Department of Computer Science & Applications

A.S College, Khanna

Practical file (BCA-16-503)

(2024-2025)

5th semester

Submitted by: Submitted to:

Sukhdeep Singh(4606) **Dr. Rajan Manro**

Contents

❖ What is Java

Java is a high-level, object-oriented programming language designed for portability and performance. It enables developers to create applications for various platforms through the Java Virtual Machine (JVM). Known for its "write once, run anywhere" capability, it supports robust security and concurrency. Its versatility makes it suitable for various applications, including web development, mobile applications

➢ Features of Java

- Robustness
- Pure Object-oriented programming
- Platform independence
- Multithreading
- Security
- Simplicity
- High performance
- Scalability
- Compiled and Interpreted
- Portability
- Automatic Garbage Collection
- Rich Standard Library

❖ History of Java

Java was developed by Sun Microsystems, with the project initiated in 1991 and its first public release in 1995. Originally called Oak, it aimed to create software for interactive television but shifted focus to web applications due to the growing Internet. Java emphasized portability, security, and ease of use, leading to its famous "write once, run anywhere" capability via the Java Virtual Machine (JVM). Over the years, Java has evolved with numerous updates, becoming a dominant language for enterprise solutions, Android apps, and big data, supported by a strong developer community.

1991

The Birth of Java

A project called “Green Project” was initiated by James Gosling and Patrick Naughton at Sun Microsystems. The goal was to create a language for small electrical devices

1995

Copyright issue and Official Release

“Oak” was renamed to “Java” due to trademark issues. Java 1.0 was officially released by Sun Microsystems introducing the core features such as “Write Once, Run Everywhere”(WORA)

1996

Java Development Kit (JDK) 1.0

The first official Java Development Kit (JDK 1.0) was launched. It laid the foundation for Java applets

2009

Oracle acquires Sun Microsystems

Oracle Corporation acquired Sun Microsystems, becoming the new steward of Java

2014

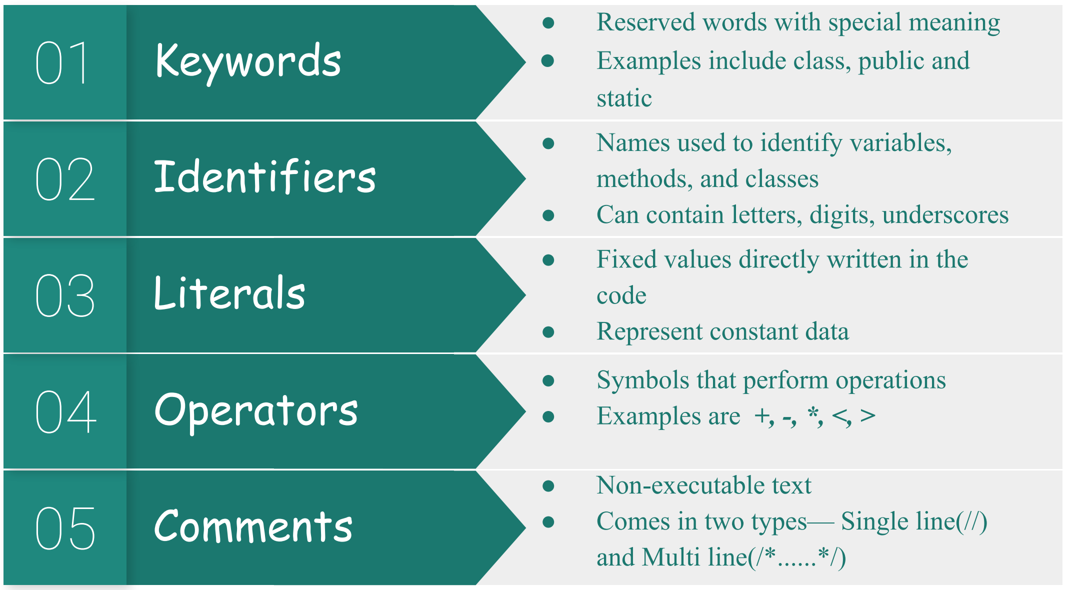
Project Valhalla

Project Valhalla aims to enhance Java by introducing value types, which behaves like primitives, but can represent more complex data structures

❖ Tokens

- ❑ Tokens are the smallest units of a program
- ❑ Tokens are recognized and processed by the compiler
- ❑ Tokens serve as building blocks of Java syntax, tokens convey meaning to the code
- ❑ Their structure is essential for effective programming

➢ Various types of Tokens are:



01	Keywords	• Reserved words with special meaning • Examples include class, public and static
02	Identifiers	• Names used to identify variables, methods, and classes • Can contain letters, digits, underscores
03	Literals	• Fixed values directly written in the code • Represent constant data
04	Operators	• Symbols that perform operations • Examples are +, -, *, <, >
05	Comments	• Non-executable text • Comes in two types— Single line(//) and Multi line(/*......*/)

➢ Why use Tokens in java

- ❑ Structured code
- ❑ Compiler understanding
- ❑ Error Detection
- ❑ Readability
- ❑ Modularity and Reusability

❖ Byte Code

Bytecode in Java is an intermediate representation of Java source code that is compiled by the Java compiler (javac). Instead of converting Java code directly into machine code for a specific hardware architecture, the Java compiler generates bytecode, which is platform-independent and can be executed on any machine that has the Java Virtual Machine (JVM). Java bytecode is stored in ***.class*** files, which contain the compiled binary representation of Java source code,

Program

Source code in Java outlines the program's logic, structure, and behavior through classes and methods. Well-organized code is crucial for documentation, maintenance, and understanding the application

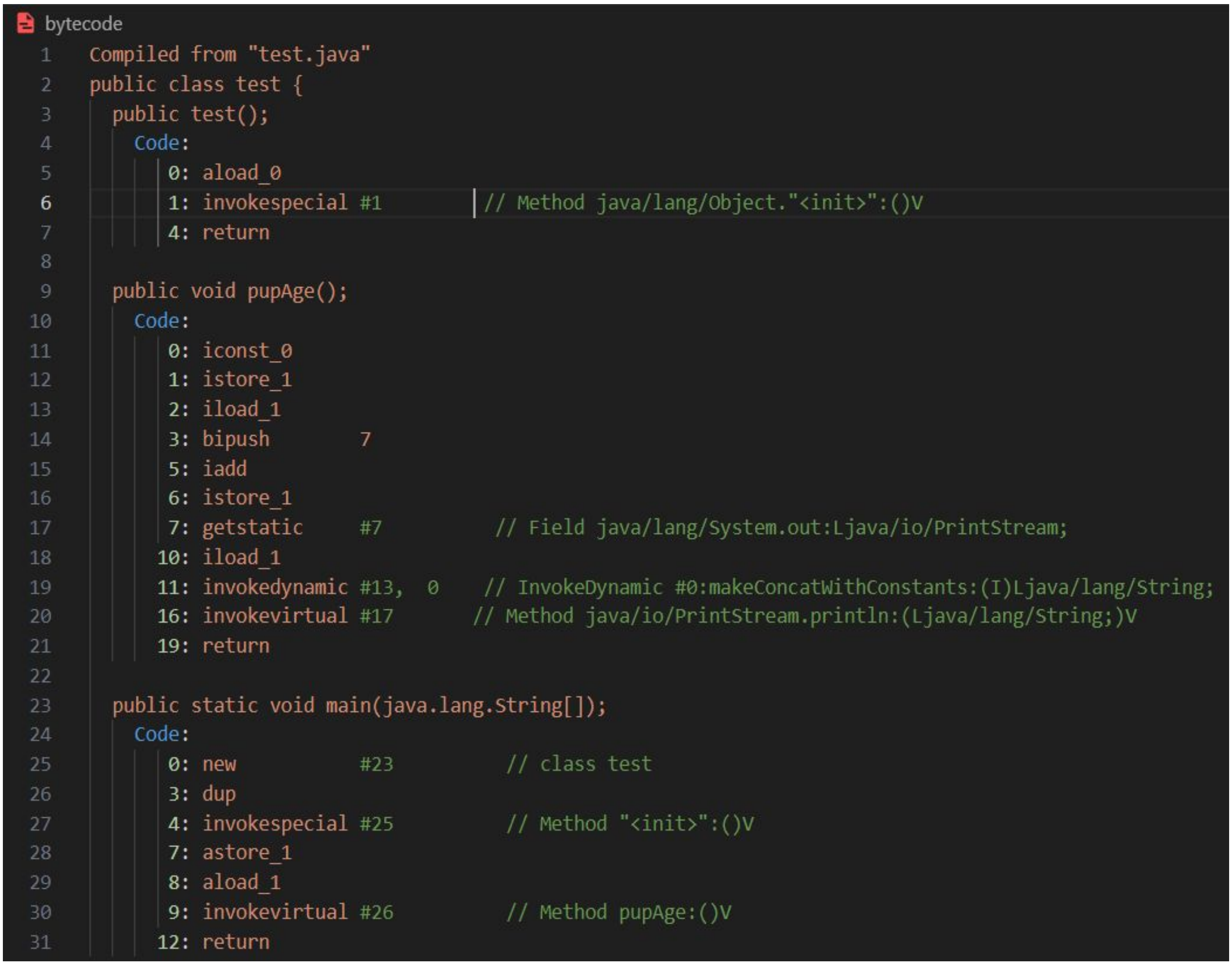
Compiler

The compiler translates Java source code into bytecode, enabling execution on any platform with a JVM. It checks for syntax errors and optimizes the code for performance during this process.

Byte Code

Bytecode is an intermediate, platform-independent representation of Java source code, generated by the compiler. It is executed by the Java Virtual Machine (JVM), allowing portability across different systems..

What Bytecode looks like


bytecode

```
Compiled from "test.java"
public class test {
  public test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void pupAge();
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush        7
       5: iadd
       6: istore_1
       7: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
      10: iload_1
      11: invokedynamic #13,  0             // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
      16: invokevirtual #17                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      19: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #23                 // class test
       3: dup
       4: invokespecial #25                 // Method "<init>":()V
       7: astore_1
       8: aload_1
       9: invokevirtual #26                 // Method pupAge:()V
      12: return
```

❖ Constructors

Constructors in Java is a terminology used to construct something in our programs. A constructor in java is a ***special method*** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

➢ Characteristics of Constructors

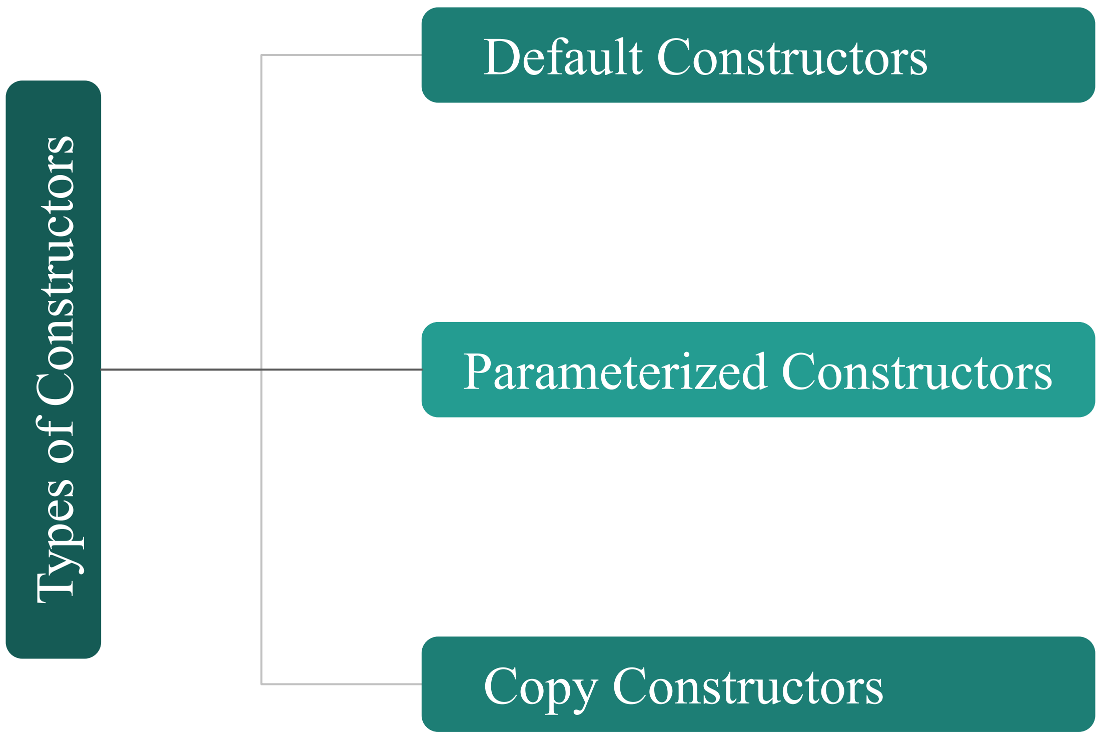
- ❏ Same name as the class they belong to
- ❏ No return type, not even void
- ❏ Allow overloading (multiple constructors with different parameters
- ❏ Can invoke superclass constructors using **super()**
- ❏ Can include access modifiers (public, private, etc.) to control visibility

➢ Difference between Java Constructors and Java Methods

- ★ Constructors initialize objects; methods perform actions or operations on objects
- ★ Constructors share the same name as the class; methods have their own distinct names

- ★ Constructors do not have a return type, not even void; methods must have a return type
- ★ Constructors are called when an object is created; methods are called on existing objects
- ★ Both can be overloaded, but constructors cannot be overridden

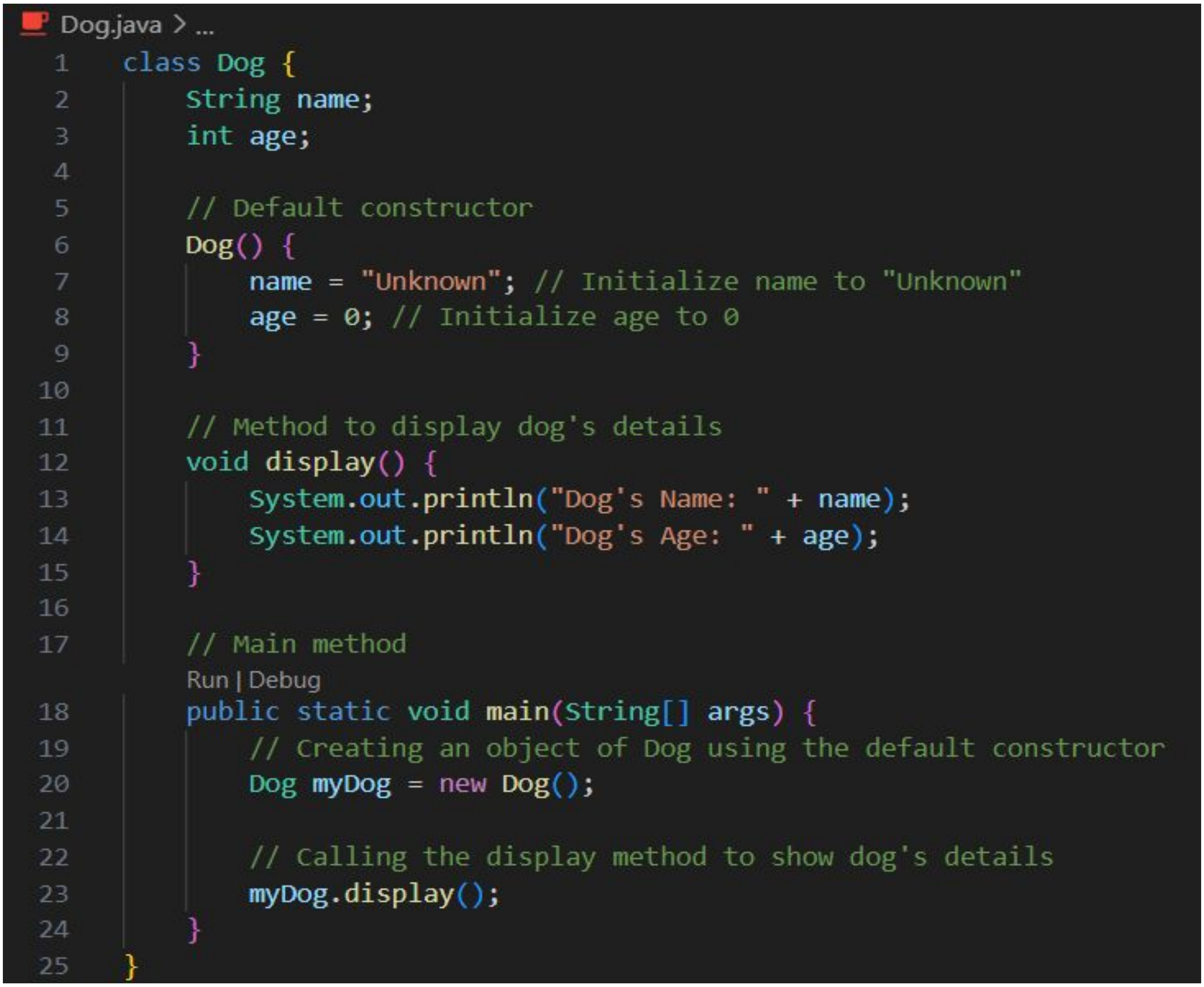
➢ Various Types:



1) <u>Default Constructors</u>

- ❑ The default constructor has no parameters
- ❑ If no constructors are explicitly defined in a class, Java automatically provides default constructor

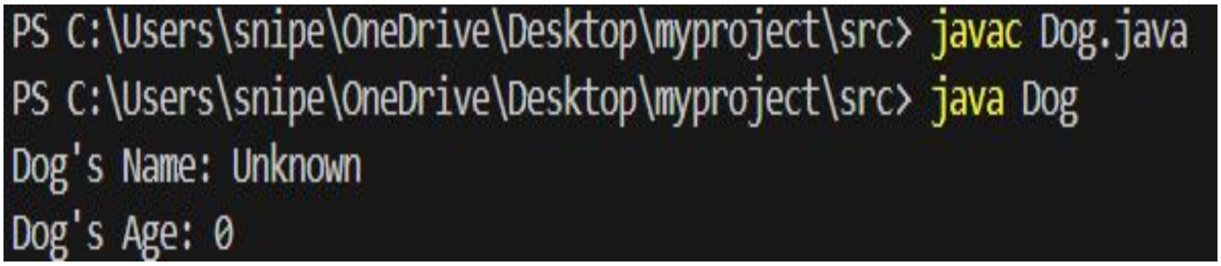
# Example code of Default Constructor



Dog.java > ...

```java
class Dog {
    String name;
    int age;

    // Default constructor
    Dog() {
        name = "Unknown"; // Initialize name to "Unknown"
        age = 0; // Initialize age to 0
    }

    // Method to display dog's details
    void display() {
        System.out.println("Dog's Name: " + name);
        System.out.println("Dog's Age: " + age);
    }

    // Main method
    Run | Debug
    public static void main(String[] args) {
        // Creating an object of Dog using the default constructor
        Dog myDog = new Dog();

        // Calling the display method to show dog's details
        myDog.display();
    }
}
```

➔ Output

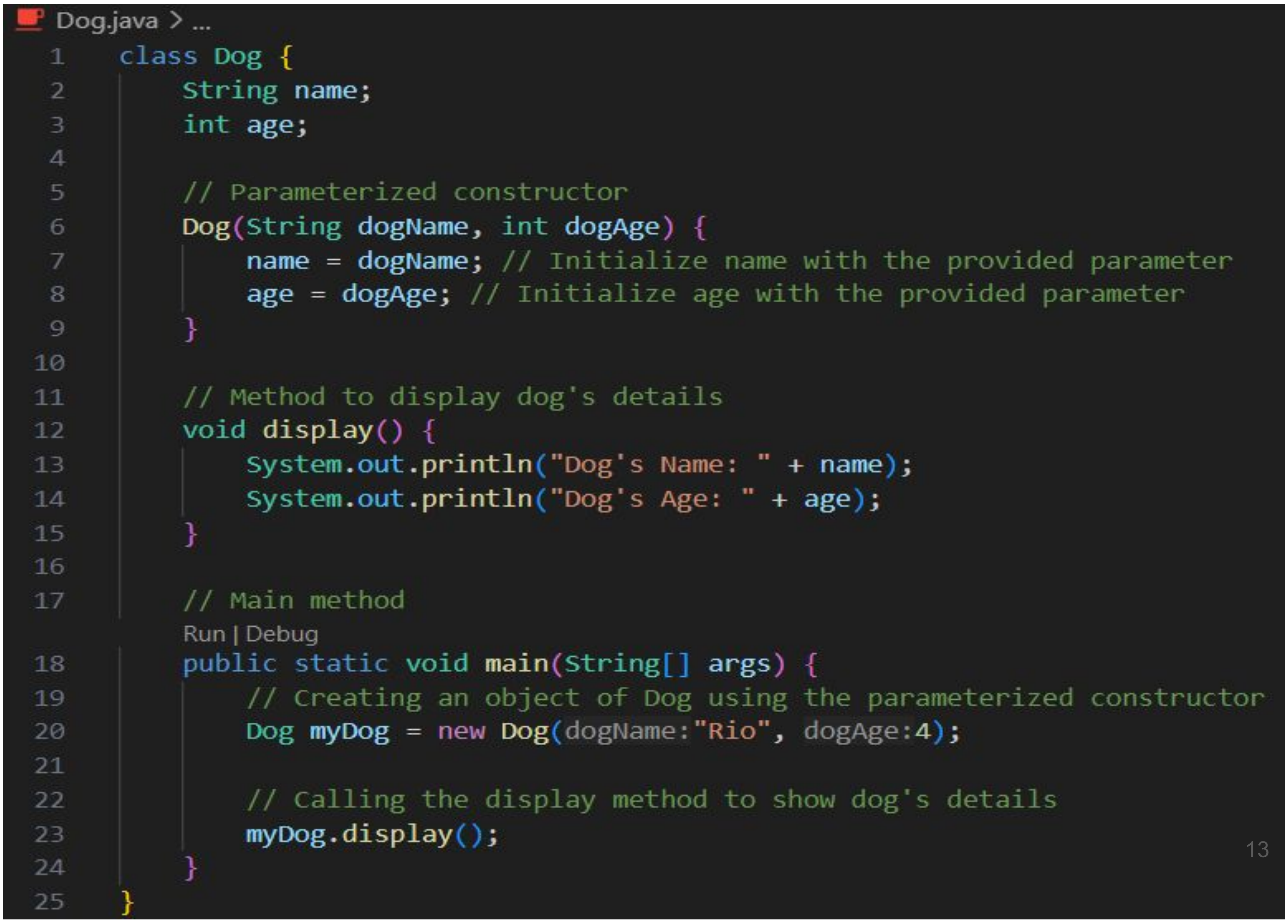


```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> javac Dog.java
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java Dog
Dog's Name: Unknown
Dog's Age: 0
```

2) Parameterized Constructor

- ❑ A parameterized constructor takes arguments to initialize an object with specific values
- ❑ It allows you to set initial values for fields during object creation
- ❑ It does not initialize fields with default values
- ❑ When we create an object using a parameterized constructor, we must pass the appropriate arguments

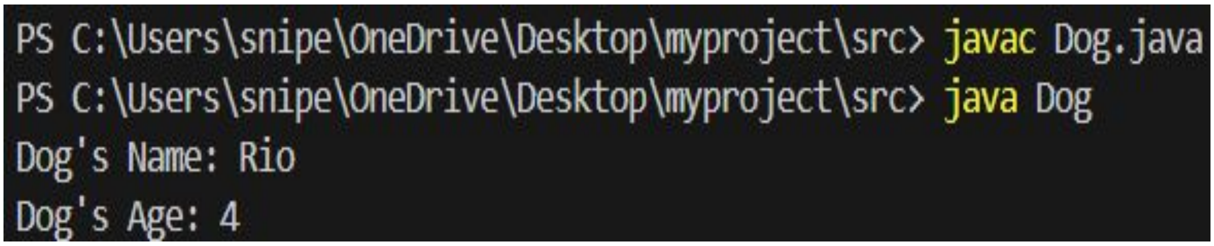
# Example code of Parameterized Constructor



Dog.java > ...

```java
class Dog {
    String name;
    int age;

    // Parameterized constructor
    Dog(String dogName, int dogAge) {
        name = dogName; // Initialize name with the provided parameter
        age = dogAge; // Initialize age with the provided parameter
    }

    // Method to display dog's details
    void display() {
        System.out.println("Dog's Name: " + name);
        System.out.println("Dog's Age: " + age);
    }

    // Main method
    Run | Debug
    public static void main(String[] args) {
        // Creating an object of Dog using the parameterized constructor
        Dog myDog = new Dog(dogName:"Rio", dogAge:4);

        // Calling the display method to show dog's details
        myDog.display();
    }
}
```

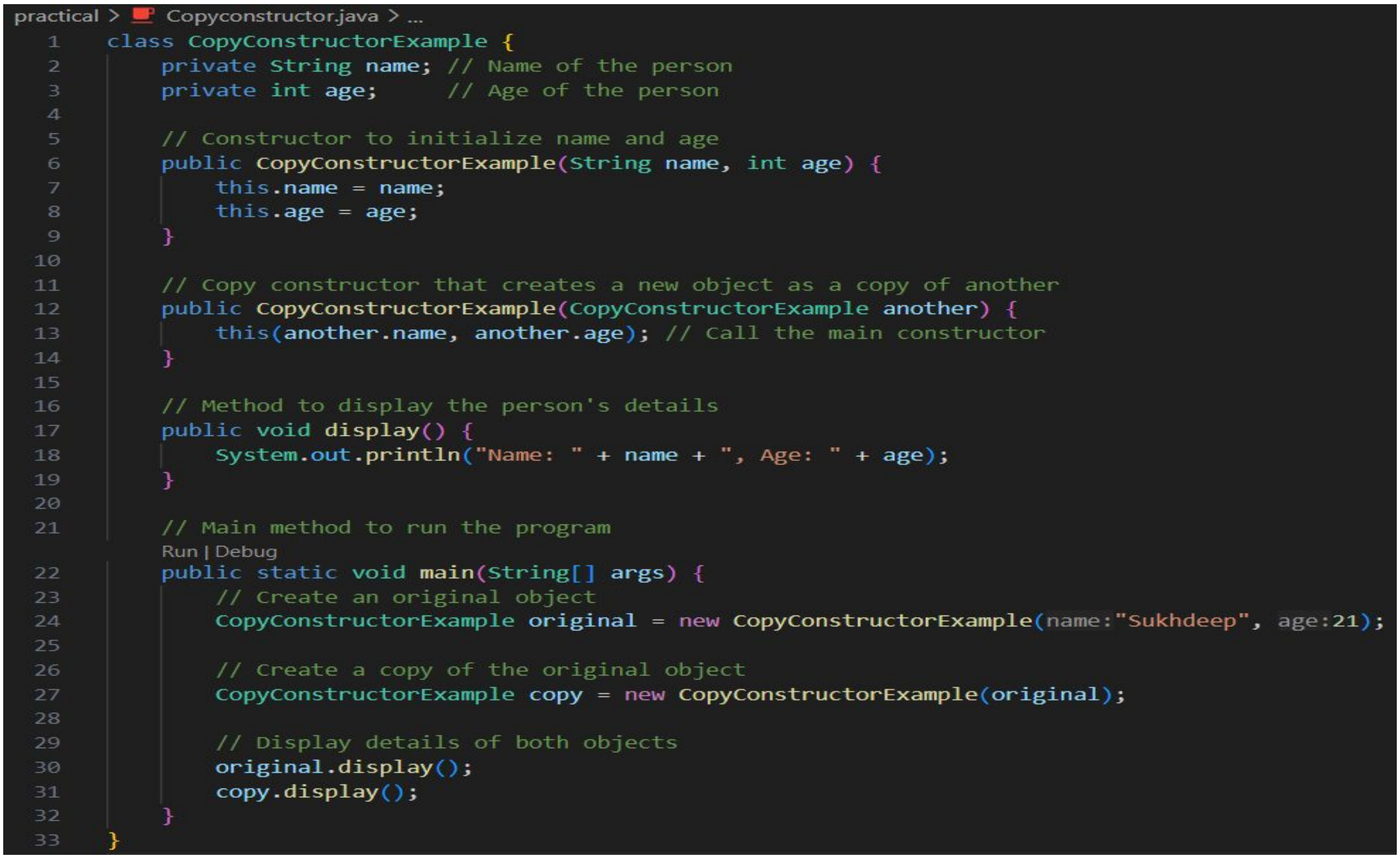
➔ **Output**



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> javac Dog.java
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java Dog
Dog's Name: Rio
Dog's Age: 4
```

3) <u>*Copy Constructor*</u>

- ❑ Create a new object as a copy of an existing object
- ❑ Typically defined as a constructor that takes an object of the same class as a parameter
- ❑ Can be overloaded with different parameters for different types of copying
- ❑ Not built-in; must be manually defined in the class



practical > Copyconstructor.java > ...

```java
class CopyConstructorExample {
    private String name; // Name of the person
    private int age;     // Age of the person

    // Constructor to initialize name and age
    public CopyConstructorExample(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy constructor that creates a new object as a copy of another
    public CopyConstructorExample(CopyConstructorExample another) {
        this(another.name, another.age); // Call the main constructor
    }

    // Method to display the person's details
    public void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }

    // Main method to run the program
    Run | Debug
    public static void main(String[] args) {
        // Create an original object
        CopyConstructorExample original = new CopyConstructorExample(name:"Sukhdeep", age:21);

        // Create a copy of the original object
        CopyConstructorExample copy = new CopyConstructorExample(original);

        // Display details of both objects
        original.display();
        copy.display();
    }
}
```

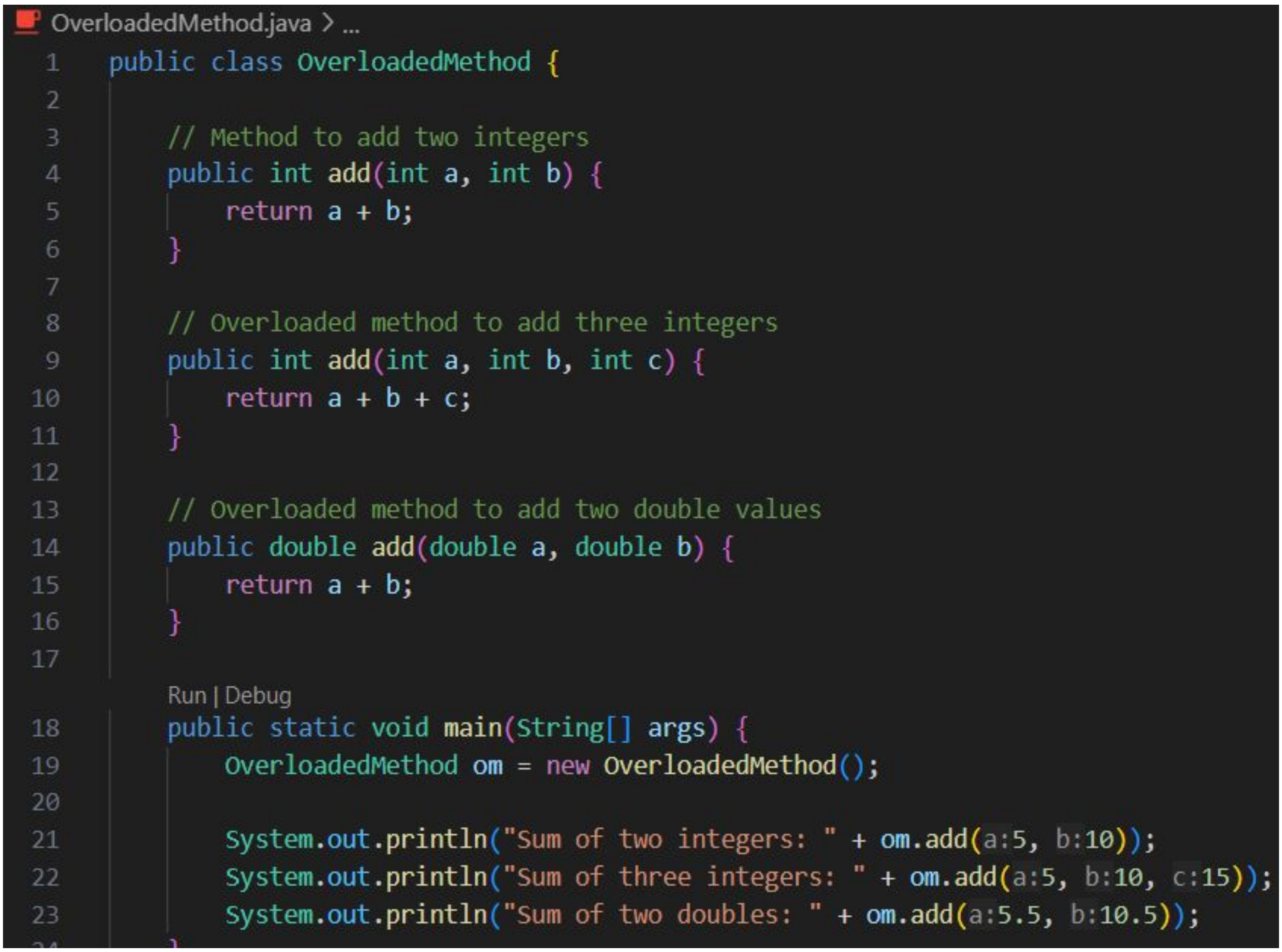
❖ Method Overloading

Method overloading in Java refers to the capability of a class to have multiple methods with the same name but different parameter lists, which can include varying types, numbers, or order of parameters. This feature displays ***runtime polymorphism***, as the method to be executed is determined at runtime based on the arguments passed.

➢ Why to Overload a Method

- ❏ Facilitates cleaner and more organized code structure
- ❏ Allows methods to perform similar tasks with varying logic based on input types
- ❏ Can improve performance by reducing the number of method names in a class
- ❏ Makes it easier for developers to understand the functionality
- ❏ Supports flexible method calls in different situations
- ❏ Helps to create more user-friendly APIs
- ❏ Keeps the code organized and tidy
- ❏ Improves user experience by offering multiple ways to interact with the same functionality
- ❏ It allows methods to evolve over time by adding new functionality without breaking backward compatibility.

# Example code of Method Overloading

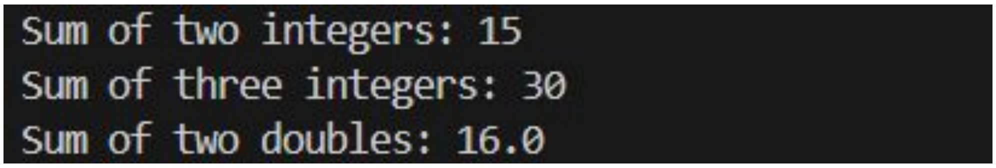


OverloadedMethod.java > ...

Run | Debug

```java
public class OverloadedMethod {

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

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two double values
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        OverloadedMethod om = new OverloadedMethod();

        System.out.println("Sum of two integers: " + om.add(a:5, b:10));
        System.out.println("Sum of three integers: " + om.add(a:5, b:10, c:15));
        System.out.println("Sum of two doubles: " + om.add(a:5.5, b:10.5));
```

➔ Output



```
Sum of two integers: 15
Sum of three integers: 30
Sum of two doubles: 16.0
```

1) Inheritance

Inheritance in Java is a mechanism where one class (subclass)inherits the properties and behaviours from another class(superclass). It allows hierarchical classification and promotes code reusability by enabling a class to acquire the functionality of another class

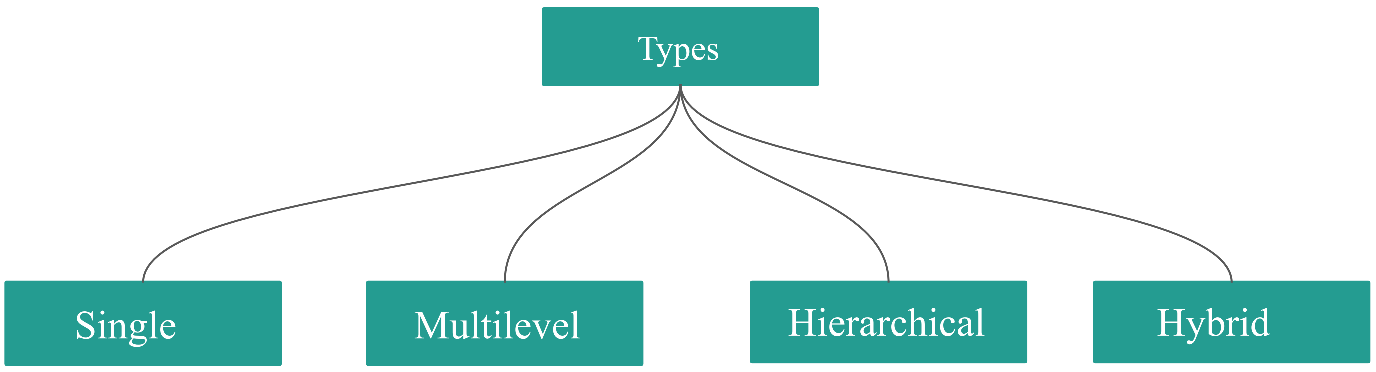
❑ Allows a class to inherit fields and methods from another class, promoting code reuse

❑ The class that inherits is called the subclass, and the class it inherits from is called superclass

❑ Inheritance allows access to protected members of the superclass

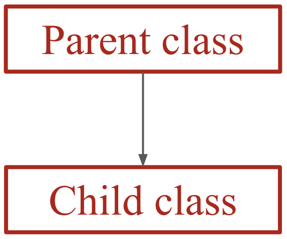
❑ A subclass can override methods of the superclass to provide specific behaviour

➢ Types of Inheritance

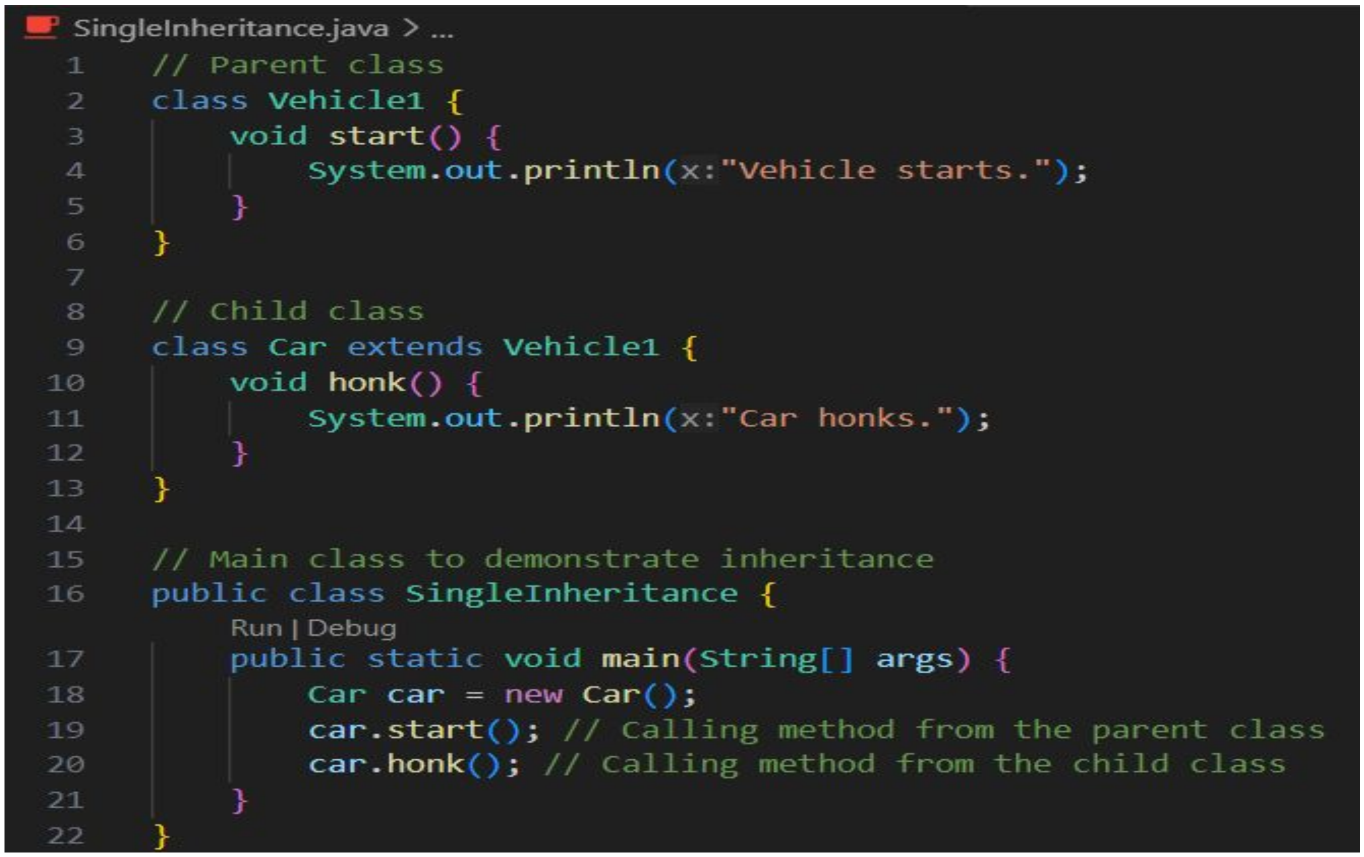


1) Single Inheritance

- ❑ Single inheritance allows a class (child class) to inherit from one other class (parent class)
- ❑ The child class can access public and protected members (methods and variables) of the parent class
- ❑ A child class can override methods from the parent class to provide specific implementations

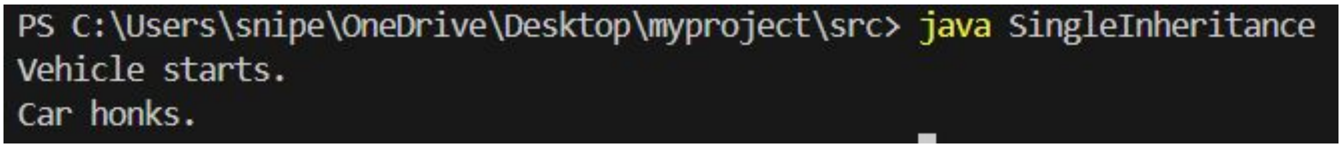


# Example code of Single Inheritance



```java
// Parent class
class Vehicle1 {
    void start() {
        System.out.println(x:"Vehicle starts.");
    }
}

// Child class
class Car extends Vehicle1 {
    void honk() {
        System.out.println(x:"Car honks.");
    }
}

// Main class to demonstrate inheritance
public class SingleInheritance {
    public static void main(String[] args) {
        Car car = new Car();
        car.start(); // Calling method from the parent class
        car.honk(); // Calling method from the child class
    }
}
```

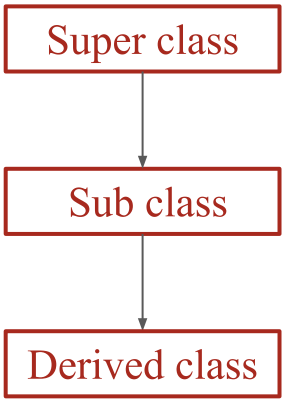
➔ **Output**



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java SingleInheritance
Vehicle starts.
Car honks.
```

2) <u>Multilevel Inheritance</u>

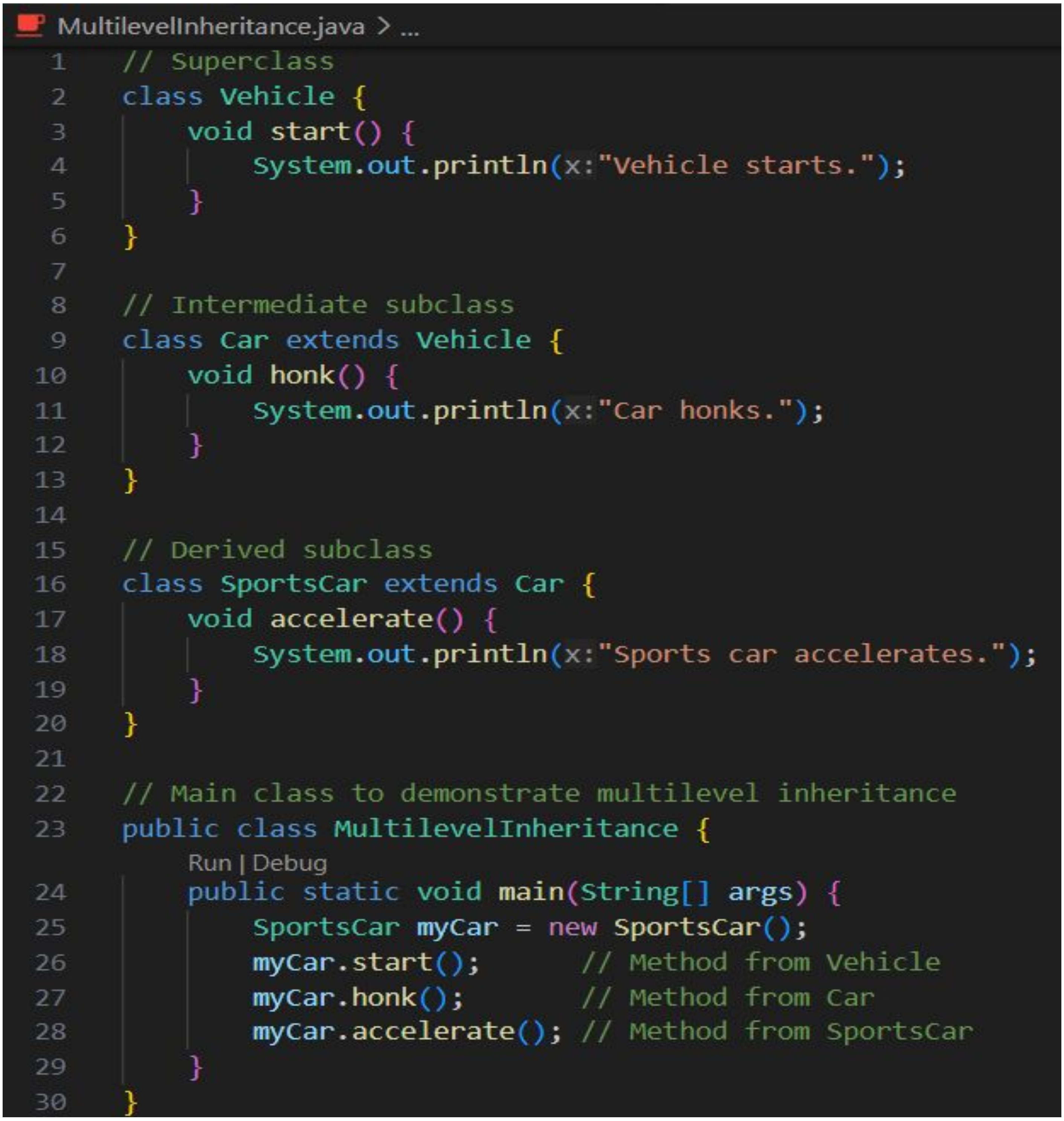
- ❑ In multilevel inheritance, a class inherits from a superclass, which itself is a subclass of another class
- ❑ It establishes a hierarchy of classes, allowing multiple levels of inheritance
- ❑ Each subclass can inherit features from its immediate superclass as well as from its ancestor classes
- ❑ Constructors of each superclass are called in order from the top of the hierarchy down to the subclass



- ❑ Example Structure:

  Class A (super class) → Class B extends A (sub class) → Class C extends B (derived class)

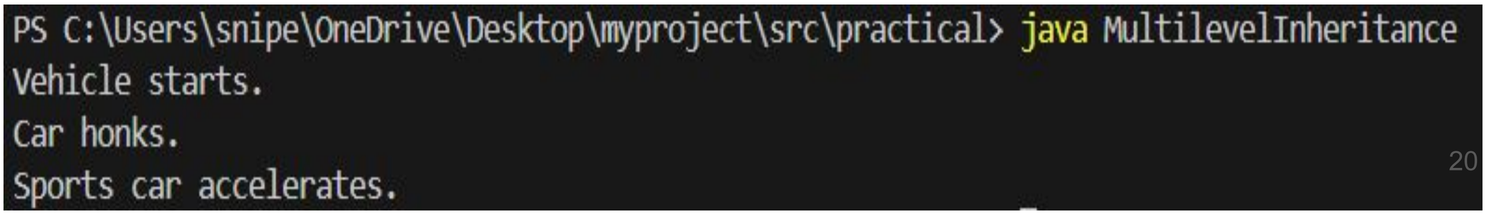
# Example code of Multilevel Inheritance



MultilevelInheritance.java > ...

```java
// Superclass
class Vehicle {
    void start() {
        System.out.println(x:"Vehicle starts.");
    }
}

// Intermediate subclass
class Car extends Vehicle {
    void honk() {
        System.out.println(x:"Car honks.");
    }
}

// Derived subclass
class SportsCar extends Car {
    void accelerate() {
        System.out.println(x:"Sports car accelerates.");
    }
}

// Main class to demonstrate multilevel inheritance
public class MultilevelInheritance {
    Run | Debug
    public static void main(String[] args) {
        SportsCar myCar = new SportsCar();
        myCar.start();      // Method from Vehicle
        myCar.honk();       // Method from Car
        myCar.accelerate(); // Method from SportsCar
    }
}
```

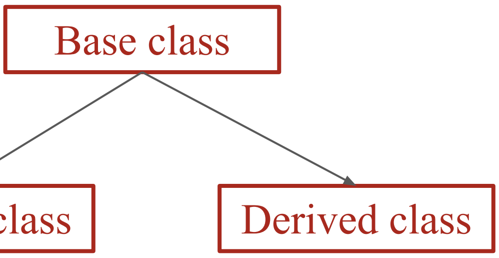
➔ Output



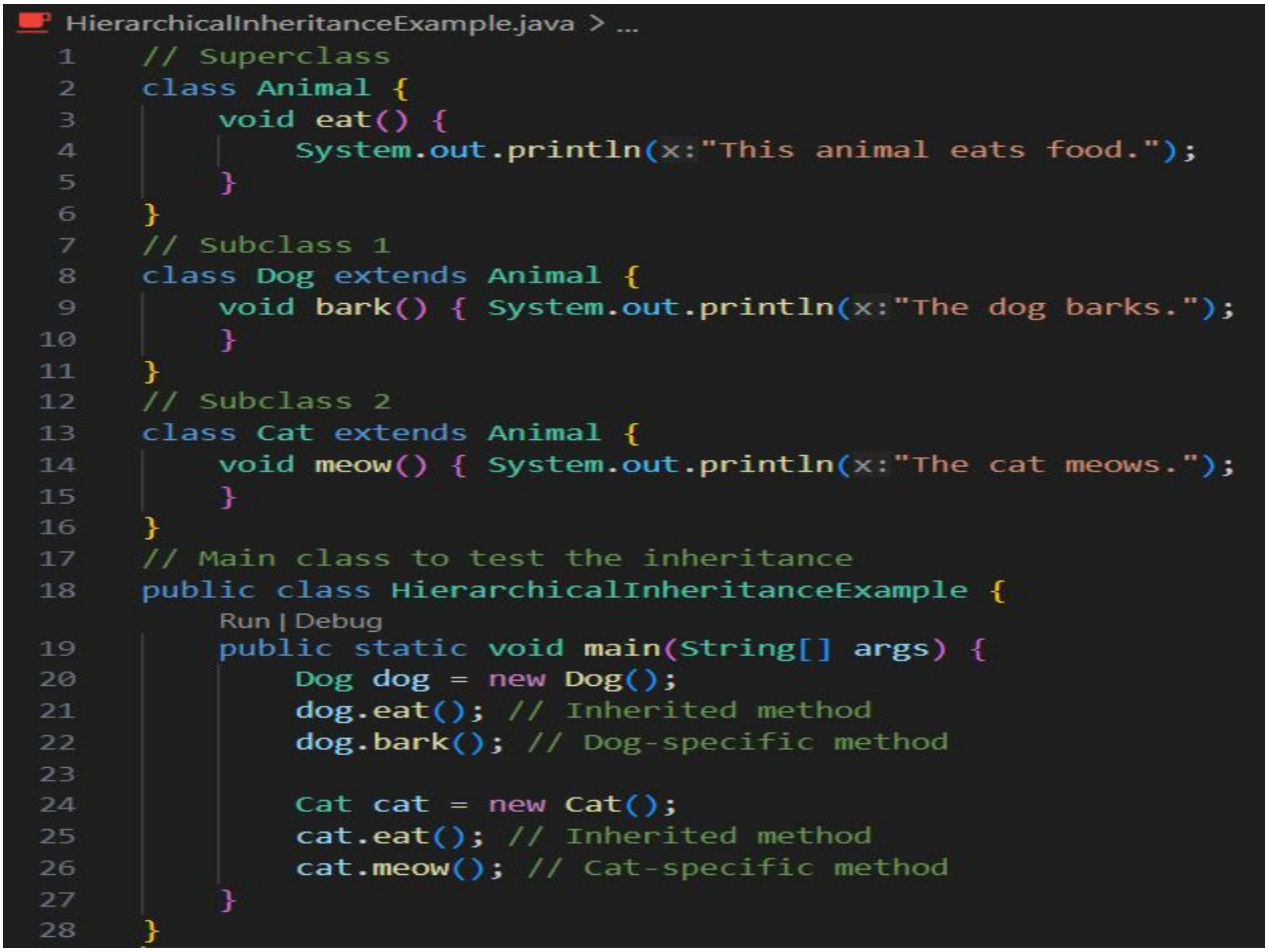
```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src\practical> java MultilevelInheritance
Vehicle starts.
Car honks.
Sports car accelerates.
```

3) Hierarchical Inheritance

- ❑ Hierarchical inheritance occurs when multiple subclasses inherit from a single superclass
- ❑ It resembles a tree structure, where the superclass is the root, and subclasses branch off from it
- ❑ This inheritance type enables polymorphism, allowing objects of different subclasses to be treated as objects of the superclass

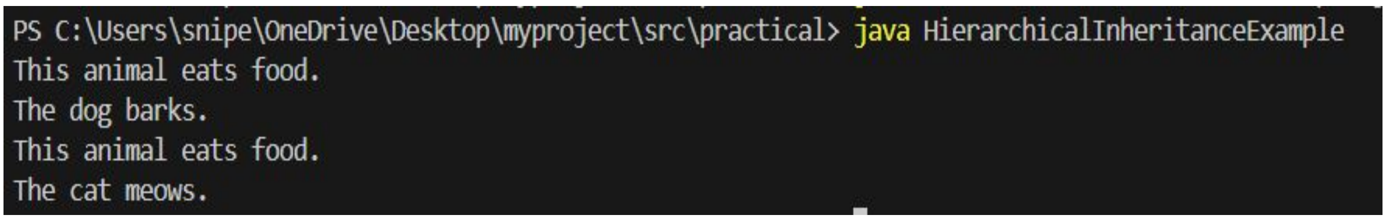


# *Example code of Hierarchical Inheritance*



```java
// Superclass
class Animal {
    void eat() {
        System.out.println(x:"This animal eats food.");
    }
}
// Subclass 1
class Dog extends Animal {
    void bark() { System.out.println(x:"The dog barks.");
    }
}
// Subclass 2
class Cat extends Animal {
    void meow() { System.out.println(x:"The cat meows.");
    }
}
// Main class to test the inheritance
public class HierarchicalInheritanceExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Inherited method
        dog.bark(); // Dog-specific method

        Cat cat = new Cat();
        cat.eat(); // Inherited method
        cat.meow(); // Cat-specific method
    }
}
```

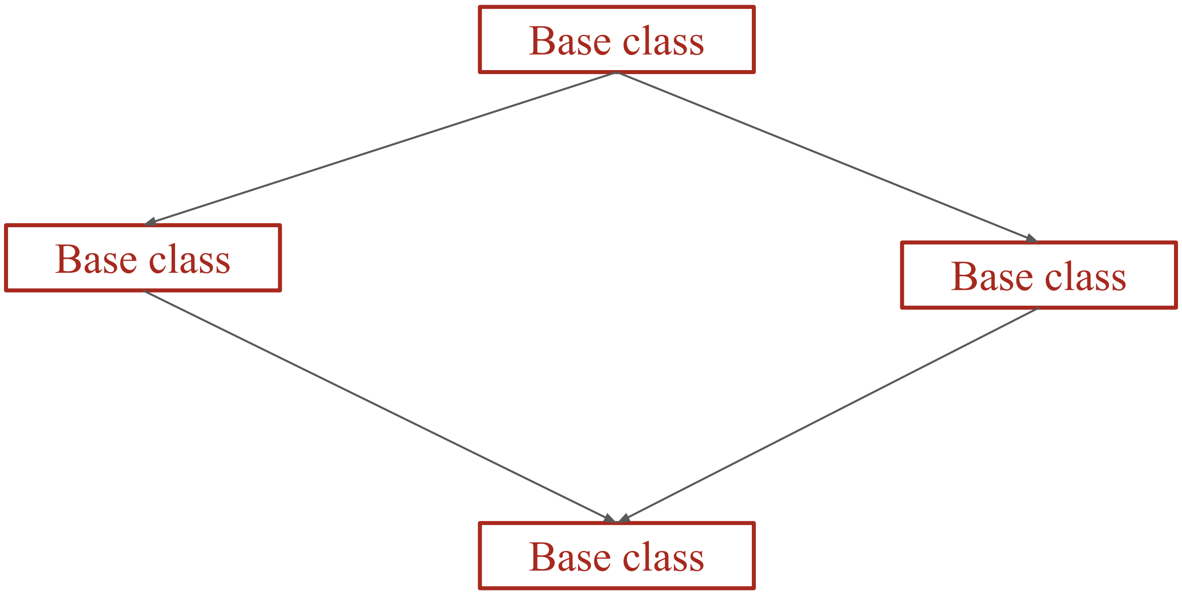
➔ **Output**



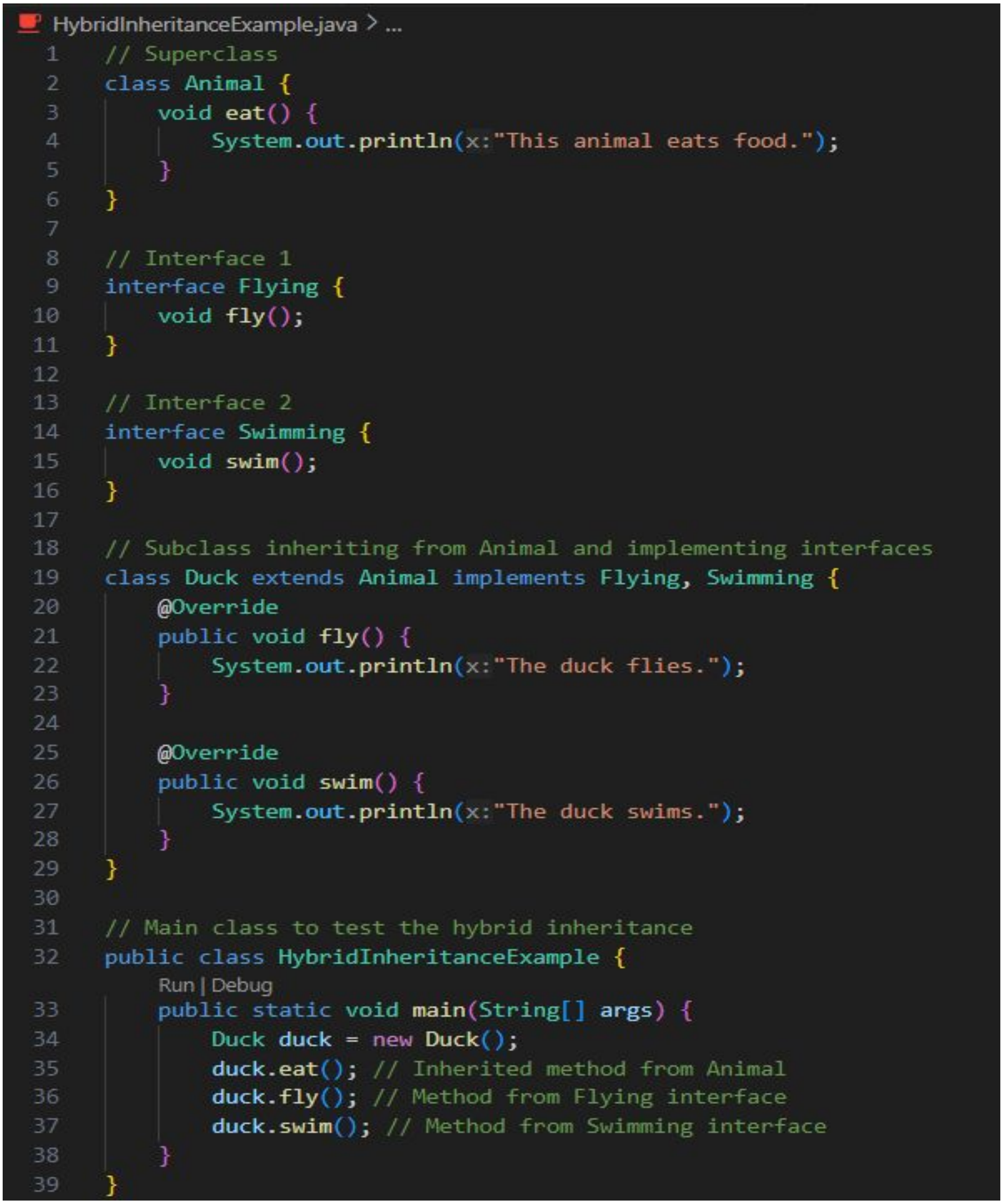
```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src\practical> java HierarchicalInheritanceExample
This animal eats food.
The dog barks.
This animal eats food.
The cat meows.
```

4) Hybrid Inheritance

- ❑ Hybrid inheritance combines multiple inheritance types, such as hierarchical and multi-level inheritance
- ❑ It allows a class to inherit from more than one superclass, enabling the use of features from multiple classes
- ❑ Java does not support hybrid inheritance directly through classes to avoid ambiguity
- ❑ This approach promotes code reusability and flexibility by allowing a class to adopt behaviors from multiple sources
- ❑ Method overriding and polymorphism are supported in hybrid inheritance scenarios



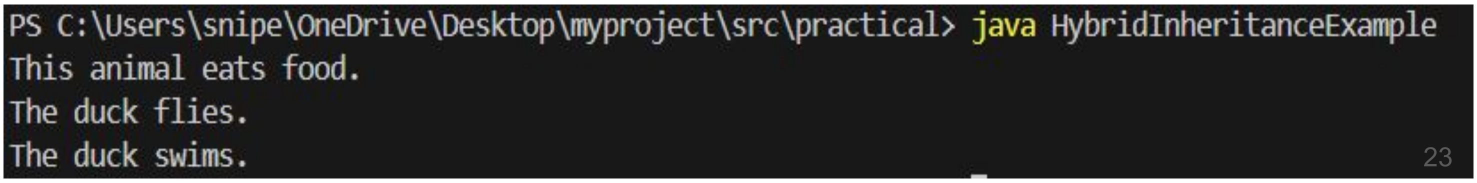
# Example code of Hybrid Inheritance



HybridInheritanceExample.java > ...

```java
// Superclass
class Animal {
    void eat() {
        System.out.println(x:"This animal eats food.");
    }
}

// Interface 1
interface Flying {
    void fly();
}

// Interface 2
interface Swimming {
    void swim();
}

// Subclass inheriting from Animal and implementing interfaces
class Duck extends Animal implements Flying, Swimming {
    @Override
    public void fly() {
        System.out.println(x:"The duck flies.");
    }

    @Override
    public void swim() {
        System.out.println(x:"The duck swims.");
    }
}

// Main class to test the hybrid inheritance
public class HybridInheritanceExample {
    Run | Debug
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.eat(); // Inherited method from Animal
        duck.fly(); // Method from Flying interface
        duck.swim(); // Method from Swimming interface
    }
}
```

➔ **Output**



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src\practical> java HybridInheritanceExample
This animal eats food.
The duck flies.
The duck swims.
```

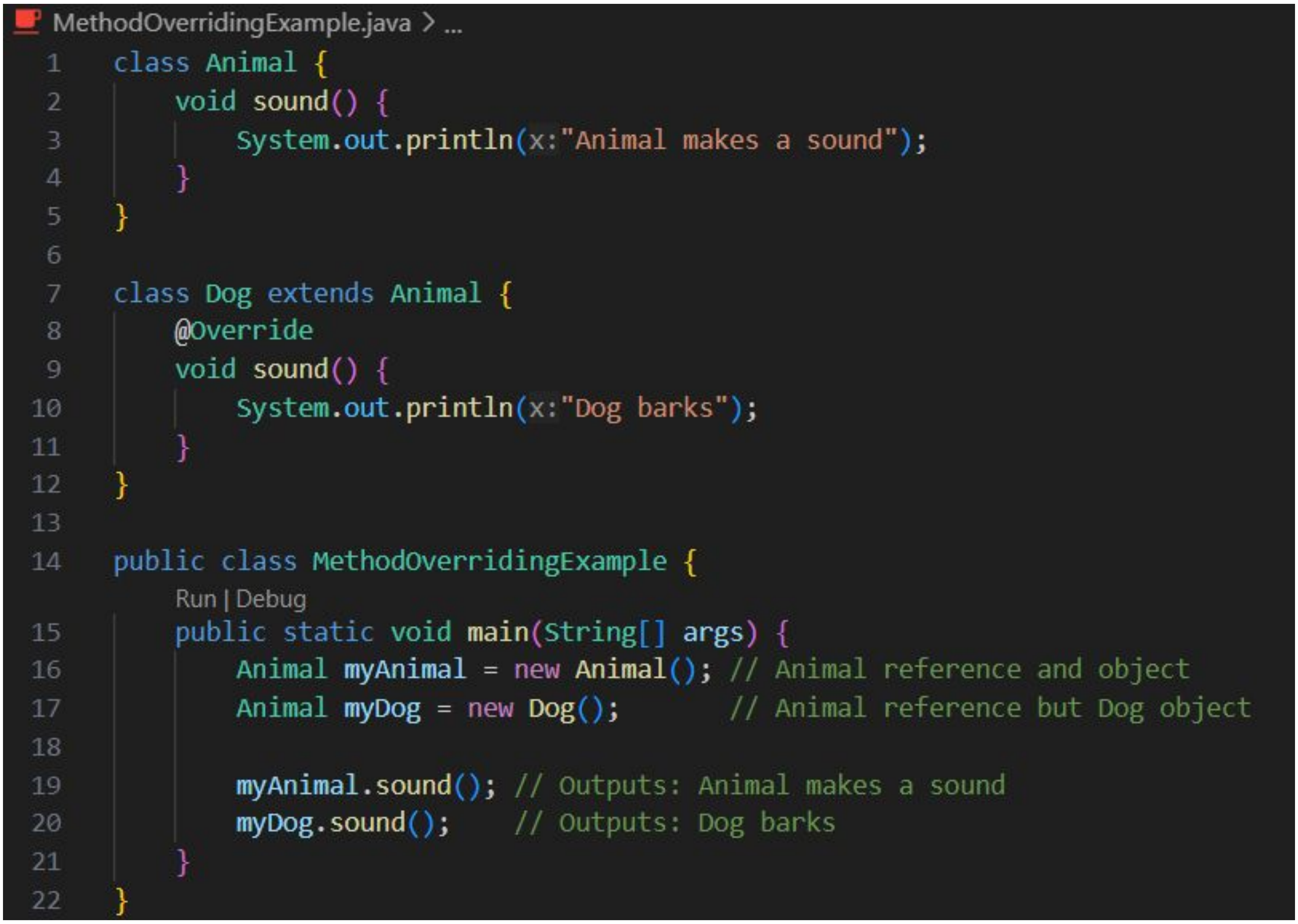
❖ Method Overriding

Method overriding in Java occurs when a subclass defines a method that has the same name, return type, and parameters as a method in its superclass. This allows the subclass to provide its own specific implementation of that method, effectively replacing the behavior defined in the superclass. Overriding enables ***dynamic method dispatch*** during runtime.

➢ Advantages of Method Overriding

- ❏ Enables runtime polymorphism, allowing methods to be determined at runtime
- ❏ Allows subclasses to provide specific implementations of inherited methods
- ❏ Promotes code reusability by extending existing classes
- ❏ Enhances flexibility and adaptability in code behavior
- ❏ Simplifies code maintenance by centralizing common functionality in the superclass
- ❏ Supports cleaner and more organized code structure
- ❏ Promotes a clear hierarchy in class relationships, enhancing understanding
- ❏ Provides opportunities for implementing custom logic in derived classes
- ❏ Overriding enables runtime behavior changes, as methods can be dynamically selected based on the actual object type.

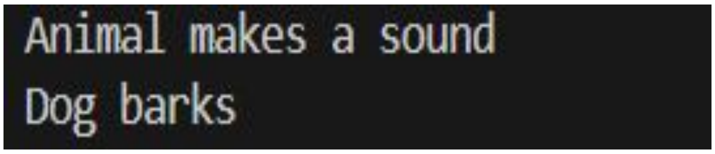
# Example code of Method Overriding



MethodOverridingExample.java > ...

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

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

public class MethodOverridingExample {
    Run | Debug
    public static void main(String[] args) {
        Animal myAnimal = new Animal(); // Animal reference and object
        Animal myDog = new Dog();       // Animal reference but Dog object

        myAnimal.sound(); // Outputs: Animal makes a sound
        myDog.sound();    // Outputs: Dog barks
    }
}
```

➔ **Output**



```
Animal makes a sound
Dog barks
```

❖ Recursion

➔ Definition

Recursion is a programming technique where a method calls itself to solve a problem

➔ Self-Invocation

The method executes by invoking itself with modified parameters

➔ Base Case

A condition that stops the recursive calls to prevent infinite loops

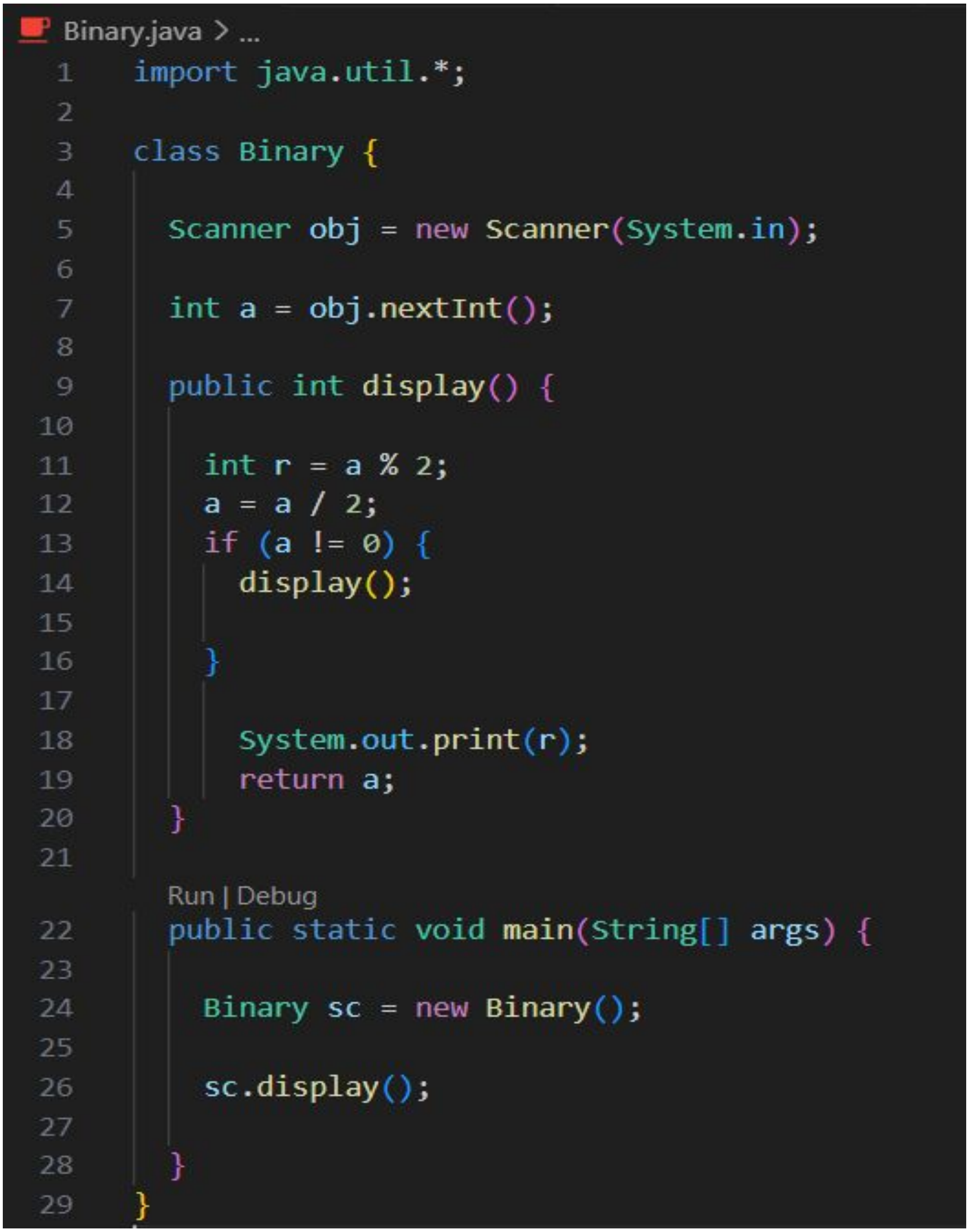
➔ Recursive Case

The scenario where the method continues to call itself to break down the problem

➔ Usage

Commonly used for tasks like calculating factorials, navigating trees or solving complex mathematical problems

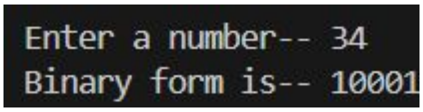
# Example code of Recursion



Binary.java > ...

```java
import java.util.*;

class Binary {

  Scanner obj = new Scanner(System.in);

  int a = obj.nextInt();

  public int display() {

    int r = a % 2;
    a = a / 2;
    if (a != 0) {
      display();

    }

      System.out.print(r);
      return a;
  }

  Run | Debug
  public static void main(String[] args) {

    Binary sc = new Binary();

    sc.display();

  }
}
```

➔ Output

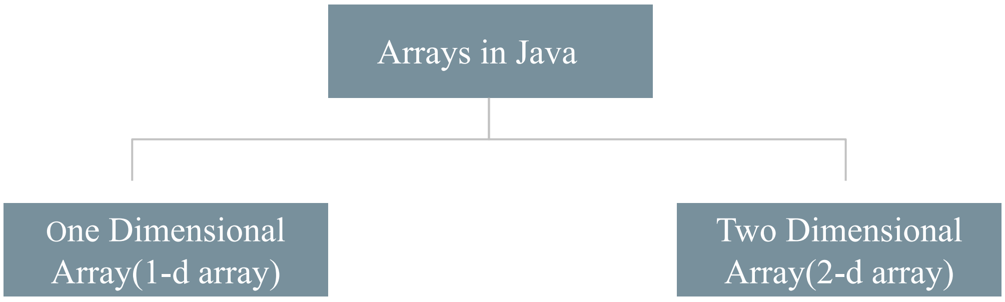


```
Enter a number-- 34
Binary form is-- 10001
```

❖ Arrays in Java

- An array is a data structure that stores a fixed-size sequence of elements of the same type
- Array elements are accessed using a zero-based index, starting from 0
- The size of an array is defined when it is created and cannot be changed afterward
- Arrays can only hold elements of the same data type (e.g., *int, String, float*
- Arrays allocate contiguous memory locations for storing elements
- Accessing elements in an array using an index is fast, with constant time complexity **O(1)**

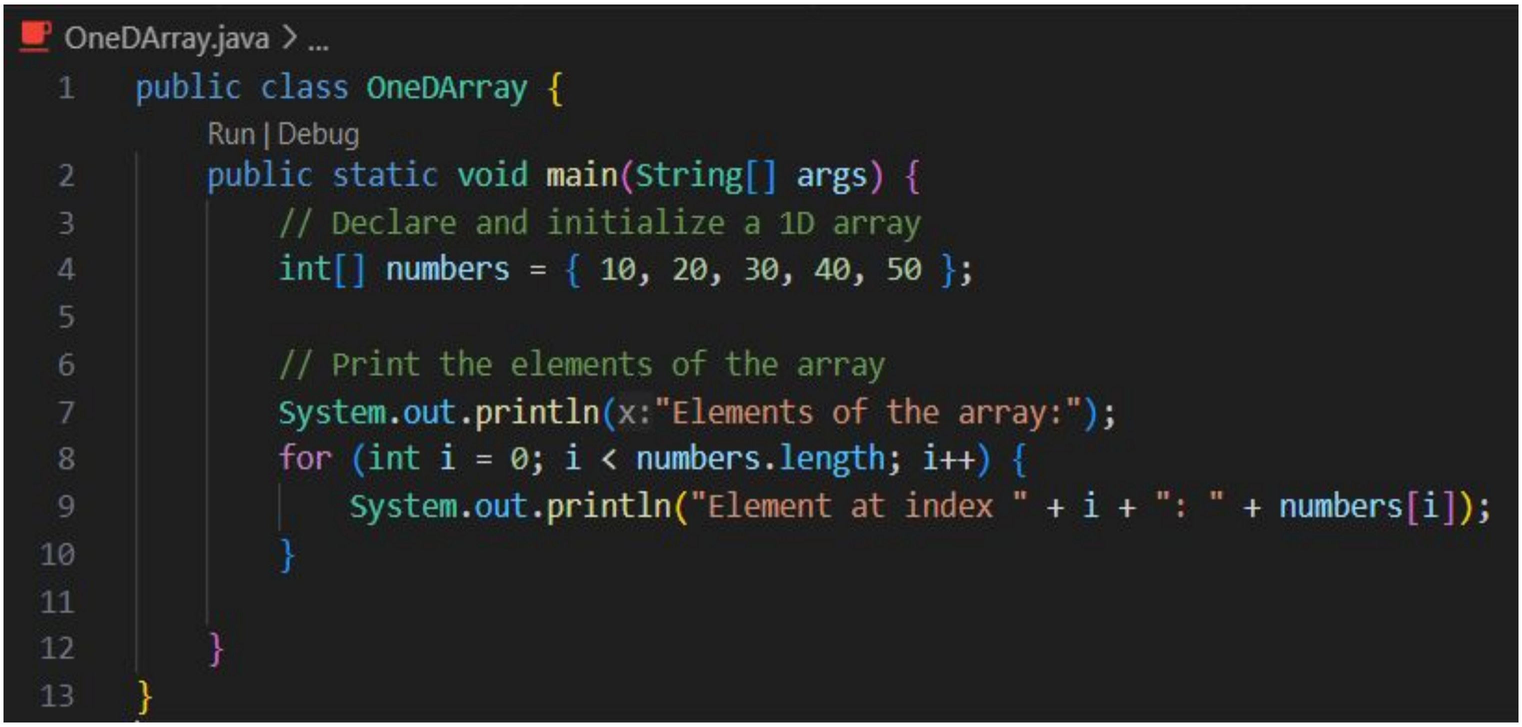
➢ Types of Array


Arrays in Java
One Dimensional Array(1-d array)
Two Dimensional Array(2-d array)

➢ *One Dimensional Array*

- ❑ A 1D array represents a list of elements stored in a single row
- ❑ The elements in 1D array can be accessed specifying the index in square brackets like ***array[index]***

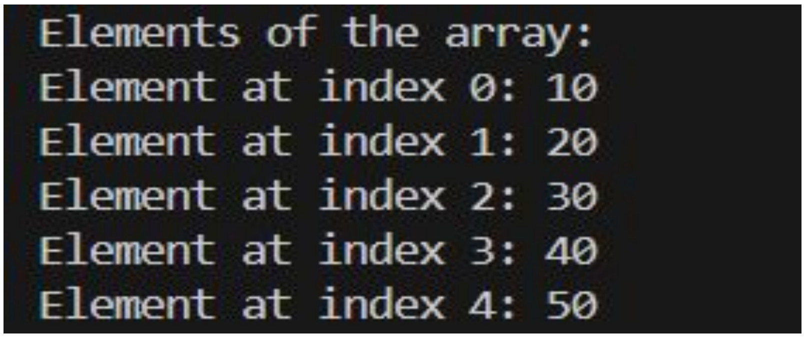
Example code of One Dimensional Array



OneDArray.java > ...

```java
public class OneDArray {
    Run | Debug
    public static void main(String[] args) {
        // Declare and initialize a 1D array
        int[] numbers = { 10, 20, 30, 40, 50 };

        // Print the elements of the array
        System.out.println(x:"Elements of the array:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Element at index " + i + ": " + numbers[i]);
        }

    }
}
```

➔ *Output*

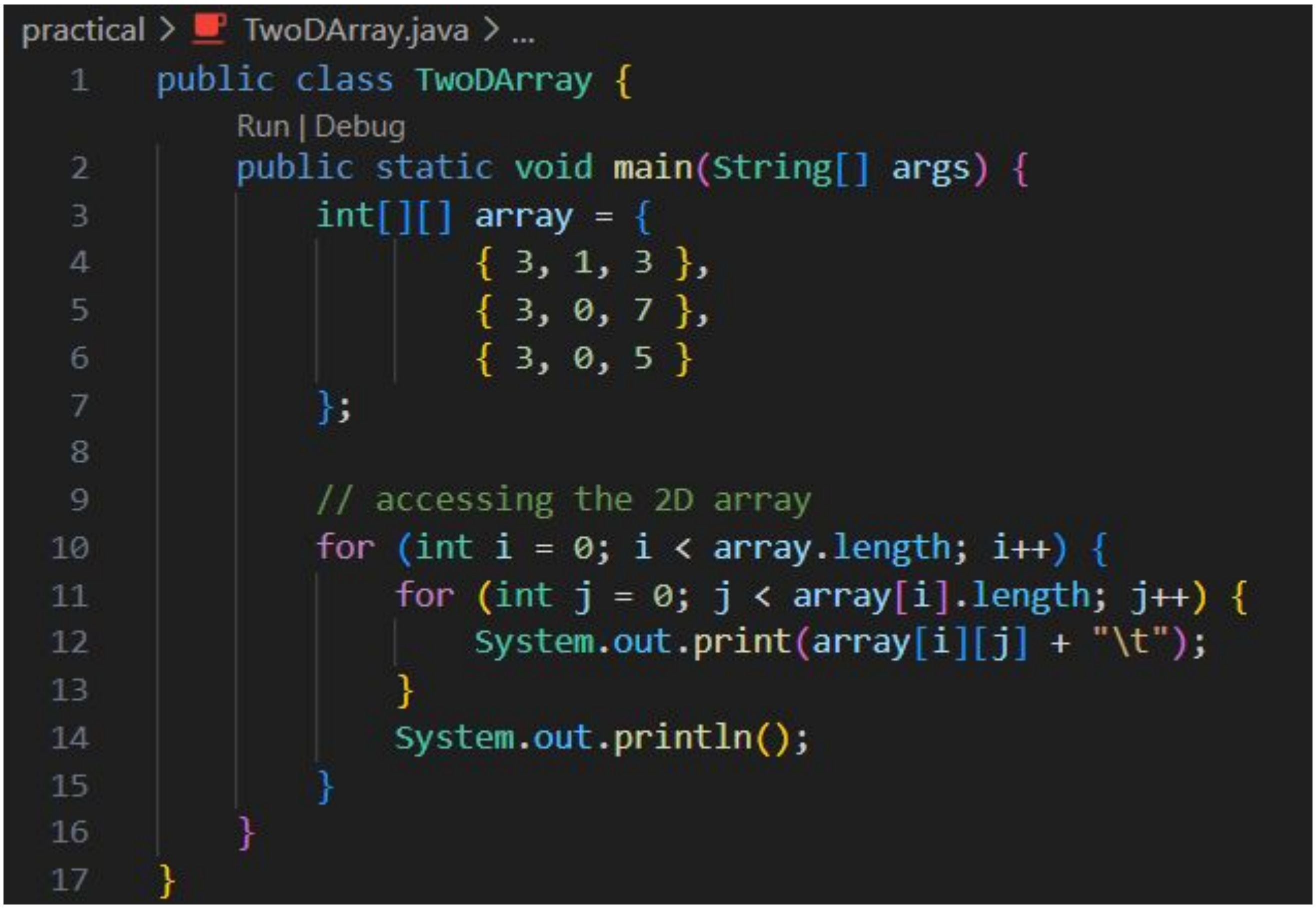


```
Elements of the array:
Element at index 0: 10
Element at index 1: 20
Element at index 2: 30
Element at index 3: 40
Element at index 4: 50
```

➢ Two Dimensional Array

- ❑ A 2D array in Java is an array of arrays, allowing data to be stored in a grid format, declared using syntax like ***int[][] arrayName = new int[rows][columns];***
- ❑ Elements are accessed using two indices, e.g., ***arrayName[row][column]***

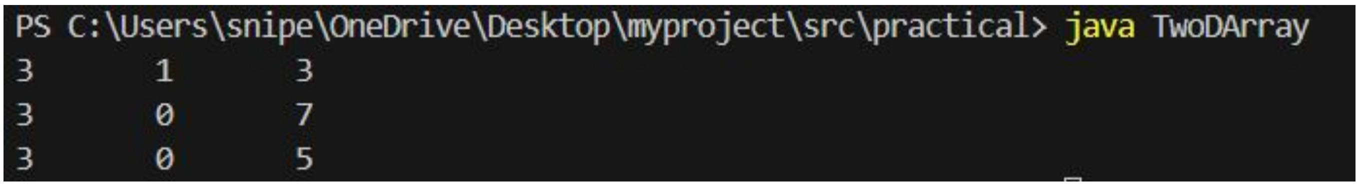
# Example code of 2D Array



practical > TwoDArray.java > ...

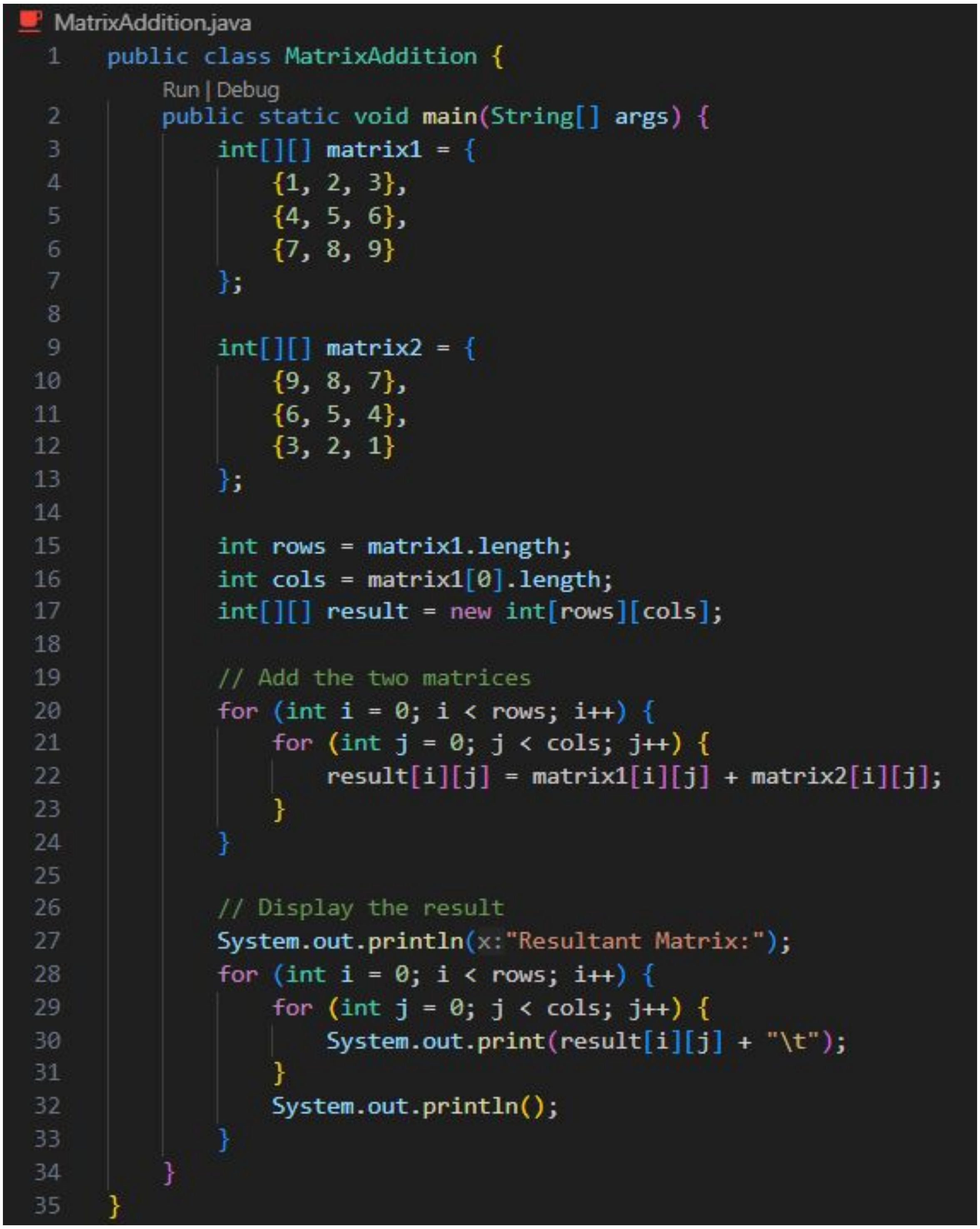
```java
public class TwoDArray {
    Run | Debug
    public static void main(String[] args) {
        int[][] array = {
                { 3, 1, 3 },
                { 3, 0, 7 },
                { 3, 0, 5 }
        };

        // accessing the 2D array
        for (int i = 0; i < array.length; i++) {
            for (int j = 0; j < array[i].length; j++) {
                System.out.print(array[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

➔ Output



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src\practical> java TwoDArray
3       1       3
3       0       7
3       0       5
```

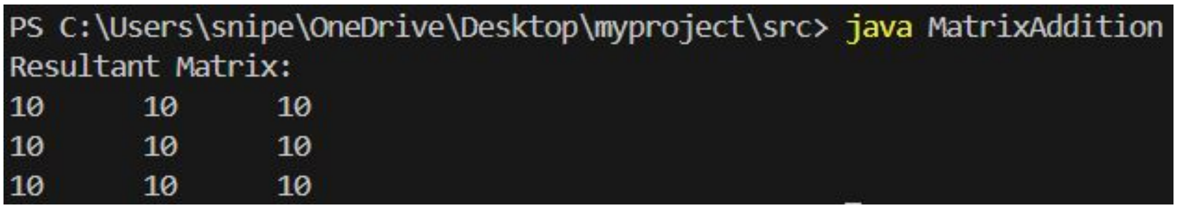
# Addition of Two Matrices



MatrixAddition.java

```java
public class MatrixAddition {
    Run | Debug
    public static void main(String[] args) {
        int[][] matrix1 = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int[][] matrix2 = {
            {9, 8, 7},
            {6, 5, 4},
            {3, 2, 1}
        };

        int rows = matrix1.length;
        int cols = matrix1[0].length;
        int[][] result = new int[rows][cols];

        // Add the two matrices
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }

        // Display the result
        System.out.println(x:"Resultant Matrix:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(result[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

➔ **Output**



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java MatrixAddition
Resultant Matrix:
10      10      10
10      10      10
10      10      10
```

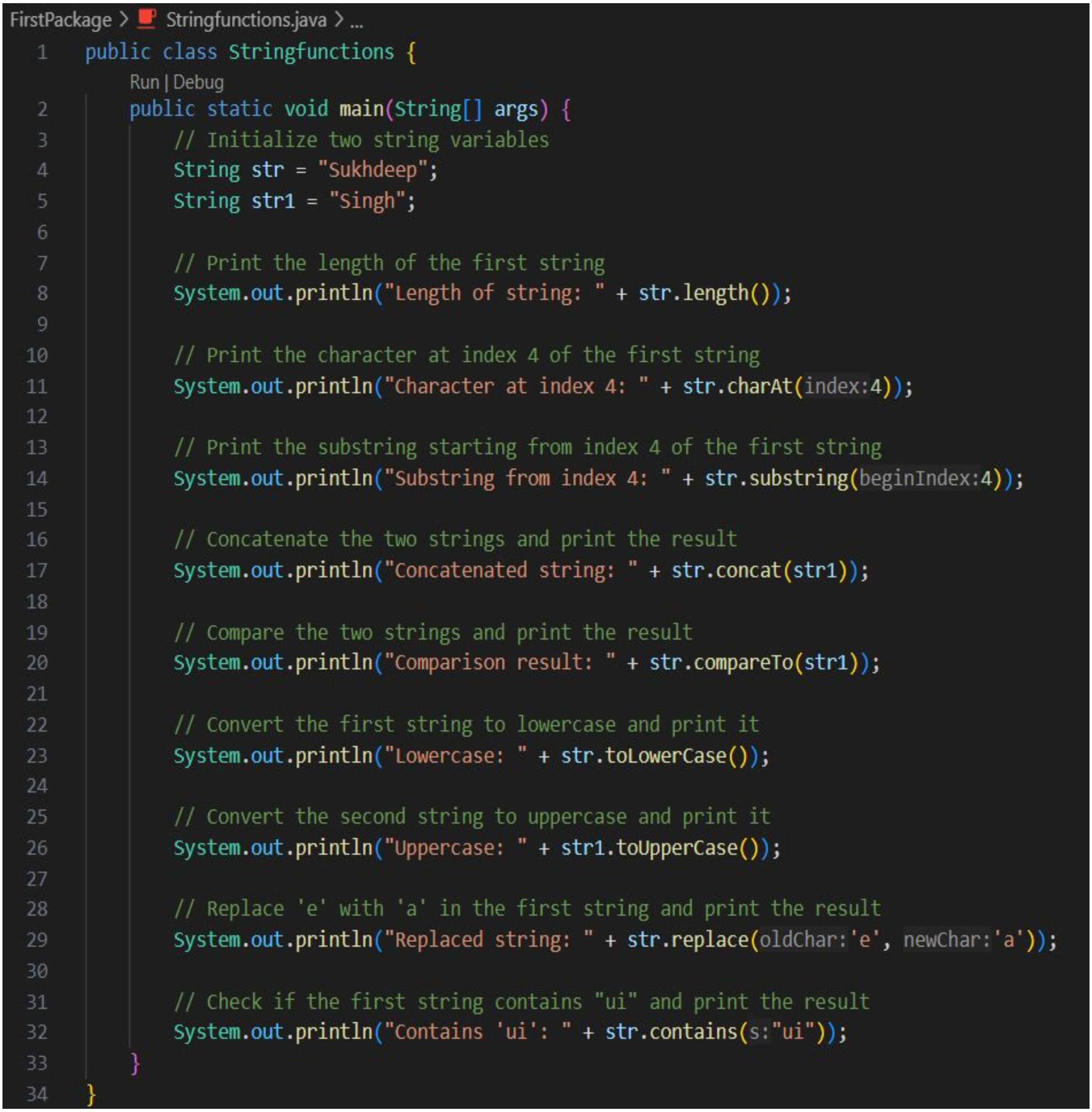
❖ String class

In Java, the ***`String`*** class is used to represent text as a sequence of characters. Strings are immutable, meaning once created, they cannot be altered. You can create strings using string literals or constructors, and various methods allow for operations like concatenation, searching, and comparison. This makes it a fundamental part of handling text in Java applications.

➢ Features of String class

- ❑ The String class is immutable; strings cannot be changed once created
- ❑ Strings can be created using string literals or the String constructor
- ❑ Java uses a string pool for efficient storage and comparison
- ❑ Concatenation can be done using the + operator or the concat() method.
- ❑ Provides methods for:
- Finding the length of a string (length())
- Extracting substrings (substring())
- Searching for characters or substrings (indexOf(), lastIndexOf())
- Comparing strings (equals(), compareTo()

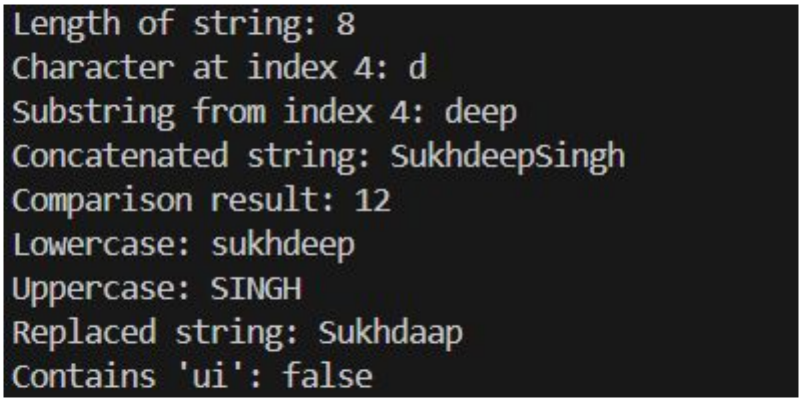
# Example code of String class and it's functions



FirstPackage > Stringfunctions.java > ...

```java
public class Stringfunctions {
    Run | Debug
    public static void main(String[] args) {
        // Initialize two string variables
        String str = "Sukhdeep";
        String str1 = "Singh";

        // Print the length of the first string
        System.out.println("Length of string: " + str.length());

        // Print the character at index 4 of the first string
        System.out.println("Character at index 4: " + str.charAt(index:4));

        // Print the substring starting from index 4 of the first string
        System.out.println("Substring from index 4: " + str.substring(beginIndex:4));

        // Concatenate the two strings and print the result
        System.out.println("Concatenated string: " + str.concat(str1));

        // Compare the two strings and print the result
        System.out.println("Comparison result: " + str.compareTo(str1));

        // Convert the first string to lowercase and print it
        System.out.println("Lowercase: " + str.toLowerCase());

        // Convert the second string to uppercase and print it
        System.out.println("Uppercase: " + str1.toUpperCase());

        // Replace 'e' with 'a' in the first string and print the result
        System.out.println("Replaced string: " + str.replace(oldChar:'e', newChar:'a'));

        // Check if the first string contains "ui" and print the result
        System.out.println("Contains 'ui': " + str.contains(s:"ui"));
    }
}
```

➔ **Output**



```
Length of string: 8
Character at index 4: d
Substring from index 4: deep
Concatenated string: SukhdeepSingh
Comparison result: 12
Lowercase: sukhdeep
Uppercase: SINGH
Replaced string: Sukhdaap
Contains 'ui': false
```

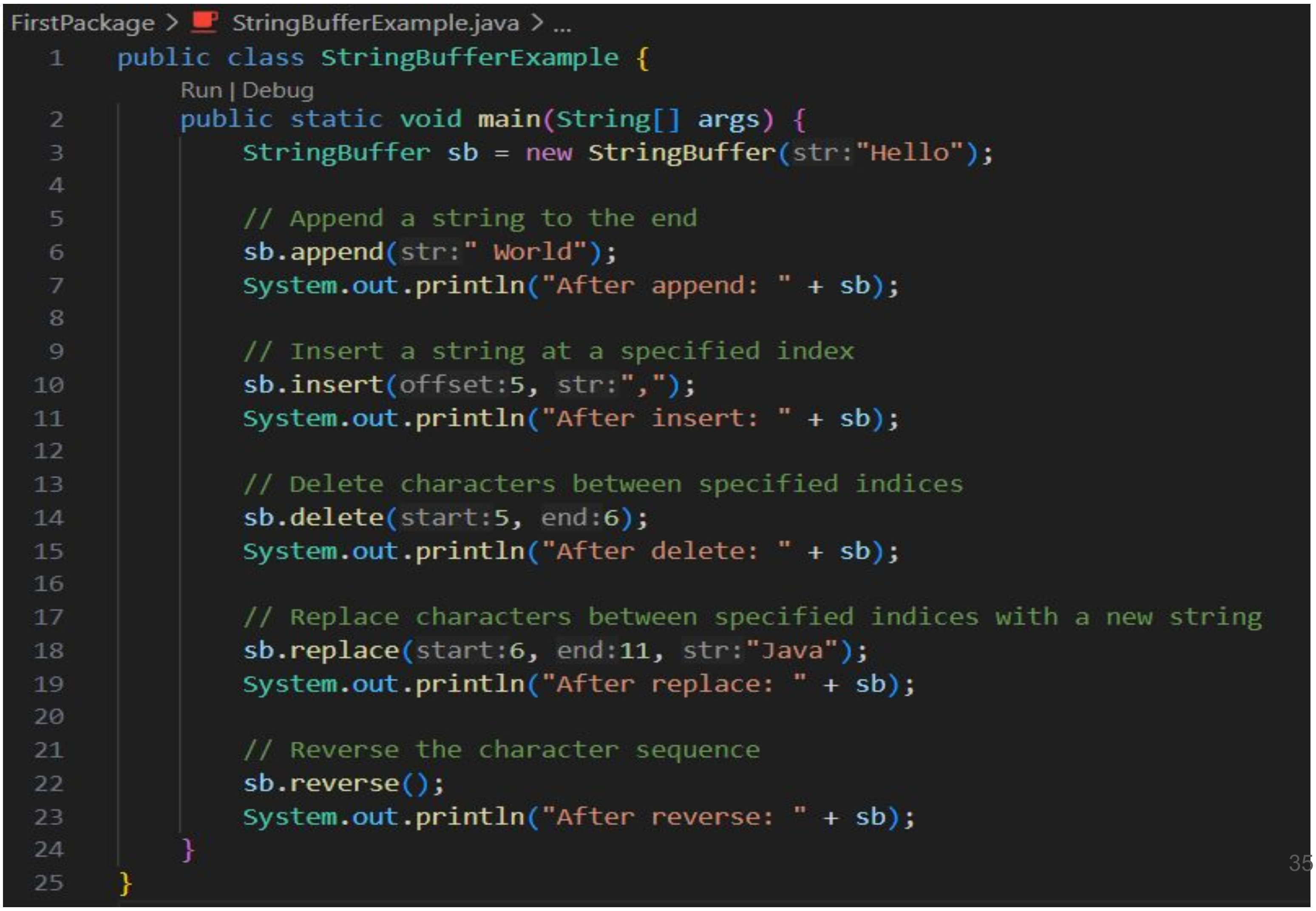
❖ Stringbuffer class

- ❑ The StringBuffer class is used for creating ***mutable*** sequences of characters
- ❑ Unlike String, StringBuffer allows modifications without creating new objects
- ❑ Provides methods for appending, inserting, deleting, and replacing characters
- ❑ Thread-safe and synchronized, making it suitable for concurrent access
- ❑ Ideal for scenarios where strings need to be built and manipulated efficiently
- ❑ More efficient for frequent string modifications compared to String

❑ Various functions of Stringbuffer class are:

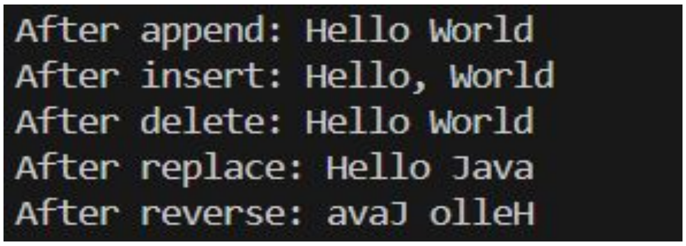
- **append(String str)**: Adds a string to the end.
- **insert(int offset, String str)**: Inserts a string at a specified index
- **delete(int start, int end)**: Removes characters between specified indices
- **replace(int start, int end, String str)**: Replaces characters between specified indices with a new string
- **reverse()**: Reverses the character sequence
- **capacity()**: Returns the allocated space for characters

# *Example code of Stringbuffer class and functions*



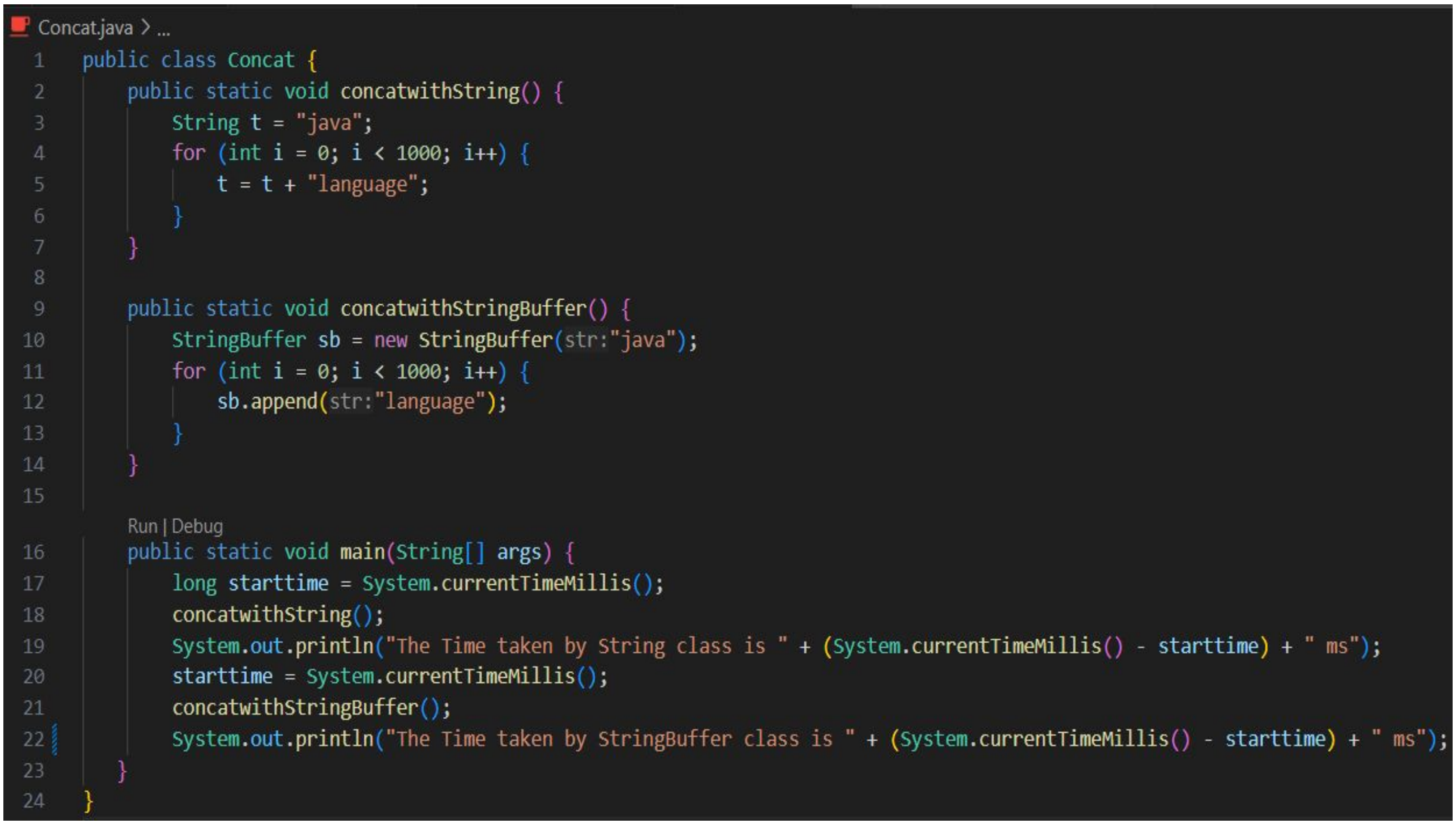
```java
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer(str:"Hello");

        // Append a string to the end
        sb.append(str:" World");
        System.out.println("After append: " + sb);

        // Insert a string at a specified index
        sb.insert(offset:5, str:",");
        System.out.println("After insert: " + sb);

        // Delete characters between specified indices
        sb.delete(start:5, end:6);
        System.out.println("After delete: " + sb);

        // Replace characters between specified indices with a new string
        sb.replace(start:6, end:11, str:"Java");
        System.out.println("After replace: " + sb);

        // Reverse the character sequence
        sb.reverse();
        System.out.println("After reverse: " + sb);
    }
}
```

➔ Output



```
After append: Hello World
After insert: Hello, World
After delete: Hello World
After replace: Hello Java
After reverse: avaJ olleH
```

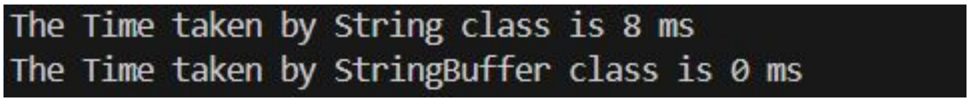
➢ Example code to show difference b/w String and StringBuffer class



Concat.java > ...

```java
public class Concat {
    public static void concatwithString() {
        String t = "java";
        for (int i = 0; i < 1000; i++) {
            t = t + "language";
        }
    }

    public static void concatwithStringBuffer() {
        StringBuffer sb = new StringBuffer(str:"java");
        for (int i = 0; i < 1000; i++) {
            sb.append(str:"language");
        }
    }

    Run | Debug
    public static void main(String[] args) {
        long starttime = System.currentTimeMillis();
        concatwithString();
        System.out.println("The Time taken by String class is " + (System.currentTimeMillis() - starttime) + " ms");
        starttime = System.currentTimeMillis();
        concatwithStringBuffer();
        System.out.println("The Time taken by StringBuffer class is " + (System.currentTimeMillis() - starttime) + " ms");
    }
}
```

➔ Output



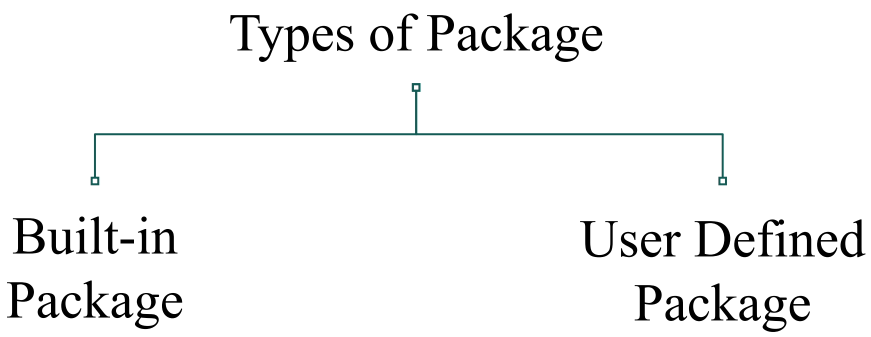
```
The Time taken by String class is 8 ms
The Time taken by StringBuffer class is 0 ms
```

➢ Comparison Chart between String and StringBuffer class

	String		StringBuffer	
01	Immutable; once created, its value cannot be changed	01 Mutability	Mutable; allows modification of its content	01
02	Less efficient for frequent modifications	02 Performance	More efficient for modifications	02
03	Used for fixed or rarely changed text	03 Usage	Used when there are multiple modifications to string data	03

❖ Packages

In Java, a package is a way to organize related classes and files. It helps avoid name conflicts, keeps the code clean, and makes it easier to manage large programs. Packages also control access to classes and their members, improving security and maintainability. Using packages allows developers to logically group components, making code more structured and readable. Packages are of two types:



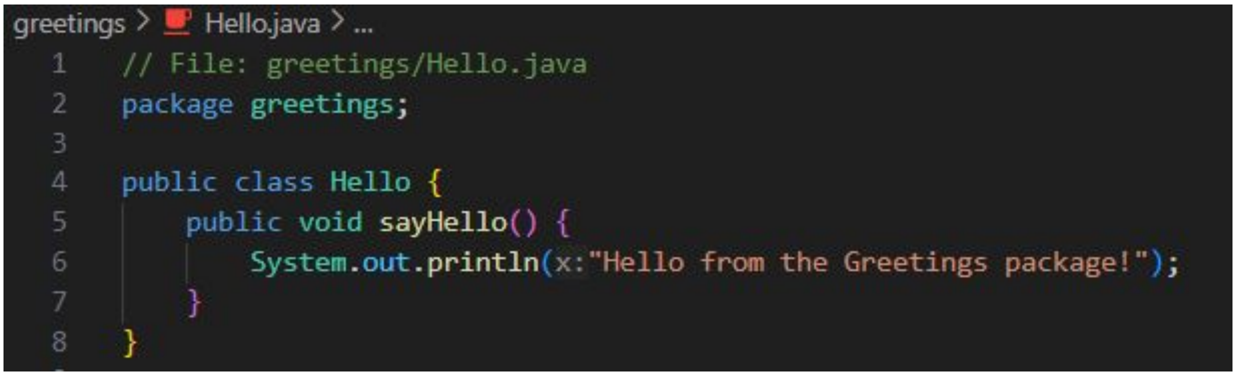
1) Built-in Package

- ❑ Built-in packages are pre-written collections of classes and interfaces in Java
- ❑ They are provided by the Java Development Kit (JDK) to simplify common programming tasks
- ❑ Examples include ***java.util*** (for collections), ***java.io*** (for input/output), and ***java.lang*** (for fundamental classes like String and Math).
- ❑ They are automatically available to Java programs without needing to write custom code for basic features

2) User Defined Package

- ❑ User-defined packages are created by developers to organize and group their own classes and interfaces.
- ❑ You define a user-defined package using the ***package*** keyword at the beginning of a Java file
- ❑ They help keep code modular, making it easier to manage large applications
- ❑ These packages can help avoid naming conflicts between classes in different projects

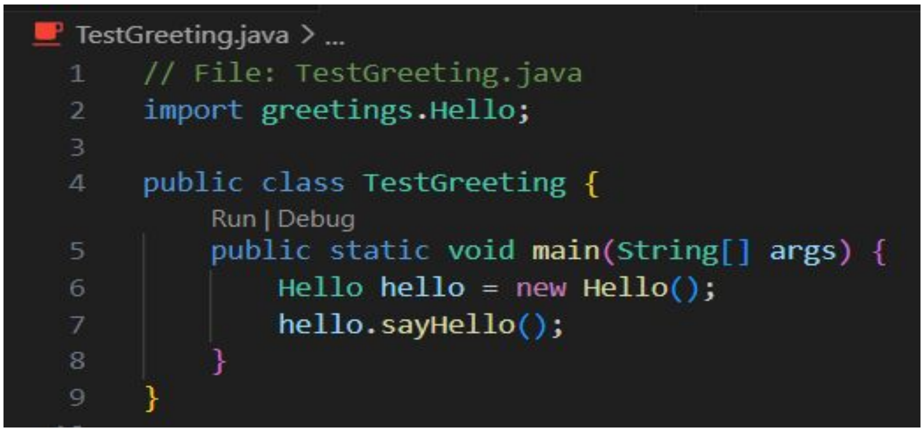
➢ Example code of Creating a Package



greetings > Hello.java > ...

```java
// File: greetings/Hello.java
package greetings;

public class Hello {
    public void sayHello() {
        System.out.println(x:"Hello from the Greetings package!");
    }
}
```

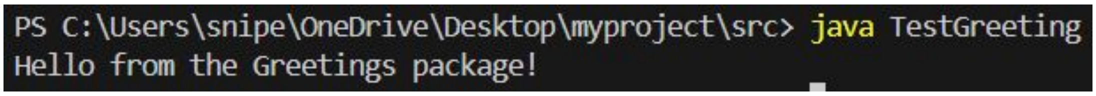
➢ Use the User Defined Package



TestGreeting.java > ...

```java
// File: TestGreeting.java
import greetings.Hello;

public class TestGreeting {
    Run | Debug
    public static void main(String[] args) {
        Hello hello = new Hello();
        hello.sayHello();
    }
}
```

➔ Output



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java TestGreeting
Hello from the Greetings package!
```

❖ Interface

An interface in Java is like a blueprint for classes. It specifies a list of methods that the classes must implement, but it doesn't provide the actual code for them. It allows different classes to implement the same set of methods, promoting ***abstraction*** and ***multiple inheritance,*** enhancing code modularity and flexibility

➢ Characteristics of Interface

- ❑ Sets rules for classes to follow
- ❑ Only has method names, not the code
- ❑ Classes can use multiple interfaces
- ❑ Can include constants (fixed values)
- ❑ Good for sharing behaviors across different classes

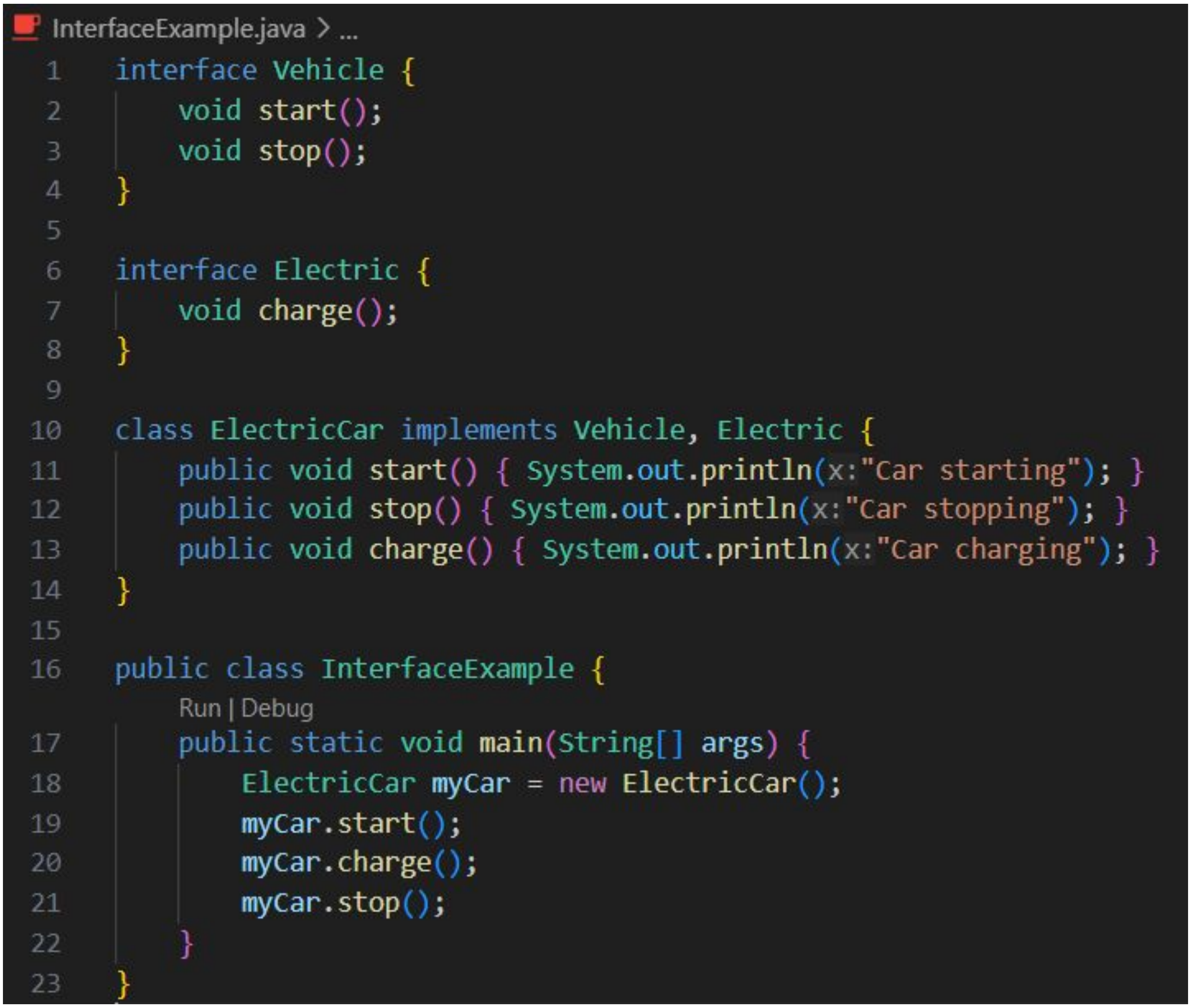
➢ Key differences b/w Interface and Abstract class

- ★ Interfaces can only declare methods while abstract classes can have both abstract methods and concrete methods
- ★ A class can implement multiple interfaces but can inherit from only one abstract class
- ★ Methods in interfaces are implicitly public. Abstract classes can have various access modifiers

★ Interfaces cannot have instance variables, only constants. Abstract classes can have instance variables and constructors

★ Interfaces are primarily used to define a contract that multiple classes can implement, while abstract classes are used to share code among related classes

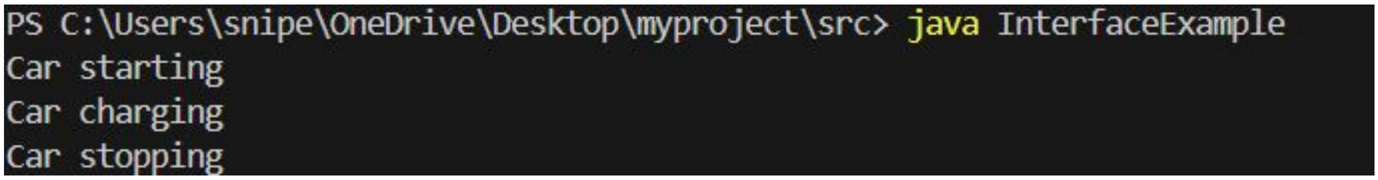
➢ Example code of implementing Interface



InterfaceExample.java > ...

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

interface Electric {
    void charge();
}

class ElectricCar implements Vehicle, Electric {
    public void start() { System.out.println(x:"Car starting"); }
    public void stop() { System.out.println(x:"Car stopping"); }
    public void charge() { System.out.println(x:"Car charging"); }
}

public class InterfaceExample {
    Run | Debug
    public static void main(String[] args) {
        ElectricCar myCar = new ElectricCar();
        myCar.start();
        myCar.charge();
        myCar.stop();
    }
}
```

➔ Output



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java InterfaceExample
Car starting
Car charging
Car stopping
```

❖ Exception Handling

Exception handling in Java is a programming practice used to manage errors and unexpected events that occur during program execution. It enables developers to define how the program should respond to different types of exceptions, ensuring that issues are addressed without causing the entire application to crash or misbehave.

➢ Features of Exception Handling

- ❑ Exception handling separates error-handling code from regular code, improving readability
- ❑ It allows developers to catch specific exceptions, enabling tailored responses to different error types
- ❑ Certain code can be executed regardless of whether an error occurs, useful for resource management
- ❑ Java distinguishes between checked exceptions, which must be declared or handled, and unchecked exceptions, which do not
- ❑ Exception handling contributes to the robustness of applications
- ❑ Developers can create their own exception classes, enhancing flexibility in error handling
- ❑ Exception handling can improve user experience by providing meaningful error messages instead of abrupt crashes

➢ Components of Exception Handling

1. Exception Class: A class that defines the type of exception. Java provides built-in exception classes (like `NullPointerException`, `IOException`) and allows the creation of custom exceptions.

2. Throw Statement: Used to explicitly throw an exception, either a built-in or a user-defined one.

3. Try Block: A block of code that is monitored for exceptions. If an exception occurs within this block, it is thrown for handling.

4. Catch Block: This block catches exceptions thrown from the associated try block. It specifies the type of exception it can handle and contains code to manage the exception.

5. Finally Block: An optional block that executes after the try and catch blocks, regardless of whether an exception was thrown or caught. It's typically used for resource cleanup.

6. Throws Keyword: Used in method signatures to declare that a method can throw specific exceptions, allowing higher-level methods to handle them.

7. Custom Exception Classes: User-defined classes that extend `Exception` or `RuntimeException`, allowing developers to create specific exception types for their applications.

These components work together to provide a ***robust*** framework for managing errors

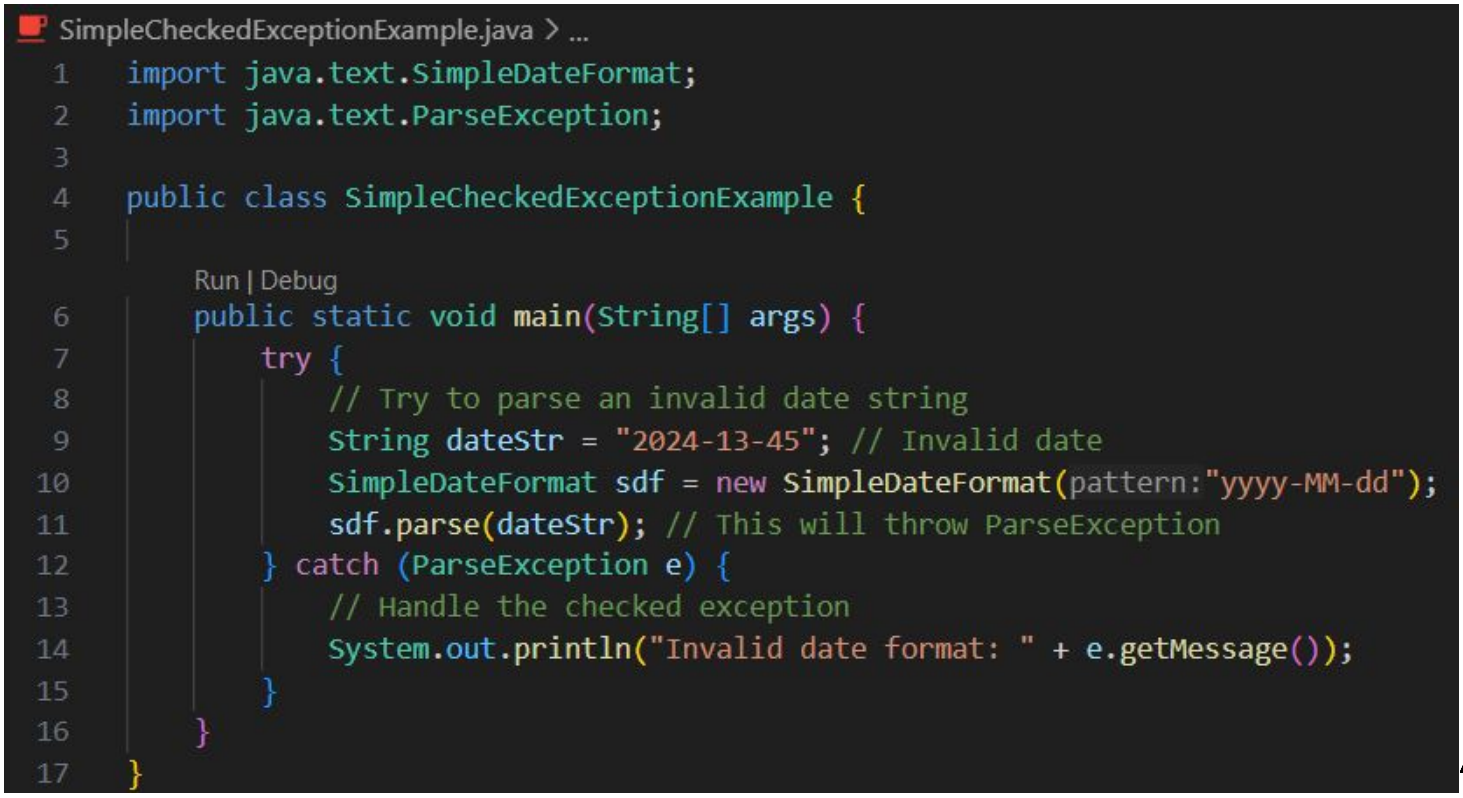
➢ Types of Exception Handling

In Java, exception handling can be categorized into two main types:

★ Checked Exceptions

- ❑ These are exceptions that must be either caught or declared in the method signature using the ***throws*** keyword.
- ❑ They are checked at compile time, meaning the compiler verifies that the code handles them appropriately.
- ❑ Examples include ***IOException, SQLException, and FileNotFoundException***

➢ Example code of Checked Exception



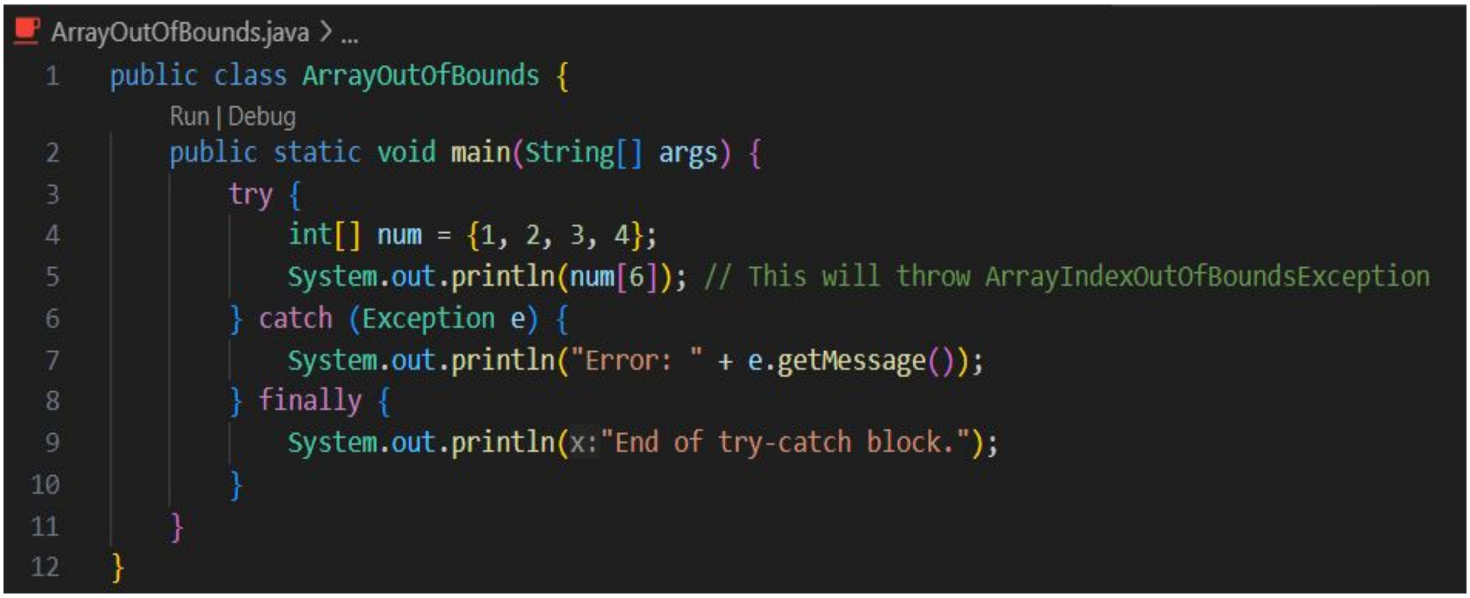
SimpleCheckedExceptionExample.java > ...

```java
import java.text.SimpleDateFormat;
import java.text.ParseException;

public class SimpleCheckedExceptionExample {

    Run | Debug
    public static void main(String[] args) {
        try {
            // Try to parse an invalid date string
            String dateStr = "2024-13-45"; // Invalid date
            SimpleDateFormat sdf = new SimpleDateFormat(pattern:"yyyy-MM-dd");
            sdf.parse(dateStr); // This will throw ParseException
        } catch (ParseException e) {
            // Handle the checked exception
            System.out.println("Invalid date format: " + e.getMessage());
        }
    }
}
```

★ Unchecked Exceptions

- ❑ These exceptions do not need to be declared or caught. They can occur during runtime, and the compiler does not check for them
- ❑ Unchecked exceptions are subclasses of RuntimeException
- ❑ Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException

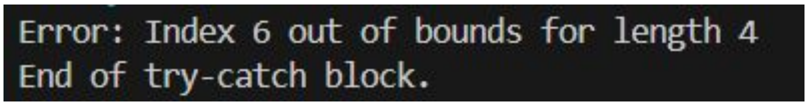
➢ Example code of Unchecked Exception



ArrayOutOfBounds.java > ...

```java
public class ArrayOutOfBounds {
    Run | Debug
    public static void main(String[] args) {
        try {
            int[] num = {1, 2, 3, 4};
            System.out.println(num[6]); // This will throw ArrayIndexOutOfBoundsException
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println(x:"End of try-catch block.");
        }
    }
}
```

➔ Output



```
Error: Index 6 out of bounds for length 4
End of try-catch block.
```

❖ Threads

Threads are independent sequences of execution that allow a program to perform multiple tasks simultaneously. They enable concurrent processing, enhancing application responsiveness and efficiency. By using threads, developers can improve resource utilization and manage complex tasks more effectively in a multi-core environment. Proper management of threads can also prevent issues like race conditions and deadlocks, ensuring smooth execution.

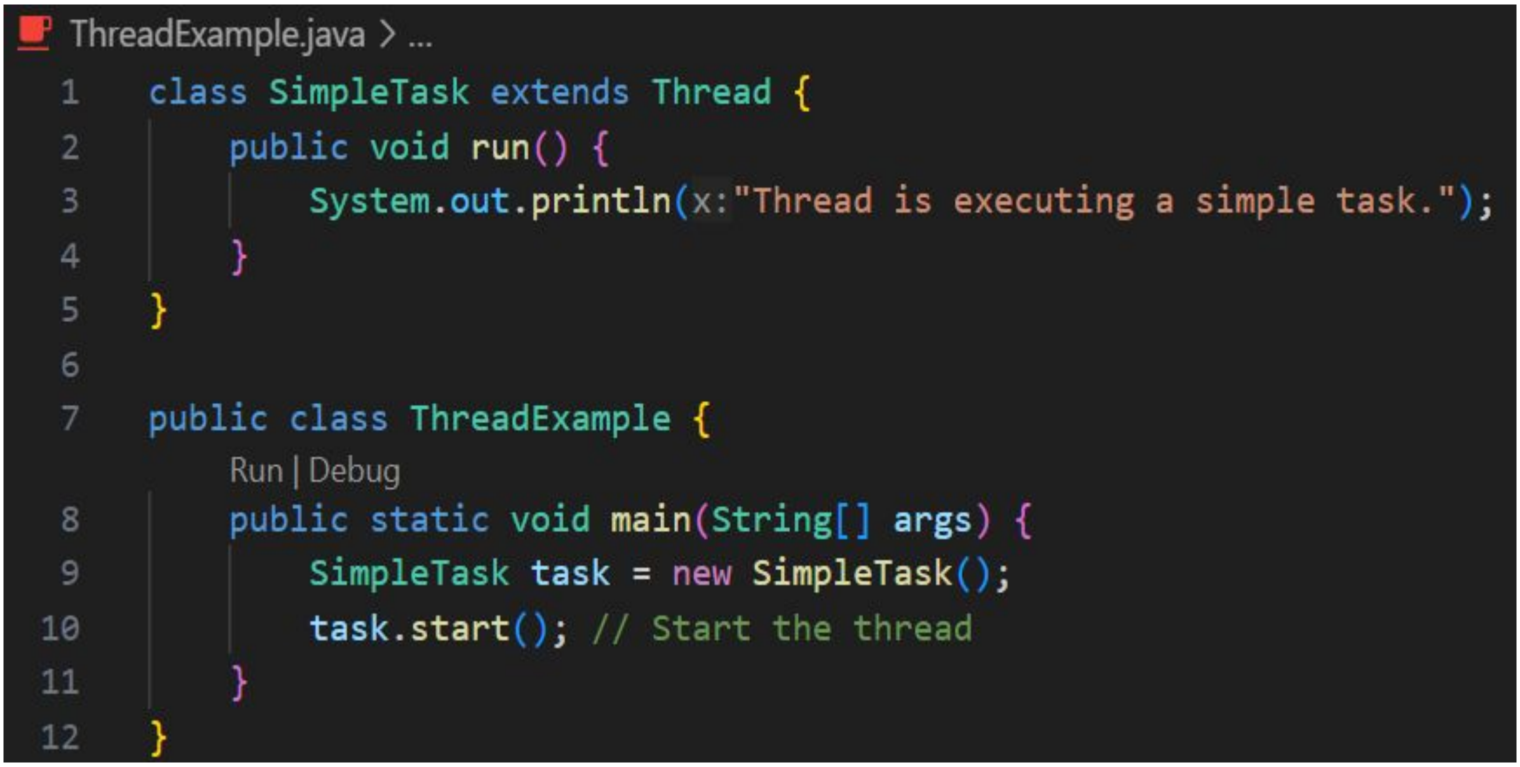
➢ Why to use Threads in Java

❏ Enables concurrent execution of tasks, improving performance and responsiveness
❏ Utilizes multi-core processors effectively, enhancing resource utilization
❏ Allows for background processing, keeping the main application responsive
❏ Facilitates easier management of complex applications with asynchronous operations
❏ Supports improved application structure by separating tasks into distinct threads
❏ Simplifies handling of I/O operations, preventing blocking of the main thread
❏ Enhances user experience in GUI applications by keeping interfaces responsive during lengthy operations
❏ Reduces latency in applications by performing time-consuming tasks in parallel

➢ Implementation

In Java, Threads can be implemented by two ways:

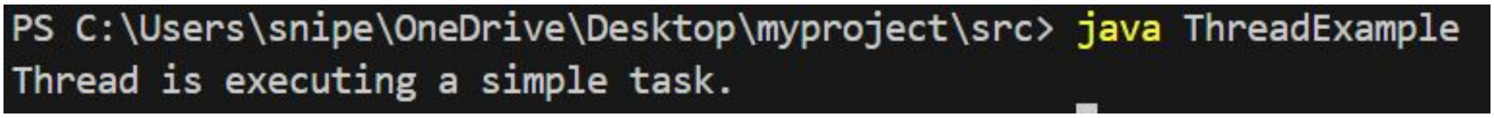
1) Extending the Thread class



ThreadExample.java > ...

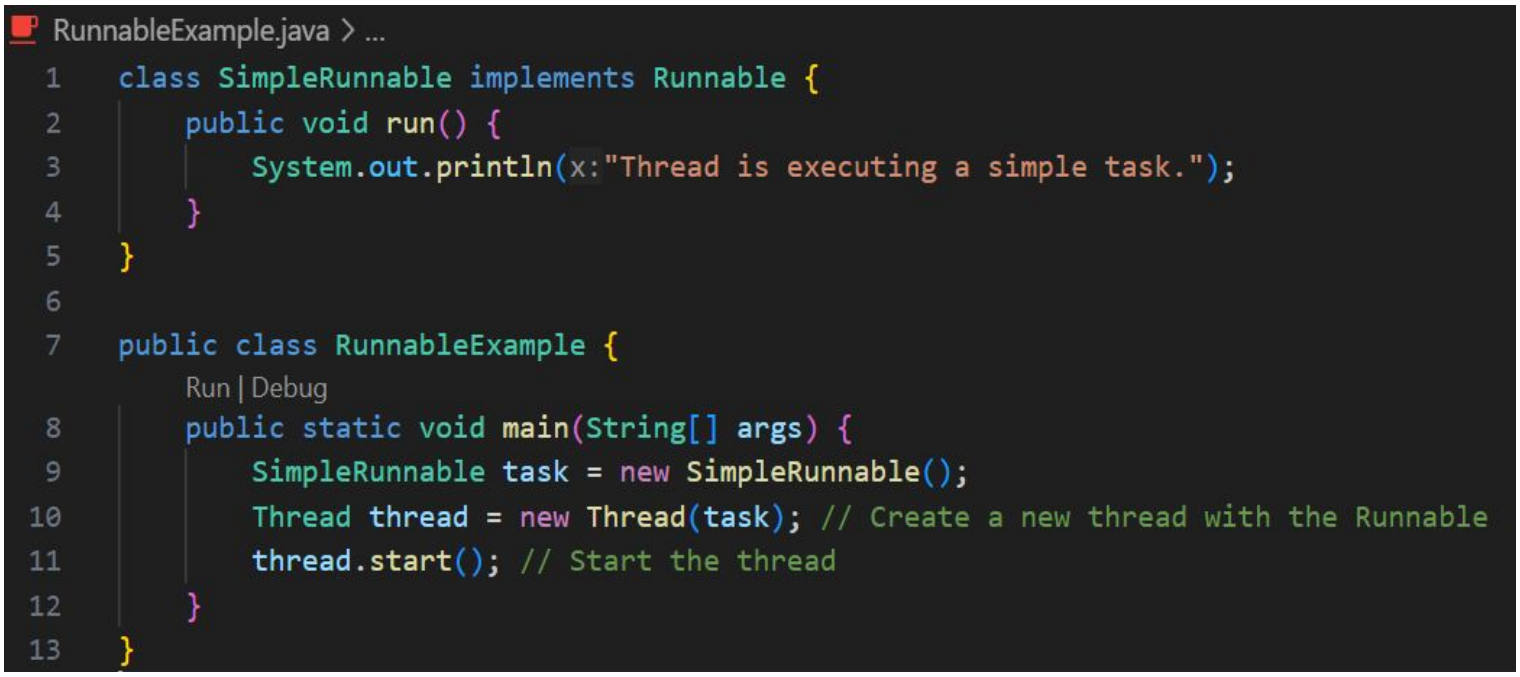
```java
class SimpleTask extends Thread {
    public void run() {
        System.out.println(x:"Thread is executing a simple task.");
    }
}

public class ThreadExample {
    Run | Debug
    public static void main(String[] args) {
        SimpleTask task = new SimpleTask();
        task.start(); // Start the thread
    }
}
```

➔ Output



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java ThreadExample
Thread is executing a simple task.
```

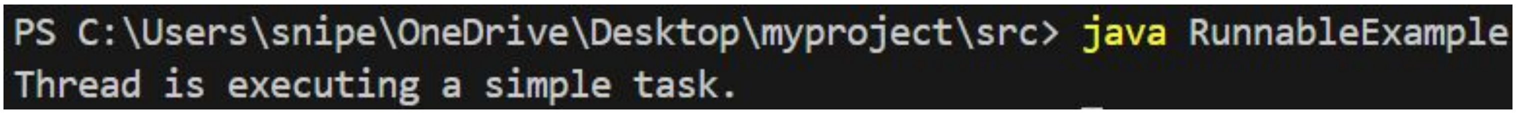
2) Implementing the Runnable Interface



RunnableExample.java > ...

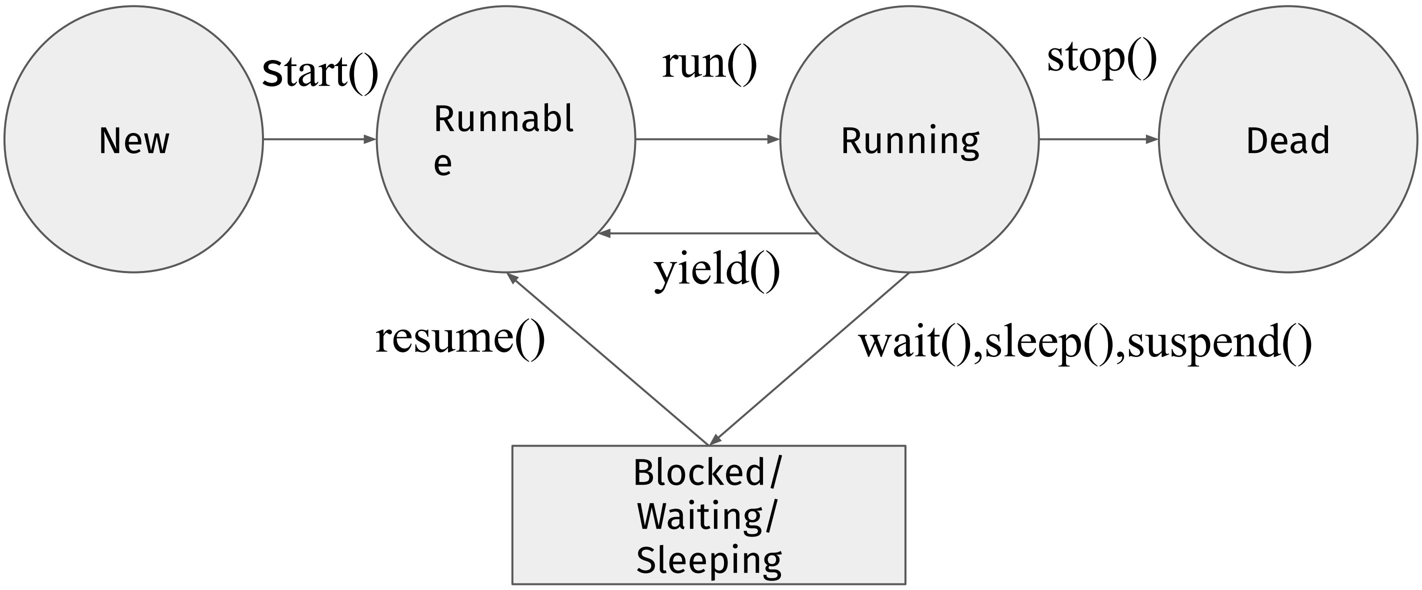
```java
class SimpleRunnable implements Runnable {
    public void run() {
        System.out.println(x:"Thread is executing a simple task.");
    }
}

public class RunnableExample {
    Run | Debug
    public static void main(String[] args) {
        SimpleRunnable task = new SimpleRunnable();
        Thread thread = new Thread(task); // Create a new thread with the Runnable
        thread.start(); // Start the thread
    }
}
```

➔ **Output**



```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java RunnableExample
Thread is executing a simple task.
```

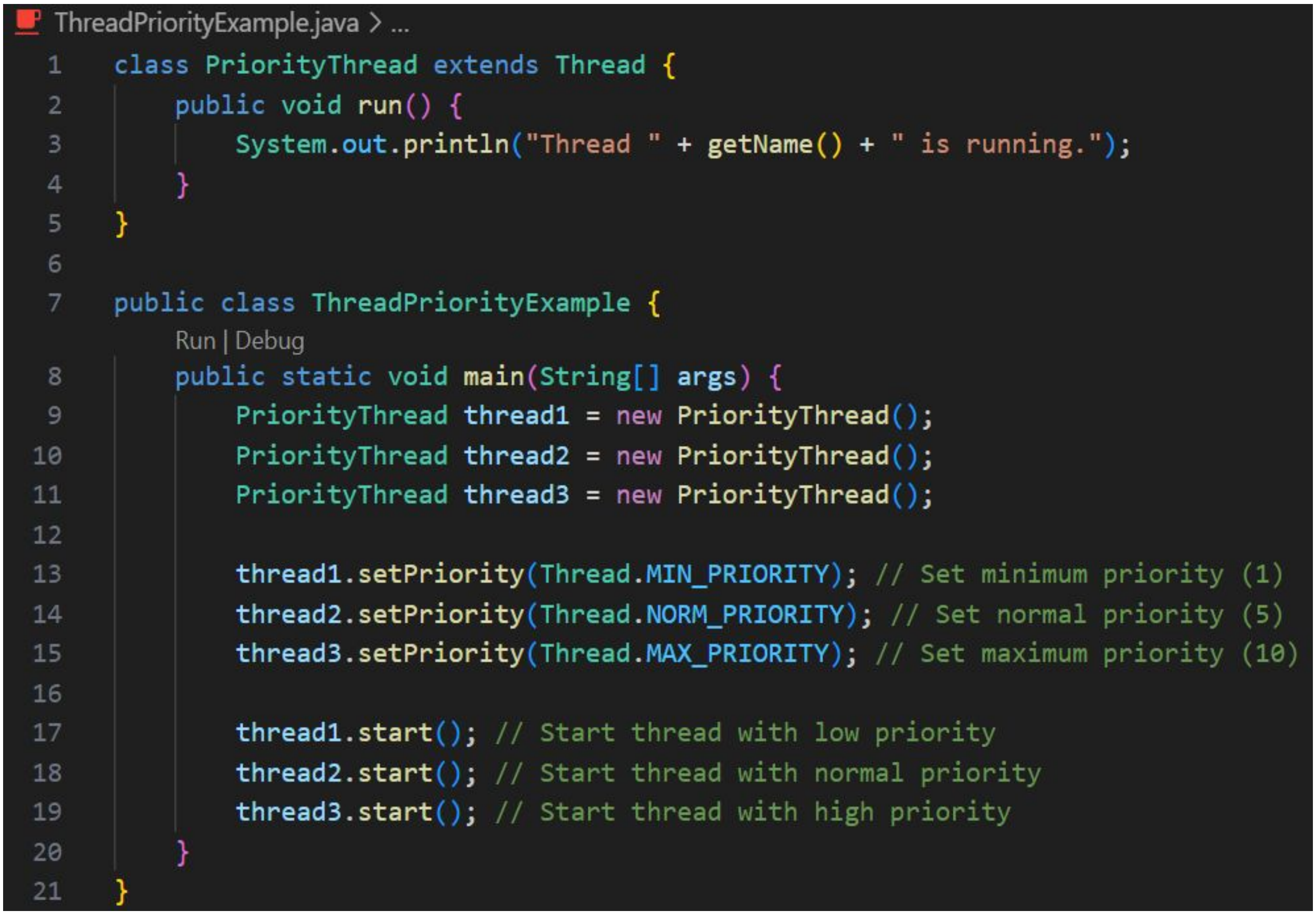
★ *Life cycle of Thread*



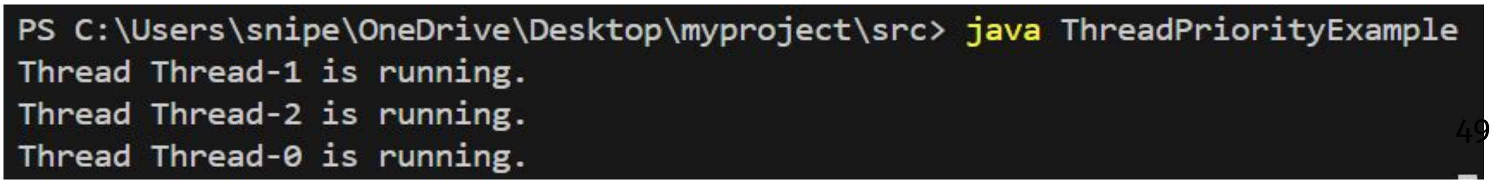
➢ Thread Priorities

Thread priorities allows to indicate the relative importance of different threads. Each thread can be assigned a priority, which is a hint to the thread scheduler about how to allocate CPU time. Thread priorities range from 1 ***(minimum priority)*** to 10 ***(maximum priority)***, with 5 being the ***default priority.***

➢ Example code of Thread Priorities


ThreadPriorityExample.java > ...

```java
class PriorityThread extends Thread {
    public void run() {
        System.out.println("Thread " + getName() + " is running.");
    }
}

public class ThreadPriorityExample {
    Run | Debug
    public static void main(String[] args) {
        PriorityThread thread1 = new PriorityThread();
        PriorityThread thread2 = new PriorityThread();
        PriorityThread thread3 = new PriorityThread();

        thread1.setPriority(Thread.MIN_PRIORITY); // Set minimum priority (1)
        thread2.setPriority(Thread.NORM_PRIORITY); // Set normal priority (5)
        thread3.setPriority(Thread.MAX_PRIORITY); // Set maximum priority (10)

        thread1.start(); // Start thread with low priority
        thread2.start(); // Start thread with normal priority
        thread3.start(); // Start thread with high priority
    }
}
```

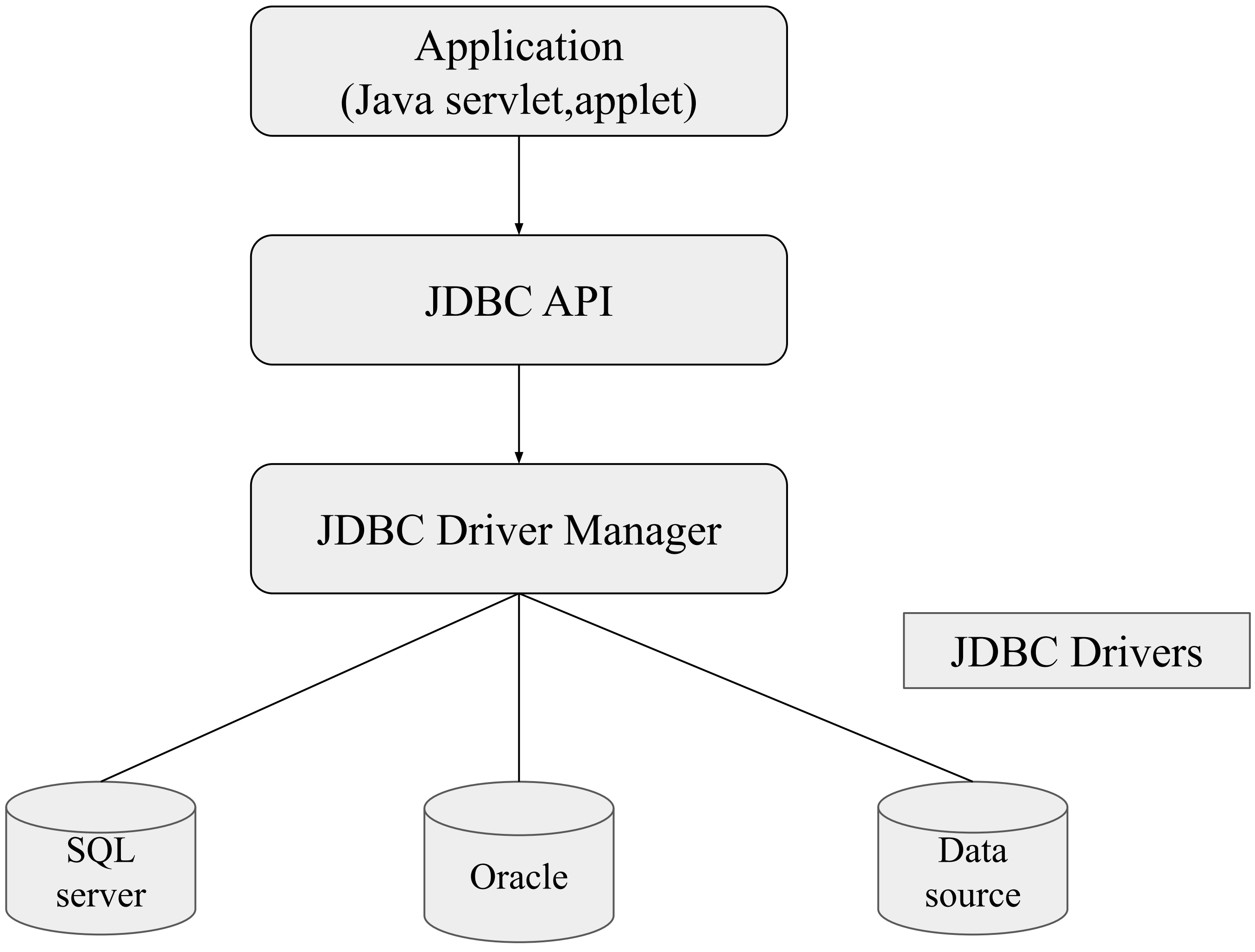


```
PS C:\Users\snipe\OneDrive\Desktop\myproject\src> java ThreadPriorityExample
Thread Thread-1 is running.
Thread Thread-2 is running.
Thread Thread-0 is running.
```

❖ JDBC

JDBC (Java Database Connectivity) is an API in Java that allows applications to interact with relational databases. It provides a standard interface for connecting, executing SQL queries, and retrieving results. JDBC supports various database drivers, enabling seamless communication between Java applications and different database systems, ensuring efficient data manipulation and retrieval through SQL.

➢ Architecture of JDBC



★ Description

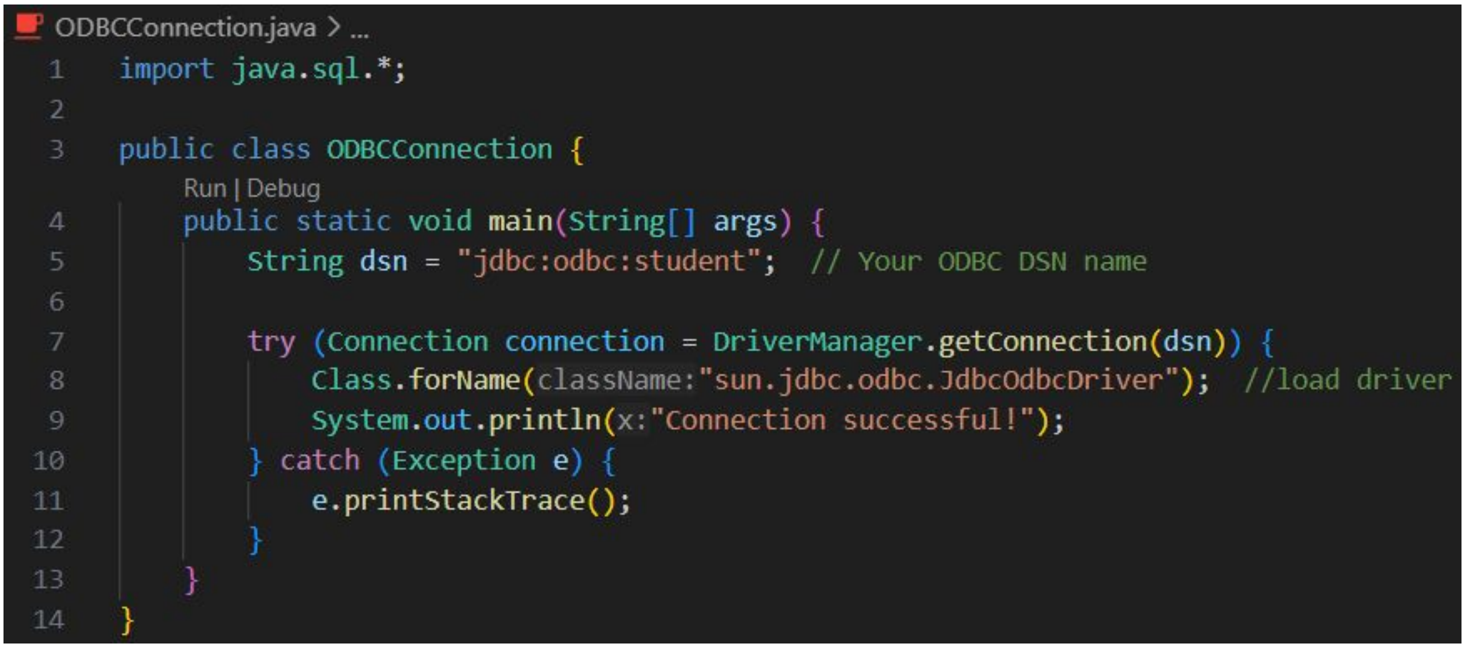
1) **Application**: It is a java applet or a servlet that communicates with a data source.

2) **The JDBC API:** The JDBC API allows Java programs to execute SQL statements and retrieve results.

3) **Driver Manager:** It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

4) **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

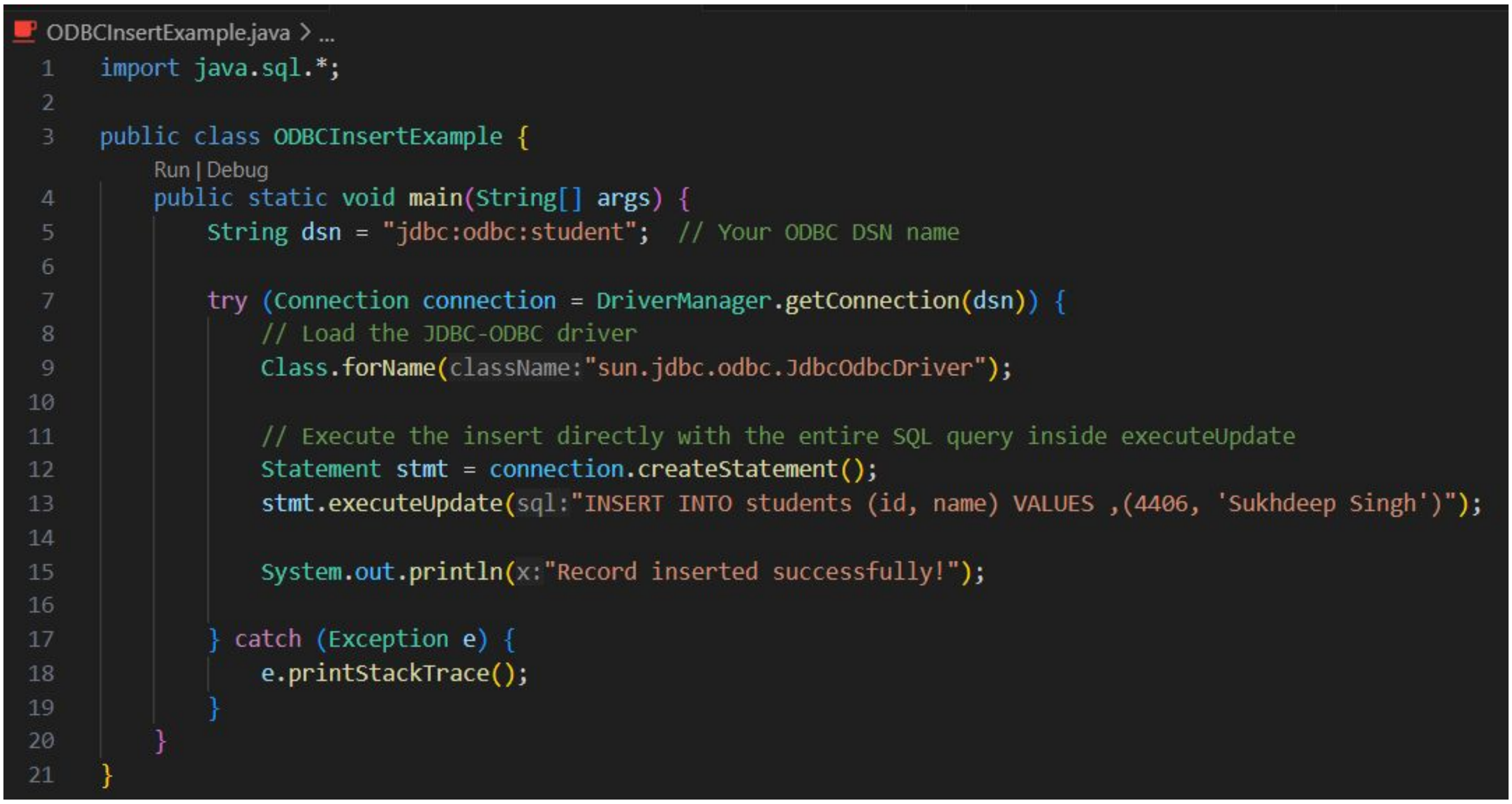
➢ Example code of Setting up connection



```
ODBCConnection.java > ...
import java.sql.*;

public class ODBCConnection {
    Run | Debug
    public static void main(String[] args) {
        String dsn = "jdbc:odbc:student";  // Your ODBC DSN name

        try (Connection connection = DriverManager.getConnection(dsn)) {
            Class.forName(className:"sun.jdbc.odbc.JdbcOdbcDriver");  //load driver
            System.out.println(x:"Connection successful!");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

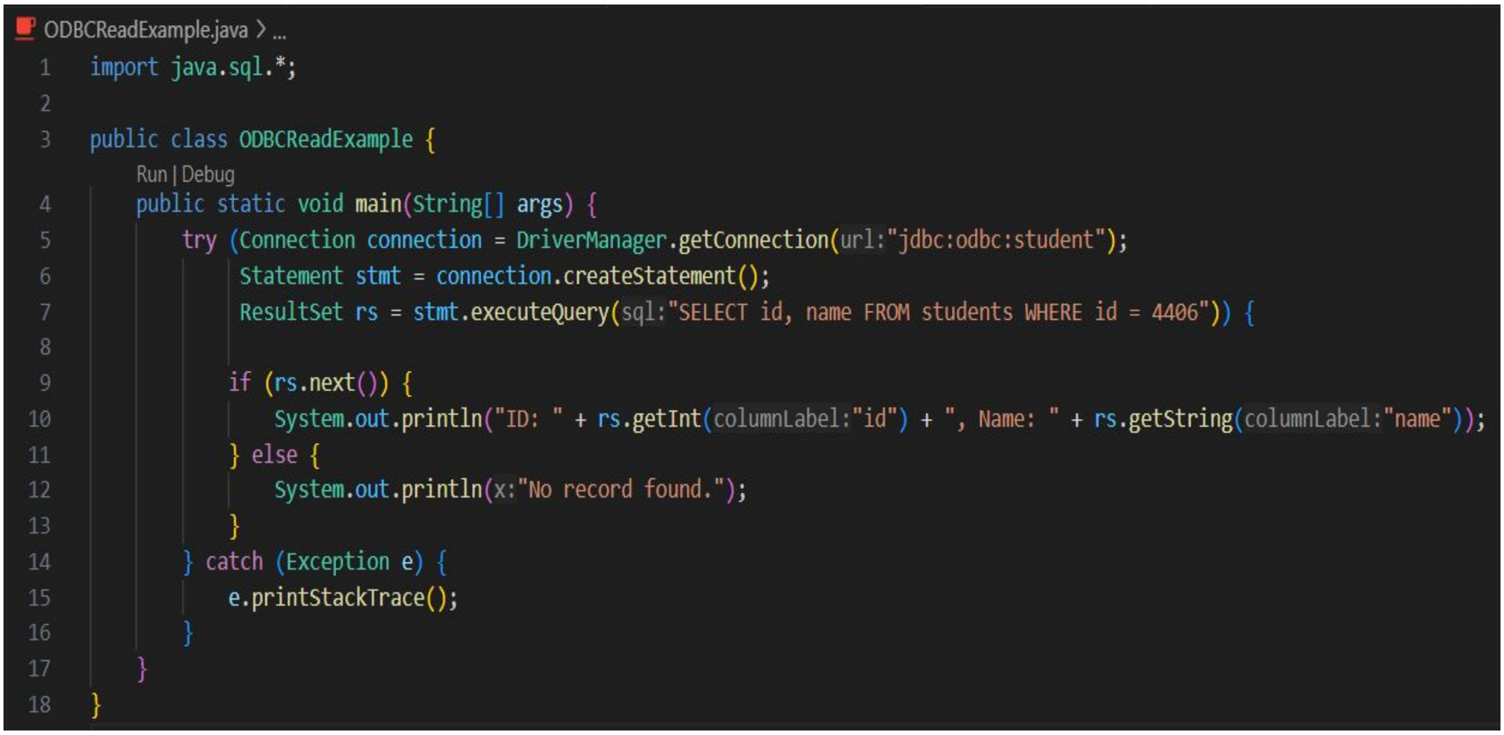
❖ CRUD Operations

➢ Example code of Creating a Record


ODBCInsertExample.java > ...

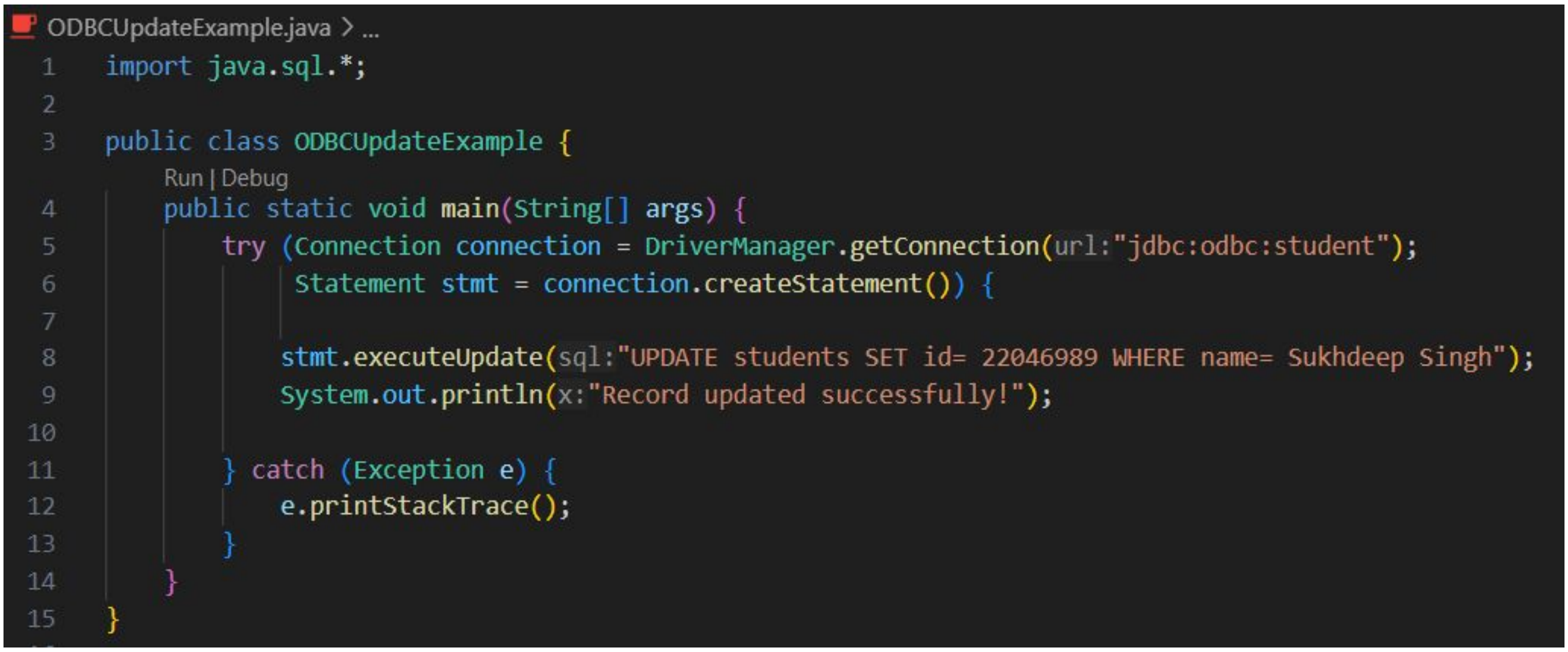
```java
import java.sql.*;

public class ODBCInsertExample {
    Run | Debug
    public static void main(String[] args) {
        String dsn = "jdbc:odbc:student";  // Your ODBC DSN name

        try (Connection connection = DriverManager.getConnection(dsn)) {
            // Load the JDBC-ODBC driver
            Class.forName(className:"sun.jdbc.odbc.JdbcOdbcDriver");

            // Execute the insert directly with the entire SQL query inside executeUpdate
            Statement stmt = connection.createStatement();
            stmt.executeUpdate(sql:"INSERT INTO students (id, name) VALUES ,(4406, 'Sukhdeep Singh')");

            System.out.println(x:"Record inserted successfully!");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

➢ Example code of Reading a Record


ODBCReadExample.java > ...

```java
import java.sql.*;

public class ODBCReadExample {
    Run | Debug
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(url:"jdbc:odbc:student");
             Statement stmt = connection.createStatement();
             ResultSet rs = stmt.executeQuery(sql:"SELECT id, name FROM students WHERE id = 4406")) {

            if (rs.next()) {
                System.out.println("ID: " + rs.getInt(columnLabel:"id") + ", Name: " + rs.getString(columnLabel:"name"));
            } else {
                System.out.println(x:"No record found.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

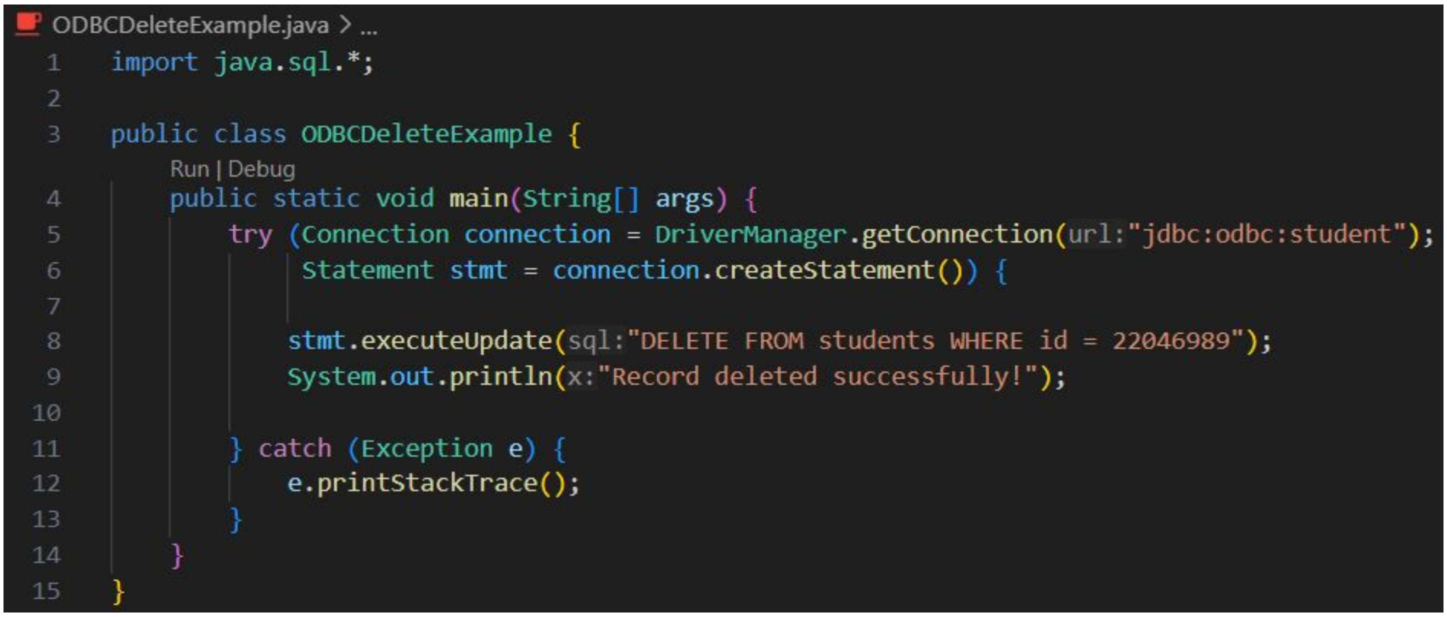
➢ Example code of Updating a Record



ODBCUpdateExample.java > ...

```java
import java.sql.*;

public class ODBCUpdateExample {
    Run | Debug
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(url:"jdbc:odbc:student");
             Statement stmt = connection.createStatement()) {

            stmt.executeUpdate(sql:"UPDATE students SET id= 22046989 WHERE name= Sukhdeep Singh");
            System.out.println(x:"Record updated successfully!");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

➢ Example code of Deleting a Record



ODBCDeleteExample.java > ...

```java
import java.sql.*;

public class ODBCDeleteExample {
    Run | Debug
    public static void main(String[] args) {
        try (Connection connection = DriverManager.getConnection(url:"jdbc:odbc:student");
             Statement stmt = connection.createStatement()) {

            stmt.executeUpdate(sql:"DELETE FROM students WHERE id = 22046989");
            System.out.println(x:"Record deleted successfully!");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

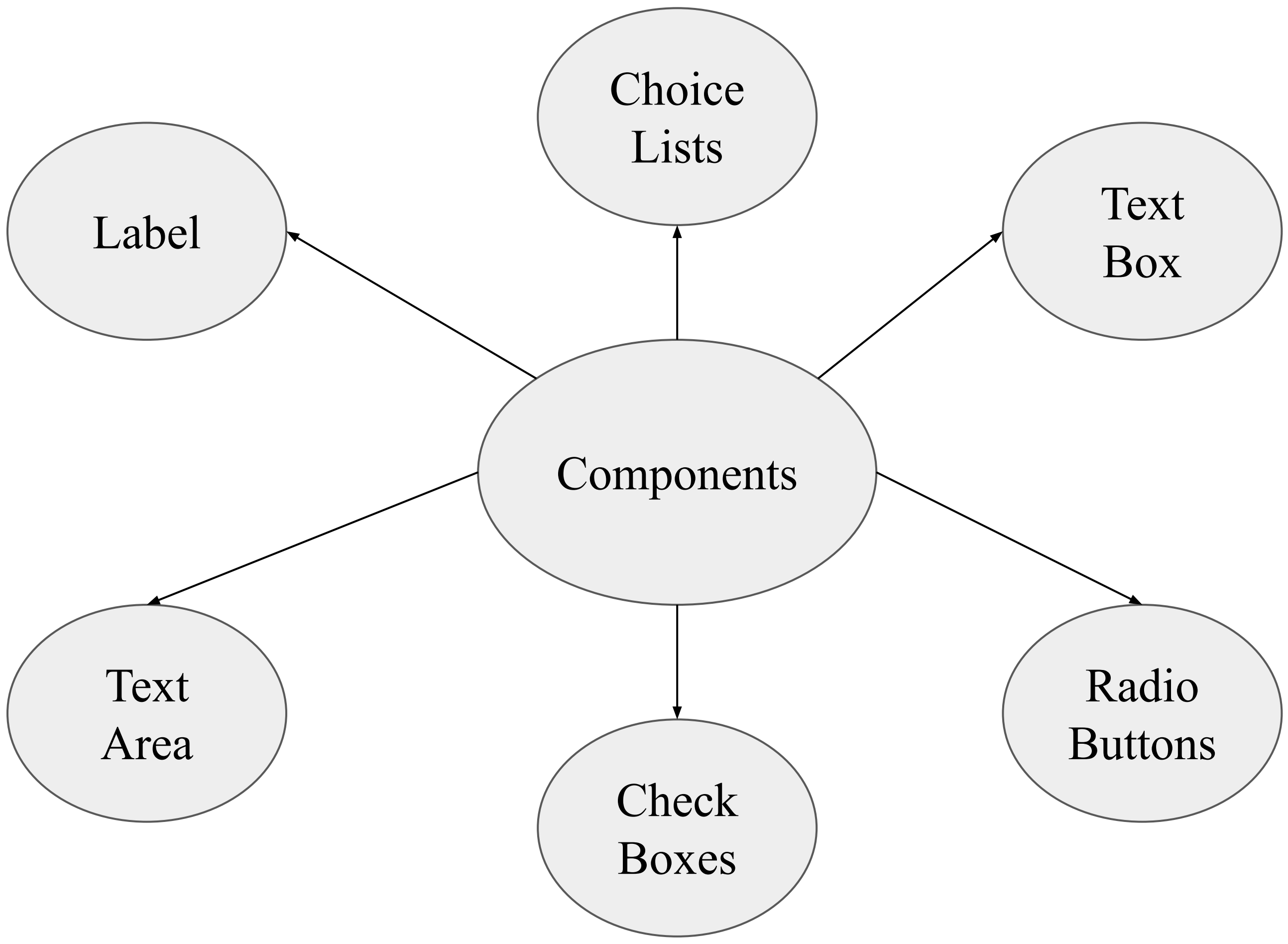
❖ AWT Controls

Java AWT **(Abstract Window Toolkit)** is a set of tools in Java that helps you create graphical user interfaces (GUIs) for desktop applications. It provides basic building blocks like buttons, text boxes, labels, and windows, allowing you to make interactive programs that users can interact with.

➢ Key features of Java AWT

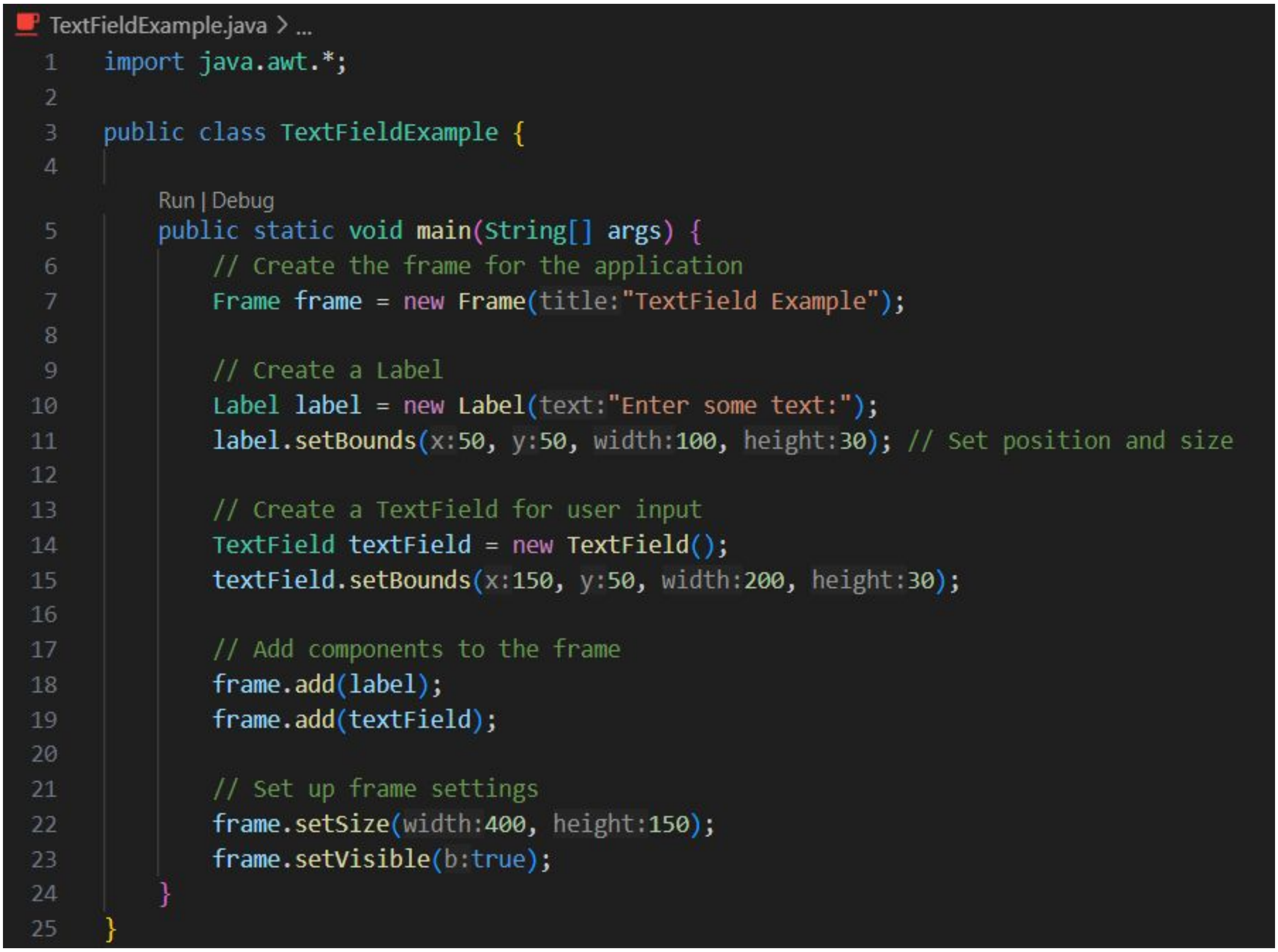
- ❑ Provides basic GUI components like buttons, text fields, labels, and checkboxes
- ❑ Handles user events such as clicks, key presses, and mouse movements
- ❑ Allows drawing of 2D graphics, such as lines, circles, and other shapes
- ❑ Includes container classes like **Frame, Panel,** and **Dialog** for organizing and arranging components.
- ❑ Can be used to create windowed applications with interactive user interfaces.
- ❑ AWT is platform-dependent in the sense that it uses the native operating system's windowing system
- ❑ Supports multi-threading to handle multiple tasks simultaneously, ensuring better performance in GUI applications
- ❑ Provides methods to create menus, pop-up dialogs, and file selection windows

➢ Various AWT Components



1) **Label :** Displays a non-editable text, often used for descriptions or instructions
2) **TextBox :** A single-line input field for text entry
3) **TextArea :** A multiline input field for longer text entry
4) **Radio Button :** A button in a group where only one can be selected at a time
5) **Choice List :** A drop-down list allowing the user to select one option from a list
6) **Check Boxes :** A checkable option allowing a user to select or deselect a choice

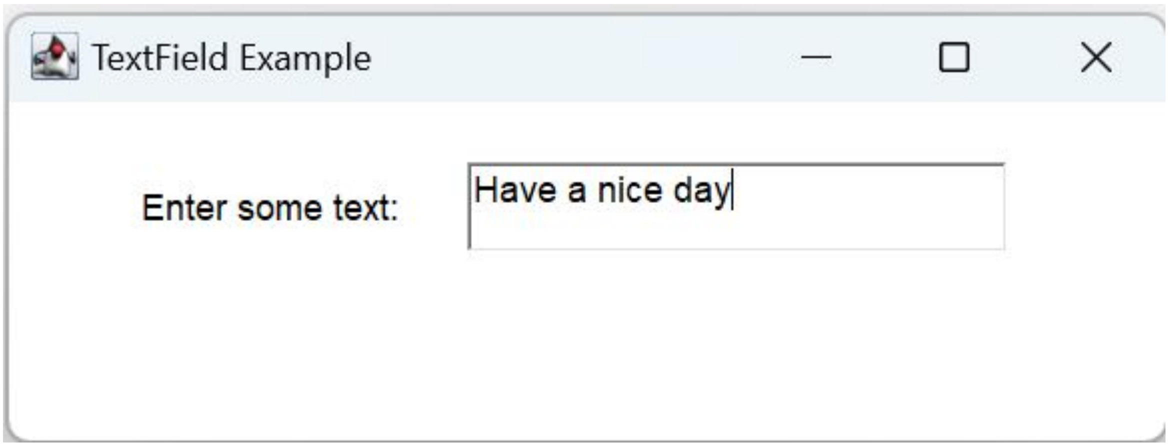
➢ Example code of TextField



TextFieldExample.java > ...

```java
import java.awt.*;

public class TextFieldExample {

    Run | Debug
    public static void main(String[] args) {
        // Create the frame for the application
        Frame frame = new Frame(title:"TextField Example");

        // Create a Label
        Label label = new Label(text:"Enter some text:");
        label.setBounds(x:50, y:50, width:100, height:30); // Set position and size

        // Create a TextField for user input
        TextField textField = new TextField();
        textField.setBounds(x:150, y:50, width:200, height:30);

        // Add components to the frame
        frame.add(label);
        frame.add(textField);

        // Set up frame settings
        frame.setSize(width:400, height:150);
        frame.setVisible(b:true);
    }
}
```

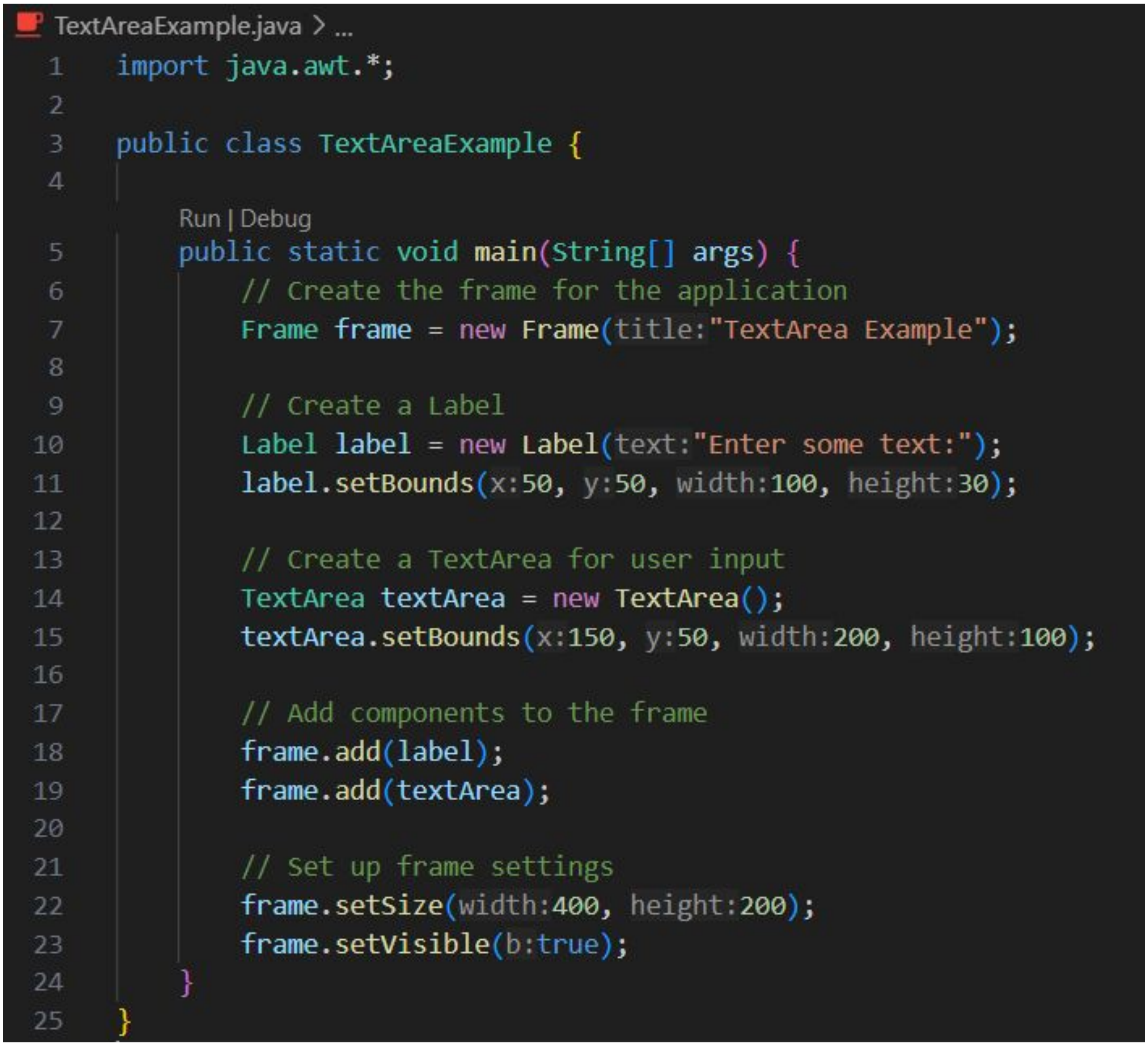
➔ Output



TextField Example

Enter some text: Have a nice day

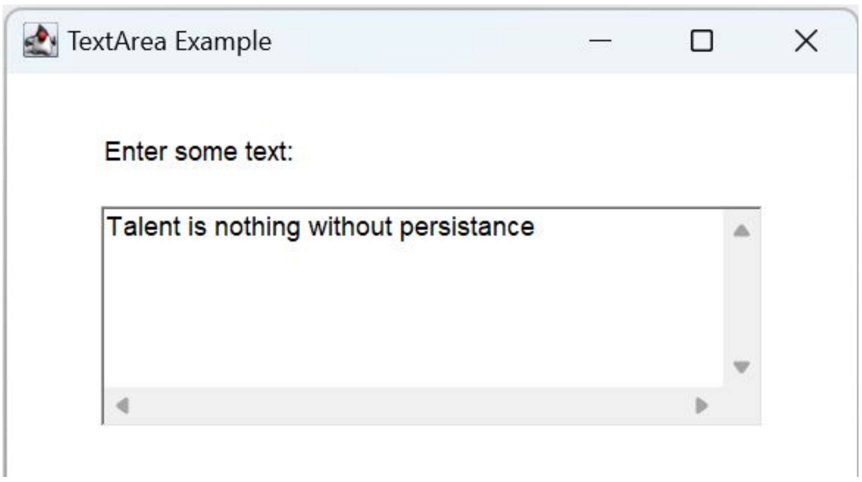
➢ Example code of TextArea



TextAreaExample.java > ...

```java
import java.awt.*;

public class TextAreaExample {

    Run | Debug
    public static void main(String[] args) {
        // Create the frame for the application
        Frame frame = new Frame(title:"TextArea Example");

        // Create a Label
        Label label = new Label(text:"Enter some text:");
        label.setBounds(x:50, y:50, width:100, height:30);

        // Create a TextArea for user input
        TextArea textArea = new TextArea();
        textArea.setBounds(x:150, y:50, width:200, height:100);

        // Add components to the frame
        frame.add(label);
        frame.add(textArea);

        // Set up frame settings
        frame.setSize(width:400, height:200);
        frame.setVisible(b:true);
    }
}
```

➔ **Output**

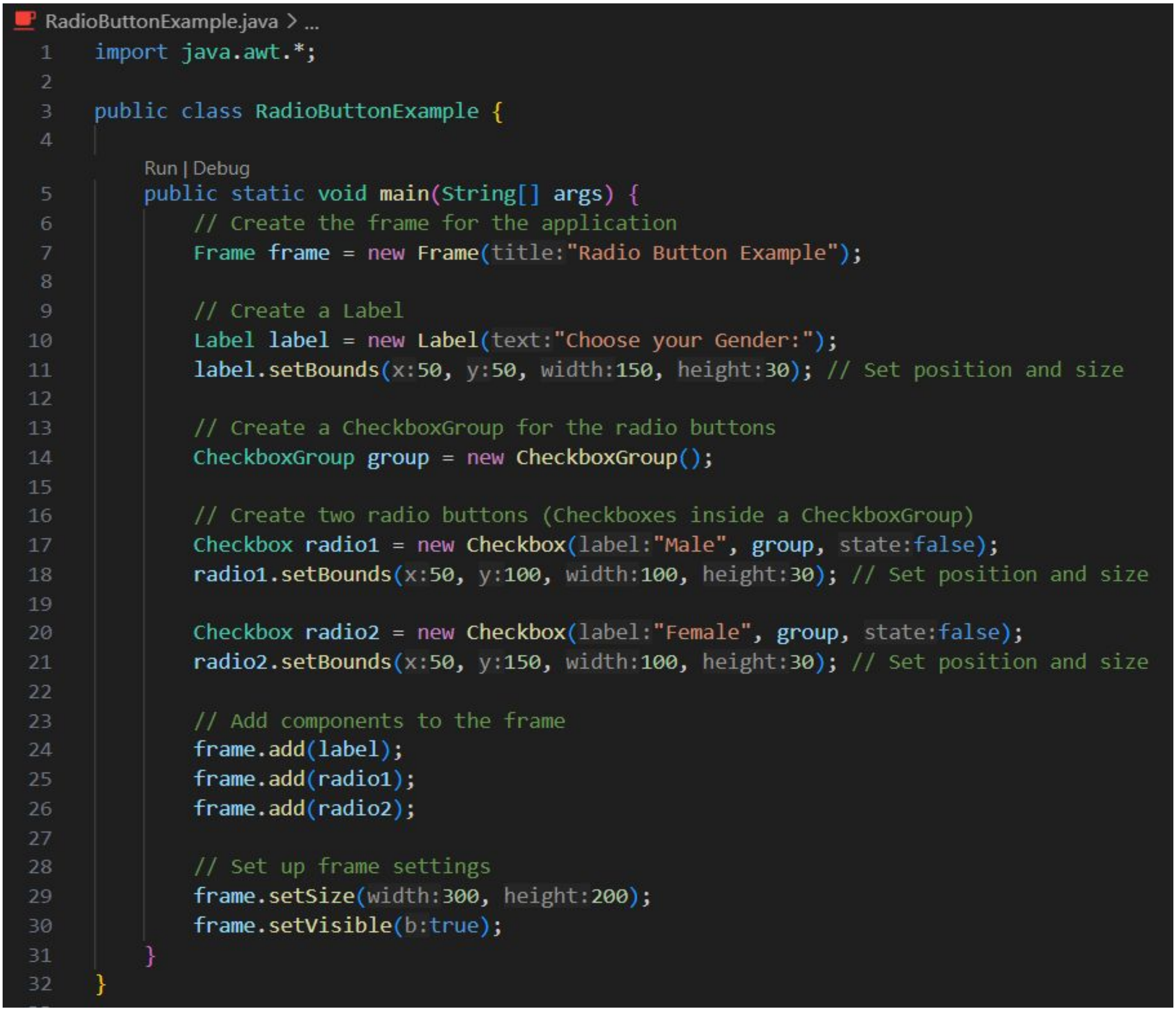


TextArea Example

Enter some text:

Talent is nothing without persistance

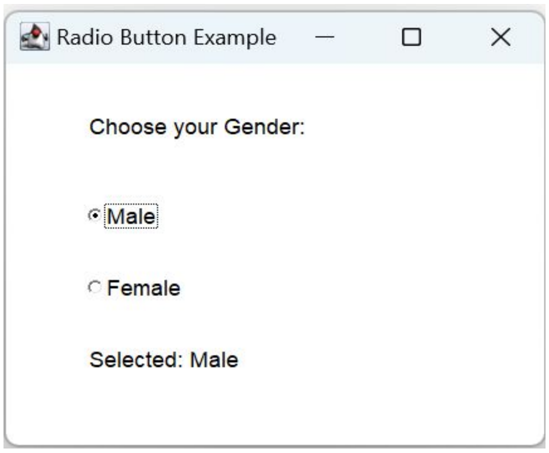
➢ Example code of Radio Button



RadioButtonExample.java > ...

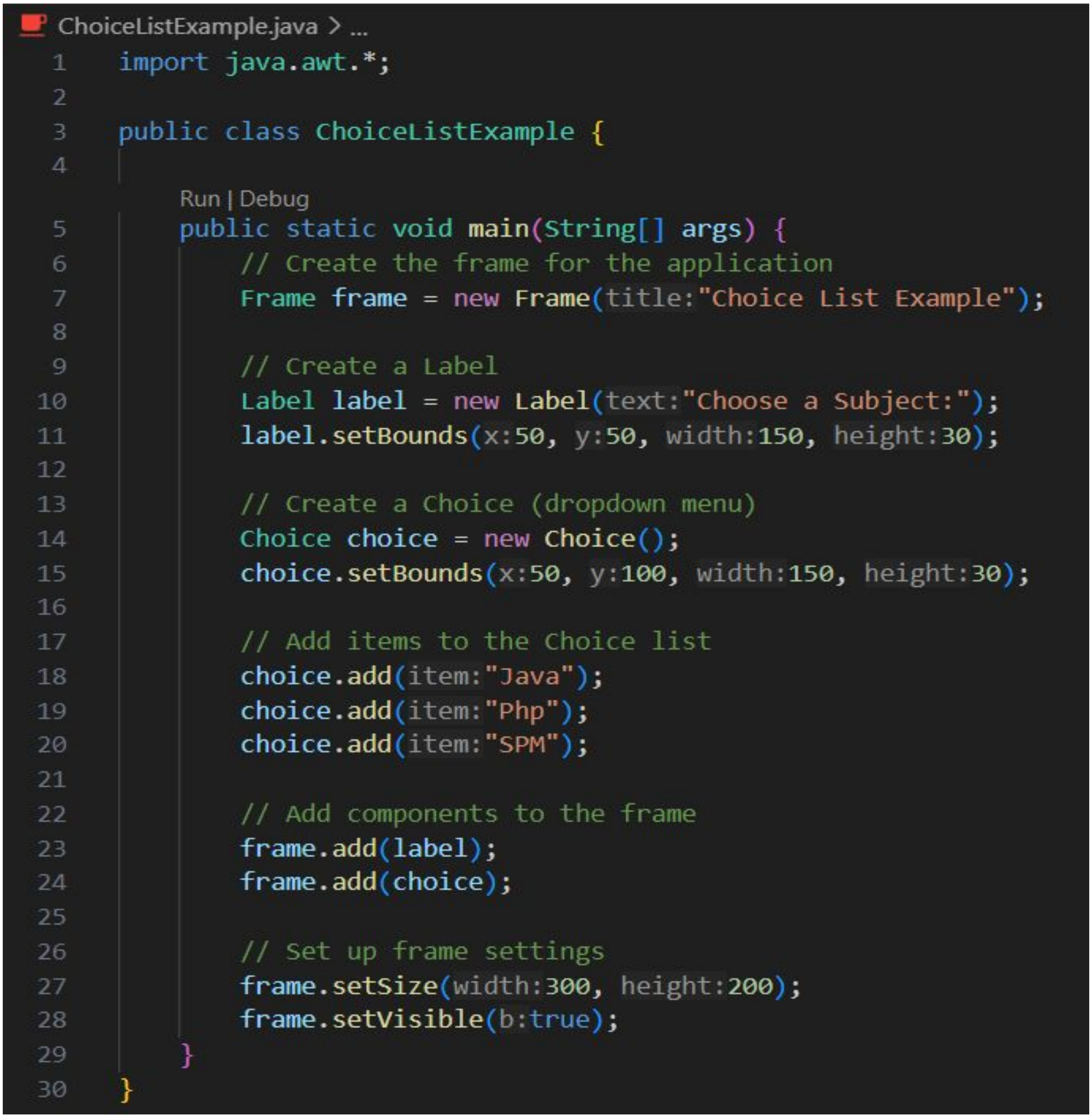
```java
import java.awt.*;

public class RadioButtonExample {

    Run | Debug
    public static void main(String[] args) {
        // Create the frame for the application
        Frame frame = new Frame(title:"Radio Button Example");

        // Create a Label
        Label label = new Label(text:"Choose your Gender:");
        label.setBounds(x:50, y:50, width:150, height:30); // Set position and size

        // Create a CheckboxGroup for the radio buttons
        CheckboxGroup group = new CheckboxGroup();

        // Create two radio buttons (Checkboxes inside a CheckboxGroup)
        Checkbox radio1 = new Checkbox(label:"Male", group, state:false);
        radio1.setBounds(x:50, y:100, width:100, height:30); // Set position and size

        Checkbox radio2 = new Checkbox(label:"Female", group, state:false);
        radio2.setBounds(x:50, y:150, width:100, height:30); // Set position and size

        // Add components to the frame
        frame.add(label);
        frame.add(radio1);
        frame.add(radio2);

        // Set up frame settings
        frame.setSize(width:300, height:200);
        frame.setVisible(b:true);
    }
}
```

➔ **Output**



```
Radio Button Example

Choose your Gender:

● Male

○ Female

Selected: Male
```

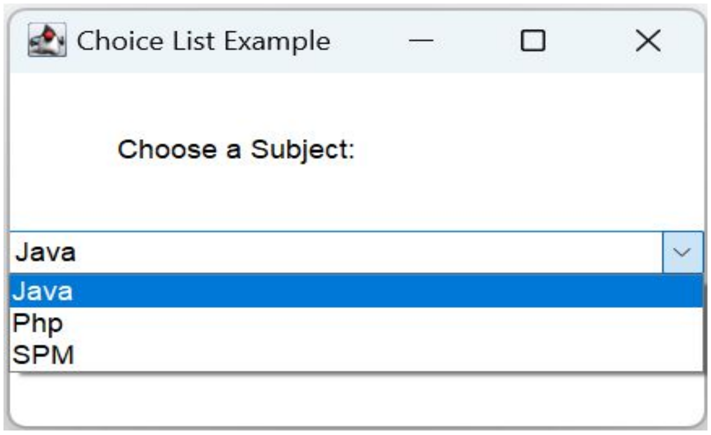
➢ Example code of Choice Lists



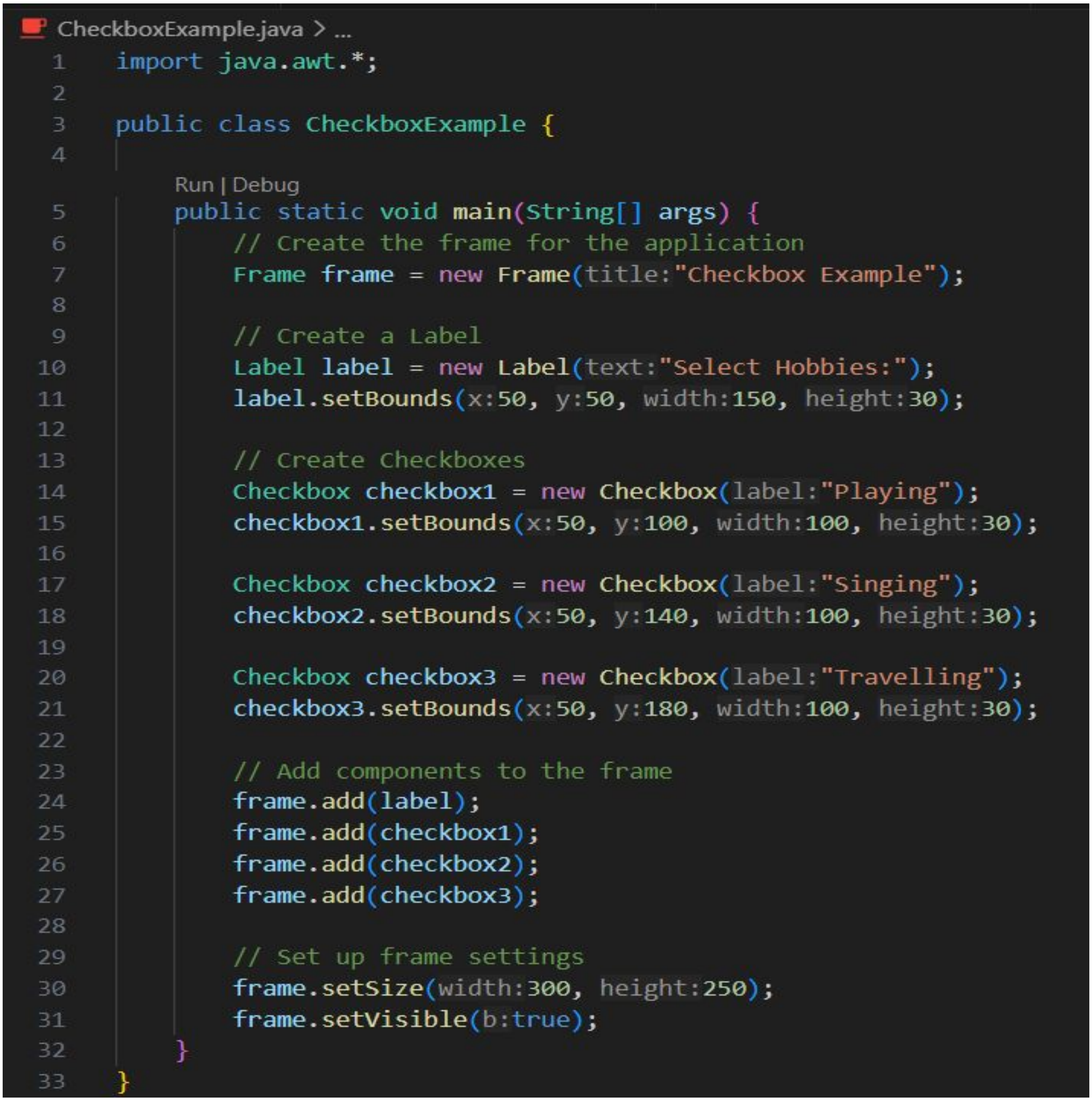
ChoiceListExample.java > ...

```java
import java.awt.*;

public class ChoiceListExample {

    Run | Debug
    public static void main(String[] args) {
        // Create the frame for the application
        Frame frame = new Frame(title:"Choice List Example");

        // Create a Label
        Label label = new Label(text:"Choose a Subject:");
        label.setBounds(x:50, y:50, width:150, height:30);

        // Create a Choice (dropdown menu)
        Choice choice = new Choice();
        choice.setBounds(x:50, y:100, width:150, height:30);

        // Add items to the Choice list
        choice.add(item:"Java");
        choice.add(item:"Php");
        choice.add(item:"SPM");

        // Add components to the frame
        frame.add(label);
        frame.add(choice);

        // Set up frame settings
        frame.setSize(width:300, height:200);
        frame.setVisible(b:true);
    }
}
```

➔ Output



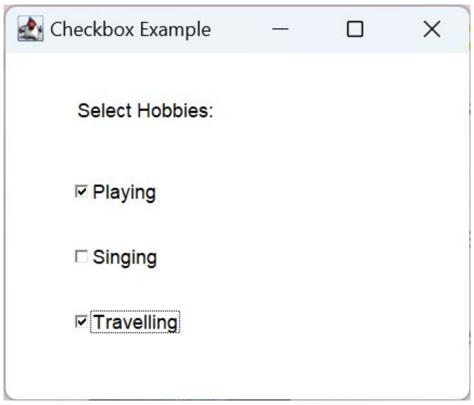
➢ Example code of Check Boxes



CheckboxExample.java > ...

```java
import java.awt.*;

public class CheckboxExample {

    Run | Debug
    public static void main(String[] args) {
        // Create the frame for the application
        Frame frame = new Frame(title:"Checkbox Example");

        // Create a Label
        Label label = new Label(text:"Select Hobbies:");
        label.setBounds(x:50, y:50, width:150, height:30);

        // Create Checkboxes
        Checkbox checkbox1 = new Checkbox(label:"Playing");
        checkbox1.setBounds(x:50, y:100, width:100, height:30);

        Checkbox checkbox2 = new Checkbox(label:"Singing");
        checkbox2.setBounds(x:50, y:140, width:100, height:30);

        Checkbox checkbox3 = new Checkbox(label:"Travelling");
        checkbox3.setBounds(x:50, y:180, width:100, height:30);

        // Add components to the frame
        frame.add(label);
        frame.add(checkbox1);
        frame.add(checkbox2);
        frame.add(checkbox3);

        // Set up frame settings
        frame.setSize(width:300, height:250);
        frame.setVisible(b:true);
    }
}
```

➔ Output



Checkbox Example

Select Hobbies:

[✓ Playing]

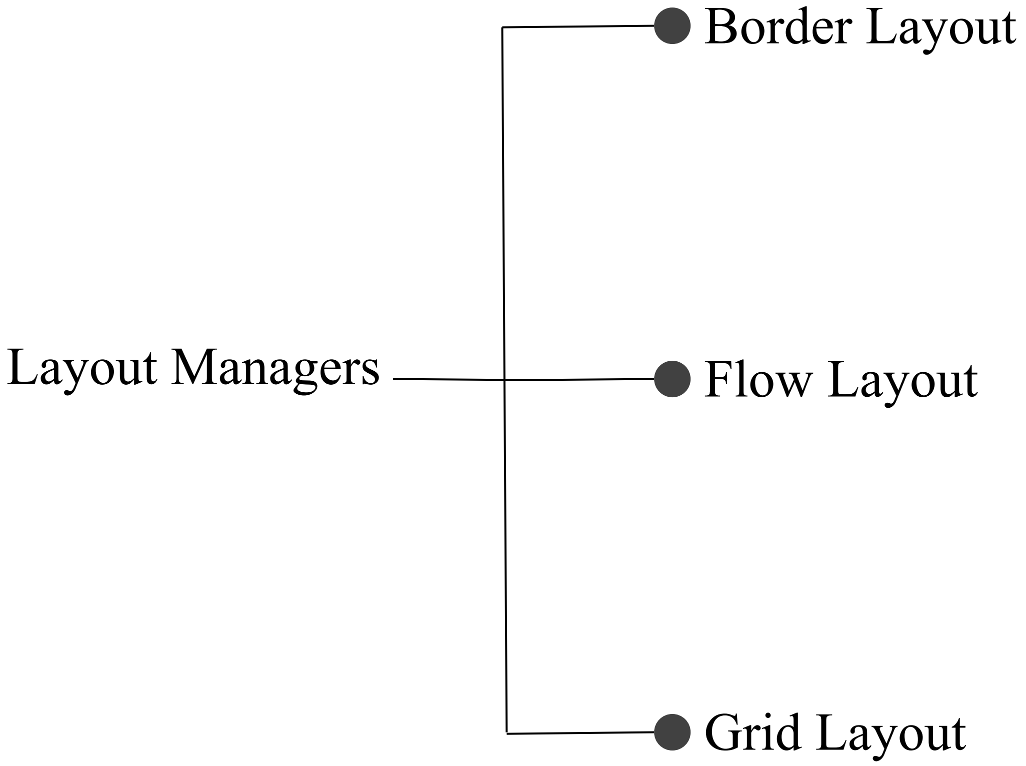
☐ Singing

[✓ Travelling]

❖ Layout Managers

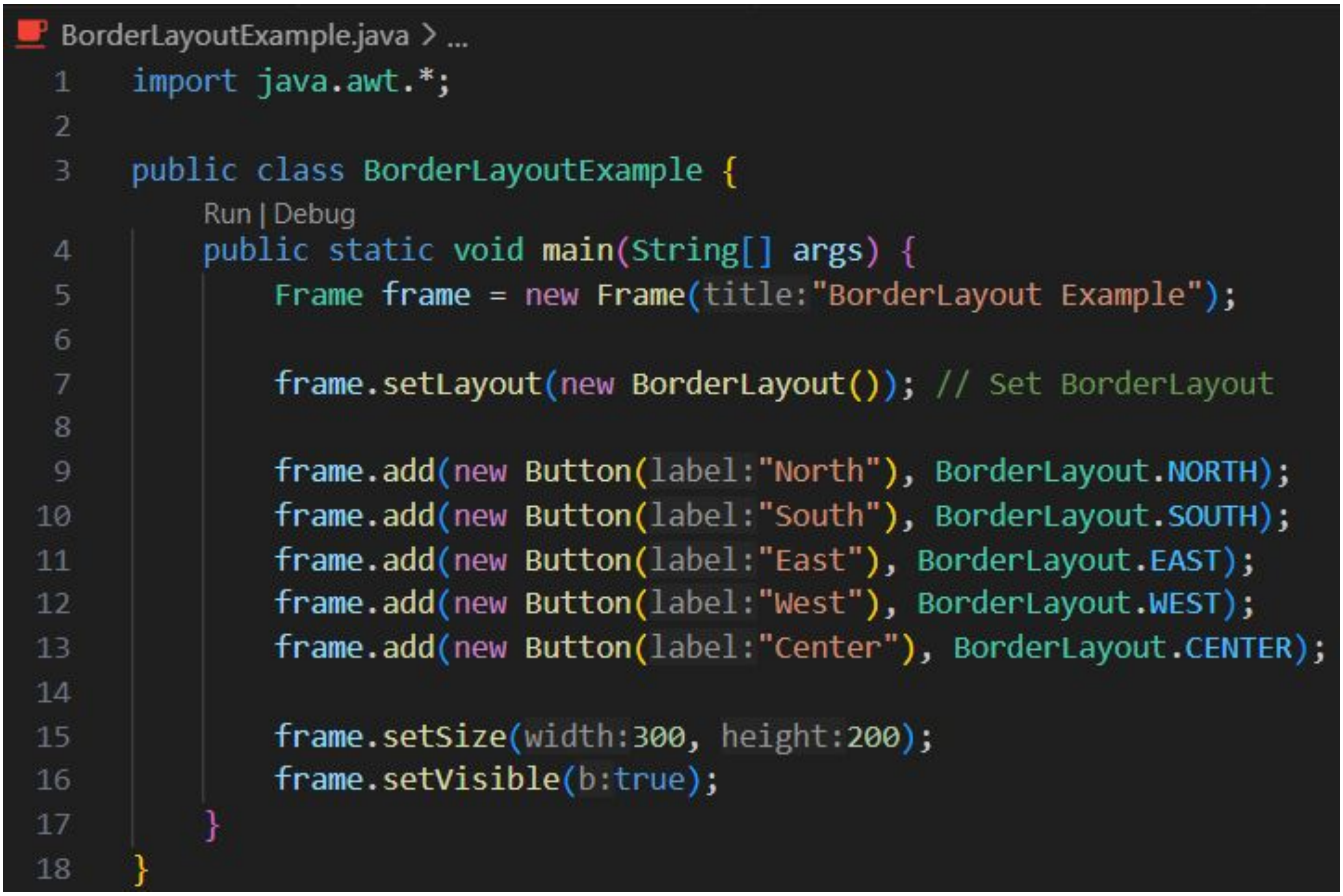
A layout manager is a tool that helps organize where the components (like buttons, text fields, etc.) go inside a container (like a window or panel). Instead of manually placing each component at an exact spot, the layout manager decides where the components should go based on certain rules

➢ Types of Layout Managers



1) **Border Layout** : BorderLayout divides the container into five regions: **North, South, East, West, and Center.** Components added to these regions will automatically resize to fill their respective areas.

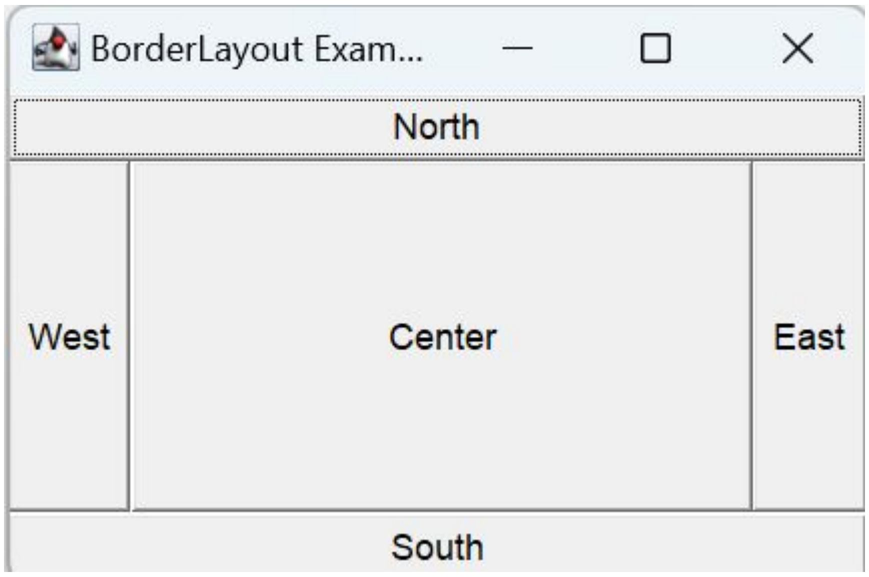
➢ Example code of Border Layout



BorderLayoutExample.java > ...

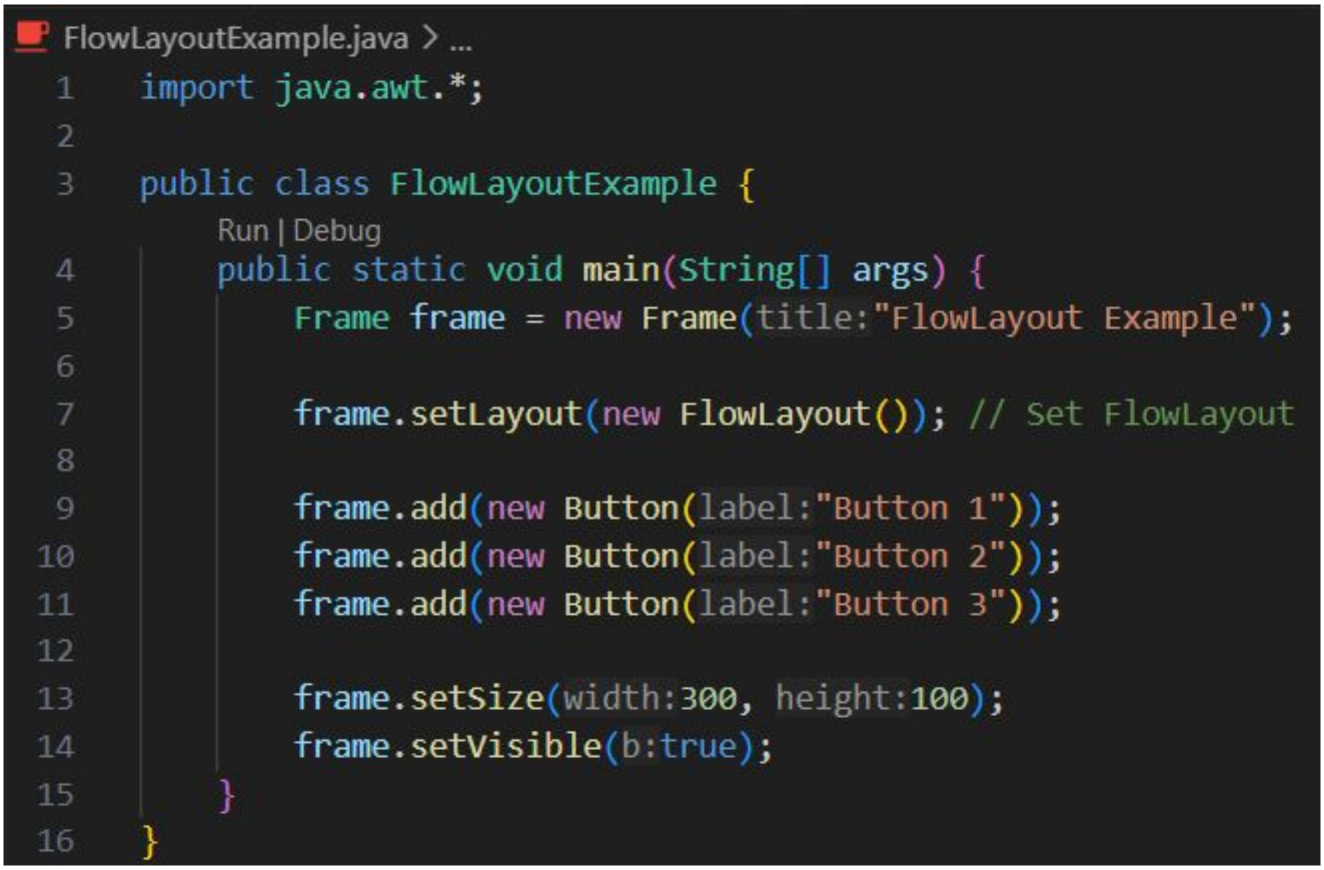
```java
import java.awt.*;

public class BorderLayoutExample {
    Run | Debug
    public static void main(String[] args) {
        Frame frame = new Frame(title:"BorderLayout Example");

        frame.setLayout(new BorderLayout()); // Set BorderLayout

        frame.add(new Button(label:"North"), BorderLayout.NORTH);
        frame.add(new Button(label:"South"), BorderLayout.SOUTH);
        frame.add(new Button(label:"East"), BorderLayout.EAST);
        frame.add(new Button(label:"West"), BorderLayout.WEST);
        frame.add(new Button(label:"Center"), BorderLayout.CENTER);

        frame.setSize(width:300, height:200);
        frame.setVisible(b:true);
    }
}
```

➔ Output



2) Flow Layout : FlowLayout arranges components in a left-to-right flow, wrapping to the next line when there's not enough space. Components are added in the order they are placed, and their size adjusts based on the container's size

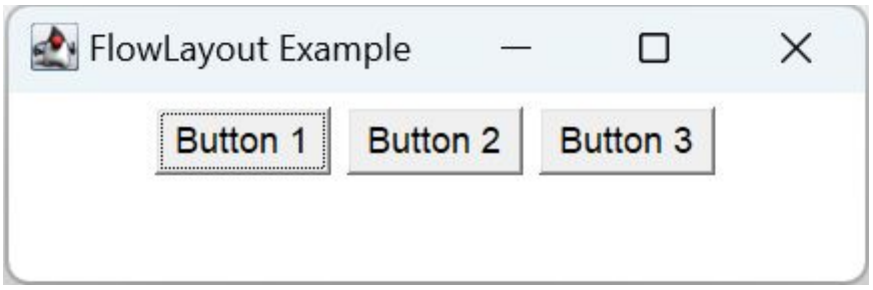
➢ Example code of Flow Layout



FlowLayoutExample.java > ...

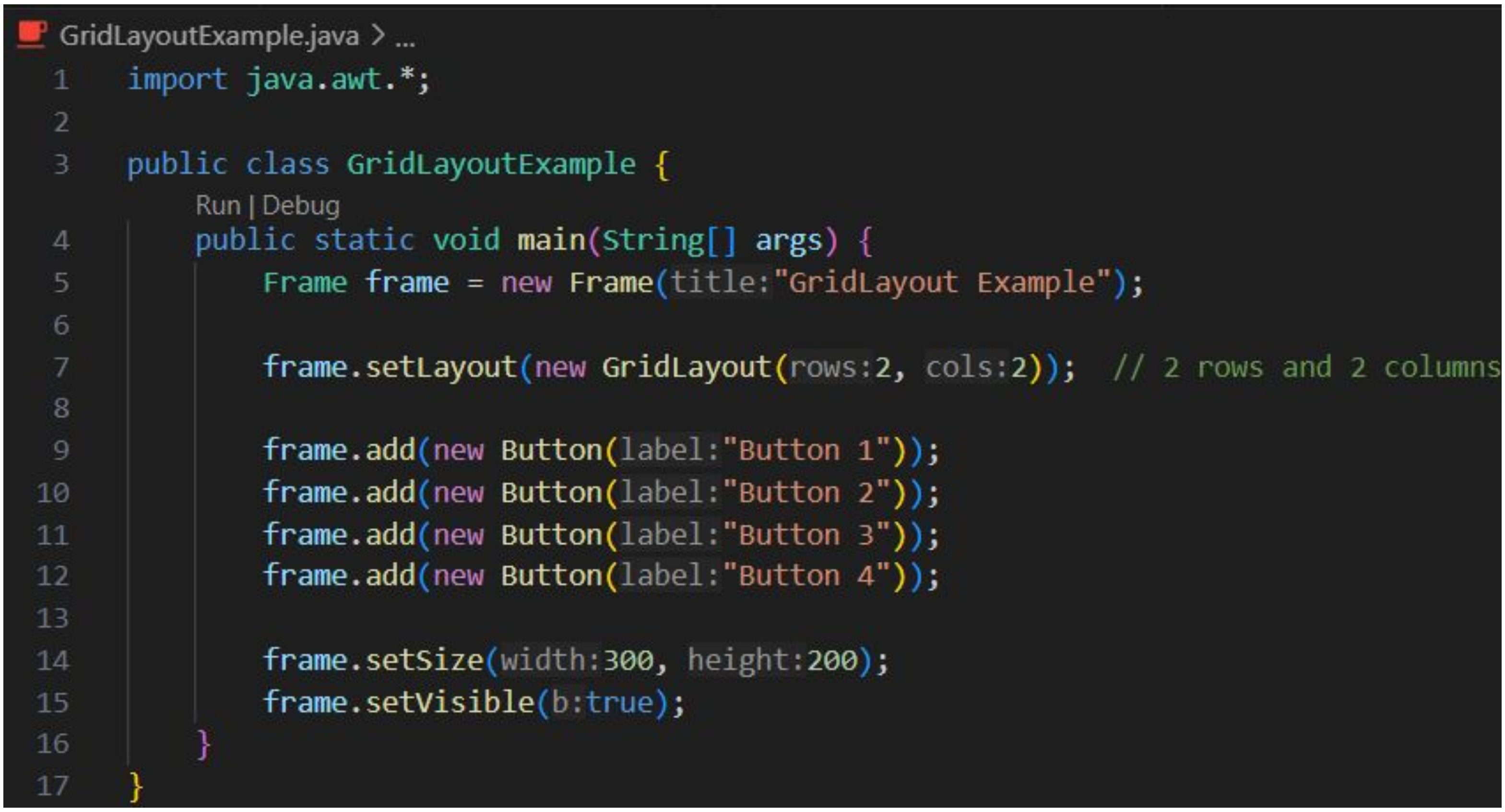
```java
import java.awt.*;

public class FlowLayoutExample {
    Run | Debug
    public static void main(String[] args) {
        Frame frame = new Frame(title:"FlowLayout Example");

        frame.setLayout(new FlowLayout()); // Set FlowLayout

        frame.add(new Button(label:"Button 1"));
        frame.add(new Button(label:"Button 2"));
        frame.add(new Button(label:"Button 3"));

        frame.setSize(width:300, height:100);
        frame.setVisible(b:true);
    }
}
```

➔ Output



3) Grid Layout : GridLayout arranges components in a grid with a specified number of rows and columns, where each cell has the same size. Components are placed in the grid, row by row, filling each cell in order

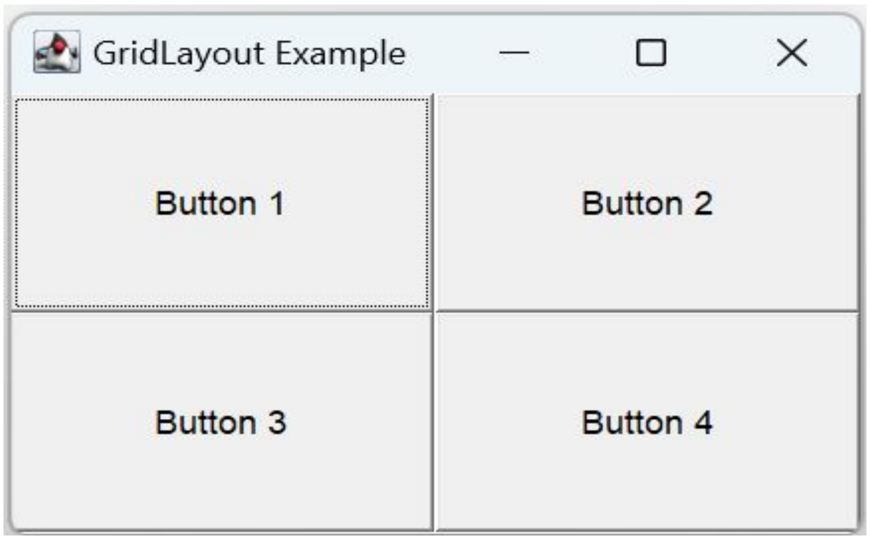
➢ Example code of Grid Layout



GridLayoutExample.java > ...

```java
import java.awt.*;

public class GridLayoutExample {
    Run | Debug
    public static void main(String[] args) {
        Frame frame = new Frame(title:"GridLayout Example");

        frame.setLayout(new GridLayout(rows:2, cols:2));  // 2 rows and 2 columns

        frame.add(new Button(label:"Button 1"));
        frame.add(new Button(label:"Button 2"));
        frame.add(new Button(label:"Button 3"));
        frame.add(new Button(label:"Button 4"));

        frame.setSize(width:300, height:200);
        frame.setVisible(b:true);
    }
}
```

➔ Output



❖ Event Handling

Event handling in Java refers to the process of responding to user actions, such as button clicks, keyboard input, or mouse movements, that trigger events. These events are captured and processed by **event listeners**, which are specific objects designed to handle particular types of events.

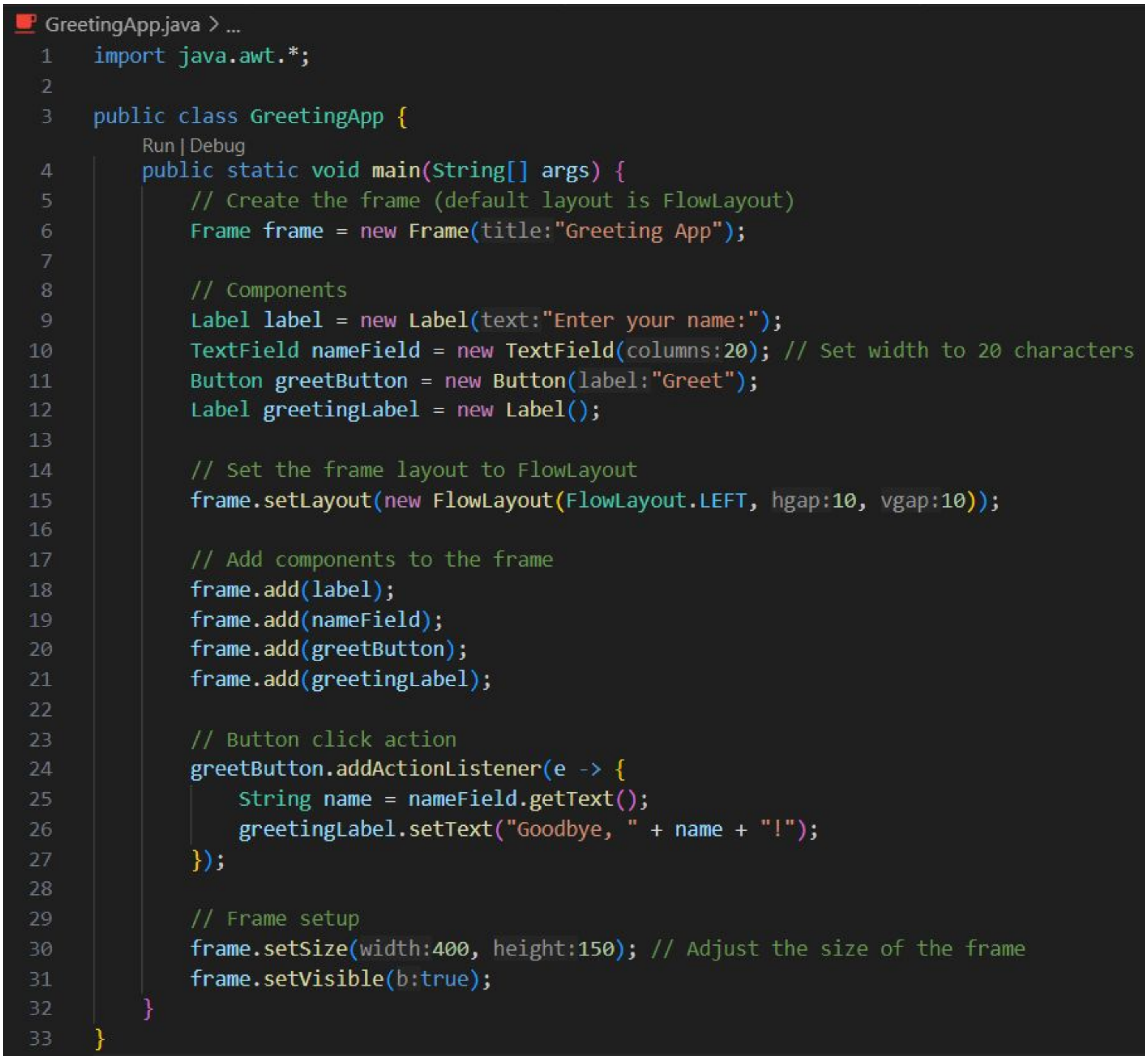
➢ Delegation model in Java

Java uses the **Delegation Model** for event handling, which means that the event source delegates the responsibility of processing the event to a listener object. In this model, the **event source** is not responsible for handling the event directly. Instead, it delegates the event to an event listener that is registered to handle the event

➔ How Java Uses the Delegation Model:

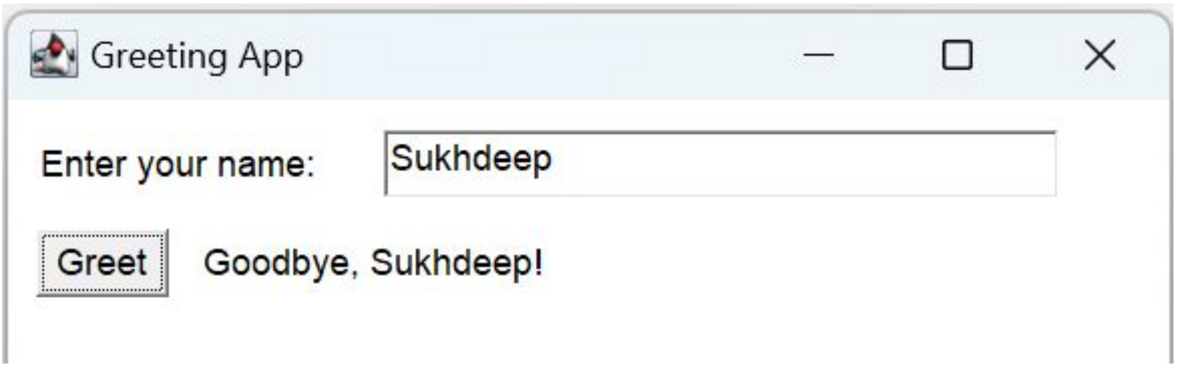
1) **Event Source :** Components like buttons, text fields, and other interactive elements generate events when users interact with them
2) **Event Listener**: An event listener in Java is an object that waits for specific events to happen, such as a button being clicked
3) **Delegation of Event Handling :** When an event is triggered (e.g., a button is clicked), the event source delegates the responsibility of handling the event to the registered event listener. The listener's corresponding method is called, and the listener performs the required action

➢ Example code of Event Handling


GreetingApp.java > ...

```java
import java.awt.*;

public class GreetingApp {
    Run | Debug
    public static void main(String[] args) {
        // Create the frame (default layout is FlowLayout)
        Frame frame = new Frame(title:"Greeting App");

        // Components
        Label label = new Label(text:"Enter your name:");
        TextField nameField = new TextField(columns:20); // Set width to 20 characters
        Button greetButton = new Button(label:"Greet");
        Label greetingLabel = new Label();

        // Set the frame layout to FlowLayout
        frame.setLayout(new FlowLayout(FlowLayout.LEFT, hgap:10, vgap:10));

        // Add components to the frame
        frame.add(label);
        frame.add(nameField);
        frame.add(greetButton);
        frame.add(greetingLabel);

        // Button click action
        greetButton.addActionListener(e -> {
            String name = nameField.getText();
            greetingLabel.setText("Goodbye, " + name + "!");
        });

        // Frame setup
        frame.setSize(width:400, height:150); // Adjust the size of the frame
        frame.setVisible(b:true);
    }
}
```

➔ Output


Greeting App
Enter your name:
Sukhdeep
Greet
Goodbye, Sukhdeep!