilevel Inheritance in Java**

Multilevel inheritance is a type of inheritance in Java where a class inherits properties and methods from another class, which in turn inherits from another class, forming a hierarchical structure. This creates a chain of inheritance, allowing for a more granular and specialized class hierarchy.

#### **Key Concepts:**

- **Parent Class (Superclass):** The base class from which properties and methods are inherited.
- **Child Class (Subclass):** The class that inherits from a parent class.
- **Inheritance Relationship:** A "is-a" relationship exists between the child class and the parent class.

#### **Benefits of Multilevel Inheritance:**

- **Code Reusability:** Common methods and attributes can be defined in the base class and reused by all derived classes.
- **Modularity:** Code is organized into well-defined hierarchies, making it easier to understand and maintain.
- **Specialization:** Each derived class can add its own specific behaviors and attributes, creating specialized classes.

#### **Limitations of Multilevel Inheritance:**

- **Complex Hierarchies:** Deep inheritance hierarchies can make code difficult to understand and maintain.
- **Diamond Problem:** Multiple inheritance can lead to ambiguity in method resolution. Java addresses this by not directly supporting multiple inheritance.

## Program:

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

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

class BabyDog extends Dog {
    void weep() {
        System.out.println("weeping...");
    }
}

class TestInheritance2 {
    public static void main(String args[]) {
        BabyDog d = new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac TestInheritance2.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java TestInheritance2
```

```
weeping...
```

```
barking...
```

```
eating...
```

**c) Aim:** Write a java program to demonstrate Hierarchical Inheritance.

## **Description:**

### **Hierarchical Inheritance in Java**

Hierarchical inheritance is a type of inheritance in Java where multiple classes inherit from a single parent class. This creates a hierarchical structure, promoting code reusability, modularity, and polymorphism.

#### **Key Concepts:**

- **Parent Class (Superclass):** The base class from which properties and methods are inherited.
- **Child Classes (Subclasses):** Classes that inherit from the parent class.
- **Inheritance Relationship:** A "is-a" relationship exists between the child class and the parent class.

#### **Benefits of Hierarchical Inheritance:**

- **Code Reusability:** Common methods and attributes defined in the parent class can be reused by child classes.
- **Modularity:** Code is organized into well-defined hierarchies, making it easier to understand and maintain.
- **Polymorphism:** Objects of child classes can be treated as objects of the parent class, enabling flexible code design and dynamic method dispatch.

#### **Limitations and Considerations:**

- **Complex Hierarchies:** Deep inheritance hierarchies can make code difficult to understand and maintain.
- **Tight Coupling:** Overreliance on inheritance can lead to tight coupling between classes, making it harder to modify and extend the code.

## Program:

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

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
}

class TestInheritance {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        dog.bark();
        dog.eat();
        cat.meow();
        cat.eat();
    }
}
```



## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac TestInheritance.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java TestInheritance
```

```
Barking...
```

```
Eating...
```

```
Meowing...
```

```
Eating...
```

**d) Aim:** Write a java program to demonstrate Hybrid Inheritance.

### **Description:**

Hybrid Inheritance in Java:

Hybrid inheritance is a combination of multiple inheritance types, such as single inheritance and multiple inheritance. However, Java doesn't directly support multiple inheritance to avoid the "diamond problem," where ambiguity arises when two parent classes have a method with the same name.

To achieve the concept of hybrid inheritance in Java, we often use interfaces and abstract classes.

### **How Hybrid Inheritance Works:**

- **Interface Implementation:** A class can implement multiple interfaces, allowing it to inherit multiple behaviors.
- **Class Inheritance:** A class can extend only one parent class, but it can implement multiple interfaces.
- **Diamond Problem Avoidance:** Interfaces don't contain method implementations, so there's no ambiguity when multiple interfaces have methods with the same name.

### **Benefits of Hybrid Inheritance:**

- **Flexibility:** Allows for more complex class hierarchies and relationships.
- **Code Reusability:** Common methods and attributes can be defined in interfaces and base classes.
- **Polymorphism:** Objects of a class can be treated as objects of its parent class and implemented interfaces.

### **Limitations and Considerations:**

- **Complex Design:** Can lead to complex class hierarchies if not used carefully.
- **Multiple Inheritance Limitations:** While interfaces provide a way to achieve multiple inheritance-like behavior, they don't allow for inheritance of implementation details.

## Program:

```
interface Parent {
    void showParent();
}

class Child1 implements Parent {
    public void showParent() {
        System.out.println("This is the Parent interface, implemented in Child1.");
    }
    public void showChild1() {
        System.out.println("This is Child1 class.");
    }
}

class Child2 implements Parent {
    public void showParent() {
        System.out.println("This is the Parent interface, implemented in Child2.");
    }
    public void showChild2() {
        System.out.println("This is Child2 class.");
    }
}

class GrandChild extends Child1 {
    public void showGrandChild() {
        System.out.println("This is the GrandChild class, inheriting from Child1.");
    }
}

public class HybridInheritance {
    public static void main(String[] args) {
```

```
GrandChild grandChild = new GrandChild();  
grandChild.showParent();  
grandChild.showChild1();  
grandChild.showGrandChild();
```

```
Child2 child2 = new Child2();  
child2.showParent();  
child2.showChild2();  
}  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac HybridInheritance.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java HybridInheritance
```

This is the Parent interface, implemented in Child1.

This is Child1 class.

This is the GrandChild class, inheriting from Child1.

This is the Parent interface, implemented in Child2.

This is Child2 class.

**II. Super:** Super keyword is used to refer immediate parent class variables, immediate parent class methods, immediate parent class constructors.

**Aim:** Write a program to demonstrate Super keyword

**Description:**

**Key Points:**

- The super keyword is used to refer to the parent class object.
- It can be used to access both fields and methods of the parent class.
- When used within a constructor, it must be the first statement to call the parent class constructor.
- It's essential to use super judiciously to avoid ambiguity and potential conflicts.

**Common Use Cases:**

- Resolving Ambiguity: When a child class has a method with the same name as a parent class method, super can be used to explicitly call the parent class's version.
- Accessing Hidden Members: If a child class member has the same name as a parent class member, super can be used to access the parent class's member.
- Initializing Parent Class Members: In constructors, super can be used to pass arguments to the parent class constructor.

**Best Practices:**

- Use super to access members of the parent class when necessary, but avoid overuse.
- Ensure that you understand the inheritance hierarchy and the specific members you want to access.
- Test your code thoroughly to ensure that super is used correctly and doesn't introduce unintended side effects.

## Program:

```
class Animal {  
    String color = "white";  
}  
  
class Dog extends Animal {  
    String color = "black";  
  
    void printColor() {  
        System.out.println(color);  
        System.out.println(super.color);  
    }  
}  
  
class TestSuper1 {  
    public static void main(String args[]) {  
        Dog d = new Dog();  
        d.printColor();  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac TestSuper1.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java TestSuper1
```

```
black
```

```
white
```



### III. Abstract:

An abstract class is a class that cannot be instantiated directly. It serves as a blueprint for other classes, providing a common set of methods and variables. Abstract classes are often used to define a general interface for a group of related classes.

**Aim:** Write a java program to demonstrate Abstract class.

### Description:

#### Key Characteristics of Abstract Classes:

##### 1. Abstract Methods:

- An abstract method is a method declared without a body.
- Subclasses must provide concrete implementations for these methods.
- Abstract methods are declared using the abstract keyword.

##### 2. Concrete Methods:

- Abstract classes can contain concrete methods, which are methods with a body.
- These methods are inherited by subclasses.

#### Why Use Abstract Classes?

- **Code Reusability:** By defining common methods in an abstract class, you can avoid code duplication in subclasses.
- **Enforcing a Contract:** Abstract classes can be used to enforce a specific contract on subclasses, ensuring that they implement certain methods.
- **Creating Base Classes:** Abstract classes provide a foundation for creating hierarchies of

## Program:

```
abstract class Shape {
    abstract void draw();
    public void erase() {
        System.out.println("Erasing shape");
    }
}

class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

class Square extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a square");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle();
        Shape shape2 = new Square();
        shape1.draw();
        shape2.draw();
        shape1.erase();
        shape2.erase();
    }
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac Main.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java Main
```

```
Drawing a circle
```

```
Drawing a square
```

```
Erasing shape
```

```
Erasing shape
```

## V. Final:

The final keyword in Java is a powerful modifier that can be applied to variables, methods, and classes. When used, it imposes certain restrictions on how these elements can be used.

**a) Aim:** Write a java program to demonstrate Final Variable.

### Description:

#### Final Variables in Java

A final variable in Java is a variable whose value cannot be changed once it's assigned. This makes it a constant. Once a value is assigned to a final variable, it remains fixed throughout the program's execution.

#### Key Points:

- **Immutable:** The value of a final variable cannot be reassigned.
- **Constant:** final variables are often used to create constants.
- **Declaration:**

```
final data_type variable_name = value;
```

#### Benefits of Using final Variables:

- **Readability:** final variables make code more readable and understandable, as their values are fixed.
- **Efficiency:** The compiler can optimize code involving final variables, as their values are known at compile time.
- **Security:** final variables prevent accidental modification of values, enhancing code reliability.
- **Immutability:** By making variables final, you can create immutable objects, which are essential for thread safety and efficient programming.

## Program:

```
class example {  
  
    final int speed_limit = 60;  
  
    void m1() {  
  
        System.out.println(speed_limit);  
  
    }  
  
    public static void main(String args[]) {  
  
        example obj = new example();  
  
        obj.m1();  
  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac example.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java example
```

```
60
```

**b)Aim:** Write a java program to demonstrate Final Method.

## **Description:**

### **Final Methods in Java**

A final method in Java is a method that cannot be overridden by subclasses. Once declared as final, the method's implementation is fixed and cannot be modified by inheriting classes.

### **Key Points:**

- **Prevents Overriding:** The final keyword ensures that the method's behavior remains consistent across the inheritance hierarchy.
- **Enhances Code Stability:** By making methods final, you can prevent unintended modifications that might break the code's functionality.
- **Performance Optimization:** In some cases, the compiler can optimize final methods, leading to potential performance improvements.

### **Use Cases:**

- **Core Library Methods:** Many core library classes, such as String and Integer, use final methods to ensure their behavior remains consistent across different versions of the Java platform.
- **Security-Critical Methods:** Methods that are critical to the security of an application can be made final to prevent malicious tampering.
- **Performance-Critical Methods:** In some cases, making a method final can allow the compiler to optimize its implementation.

## **Program:**

```
class example2 {  
  
    int a = 10, b = 20;  
  
    final void add() {  
  
        System.out.println("Addition is: " + (a + b));  
  
    }  
  
    public static void main(String[] args) {  
  
        example2 obj = new example2();  
  
        obj.add();  
  
    }  
}
```



## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac example2.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java example2
```

```
Addition is: 30
```

**c)Aim:** Write a java Program to demonstrate Final class.

## **Description:**

### **Final Classes in Java**

A final class in Java is a class that cannot be extended or subclassed. Once a class is declared as final, it becomes the end of an inheritance hierarchy.

### **Why Use Final Classes?**

- **Security:** By making a class final, you can prevent unauthorized modification of its behavior.
- **Efficiency:** The JVM can optimize the code for final classes, as it knows that there won't be any subclasses.
- **Immutability:** final classes can be used to create immutable objects, which are essential for thread safety and efficient programming.

### **Key Points:**

- final classes cannot be inherited.
- They are often used to create immutable objects.
- They can improve code security and efficiency.
- Use final classes judiciously to balance flexibility and security.

### **When to Use Final Classes:**

- **Utility Classes:** Classes that provide utility methods can be made final to prevent unintended subclassing.
- **Immutable Classes:** Classes that represent immutable data structures should be made final to ensure their immutability.
- **Security-Sensitive Classes:** Classes that handle sensitive information can be made final to protect their integrity.

## Program:

```
final class example3 {  
  
    int a = 10, b = 20;  
  
    void mul() {  
  
        System.out.println("Multiplication is: " + (a * b));  
  
    }  
  
    public static void main(String[] args) {  
  
        example3 obj = new example3(); // Corrected class name  
  
        obj.mul();  
  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac example2.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java example2
```

```
Multiplication is: 200
```

**6. Aim:** Write a java program for dynamic dispatch of methods.

### **Description:**

#### **Dynamic Dispatch (Runtime Polymorphism) in Java**

**Dynamic dispatch**, also known as **runtime polymorphism**, is a fundamental concept in object-oriented programming that allows you to call methods on objects whose actual type is determined at runtime, rather than at compile time. This flexibility is achieved through inheritance and method overriding.

#### **How Dynamic Dispatch Works:**

1. **Inheritance:** A subclass inherits properties and methods from its parent class.
2. **Method Overriding:** A subclass can provide its own implementation for a method inherited from the parent class.
3. **Reference Variable:** A reference variable can refer to objects of its declared type or any of its subclasses.
4. **Method Call:** When a method is called on a reference variable, the actual method to be executed is determined at runtime based on the object's type.

#### **Key Points:**

- Dynamic dispatch is a powerful mechanism for achieving flexible and reusable code.
- It allows you to write code that can work with objects of different types without knowing their exact type at compile time.
- It's essential for designing object-oriented systems that are extensible and maintainable.
- By understanding dynamic dispatch, you can create more efficient and elegant Java applications.

## Program:

```
class Phone {  
    public void showTime() {  
        System.out.println("Time is the same");  
    }  
    public void on() {  
        System.out.println("Turning on phone");  
    }  
}  
  
class MastPhone extends Phone {  
    public void music() {  
        System.out.println("Playing music");  
    }  
    @Override  
    public void on() {  
        System.out.println("Turning on MastPhone");  
    }  
}  
  
class SmartPhone extends Phone {  
    public void music() {  
        System.out.println("Playing music");  
    }  
    @Override  
    public void on() {  
        System.out.println("Turning on SmartPhone");  
    }  
}  
  
class Demo {
```

```
public static void main(String[] args) {  
    Phone obj = new SmartPhone();  
    obj.showTime();  
    obj.on();  
}  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac Demo.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java Demo
```

```
Time is the same
```

```
Turning on SmartPhone
```



**7. Aim:** Write a java program to create and demonstrate packages.

## **Description:**

### **What is a Package?**

A package in Java is a mechanism used to organize a set of related classes and interfaces. It provides a hierarchical namespace, preventing naming conflicts between different classes and interfaces. Packages are essential for modularizing code, improving code organization, and promoting code reusability.

### **Why Use Packages?**

- **Namespace Management:** Packages provide a way to avoid naming conflicts between classes with the same name.
- **Code Organization:** Related classes and interfaces can be grouped together into logical packages.
- **Modularity:** Packages can be used to create modular components that can be reused in different projects.
- **Access Control:** Packages can control the visibility of classes and interfaces to other packages.

### **Package Declaration:**

A package is declared at the beginning of a Java source file using the package keyword:

### **Package Structure:**

Java packages follow a hierarchical structure, similar to a file system. For example, the `java.util` package is a subpackage of the `java` package.

### **Default Package:**

If a class is not explicitly declared within a package, it belongs to the default package. However, it's generally recommended to organize classes into packages to improve code organization and avoid potential naming conflicts.

## Program:

```
package mypack;

public class Addition {

    int a, b;

    public int sum(int p, int q) {

        a = p;

        b = q;

        return a + b;

    }

}
```

```
package mypack;

public class Subtraction {

    int a, b;

    public int sub(int p, int q) {

        a = p;

        b = q;

        return a - b;

    }

}
```

```
import mypack.Subtraction;

import mypack.Addition;

class Test {
```

```
public static void main(String[] args) {  
  
    Addition O1 = new Addition();  
  
    int r = O1.sum(10, 20);  
  
    System.out.println("Sum=" + r);  
  
    Subtraction O2 = new Subtraction();  
  
    int s = O2.sub(10, 20);  
  
    System.out.println("Sub=" + s);  
  
}  
  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac Test.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java Test
```

```
Sum=30
```

```
Sub=-10
```

**8. Aim:** Write a java program to implement the concept of package and interface.

### **Description:**

#### **What is a Package?**

A package in Java is a mechanism used to organize a set of related classes and interfaces. It provides a hierarchical namespace, preventing naming conflicts between different classes and interfaces. Packages are essential for modularizing code, improving code organization, and promoting code reusability.

#### **Why Use Packages?**

- **Namespace Management:** Packages provide a way to avoid naming conflicts between classes with the same name.
- **Code Organization:** Related classes and interfaces can be grouped together into logical packages.
- **Modularity:** Packages can be used to create modular components that can be reused in different projects.
- **Access Control:** Packages can control the visibility of classes and interfaces to other packages.

#### **What is an Interface?**

An interface in Java is a blueprint of a class. It defines a set of abstract methods that a class must implement. Interfaces are used to achieve abstraction and polymorphism.

#### **Key Characteristics of Interfaces:**

- **Abstract Methods:** Interfaces only contain abstract methods.
- **Multiple Inheritance:** A class can implement multiple interfaces.
- **No Constructor:** Interfaces cannot have constructors.
- **All Methods are Public and Abstract:** By default, methods in an interface are public and abstract.

## Program:

```
package myinterface;

interface FirstInterface {
    final int b = 10;
    public void message();
}

interface SecondInterface {
    public void Display();
}

public class interfacepackage implements FirstInterface, SecondInterface {
    public void message() {
        System.out.println("Welcome to first interface");
    }
    public void Display() {
        System.out.println("Welcome to second interface...");
    }
}

import myinterface.interfacepackage;

class TestInterfacePackage {
    public static void main(String[] args) {
        interfacepackage obj = new interfacepackage();
        obj.message();
        obj.Display();
        System.out.println(obj.b);
    }
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac TestInterfacePackage.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java TestInterfacePackage
```

```
Welcome to first interface
```

```
Welcome to second interface...
```

```
10
```

**9. Aim:** Write a program to illustrate exception handling mechanism using multiple catch clauses.

### **Description:**

#### **Exception Handling with Multiple Catch Clauses in Java**

Exception handling is a mechanism in Java that allows you to gracefully handle errors and exceptions that may occur during program execution. By using try, catch, and finally blocks, you can anticipate and respond to potential exceptions, preventing your program from crashing and providing a more robust and user-friendly experience.

#### **Multiple Catch Clauses:**

When there's a possibility of multiple types of exceptions, you can use multiple catch blocks to handle them specifically. Each catch block should catch a specific type of exception.

#### **Key Points:**

- **Order of catch Blocks:** The order of catch blocks matters. More specific exceptions should be caught first.
- **Parent Exception:** A catch block can catch a parent exception type to handle multiple exceptions in a single block.
- **finally Block:** It's guaranteed to execute, making it ideal for cleanup operations.
- **Exception Hierarchy:** Understanding the Java exception hierarchy can help you choose the appropriate exception types to catch.

By effectively using multiple catch blocks, you can create more robust and user-friendly applications that can handle various exceptions gracefully.



## Program:

```
import java.lang.*;

public class Multiplecatch3 {

    public static void main(String[] args) {

        try {

            int a[] = new int[5];

            a[5] = 30 / 0;

            System.out.println(a[10]);

        } catch (ArithmeticException e) {

            System.out.println("Arithmetic Exception occurs");

        } catch (ArrayIndexOutOfBoundsException e) {

            System.out.println("ArrayIndexOutOfBounds Exception occurs");

        } catch (Exception e) {

            System.out.println("Parent Exception occurs");

        } finally {

            System.out.println("rest of the code");

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac Multiplecatch3.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java Multiplecatch3
```

```
Arithmetic Exception occurs
```

```
rest of the code
```

**10. Aim:** Write a java program to implement the concept of multi threading.

## **Description:**

### **Multithreading in Java**

Multithreading is a programming technique that allows a program to execute multiple threads concurrently, improving performance and responsiveness. In Java, threads are lightweight processes that share the same memory space but execute independently.

### **Key Concepts:**

1. **Thread:** A thread is the smallest unit of execution within a process.
2. **Concurrency:** Multiple threads executing simultaneously.
3. **Parallelism:** Multiple threads executing on different cores.

### **Thread Lifecycle:**

1. **New:** The thread is created but not yet started.
2. **Runnable:** The thread is ready to run but waiting for CPU time.
3. **Running:** The thread is currently executing.
4. **Blocked:** The thread is waiting for a resource or event.

### **Thread Synchronization:**

When multiple threads access shared resources, it's essential to synchronize their access to prevent data corruption and race conditions. Java provides synchronization mechanisms like:

1. **synchronized Keyword:**
  - Can be applied to methods or blocks of code.
  - Ensures that only one thread can execute the synchronized code at a time.
2. **wait() and notify() Methods:**
  - Used for inter-thread communication and synchronization.

## Program:

```
class AThread extends Thread {  
    public void run() {  
        for (int i = 1; i <= 10; i++) {  
            System.out.println("i=" + i);  
        }  
    }  
}  
  
class BThread extends Thread {  
    public void run() {  
        for (int j = 11; j <= 20; j++) {  
            System.out.println("j=" + j);  
        }  
    }  
}  
  
class CThread extends Thread {  
    public void run() {  
        for (int k = 21; k <= 30; k++) {  
            System.out.println("k=" + k);  
        }  
    }  
}
```

```
public class ThreadClass {  
    public static void main(String args[]) {  
        AThread t1 = new AThread();  
        t1.start();  
        BThread t2 = new BThread();  
        t2.start();  
        CThread t3 = new CThread();  
        t3.start();  
    }  
}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac ThreadClass.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java ThreadClass
```

```
i=1
```

```
k=21
```

```
j=11
```

```
j=12
```

```
j=13
```

```
j=14
```

```
j=15
```

```
i=2
```

```
i=3
```

```
i=4
```

```
i=5
```

```
i=6
```

```
i=7
```

```
i=8
```

```
i=9
```

k=22

k=23

k=24

k=25

k=26

k=27

k=28

k=29

k=30

j=16

i=10

j=17

j=18

j=19

j=20

**11. Aim:** Write a JDBC program to perform CRUD operations.

**Description:**

**CRUD Operations in Java**

**CRUD** is an acronym for **Create, Read, Update, and Delete**. These are the four basic operations performed on data in a database. In Java, we can perform CRUD operations on databases using JDBC (Java Database Connectivity).

**A. Create Operation (Insert)**

To insert a new record into a database table, we use the INSERT INTO SQL statement.

**B. Read Operation (Select)**

To retrieve data from a database table, we use the SELECT SQL statement.

**C. Update Operation (Update)**

To modify existing data in a database table, we use the UPDATE SQL statement.

**D. Delete Operation (Delete)**

To remove a record from a database table, we use the DELETE SQL statement.

**Key Points to Remember:**

- **Prepared Statements:** Use prepared statements to prevent SQL injection attacks and improve performance.
- **Error Handling:** Use try-catch blocks to handle potential exceptions like SQLException.
- **Database Connection:** Establish a database connection using DriverManager.getConnection().
- **Closing Resources:** Always close database connections, statements, and result sets to release resources.
- **Transaction Management:** Use transactions to ensure data consistency, especially for multiple operations.



## A. Program:

### Example program for Table creation:

```
import java.sql.*;

class CreateTable {

    public static void main(String args[]) {

        try {

            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mca",
"root", "Renuka@123");

            Statement st = con.createStatement();

            st.executeUpdate("create table Student(sid int, sname varchar(20))");

            System.out.println("Student table created successfully");

        } catch (SQLException e) {

            System.out.println(e);

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac CreateTable.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java -cp mysql-connector-j-8.0.31.jar; CreateTable
```

```
Student table created successfully
```

## Program:

**Example program for Inserting records in table:**

```
import java.sql.*;

class InsertRecord {

    public static void main(String args[]) {

        try {

            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mca",
"root", "password");

            Statement st = con.createStatement();

            st.executeUpdate("insert into Student(sid, sname) values(134, Renuka)");

            System.out.println("Record inserted successfully");

        } catch (Exception e) {

            System.out.println(e);

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac InsertRecord.java  
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java -cp mysql-connector-j-8.0.31.jar;  
InsertRecord  
Record inserted successfully
```

## B. Program:

**Example program for Retrieving the records:**

```
import java.sql.*;

class RetrieveRecord {

    public static void main(String args[]) {

        try {

            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mca",
"root", "password");

            PreparedStatement ps = con.prepareStatement("select * from Student");

            ResultSet rs = ps.executeQuery();

            while (rs.next()) {

                int id = rs.getInt("sid");

                String name = rs.getString("sname");

                System.out.println(id + "\t" + name);

            }

        } catch (SQLException e) {

            System.out.println(e);

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac RetrieveRecord.java  
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java -cp mysql-connector-j-8.0.31.jar;  
Retrieverecord  
134   Renuka  
138   Durga
```

## C. Program:

**Example program for updating the records:**

```
import java.sql.*;

class UpdateRecord {

    public static void main(String... args) {

        try {

            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mca",
"root", "password");

            Statement st = con.createStatement();

            st.executeUpdate("update Student set sname='Durga' where sid=138");

            System.out.println("data updated successfully");

        } catch (SQLException e) {

            System.out.println(e);

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac UpdateRecord.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java -cp mysql-connector-j-8.0.31.jar; UpdateRecord  
data updated successfully
```



## D. Program:

**Example program for Delete the records from table:**

```
import java.sql.*;

class DeleteRecord {

    public static void main(String... args) {

        try {

            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mca",
"root", "password");

            Statement st = con.createStatement();

            st.executeUpdate("delete from Student where sid=138");

            System.out.println("data deleted successfully");

        } catch (SQLException e) {

            System.out.println(e);

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> javac DeleteRecord.java  
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java -cp mysql-connector-j-8.0.31.jar; DeleteRecord  
data deleted successfully
```

**12. Aim:** Write a program to receive two numbers from a HTML form and display their sum in HttpServletResponse.

## **Description:**

### **Servlet: The Dynamic Engine**

Servlets are Java programs that extend the capabilities of web servers. They handle dynamic content generation, database interactions, and user session management. Imagine a servlet as a skilled worker who can customize the house based on the occupant's needs, like installing specific furniture or adjusting the lighting.

### **XML: The Data Format**

XML (eXtensible Markup Language) is a flexible data format used for storing and transporting data. It's like a universal language that different systems can understand. In web applications, XML can be used to store configuration settings, data feeds, or to exchange information between the server and the client.

### **How They Work Together**

1. **HTML:** Creates the basic structure of the webpage, including the layout and content.
2. **Servlet:** Processes user requests, generates dynamic content, and interacts with databases.
3. **XML:** Stores configuration data or data to be displayed on the page.

### **A Real-World Analogy**

- **HTML:** The architect designs the blueprint, outlining the rooms, their sizes, and their positions.
- **Servlet:** The construction crew builds the house, making modifications based on the client's preferences and ensuring it's sturdy and functional.
- **XML:** The interior designer uses a blueprint to plan the furniture layout and decor, ensuring it aligns with the overall design.

## Program:

### Index.HTML:

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<meta charset="UTF-8">
```

```
<title>Insert title here</title>
```

```
</head>
```

```
<body>
```

```
<form action="add">
```

```
    Enter 1st value:<input type="text" name="num1"><br>
```

```
    Enter 2nd value:<input type="text" name="num2"><br>
```

```
    <input type="submit">
```

```
</form>
```

```
</body>
```

```
</html>
```

## Web.XML:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-
app_4_0.xsd" id="WebApp_ID" version="4.0">
```

```
<servlet>
```

```
<servlet-name>abc</servlet-name>
```

```
<servlet-class>com.mcab.Addservlet</servlet-class>
```

```
</servlet>
```

```
<servlet-mapping>
```

```
<servlet-name>abc</servlet-name>
```

```
<url-pattern>/add</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

## AddServlet.Java:

```
package com.mcab;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class Addservlet extends HttpServlet{

    private static final long serialVersionUID = 1L;

    public void service(HttpServletRequest req,HttpServletResponse res) throws IOException

    {

        int i=Integer.parseInt(req.getParameter("num1"));

        int j=Integer.parseInt(req.getParameter("num2"));

        int k=i+j;

        //System.out.println("result"+k);

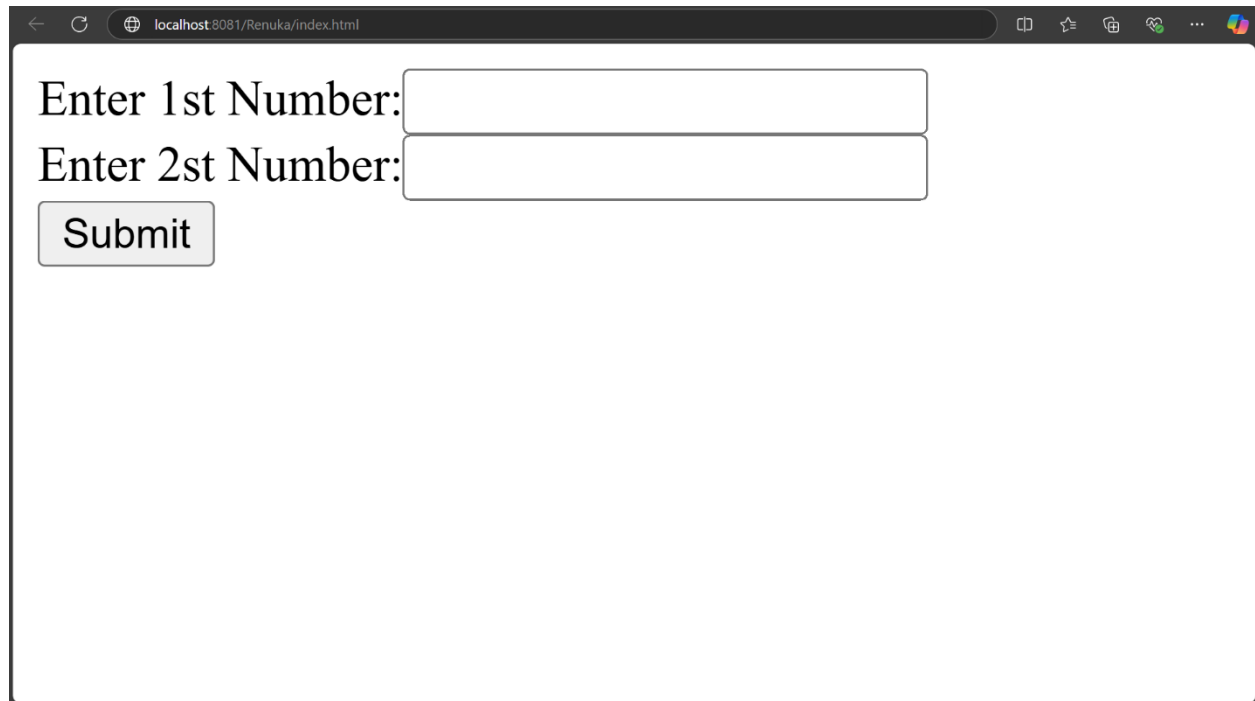
        PrintWriter out=res.getWriter();

        out.println("result is"+k);

    }

}
```

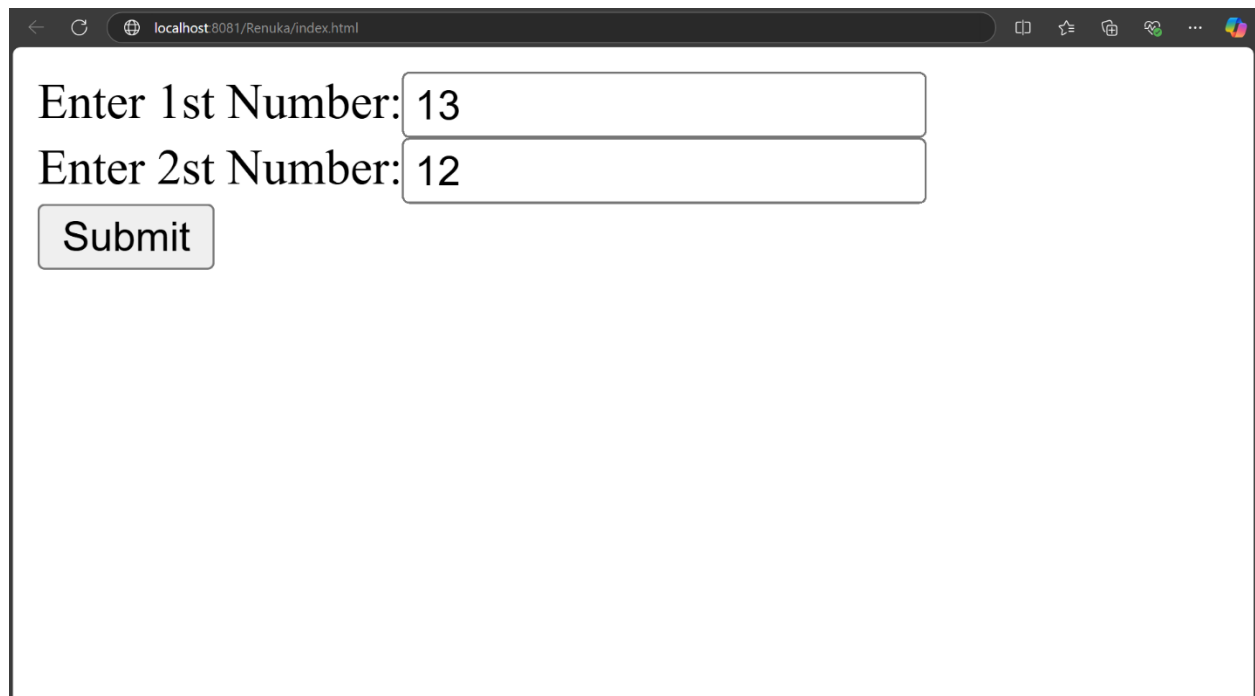
## Output:



A screenshot of a web browser window displaying a form. The address bar shows 'localhost:8081/Renuka/index.html'. The form contains two input fields and a submit button. The first input field is labeled 'Enter 1st Number:' and the second is labeled 'Enter 2st Number:'. The submit button is labeled 'Submit'.

Enter 1st Number:

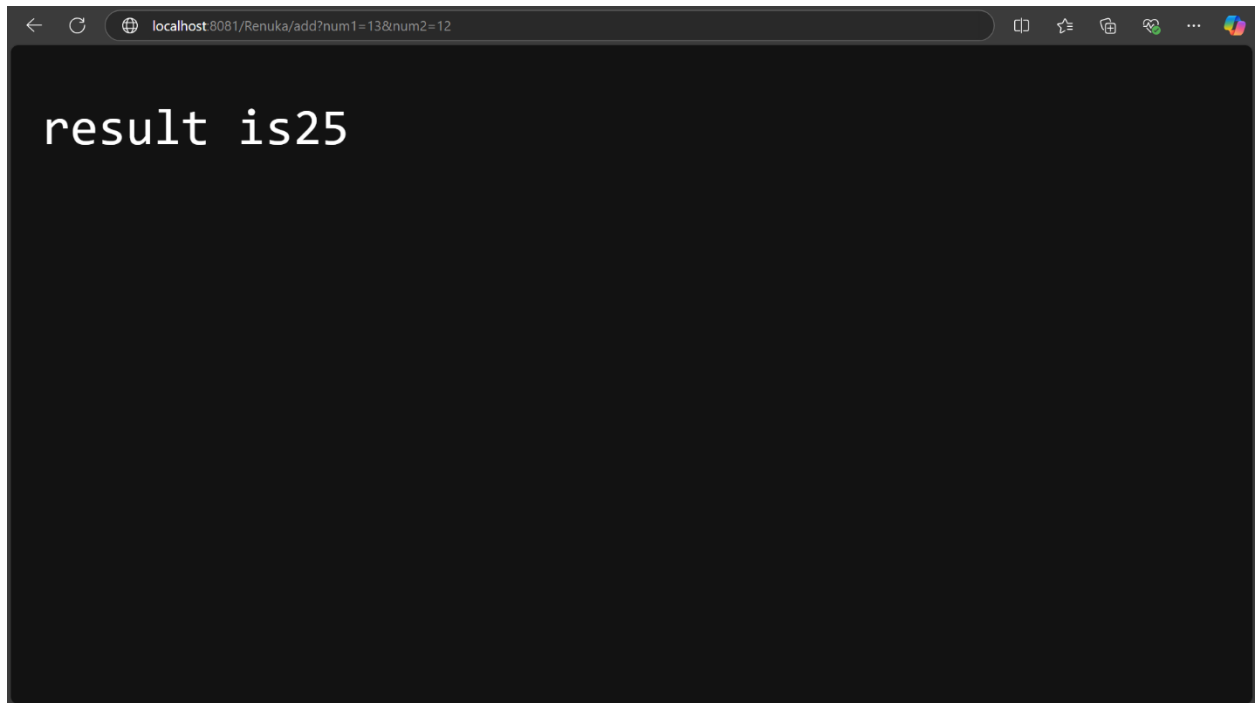
Enter 2st Number:



A screenshot of the same web browser window, but now the input fields contain the numbers 13 and 12. The submit button remains labeled 'Submit'.

Enter 1st Number:

Enter 2st Number:





**13. Aim:** Write a program to Authenticate using JSP.

**Description:**

**Authentication with JSP**

JSP enables you to create dynamic web pages that can handle user input, perform server-side logic, and manage sessions. This makes it well-suited for implementing user authentication within web applications.

**Key Steps:**

- **Create a Login Form:** Design an HTML form to collect user credentials (username and password).
- **Handle Form Submission:**
  - In the JSP page, receive the submitted data.
  - Validate user input (e.g., check for empty fields, correct data types).
  - **Authenticate:**
    - Retrieve user information from a database.
    - Compare provided credentials with stored data.
- **Session Management:**
  - If successful, create a session to track the logged-in user.
  - Store user information (e.g., username, role) in the session.
  - Redirect the user to appropriate pages (e.g., welcome page, restricted areas).
- **Handle Failures:** Display error messages and provide an opportunity to try again.

**Benefits:**

- **Dynamic Content:** Generate personalized login forms and error messages.
- **Server-Side Logic:** Perform complex authentication checks.

## Program:

### Authenticator HTML File :

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>New</title>

</head>

<body>

    <h2>Login</h2>

    <form action="authenticator.jsp" method="post">

        <label for="username">Username:</label>

        <input type="text" id="username" name="username" required>

        <br><br>

        <label for="password">Password:</label>

        <input type="password" id="password" name="password" required>

        <br><br>

        <input type="submit" value="Login">

    </form>


</body>

</html>
```

## Authenticator JSP File :

```
<%  
  
    String username = request.getParameter("username");  
  
    String password = request.getParameter("password");  
  
    String validUsername = "Renuka";  
  
    String validPassword = "Renuka@77";  
  
    if (username != null && password != null && username.equals(validUsername) && password.equals(validPassword)) {  
  
        // Successful login  
  
        out.println("<h3>Welcome, " + username + "!</h3>");  
  
        } else {  
  
        // Invalid credentials  
  
        out.println("<h3>Invalid Username or Password!</h3>");  
  
        }  
  
%>
```

## Output:



A screenshot of a web browser window. The address bar shows 'localhost:8080/Renuka/authenticate.html'. The page content includes a large heading 'Login', followed by a 'Username:' label and an empty text input field, a 'Password:' label and an empty text input field, and a 'Login' button at the bottom.

localhost:8080/Renuka/authenticate.html

# Login

Username:

Password:

Login



A screenshot of a web browser window, similar to the one above. The address bar shows 'localhost:8080/Renuka/authenticate.html'. The page content includes a large heading 'Login', followed by a 'Username:' label and a text input field containing the value 'Renuka', a 'Password:' label and a text input field containing ten dots, and a 'Login' button at the bottom.

New

localhost:8080/Renuka/authenticate.html

# Login

Username:

Password:

Login



**Welcome, Renuka!**



**Invalid Username or Password!**

**14. Aim:** Write a JSP program calculates factorial values for an integer number, while the input is taken from an HTML form.

## Description:

### Factorial Calculation in JSP

#### 1. Core Concept:

- **Factorial:** The factorial of a non-negative integer 'n' (denoted as 'n!') is the product of all positive integers less than or equal to 'n'.
- **Formula:**  $n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$

### Factorial Calculation in JSP

#### 1. Core Concept:

- **Factorial:** The factorial of a non-negative integer 'n' (denoted as 'n!') is the product of all positive integers less than or equal to 'n'.
- **Formula:**  $n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$
- **Example:**  $5! = 5 * 4 * 3 * 2 * 1 = 120$

#### 2. JSP Implementation:

- **Create an HTML form:** Allow the user to enter an integer value.
- **Handle form submission:**
  - In the JSP page, retrieve the entered integer value.
  - Use Java code within the JSP to calculate the factorial using a loop (e.g., for loop).
- **Display the result:** Display the calculated factorial value to the user.

## Program:

### Factorials HTML File :

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>New</title>

</head>

<body>

    <h2>Factorial of the given number</h2>

    <form action="factorial.jsp" method="post">

        <label for="number">Enter a number: </label>

        <input type="number" id="number" name="number" required>

        <input type="submit" value="Calculate">

    </form>

</body>

</html>
```

## Factorials JSP File:

```
<%  
  
    String numberStr = request.getParameter("number");  
  
    if (numberStr != null && !numberStr.isEmpty()) {  
  
        try {  
  
            int number = Integer.parseInt(numberStr);  
  
            long factorial = 1;  
  
            if (number >= 0) {  
  
                for (int i = 1; i <= number; i++) {  
  
                    factorial *= i;  
  
                }  
  
                out.println("<h3>Factorial of " + number + " is: " + factorial + "</h3>");  
  
            } else {  
  
                out.println("<h3>Please enter a non-negative number.</h3>");  
  
            }  
  
        } catch (NumberFormatException e) {  
  
            out.println("<h3>Invalid input. Please enter a valid number.</h3>");  
  
        }  
  
    }  
  
%>
```



**Output:**



## Factorial of the given number

Enter a number:



## Factorial of the given number

Enter a number:

---



**Factorial of 6 is: 720**

## Additional Programs:

**1. Aim:** Write a java program to demonstrate String Tokenization.

## Description:

### What is String Tokenization?

String tokenization is the process of breaking down a string into smaller units called tokens. These tokens are typically separated by delimiters, which are characters or strings that mark the boundaries between tokens.

### StringTokenizer Class

Java provides the StringTokenizer class to perform string tokenization. It offers a simple way to break a string into tokens based on a specified set of delimiters.

### Key Points:

- **Delimiter:** The delimiter is the character or pattern used to separate tokens.
- **Token:** A token is a substring that is separated by delimiters.
- **StringTokenizer vs. String.split():**
  - StringTokenizer is less flexible and can be more difficult to use in complex scenarios.
  - String.split() is more powerful and allows for more complex delimiter patterns.
- **Regular Expressions:** You can use regular expressions with String.split() to define complex delimiter patterns.
- **Tokenizing Strings with Multiple Delimiters:** You can use regular expressions to define multiple delimiters.

### Choosing the Right Approach:

- **StringTokenizer:** Simpler for basic tokenization tasks, but less flexible.
- **String.split():** More powerful and flexible, especially for complex patterns.

## Program:

```
import java.util.StringTokenizer;

public class StringTokenizationExample {

    public static void main(String[] args) {

        String sentence = "This is a sample sentence with multiple words.";

        StringTokenizer tokenizer = new StringTokenizer(sentence);

        while (tokenizer.hasMoreTokens()) {

            String token = tokenizer.nextToken();

            System.out.println(token);

        }

        String[] words = sentence.split(" ");

        for (String word : words) {

            System.out.println(word);

        }

    }

}
```

## Output:

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\java> javac StringTokenizationExample.java
```

```
C:\Users\Renuka\OneDrive\Desktop\24M11MC134\Java> java StringTokenizationExample.java
```

This

is

a

sample

sentence

with

multiple

words.

This

is

a

sample

sentence

with

multiple

words.

**2. Aim:** Write a Java program to solve producer consumer problem using synchronization.

### **Description:**

#### **Producer-Consumer Problem with Synchronization**

The producer-consumer problem is a classic concurrency problem that involves two types of threads:

- **Producers:** These threads generate data and place it into a shared buffer.
- **Consumers:** These threads consume data from the shared buffer.

The challenge arises when the buffer is either full (producer cannot add data) or empty (consumer cannot consume data). Synchronization mechanisms are crucial to prevent these situations and ensure efficient data exchange between the threads.

#### **Key Concepts:**

1. **Shared Buffer:** A common data structure (e.g., an array, queue) that both producers and consumers access.
2. **Synchronization:** Mechanisms to control access to the shared buffer and prevent race conditions. Common techniques include:
  - **Mutual Exclusion:** Only one thread can access the buffer at a time. This is often achieved using locks or semaphores.
  - **Wait and Notify:** Threads can wait for specific conditions (e.g., buffer empty/full) and be notified when those conditions change.
3. **Bounded Buffer:** In many cases, the buffer has a limited capacity. This adds another layer of complexity, as producers must wait if the buffer is full, and consumers must wait if the buffer is empty.

#### **Benefits of Synchronization:**

- Prevents race conditions and data corruption.
- Ensures efficient and orderly data exchange between threads.

## Program:

```
class Q {  
    int[] buffer = new int[5]; // Buffer size: 5  
    int in = 0;  
    int out = 0;  
    int count = 0;  
    synchronized void put(int n) {  
        while (count == buffer.length) { // Buffer full  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                System.out.println("InterruptedException caught");  
            }  
        }  
        buffer[in] = n;  
        in = (in + 1) % buffer.length; // Circular buffer  
        count++;  
        notify();  
    }  
    synchronized int get() {  
        while (count == 0) { // Buffer empty  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                System.out.println("InterruptedException caught");  
            }  
        }  
        int n = buffer[out];  
        out = (out + 1) % buffer.length; // Circular buffer  
        count--;  
        notify();  
        return n;  
    }  
}
```

## Output:

Put: 0

Got: 0

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

Got: 3

Put: 4

Got: 4

Put: 5

Got: 5

// ... and so on



**3. Aim:** Write a program to Authenticate using Servlet.

## **Description:**

## **Authenticating Login Using Servlets in Java**

### **1. Core Components:**

- **Login Form (HTML):** A web page with an HTML form for users to enter their username and password.
- **Login Servlet:** A Java Servlet that handles the form submission, validates user credentials, and responds accordingly.
- **Database:** A database (e.g., MySQL, PostgreSQL) to store user information (username, password, etc.).

### **2. Workflow:**

#### **1. User Interaction:**

- The user opens the login page in their web browser.
- They enter their username and password into the form.
- The user submits the form, and the browser sends the data to the Login Servlet.

#### **2. Servlet Processing:**

- The Login Servlet receives the username and password from the request.
- It establishes a connection to the database.
- The servlet executes an SQL query to retrieve user information based on the provided username and password.
- The servlet compares the entered credentials with the data retrieved from the database.

#### **3. Authentication Result:**

- If the credentials match, the servlet redirects the user to the protected area (e.g., a welcome page) or grants access to specific resources.
- If the credentials are incorrect, the servlet displays an error message to the user.

## Program:

### Authenticate.HTML:

```
<!DOCTYPE html>

<html>

<head>

    <title>Login Page</title>

</head>

<body>

    <h2>Login</h2>

    <form action="LOGIN" method="POST">

        <label for="username">Username:</label>

        <input type="text" id="username" name="username" required>

        <br>

        <label for="password">Password:</label>

        <input type="password" id="password" name="password" required>

        <br>

        <button type="submit">Login</button>

    </form>

</body>

</html>
```

## Login.Servlet:

```
import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/authenticator")

public class Login extends HttpServlet {

    private static final String VALID_USERNAME = "Renuka";

    private static final String VALID_PASSWORD = "Renuka@77";

    @Override

    protected void doPost(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

        response.setContentType("text/html");

        String username = request.getParameter("username");

        String password = request.getParameter("password");

        response.getWriter().println("<!DOCTYPE html>");

        response.getWriter().println("<html>");

        response.getWriter().println("<head><title>Login Result</title></head>");

        response.getWriter().println("<body>");
```

```
        if (username != null && password != null && username.equals(VALID_USERNAME) &&
password.equals(VALID_PASSWORD)) {

            response.getWriter().println("<h3>Welcome, " + username + "!</h3>");

        } else {

            response.getWriter().println("<h3>Invalid Username or Password!</h3>");

            response.getWriter().println("<a href=\"login.html\">Try Again</a>");

        }

        response.getWriter().println("</body>");

        response.getWriter().println("</html>");

    }

}
```

## Web.XML:

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <title>Login Page</title>
```

```
</head>
```

```
<body>
```

```
  <h2>Login</h2>
```

```
  <form action="LOGIN" method="POST">
```

```
    <label for="username">Username:</label>
```

```
    <input type="text" id="username" name="username" required>
```

```
    <br>
```

```
    <label for="password">Password:</label>
```

```
    <input type="password" id="password" name="password" required>
```

```
    <br>
```

```
    <button type="submit">Login</button>
```

```
  </form>
```

```
</body>
```

```
</html>
```

## Output:



A screenshot of a web browser window. The address bar shows 'localhost:8080/Renuka/authenticate.html'. The page content includes a large heading 'Login', followed by a 'Username:' label and an empty text input field, a 'Password:' label and an empty text input field, and a 'Login' button at the bottom.

localhost:8080/Renuka/authenticate.html

# Login

Username:

Password:

Login



A screenshot of a web browser window. The address bar shows 'localhost:8080/Renuka/authenticate.html'. The page content includes a large heading 'Login', followed by a 'Username:' label and a text input field containing 'Renuka', a 'Password:' label and a text input field containing ten dots, and a 'Login' button at the bottom.

New

localhost:8080/Renuka/authenticate.html

# Login

Username:

Password:

Login



**Welcome, Renuka!**



**Invalid Username or Password!**