DHRUVA KUMAR S

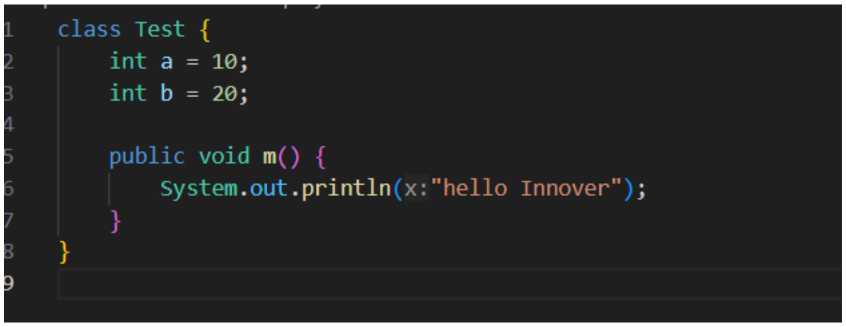
Assignment1

Date:09 June 2023

Class:

Class is a collection of objects and it doesn't It doesn't take any space on memory, Class is also called blueprint or logical entity

- A class is defined using the class keyword followed by the class name.



```
class Test {
    int a = 10;
    int b = 20;

    public void m() {
        System.out.println(x:"hello Innover");
    }
}
```

Fig1: Class

- **class**: Declares a new class.
- **Test**: The name of the class.
- **int a, int b**: Instance variables with initial values (10 and 20 respectively).
- **public**: Access modifier for the method, indicating it can be accessed from other classes.
- **void**: Indicates the method does not return a value.
- **m()**: Method name.
- **System.out.println("hello Innover");**: Prints a message to the console within the method.

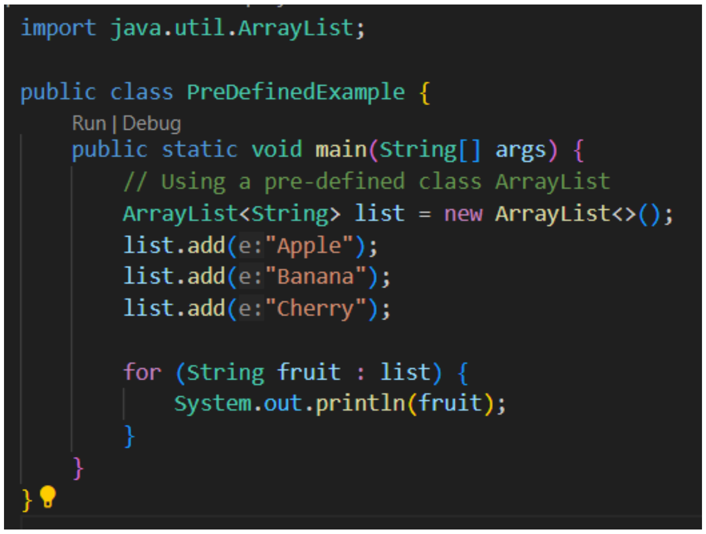
There are 2 types of Class:

Pre-Defined Classes

Pre-defined classes are classes that are part of the Java standard library. These classes are provided by the Java Development Kit (JDK) and can be readily used without the need to define them explicitly.

Examples:

- **String:** Represents a sequence of characters.
- **ArrayList**: Provides a resizable array, which can be found in the java.util package.
- **HashMap:** Implements a hash table, which maps keys to values, and is also found in the java.util package.



```java
import java.util.ArrayList;

public class PreDefinedExample {
    Run | Debug
    public static void main(String[] args) {
        // Using a pre-defined class ArrayList
        ArrayList<String> list = new ArrayList<>();
        list.add(e:"Apple");
        list.add(e:"Banana");
        list.add(e:"Cherry");

        for (String fruit : list) {
            System.out.println(fruit);
        }
    }
}
```

Fig2: Pre-defined Class "ArrayList"

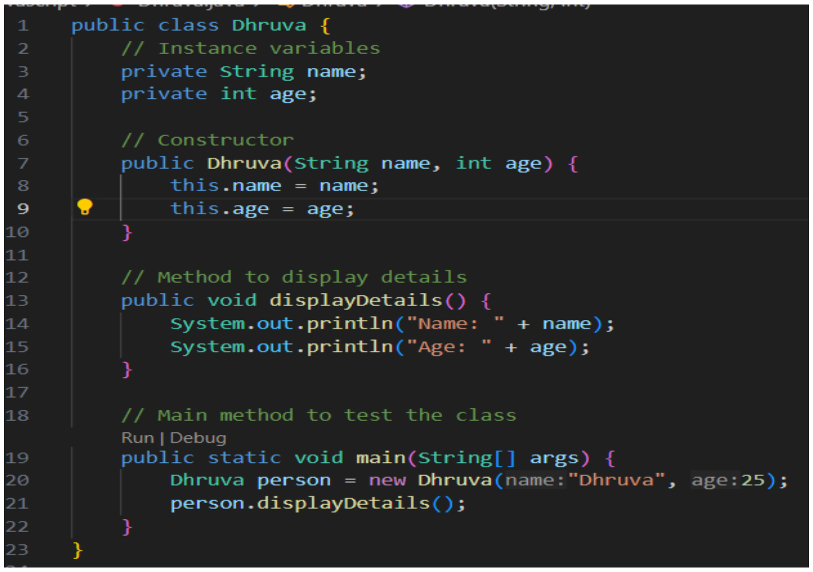
User-Defined Classes

User-defined classes are classes that are created by the programmer to fulfill specific requirements of a particular application.

Examples:

Any class defined by the programmer that is not part of the Java standard library such as

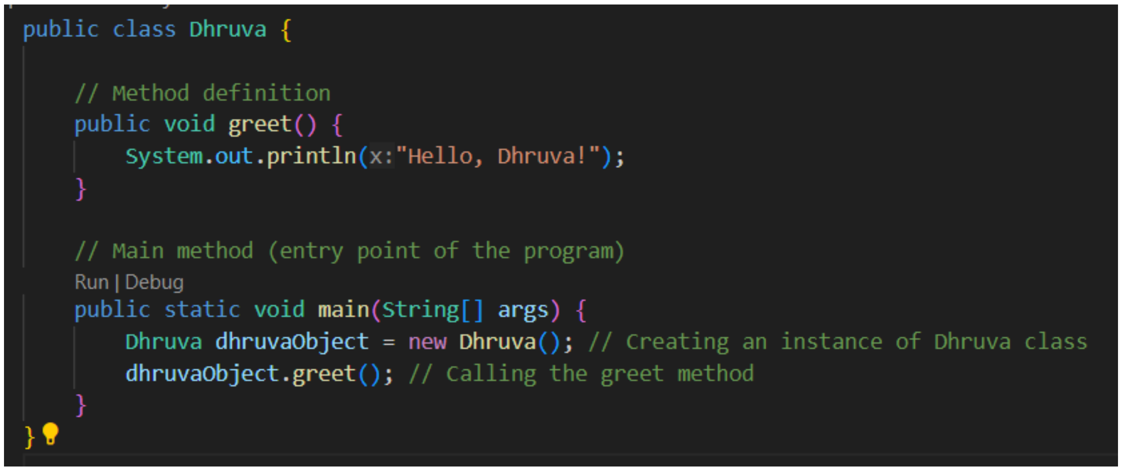
Dhruva, Car, Person, Employee etc.



```java
public class Dhruva {
    // Instance variables
    private String name;
    private int age;

    // Constructor
    public Dhruva(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to display details
    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    // Main method to test the class
    Run | Debug
    public static void main(String[] args) {
        Dhruva person = new Dhruva(name:"Dhruva", age:25);
        person.displayDetails();
    }
}
```

Fig3: User defined class as "Dhruva "

Method: In Java, a method is a block of code within a class that performs a specific task.



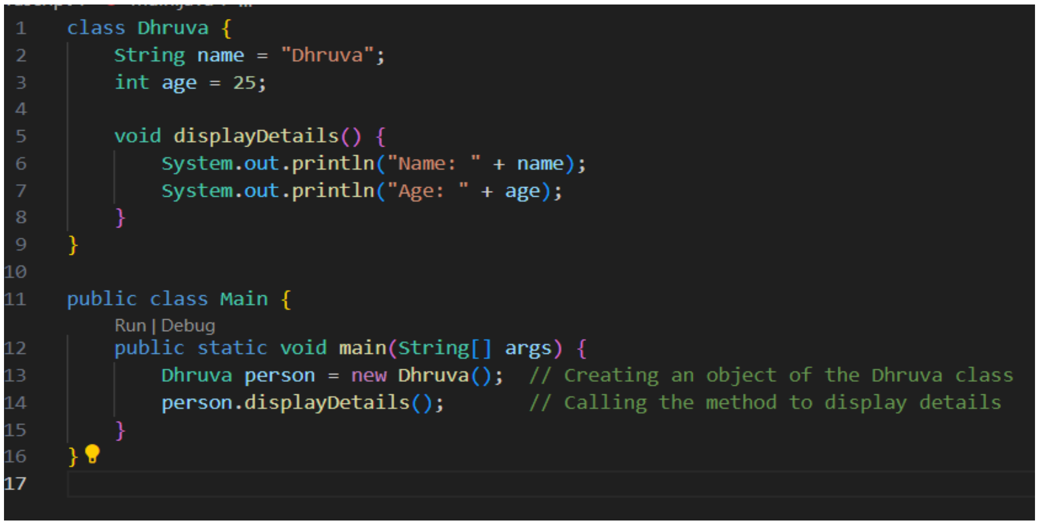
```java
public class Dhruva {

    // Method definition
    public void greet() {
        System.out.println(x:"Hello, Dhruva!");
    }

    // Main method (entry point of the program)
    Run | Debug
    public static void main(String[] args) {
        Dhruva dhruvaObject = new Dhruva(); // Creating an instance of Dhruva class
        dhruvaObject.greet(); // Calling the greet method
    }
}
```

Fig4: Method

- **greet()** is a method defined within the Dhruva class that prints "Hello, Dhruva!" to the console.
- The **main()** method is the entry point of the program, creating an instance of Dhruva and calling its greet() method.

Object:

An object is an instance of a class. It is a concrete entity based on the class blueprint that can be used to access the properties and methods defined by the class.



```java
class Dhruva {
    String name = "Dhruva";
    int age = 25;

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

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Dhruva person = new Dhruva();  // Creating an object of the Dhruva class
        person.displayDetails();       // Calling the method to display details
    }
}
```

Fig5: Object “person” Creation

- In the main method of the Main class, an object of the Dhruva class is created with Dhruva **person = new Dhruva();.**
- The **displayDetails()** method is called on the **person** object to print its details.

Access Modifiers

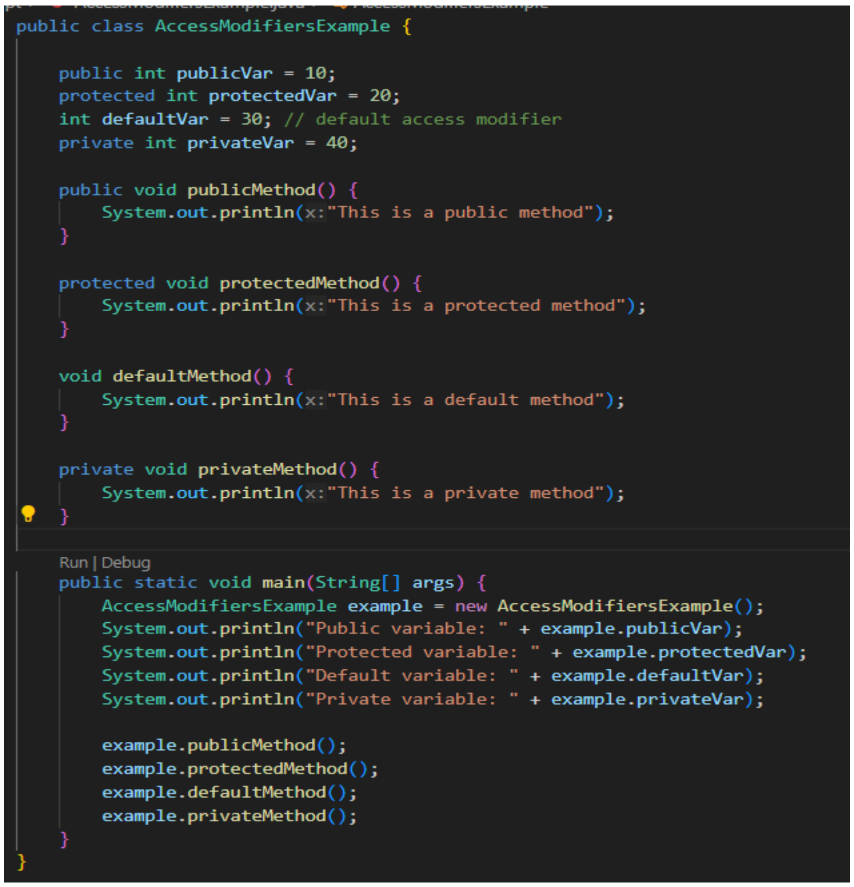
In Java, access modifiers control the visibility of classes, methods, and variables. There are four types of access modifiers:

public: Accessible from anywhere.

default: Accessible within the same package.

protected: Accessible within the same package, same class and subclasses.

private: Accessible only within the same class.



```java
public class AccessModifiersExample {

    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30; // default access modifier
    private int privateVar = 40;

    public void publicMethod() {
        System.out.println(x:"This is a public method");
    }

    protected void protectedMethod() {
        System.out.println(x:"This is a protected method");
    }

    void defaultMethod() {
        System.out.println(x:"This is a default method");
    }

    private void privateMethod() {
        System.out.println(x:"This is a private method");
    }

    Run | Debug
    public static void main(String[] args) {
        AccessModifiersExample example = new AccessModifiersExample();
        System.out.println("Public variable: " + example.publicVar);
        System.out.println("Protected variable: " + example.protectedVar);
        System.out.println("Default variable: " + example.defaultVar);
        System.out.println("Private variable: " + example.privateVar);

        example.publicMethod();
        example.protectedMethod();
        example.defaultMethod();
        example.privateMethod();
    }
}
```

Fig6: Access Modifiers

In the above fig6:

- publicVar, publicMethod(): Accessible from anywhere.
- protectedVar, protectedMethod(): Accessible within the same package and subclasses.
- defaultVar, defaultMethod(): Accessible within the same package.
- privateVar, privateMethod(): Accessible only within the same class.

Assignment 2

Date:10 June 2024

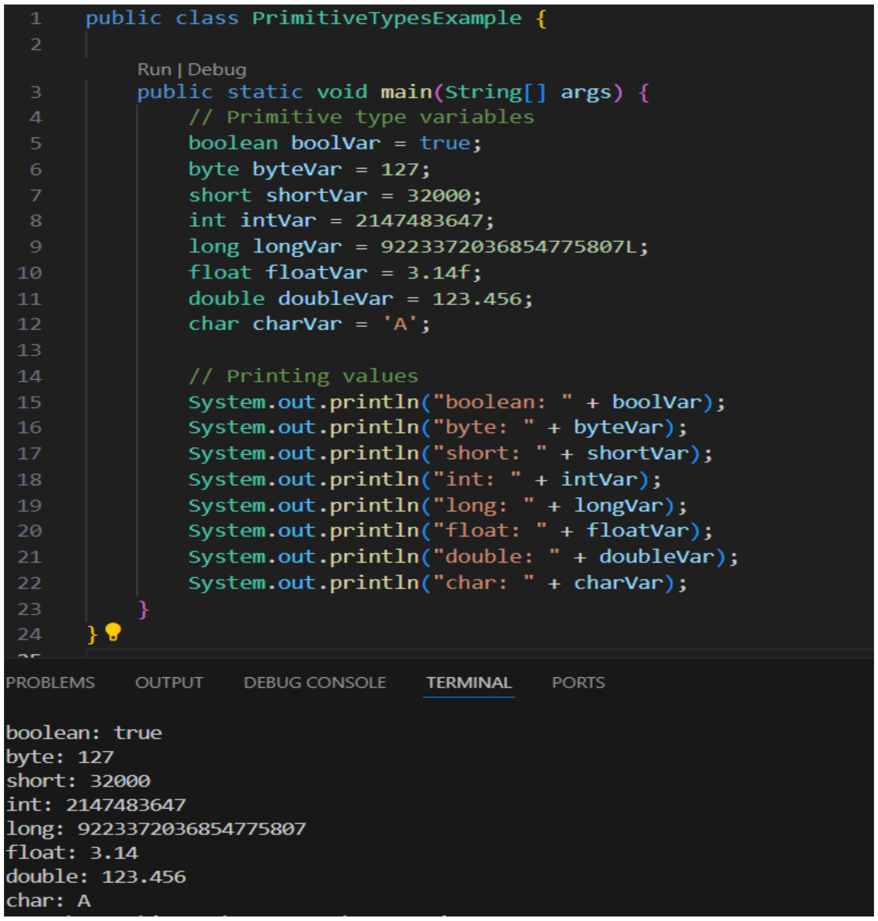
Keywords:

Keywords in Java are reserved words that have predefined meanings and purposes within the language's syntax and semantics. They cannot be used as identifiers. Because they are reserved for specific programming tasks and functionalities.

The set of keywords that are reserved words that cannot be used as variables, methods, classes, or any other identifiers:

Primitive Types:

- **boolean**: A data type that can store true or false values.
- **byte**: A data type that can store whole numbers from -128 to 127.
- **short**: A data type that can store whole numbers from -32768 to 32767.
- **int**: A data type that can store whole numbers from -2147483648 to 2147483647.
- **long**: A data type that can store larger whole numbers from -9223372036854775808 to 9223372036854775807.
- **float**: A data type that can store fractional numbers, useful when you need a fractional component.
- **double**: A data type that can store fractional numbers, useful when you need a large range of values.
- **char**: A data type that is used to store a single character.



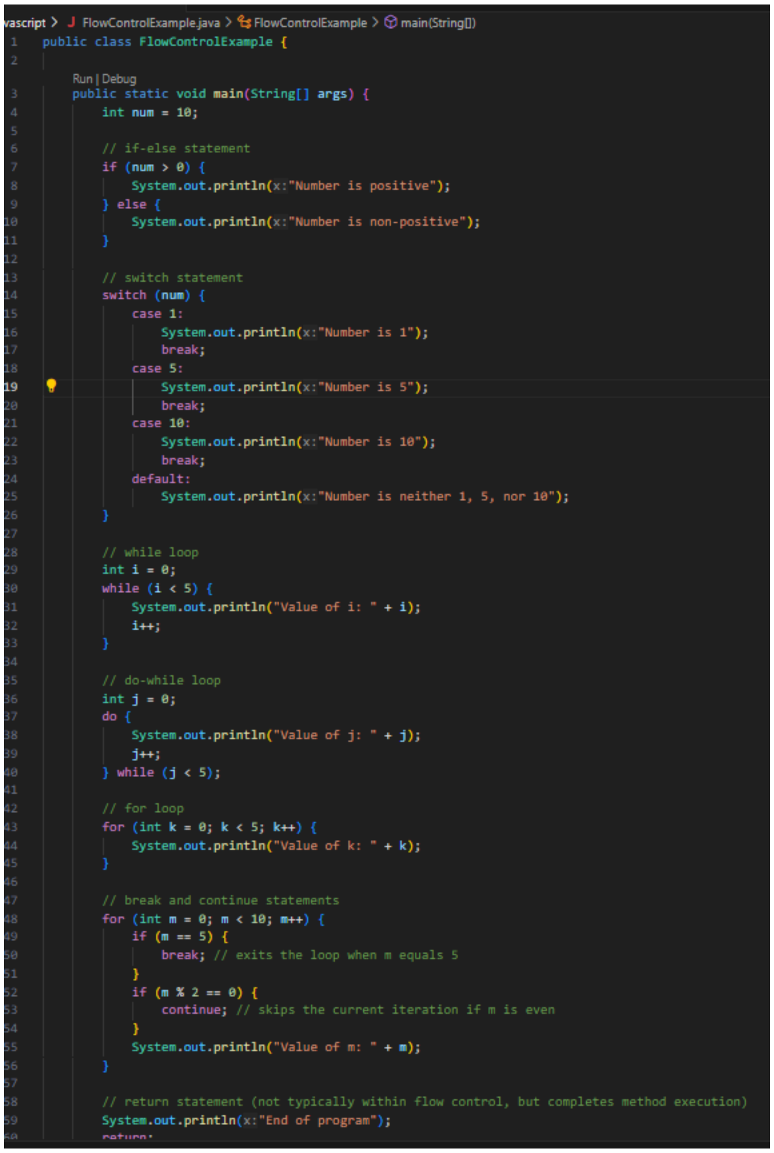
```java
public class PrimitiveTypesExample {

    Run | Debug
    public static void main(String[] args) {
        // Primitive type variables
        boolean boolVar = true;
        byte byteVar = 127;
        short shortVar = 32000;
        int intVar = 2147483647;
        long longVar = 9223372036854775807L;
        float floatVar = 3.14f;
        double doubleVar = 123.456;
        char charVar = 'A';

        // Printing values
        System.out.println("boolean: " + boolVar);
        System.out.println("byte: " + byteVar);
        System.out.println("short: " + shortVar);
        System.out.println("int: " + intVar);
        System.out.println("long: " + longVar);
        System.out.println("float: " + floatVar);
        System.out.println("double: " + doubleVar);
        System.out.println("char: " + charVar);
    }
}
```

```
boolean: true
byte: 127
short: 32000
int: 2147483647
long: 9223372036854775807
float: 3.14
double: 123.456
char: A
```

Fig1: Primitive Type keywords

Flow Control:

- **if**: Makes a conditional statement.
- **else**: Used in conjunction with if to execute a block of code if the condition is false.
- **switch**: Selects one of many code blocks to be executed.
- **case**: Marks a block of code in switch statements.
- **default**: Specifies the default block of code in a switch statement.
- **while**: Creates a while loop.
- **do**: Used in conjunction with while to create a do-while loop.
- **for**: Creates a for loop.
- **break**: Breaks out of a loop or a switch block.
- **continue**: Skips the current iteration of a loop and continues with the next iteration.
- **return**: Finishes the execution of a method and can optionally return a value.



vascript > FlowControlExample.java > FlowControlExample > main(String[])

```java
public class FlowControlExample {

    Run | Debug
    public static void main(String[] args) {
        int num = 10;

        // if-else statement
        if (num > 0) {
            System.out.println(x:"Number is positive");
        } else {
            System.out.println(x:"Number is non-positive");
        }

        // switch statement
        switch (num) {
            case 1:
                System.out.println(x:"Number is 1");
                break;
            case 5:
                System.out.println(x:"Number is 5");
                break;
            case 10:
                System.out.println(x:"Number is 10");
                break;
            default:
                System.out.println(x:"Number is neither 1, 5, nor 10");
        }

        // while loop
        int i = 0;
        while (i < 5) {
            System.out.println("Value of i: " + i);
            i++;
        }

        // do-while loop
        int j = 0;
        do {
            System.out.println("Value of j: " + j);
            j++;
        } while (j < 5);

        // for loop
        for (int k = 0; k < 5; k++) {
            System.out.println("Value of k: " + k);
        }

        // break and continue statements
        for (int m = 0; m < 10; m++) {
            if (m == 5) {
                break; // exits the loop when m equals 5
            }
            if (m % 2 == 0) {
                continue; // skips the current iteration if m is even
            }
            System.out.println("Value of m: " + m);
        }

        // return statement (not typically within flow control, but completes method execution)
        System.out.println(x:"End of program");
        return;
```

Fig2: Flow control keywords

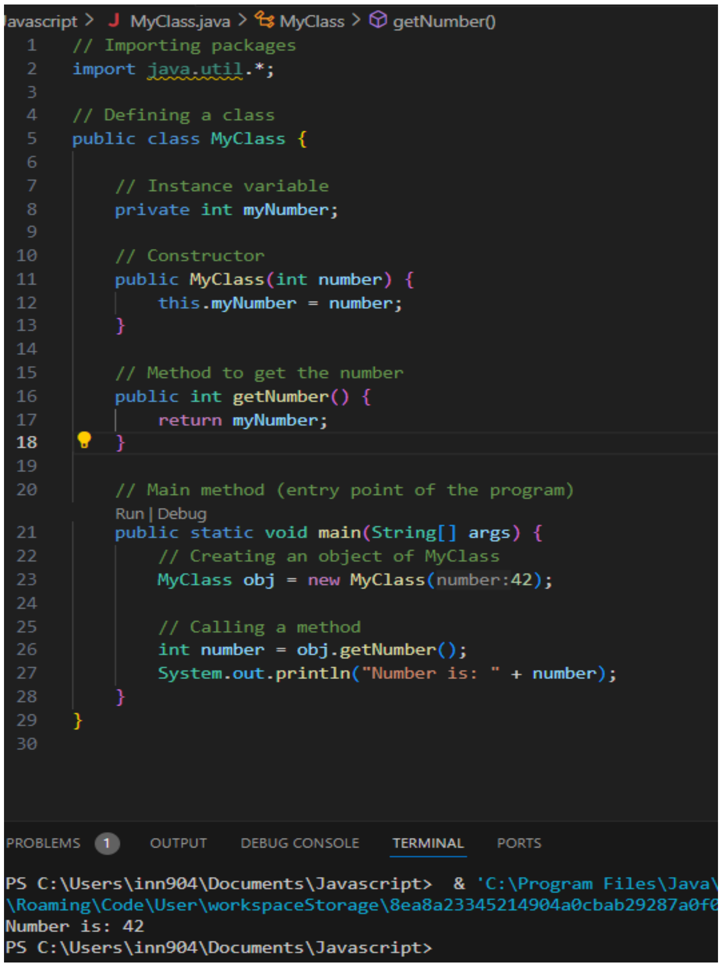
Modifiers:

- **public**: An access modifier used for classes, methods, and attributes, making them accessible by any other class.
- **private**: An access modifier used for attributes, methods, and constructors, making them only accessible within the declared class.
- **protected**: An access modifier used for attributes, methods, and constructors, making them accessible in the same package and subclasses.
- **static**: A non-access modifier used for methods and attributes. Static methods/attributes can be accessed without creating an object of a class.
- **final**: A non-access modifier used for classes, attributes, and methods, which makes them non-changeable (impossible to inherit or override).

- **abstract**: A non-access modifier used for classes and methods. An abstract class cannot be instantiated and can only be used as a superclass.
- **synchronized**: A non-access modifier, which specifies that methods can only be accessed by one thread at a time.
- **volatile**: Indicates that an attribute is not cached thread-locally, and is always read from the "main memory".
- **transient**: Used to ignore an attribute when serializing an object.

Class, Method, and Package:

- **class**: Defines a class.
- **interface**: Used to declare a special type of class that only contains abstract methods.
- **extends**: Indicates that a class is inherited from another class.
- **implements**: Implements an interface.
- **package**: Declares a package.
- **import**: Used to import a package, class, or interface.



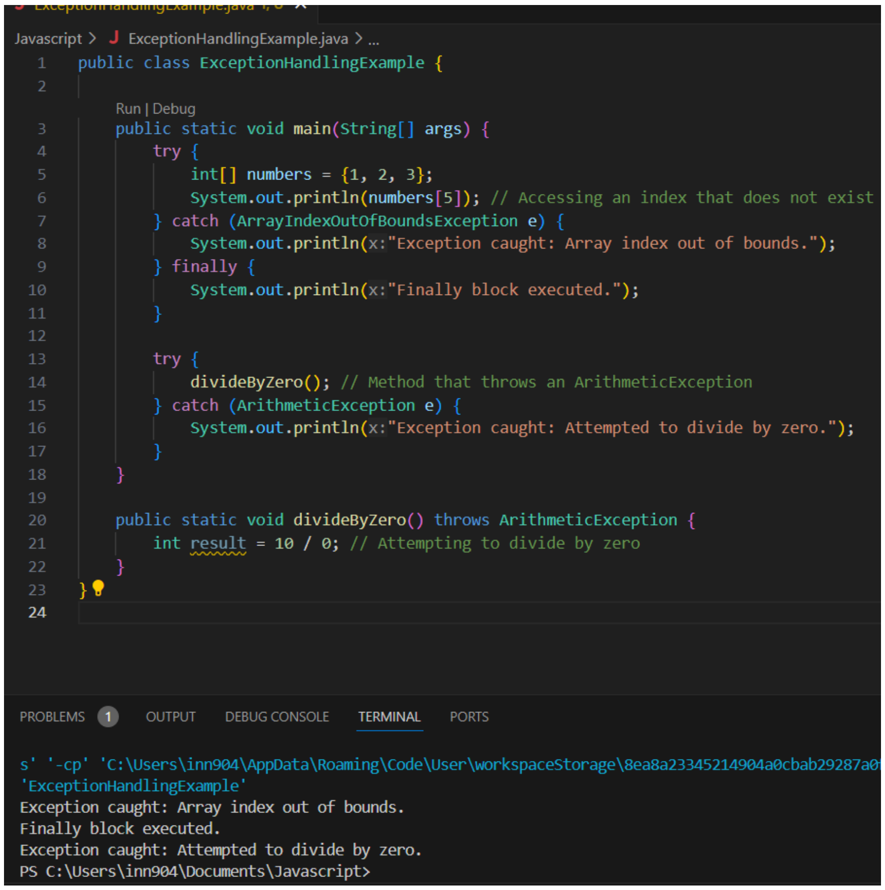
```java
// Importing packages
import java.util.*;

// Defining a class
public class MyClass {

    // Instance variable
    private int myNumber;

    // Constructor
    public MyClass(int number) {
        this.myNumber = number;
    }

    // Method to get the number
    public int getNumber() {
        return myNumber;
    }

    // Main method (entry point of the program)
    public static void main(String[] args) {
        // Creating an object of MyClass
        MyClass obj = new MyClass(number:42);

        // Calling a method
        int number = obj.getNumber();
        System.out.println("Number is: " + number);
    }
}
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\
\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab29287a0f
Number is: 42
PS C:\Users\inn904\Documents\Javascript>
```

Fig3: Keywords of Class, Method, and Package.

Exception Handling:

- **try**: Creates a try...catch statement to handle exceptions.
- **catch**: Catches exceptions generated by try statements.
- **finally**: A block of code that will be executed no matter if there is an exception or not.
- **throw**: Creates a custom error.
- **throws**: Indicates what exceptions may be thrown by a method.



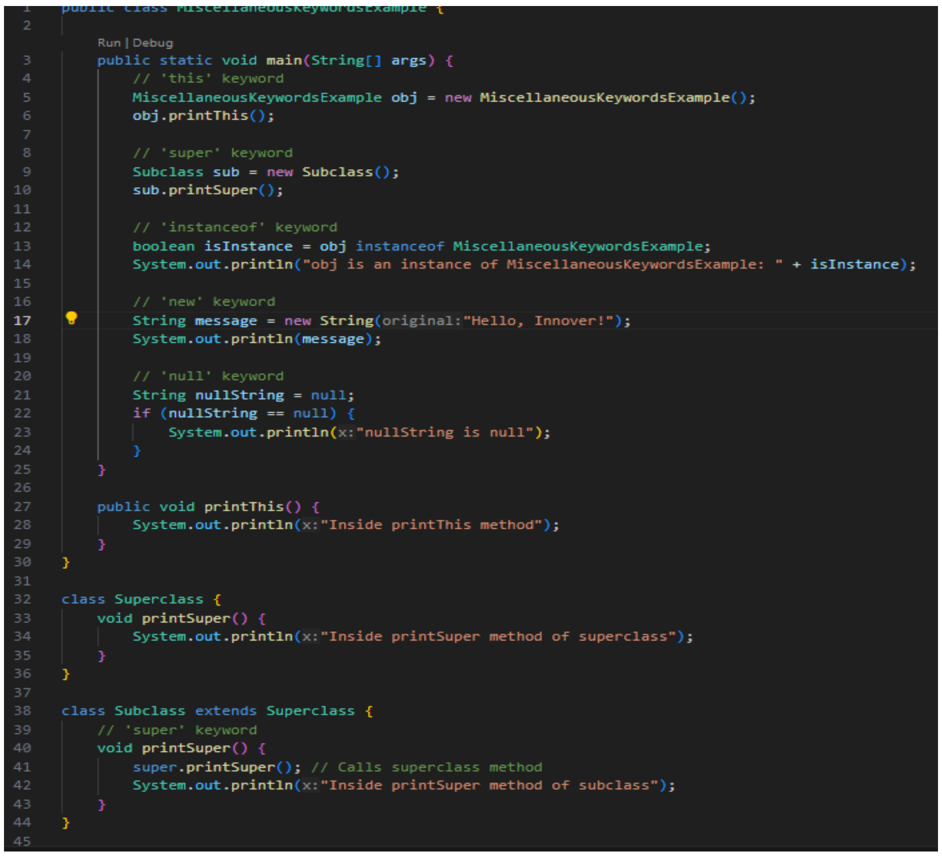
```java
public class ExceptionHandlingExample {

    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]); // Accessing an index that does not exist
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println(x:"Exception caught: Array index out of bounds.");
        } finally {
            System.out.println(x:"Finally block executed.");
        }

        try {
            divideByZero(); // Method that throws an ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println(x:"Exception caught: Attempted to divide by zero.");
        }
    }

    public static void divideByZero() throws ArithmeticException {
        int result = 10 / 0; // Attempting to divide by zero
    }
}
```

```
s' '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab29287a0
'ExceptionHandlingExample'
Exception caught: Array index out of bounds.
Finally block executed.
Exception caught: Attempted to divide by zero.
PS C:\Users\inn904\Documents\Javascript>
```

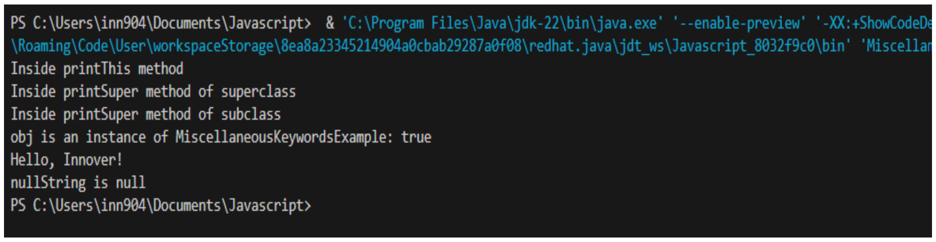
Fig: Exception Handling keywords

Miscellaneous:

- **this**: Refers to the current object in a method or constructor.
- **super**: Refers to superclass (parent) objects.
- **instanceof**: Checks whether an object is an instance of a specific class or an interface.
- **new**: Creates new objects.



```
public class MiscellaneousKeywordsExample {

    Run | Debug
    public static void main(String[] args) {
        // 'this' keyword
        MiscellaneousKeywordsExample obj = new MiscellaneousKeywordsExample();
        obj.printThis();

        // 'super' keyword
        Subclass sub = new Subclass();
        sub.printSuper();

        // 'instanceof' keyword
        boolean isInstance = obj instanceof MiscellaneousKeywordsExample;
        System.out.println("obj is an instance of MiscellaneousKeywordsExample: " + isInstance);

        // 'new' keyword
        String message = new String(original:"Hello, Innover!");
        System.out.println(message);

        // 'null' keyword
        String nullString = null;
        if (nullString == null) {
            System.out.println(x:"nullString is null");
        }
    }

    public void printThis() {
        System.out.println(x:"Inside printThis method");
    }
}

class Superclass {
    void printSuper() {
        System.out.println(x:"Inside printSuper method of superclass");
    }
}

class Subclass extends Superclass {
    // 'super' keyword
    void printSuper() {
        super.printSuper(); // Calls superclass method
        System.out.println(x:"Inside printSuper method of subclass");
    }
}
```



```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java.exe' '--enable-preview' '-XX:+ShowCodeDe
\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab29287a0f08\redhat.java\jdt_ws\Javascript_8032f9c0\bin' 'Miscellan
Inside printThis method
Inside printSuper method of superclass
Inside printSuper method of subclass
obj is an instance of MiscellaneousKeywordsExample: true
Hello, Innover!
nullString is null
PS C:\Users\inn904\Documents\Javascript>
```

Fig: Miscellaneous Keywords

Identifiers:
In Java, identifiers are names given to various program elements such as variables, methods, classes, packages, and interfaces. These names serve as labels that the programmer uses to refer to these elements in the code. Identifiers help in making code readable and understandable.

Rules for Identifiers in Java:

- **Valid Characters**:
  Must start with a letter (a-z, A-Z), underscore (_) or dollar sign ($).
  After the first character, identifiers can also contain digits (0-9).

- **Length**:
  No limit on the length of identifiers, but meaningful and concise names are encouraged for readability.

- **Case Sensitivity**:
  Java is case-sensitive, meaning myVariable and MyVariable are different identifiers.

- **Reserved Words**:
  Cannot be used as identifiers. Examples include keywords like int, class, if, etc.

- **Whitespace**:
  No spaces or special characters are allowed within identifiers.

Examples of Valid Identifiers:

- age
- _value
- $totalAmount
- firstName
- calculateArea
- MAX_VALUE
- isValid123

Examples of Invalid Identifiers:

- 2ndNumber (starts with a digit)
- total amount (contains a space)
- @username (contains a special character)
- if (reserved word)
- class (reserved word)

Assignment 3

Date:11 June 2024

OOPs: Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects., the main purpose of OOP is to deal with real world entity using programming language.

Key Concepts of OOP in Java:

Class:

A blueprint for creating objects. It defines a type of object according to the properties and methods it should contain.

Object:

An instance of a class. It is created based on the class definition.

Inheritance:

When we construct a new class from existing class in such a way that the new class access all the futures and properties of existing class called inheritance.

Or

Inheritance is one of the core concepts of Object-Oriented Programming (OOP) in Java. It allows one class (subclass or derived class) to inherit the properties and methods of another class (superclass or base class). This promotes code reuse and establishes a natural hierarchy between classes.

- In Java extends keyword is used to perform inheritance
- It provides code reusability
- We can't access private members of class throw inheritance
- A subclass contains all the features of super class so, we should create the object of subclass
- Method overriding only possible throw in a returns

Syntax:

```
class SuperClass {
   // Superclass methods and fields
}

class SubClass extends SuperClass {
   // Subclass methods and fields
```

}

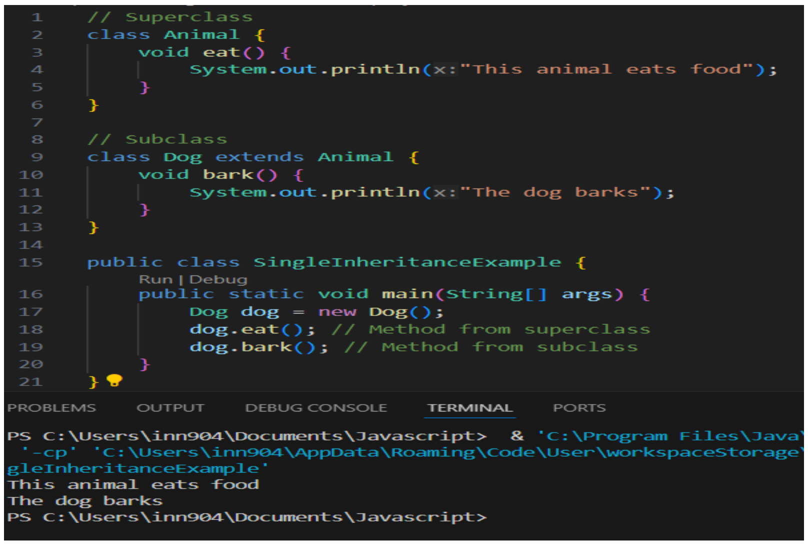
Types of Inheritance in Java

Java supports different types of inheritance:

1. **Single Inheritance**: A class inherits from one superclass.

    - It contains only one super class and only one subclass
      Syntax:

```
// Superclass
class SuperClass {
   // Superclass methods and fields
}

// Subclass
class SubClass extends SuperClass {
   // Subclass methods and fields
}
```



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

// Subclass
class Dog extends Animal {
    void bark() {
        System.out.println(x:"The dog barks");
    }
}

public class SingleInheritanceExample {
    Run | Debug
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Method from superclass
        dog.bark(); // Method from subclass
    }
}
```

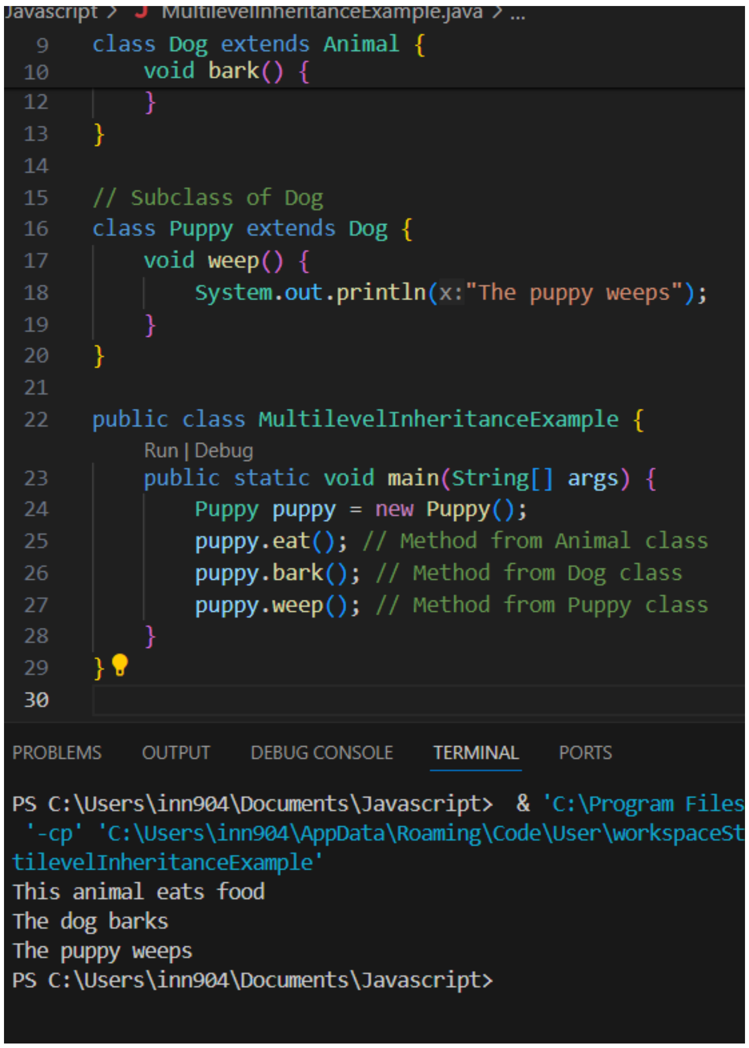
```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java
 '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceStorage
gleInheritanceExample'
This animal eats food
The dog barks
PS C:\Users\inn904\Documents\Javascript>
```

Fig1: Single Inheritance

2. **Multilevel Inheritance**: A class inherits from a superclass, which in turn inherits from another superclass.
   Or
   In multilevel inheritance, a subclass inherits from another subclass, forming a chain of inheritance.

Syntax:

```
// Superclass

class SuperClass {

   // Superclass methods and fields

}


// Subclass

class SubClass extends SuperClass {

   // Subclass methods and fields

}


// Subclass of SubClass

class SubSubClass extends SubClass {

   // SubSubClass methods and fields

}
```



Javascript > MultilevelInheritanceExample.java > ...

```java
class Dog extends Animal {
    void bark() {
    }
}

// Subclass of Dog
class Puppy extends Dog {
    void weep() {
        System.out.println(x:"The puppy weeps");
    }
}

public class MultilevelInheritanceExample {
    Run | Debug
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.eat(); // Method from Animal class
        puppy.bark(); // Method from Dog class
        puppy.weep(); // Method from Puppy class
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

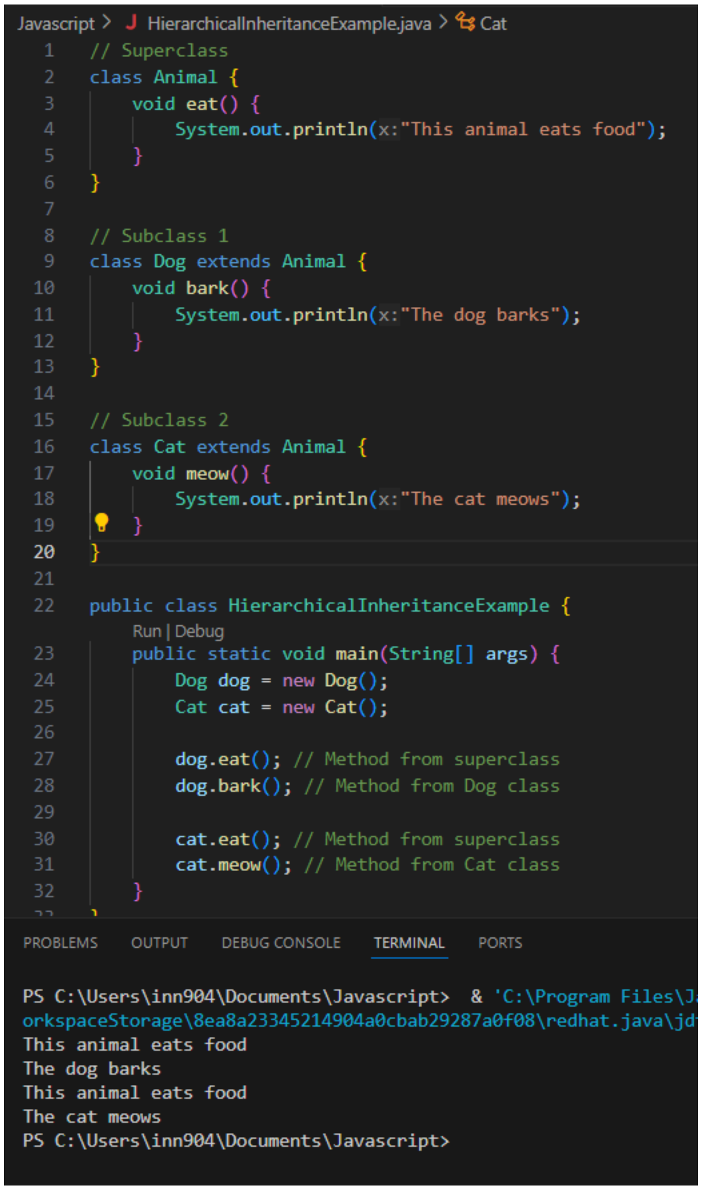
```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files
 '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceSt
tilevelInheritanceExample'
This animal eats food
The dog barks
The puppy weeps
PS C:\Users\inn904\Documents\Javascript>
```

Fig2: Multilevel Inheritance

3. **Hierarchical Inheritance**: In hierarchical inheritance, multiple subclasses inherit from a single superclass.

   Syntax:

   ```
   // Superclass

   class SuperClass {

      // Superclass methods and fields

   }



   // Subclass 1

   class SubClass1 extends SuperClass {

      // SubClass1 methods and fields

   }



   // Subclass 2

   class SubClass2 extends SuperClass {

      // SubClass2 methods and fields

   }
   ```



Javascript > HierarchicalInheritanceExample.java > Cat

```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");
    }
}

public class HierarchicalInheritanceExample {
    Run | Debug
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();

        dog.eat(); // Method from superclass
        dog.bark(); // Method from Dog class

        cat.eat(); // Method from superclass
        cat.meow(); // Method from Cat class
    }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

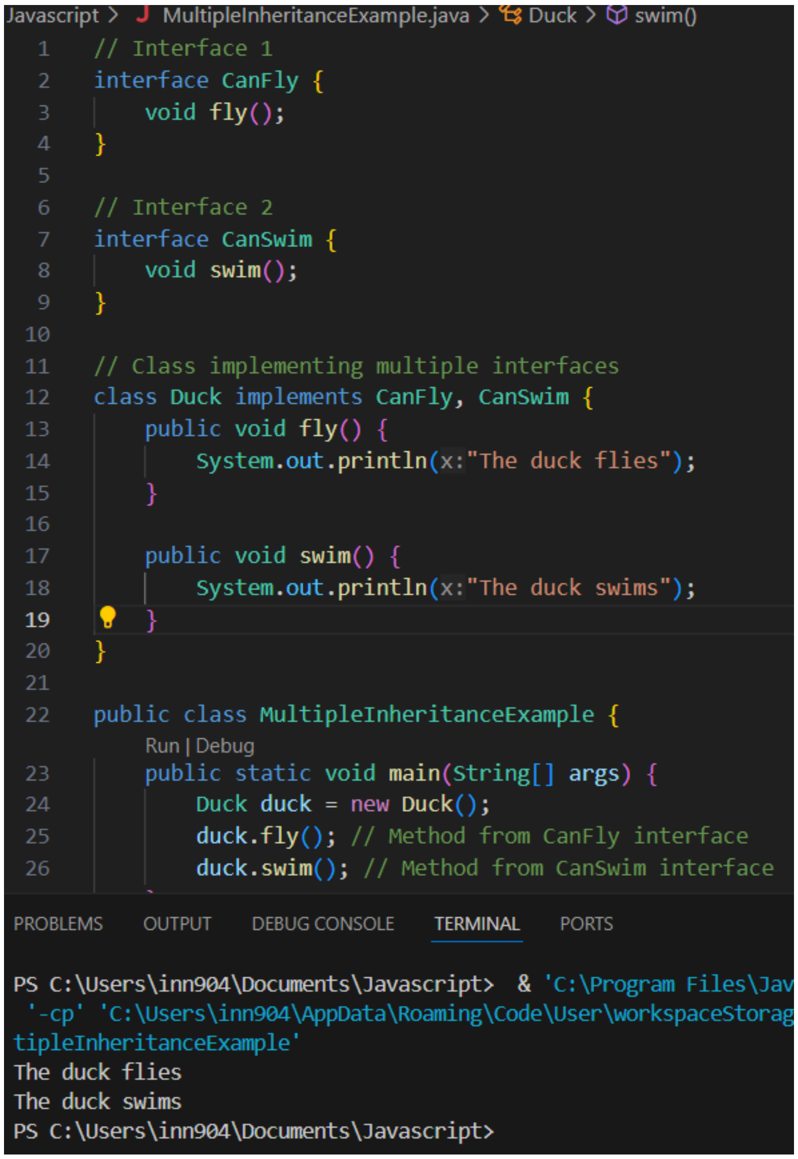
```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\J
orkspaceStorage\8ea8a23345214904a0cbab29287a0f08\redhat.java\jd
This animal eats food
The dog barks
This animal eats food
The cat meows
PS C:\Users\inn904\Documents\Javascript>
```

Fig3: Hierarchical Inheritance

4. **Multiple Inheritance (through interfaces):** Java does not support multiple inheritance through classes. Instead, multiple inheritance is achieved using interfaces.

   Syntax:

   ```
   // Interface 1
   interface Interface1 {
      // Interface1 methods
   }

   // Interface 2
   interface Interface2 {
      // Interface2 methods
   }

   // Class implementing multiple interfaces
   class ImplementingClass implements Interface1, Interface2 {
      // Implementing methods from Interface1 and Interface2
   }
   ```


Javascript > MultipleInheritanceExample.java > Duck > swim()
Run | Debug
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS

```java
// Interface 1
interface CanFly {
    void fly();
}

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

// Class implementing multiple interfaces
class Duck implements CanFly, CanSwim {
    public void fly() {
        System.out.println(x:"The duck flies");
    }

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

public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly(); // Method from CanFly interface
        duck.swim(); // Method from CanSwim interface
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Jav
 '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceStorag
tipleInheritanceExample'
The duck flies
The duck swims
PS C:\Users\inn904\Documents\Javascript>
```

Fig 4: Multiple Inheritance (through interfaces)

Encapsulation:

It refers to the bundling /wrapping of data (variables) and methods (functions) that operate on the data into a single unit, known as a class.

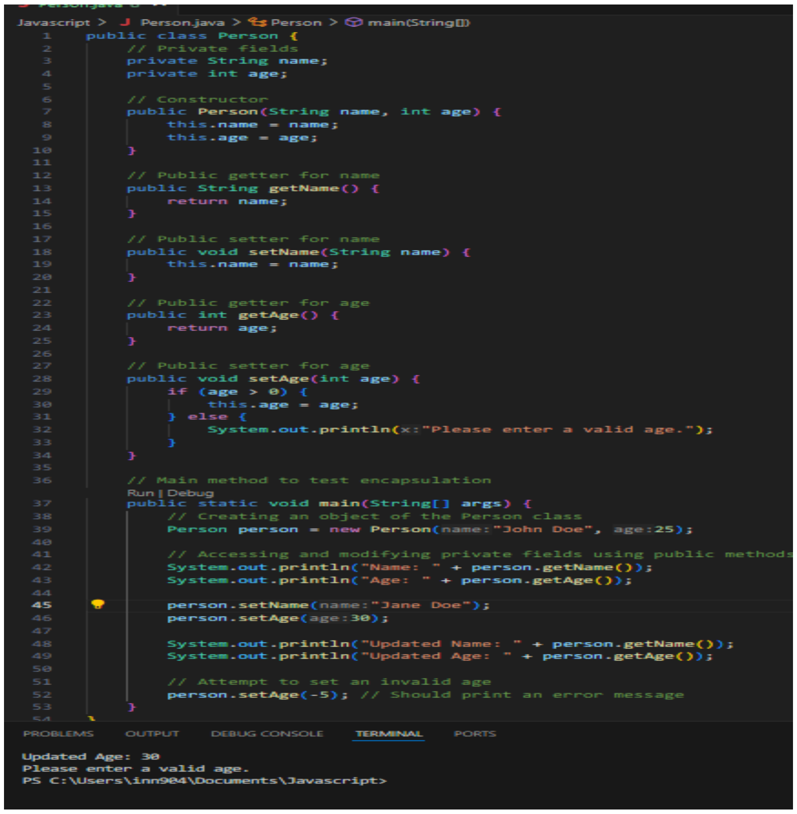
Encapsulation helps in:

- Hiding the internal state of an object from the outside world, which is known as data hiding.
- Providing controlled access to the data through public methods (getters and setters).

By using encapsulation, you can protect an object's internal state and ensure that it is only modified in a controlled manner.

Encapsulation Syntax:

- Declare the fields (variables) of a class as private.
- Provide public getter and setter methods to access and update the value of the private fields.



```java
public class Person {
    // Private fields
    private String name;
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Public getter for name
    public String getName() {
        return name;
    }

    // Public setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Public getter for age
    public int getAge() {
        return age;
    }

    // Public setter for age
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        } else {
            System.out.println(x:"Please enter a valid age.");
        }
    }

    // Main method to test encapsulation
    Run | Debug
    public static void main(String[] args) {
        // Creating an object of the Person class
        Person person = new Person(name:"John Doe", age:25);

        // Accessing and modifying private fields using public methods
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());

        person.setName(name:"Jane Doe");
        person.setAge(age:30);

        System.out.println("Updated Name: " + person.getName());
        System.out.println("Updated Age: " + person.getAge());

        // Attempt to set an invalid age
        person.setAge(-5); // Should print an error message
    }
```

```
Updated Age: 30
Please enter a valid age.
PS C:\Users\inn904\Documents\Javascript>
```

Fig 5: Encapsulation

Explanation of the code:

Creating an Object:

- We start by creating a Person object named person with the initial values "John Doe" for the name and 25 for the age.
- The constructor of the Person class sets these initial values.

1. **Accessing Fields**:

   - The program prints the name and age of the person object:
   - It shows "John Doe" for the name.
   - It shows "25" for the age.

Modifying Fields:

- We update the name of the person object to "Jane Doe" using the setter method.
- We update the age to 30 using another setter method.
- The program prints the updated values:
  - It shows "Jane Doe" for the name.
  - It shows "30" for the age.

2. **Handling Invalid Input**:

   - When we try to set the age to -5, the setter method checks if the age is valid (greater than 0).
   - Since -5 is not valid, the program prints a message: "Please enter a valid age."

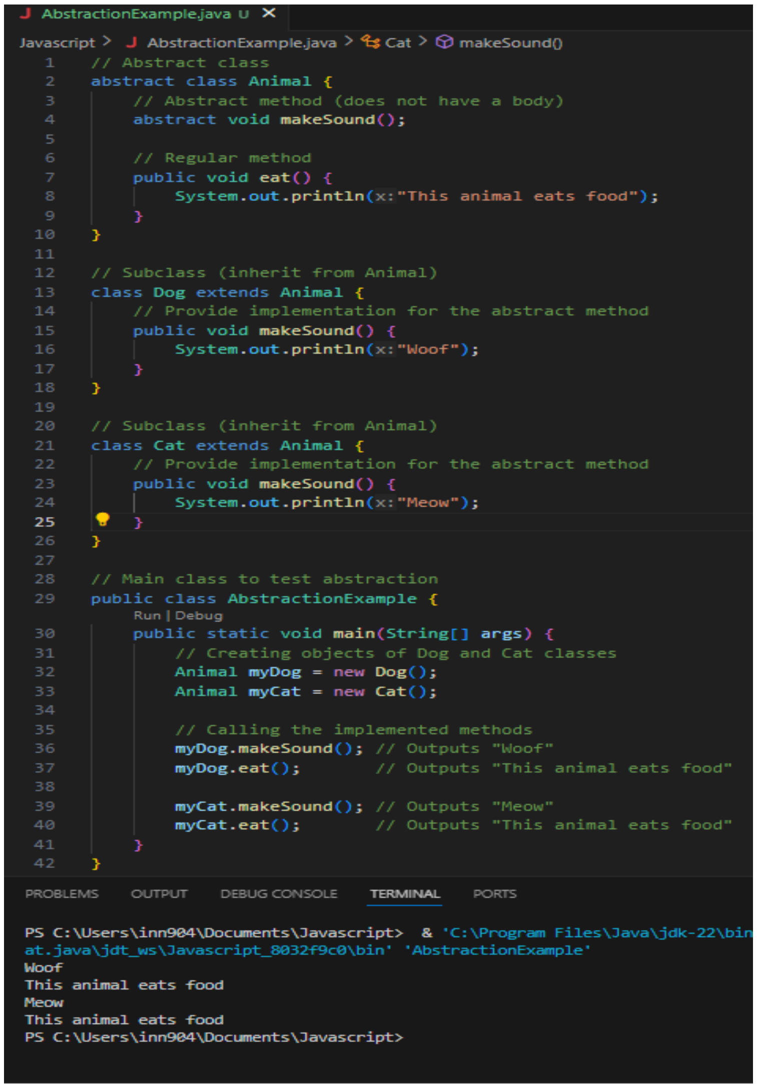
Abstraction:

It focuses on hiding the complex implementation details and showing only the essential features of an object. Abstraction helps in reducing complexity and allows the programmer to focus on interactions at a higher level.

- Abstraction is achieved using abstract classes and interfaces.

Abstract Class

An abstract class cannot be instantiated and may contain abstract methods, which are methods without a body. Subclasses of the abstract class must provide implementations for the abstract methods.



AbstractionExample.java U
Javascript > AbstractionExample.java > Cat > makeSound()
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```java
// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void makeSound();

    // Regular method
    public void eat() {
        System.out.println(x:"This animal eats food");
    }
}

// Subclass (inherit from Animal)
class Dog extends Animal {
    // Provide implementation for the abstract method
    public void makeSound() {
        System.out.println(x:"Woof");
    }
}

// Subclass (inherit from Animal)
class Cat extends Animal {
    // Provide implementation for the abstract method
    public void makeSound() {
        System.out.println(x:"Meow");
    }
}

// Main class to test abstraction
public class AbstractionExample {
    Run | Debug
    public static void main(String[] args) {
        // Creating objects of Dog and Cat classes
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        // Calling the implemented methods
        myDog.makeSound(); // Outputs "Woof"
        myDog.eat();       // Outputs "This animal eats food"

        myCat.makeSound(); // Outputs "Meow"
        myCat.eat();       // Outputs "This animal eats food"
    }
}
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin
at.java\jdt_ws\Javascript_8032f9c0\bin' 'AbstractionExample'
Woof
This animal eats food
Meow
This animal eats food
PS C:\Users\inn904\Documents\Javascript>
```

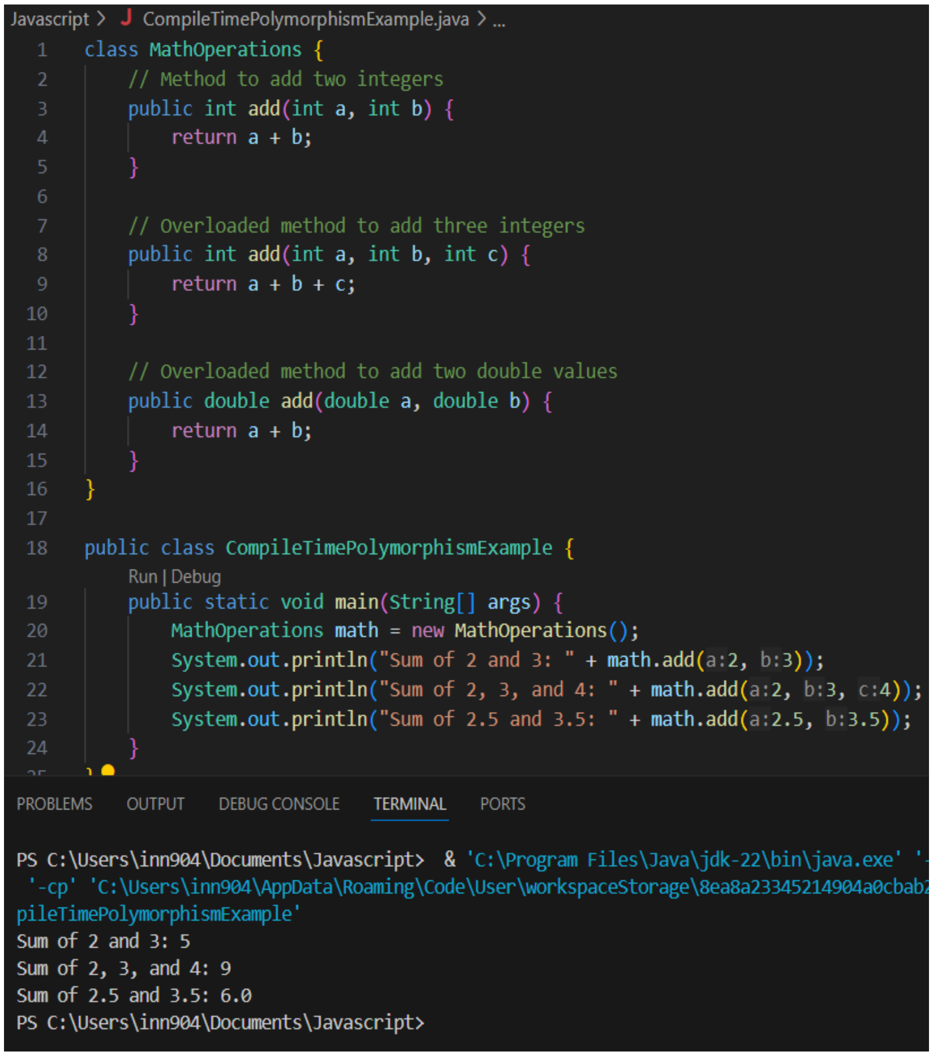
Fig 6: Abstraction

Polymorphism:

Polymorphism is one of the fundamental principles of Object-Oriented Programming (OOP). It allows objects to be treated as instances of their parent class rather than their actual class. The two main types of polymorphism are:

1. **Compile-time Polymorphism**: A polymorphism which exists at the time of compilation of the program is called compiled time polymorphism.

   Ex: Method Overloading:
   Whenever a class contain more than one method with same name and different type of parameter called Method Overloading.


Javascript > CompileTimePolymorphismExample.java > ...
Run | Debug
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

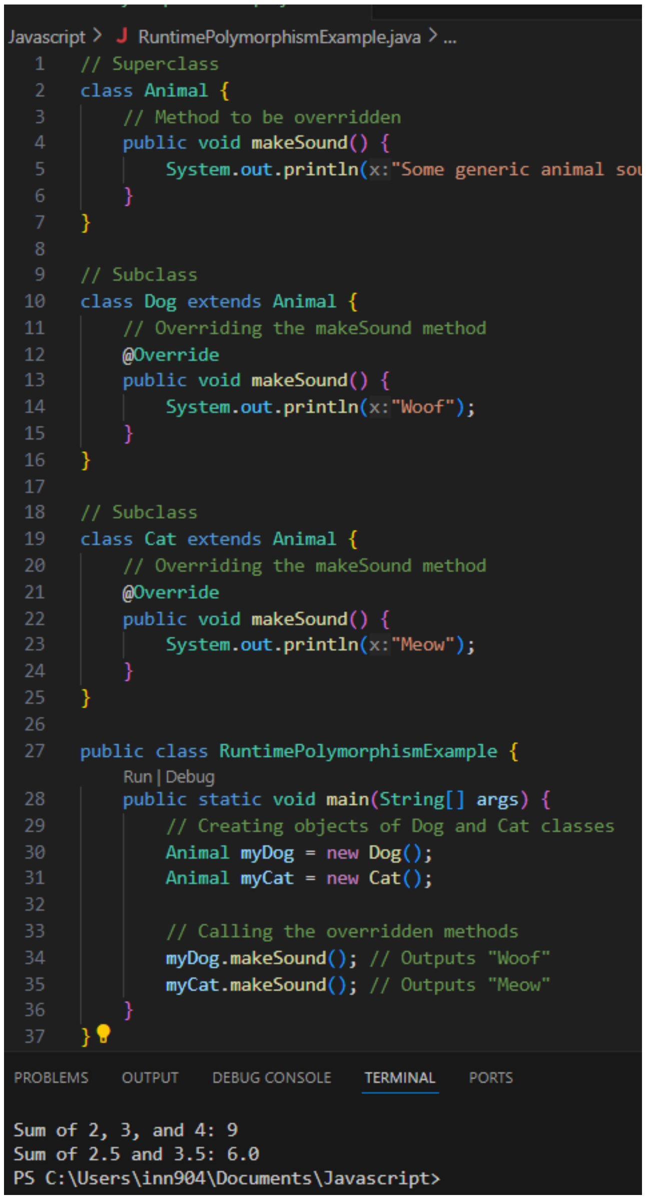
```java
class MathOperations {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // 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 class CompileTimePolymorphismExample {
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        System.out.println("Sum of 2 and 3: " + math.add(a:2, b:3));
        System.out.println("Sum of 2, 3, and 4: " + math.add(a:2, b:3, c:4));
        System.out.println("Sum of 2.5 and 3.5: " + math.add(a:2.5, b:3.5));
    }
}
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java.exe' '-
 '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab
pileTimePolymorphismExample'
Sum of 2 and 3: 5
Sum of 2, 3, and 4: 9
Sum of 2.5 and 3.5: 6.0
PS C:\Users\inn904\Documents\Javascript>
```

Fig7: Method Overloading

2. **Runtime Polymorphism:** A polymorphism which exists at the time of execution of the program is called runtime polymorphism.

   Ex: Method Overriding:
   Whenever we writing method in super and subclasses in such a way that the method name and parameter must be same .



Javascript > RuntimePolymorphismExample.java > ...

```java
// Superclass
class Animal {
    // Method to be overridden
    public void makeSound() {
        System.out.println(x:"Some generic animal sou
    }
}

// Subclass
class Dog extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println(x:"Woof");
    }
}

// Subclass
class Cat extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println(x:"Meow");
    }
}

public class RuntimePolymorphismExample {
    Run | Debug
    public static void main(String[] args) {
        // Creating objects of Dog and Cat classes
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        // Calling the overridden methods
        myDog.makeSound(); // Outputs "Woof"
        myCat.makeSound(); // Outputs "Meow"
    }
}
```

```
Sum of 2, 3, and 4: 9
Sum of 2.5 and 3.5: 6.0
PS C:\Users\inn904\Documents\Javascript>
```

Fig 8: Method Overriding

Assignment 4

Date:12 June 2024

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.
so that the normal flow of the application can be maintained.

Java uses try, catch, finally, throw, and throws keywords to handle exceptions.

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The **try and catch** keywords come in pairs
- The **finally** statement lets you execute code, after try...catch, regardless of the result OR It is a mandatory block it will execute all the time if we get exception or not.
- The **throw keyword** is used to explicitly throw an exception. This can be useful when you want to create and throw an exception based on specific conditions within your code.

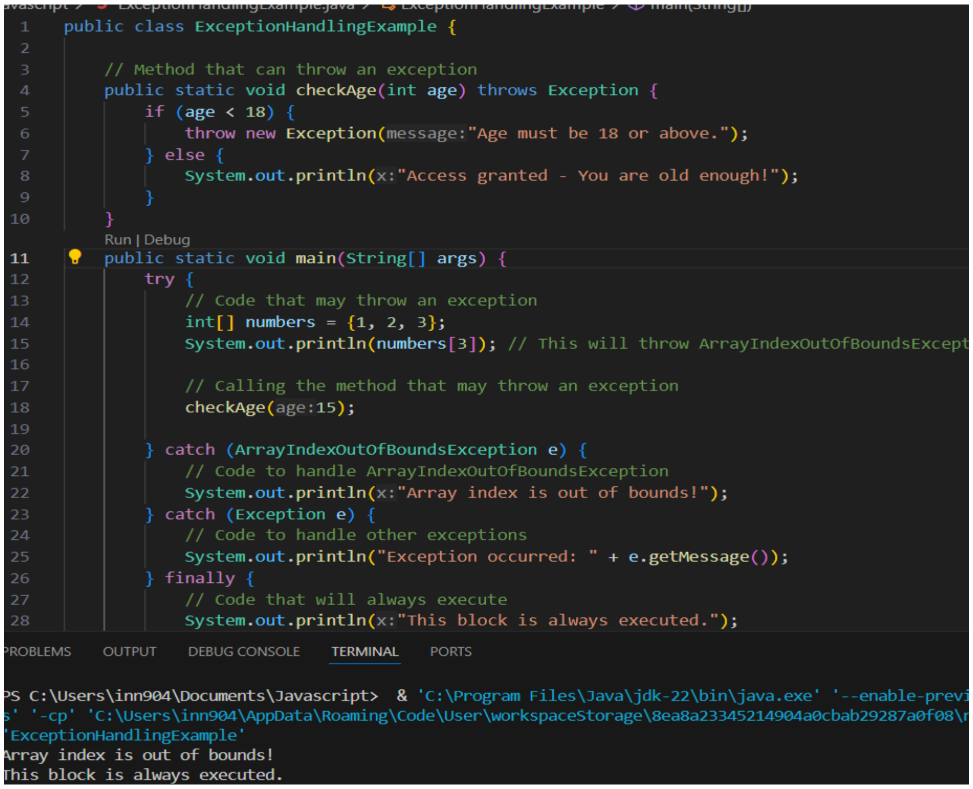
  Syntax: throw new ExceptionType("Exception message");

- The **throws keyword** is used in a method signature to declare that the method might throw one or more exceptions. This informs the caller of the method that it needs to handle these exceptions.

  Syntax:
```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2 {
   // Method body
         }
```

Syntax of Exception Handling

```
try {

   // Code that may throw an exception

} catch (ExceptionType1 e1) {

   // Code to handle ExceptionType1

} catch (ExceptionType2 e2) {

   // Code to handle ExceptionType2

} finally {

   // Code that will always execute

}
```



```java
public class ExceptionHandlingExample {

    // Method that can throw an exception
    public static void checkAge(int age) throws Exception {
        if (age < 18) {
            throw new Exception(message:"Age must be 18 or above.");
        } else {
            System.out.println(x:"Access granted - You are old enough!");
        }
    }
    Run | Debug
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[3]); // This will throw ArrayIndexOutOfBoundsExcept

            // Calling the method that may throw an exception
            checkAge(age:15);

        } catch (ArrayIndexOutOfBoundsException e) {
            // Code to handle ArrayIndexOutOfBoundsException
            System.out.println(x:"Array index is out of bounds!");
        } catch (Exception e) {
            // Code to handle other exceptions
            System.out.println("Exception occurred: " + e.getMessage());
        } finally {
            // Code that will always execute
            System.out.println(x:"This block is always executed.");
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java.exe' '--enable-previ
s' '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab29287a0f08\r
'ExceptionHandlingExample'
Array index is out of bounds!
This block is always executed.
```

Fig1: Exception Handling

In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions.

Checked Exceptions

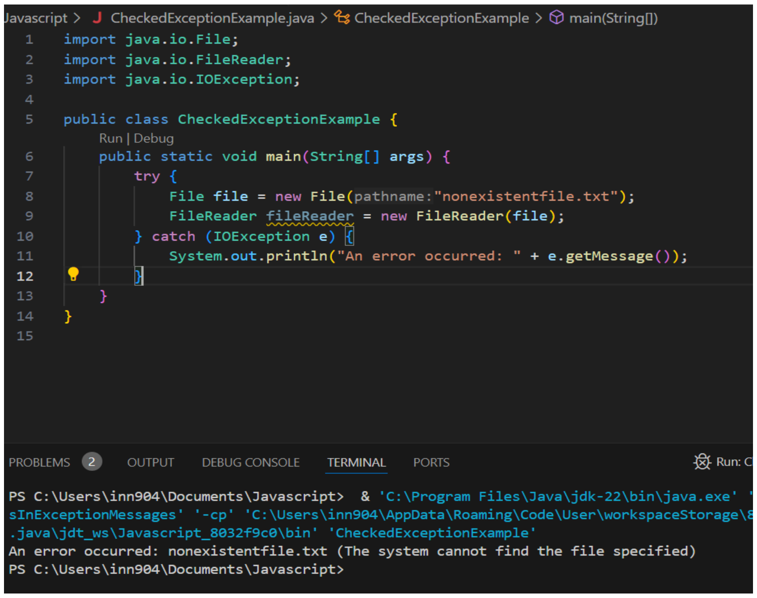
Checked exceptions are exceptions that are checked at compile-time. These are exceptions that the compiler forces you to handle explicitly. If a method throws a checked exception, it must either handle the exception using a try-catch block or declare the exception using the throws keyword in the method signature.

Characteristics of Checked Exceptions:

- Checked exceptions are derived from the Exception class (excluding RuntimeException and its subclasses).
- The compiler checks these exceptions at compile-time to ensure that they are properly handled.

Examples of Checked Exceptions:

- IOException
- SQLException
- ClassNotFoundException


Javascript > CheckedExceptionExample.java > CheckedExceptionExample > main(String[])
Run | Debug
PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Run: C

```java
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            File file = new File(pathname:"nonexistentfile.txt");
            FileReader fileReader = new FileReader(file);
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java.exe' '
sInExceptionMessages' '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceStorage\8
.java\jdt_ws\Javascript_8032f9c0\bin' 'CheckedExceptionExample'
An error occurred: nonexistentfile.txt (The system cannot find the file specified)
PS C:\Users\inn904\Documents\Javascript>
```

Fig2: Checked Exception

Unchecked Exceptions

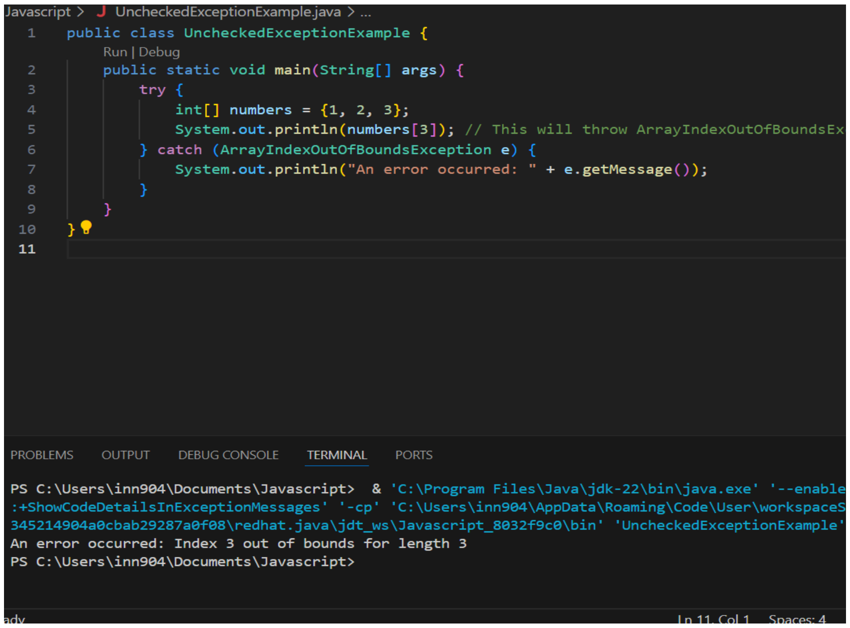
Unchecked exceptions are exceptions that are not checked at compile-time. These exceptions are derived from RuntimeException and its subclasses. The compiler does not force you to handle or declare these exceptions, although it is often a good practice to do so.

Characteristics of Unchecked Exceptions:

- Unchecked exceptions are derived from RuntimeException.
- They occur at runtime and are often indicative of programming errors, such as logic mistakes or improper use of APIs.

Examples of Unchecked Exceptions:

- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException



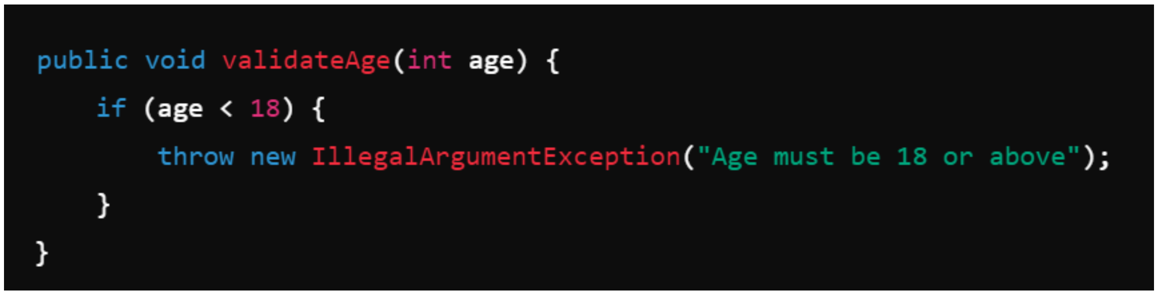
```java
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[3]); // This will throw ArrayIndexOutOfBoundsEx
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}

```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java.exe' '--enable
:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\inn904\AppData\Roaming\Code\User\workspaceS
345214904a0cbab29287a0f08\redhat.java\jdt_ws\Javascript_8032f9c0\bin' 'UncheckedExceptionExample'
An error occurred: Index 3 out of bounds for length 3
PS C:\Users\inn904\Documents\Javascript>
```

Fig3: Unchecked Exceptions

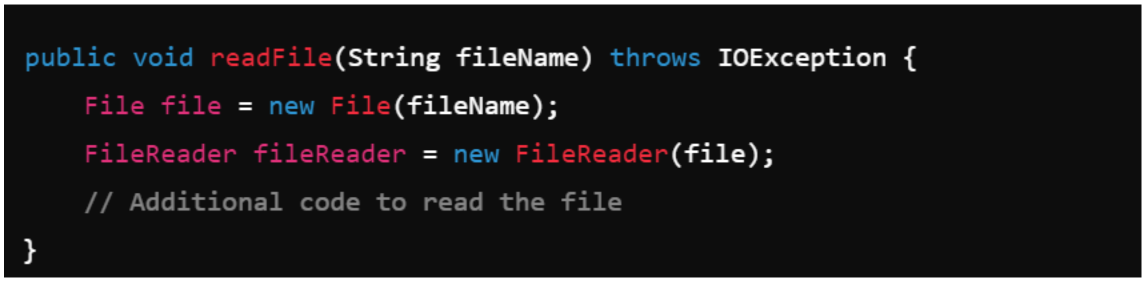
Differences between the throw and throws keywords in Java

Feature	**throw**	**throws**
Purpose	Used to explicitly throw an exception from a method or block of code.	Used in method signature to declare that a method can throw one or more exceptions
Usage	Followed by an instance of Throwable (e.g., throw new Exception("message");).	Followed by one or more exception class names (e.g., throws IOException, SQLException).
Scope	Affects the flow of control within a method.	Provides information about potential exceptions to the caller of the method.
Position	Inside the method body.	In the method signature.
Compilation	Compiler checks that the thrown exception is handled within a try-catch block or propagated up the call stack.	Compiler checks that the declared exceptions are either handled or further declared in the calling methods.
Example	throw new IOException("File not found");	public void readFile(String fileName) throws IOException



```java
public void validateAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Age must be 18 or above");
    }
}
```

Fig4: throw Example:



```java
public void readFile(String fileName) throws IOException {
    File file = new File(fileName);
    FileReader fileReader = new FileReader(file);
    // Additional code to read the file
}
```

Fig5: throws Example

Assignment 5

Date:13 June 2023

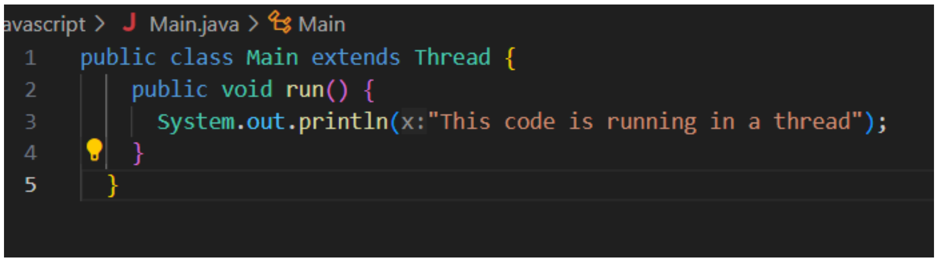
Thread:

A thread in Java is a lightweight process that allows a program to perform multiple operations concurrently. Threads are part of the Java concurrency framework and are used to achieve parallelism and improve the performance of applications. Each thread runs in the context of a process and shares the process's resources but executes independently

- Threads can be used to perform complicated tasks in the background without interrupting the main program.

- There are two ways to create a thread.

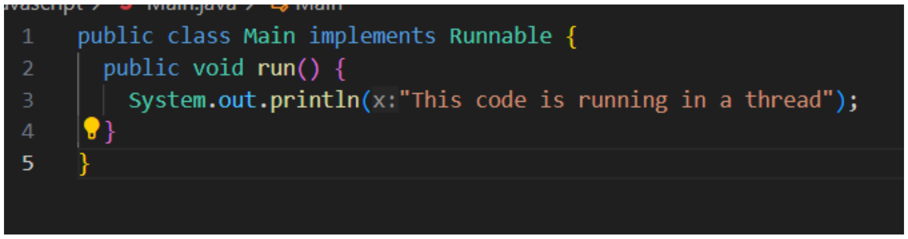
  - It can be created by extending the Thread class and overriding its run() method



```java
public class Main extends Thread {
    public void run() {
      System.out.println(x:"This code is running in a thread");
    }
  }
```

Fig1: Thread creation by Thread Class

  - Another way to create a thread is to implement the Runnable interface:



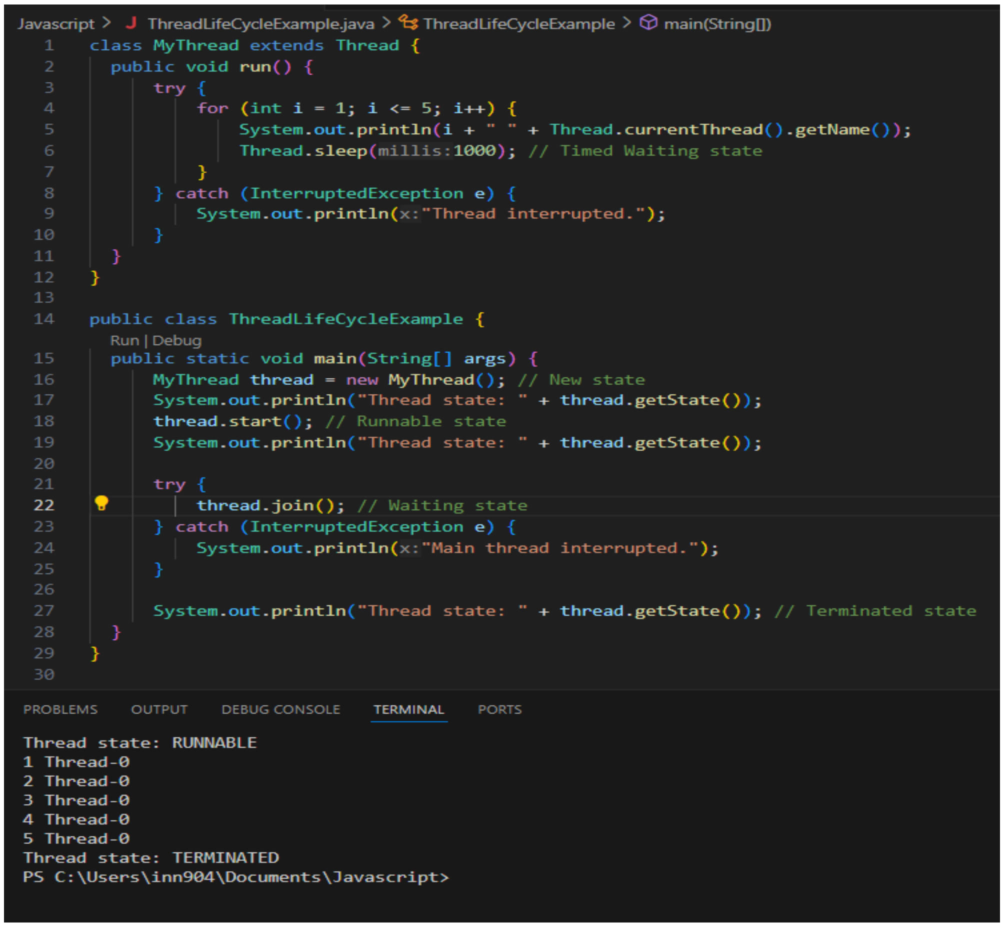
```java
public class Main implements Runnable {
  public void run() {
    System.out.println(x:"This code is running in a thread");
  }
}
```

Fig2: Thread creation by Runnable Interface

Thread Life Cycle in Java

A thread in Java goes through several states in its life cycle. These states represent the different stages of execution for a thread. The thread life cycle states are:

1. **New**: A thread is in this state when it is created but not yet started.
2. **Runnable**: A thread transitions to this state when the start() method is called. The thread is ready to run but may be waiting for CPU time.
3. **Blocked**: A thread is in this state when it is waiting to acquire a monitor lock to enter or re-enter a synchronized block or method.
4. **Waiting**: A thread is in this state when it is waiting indefinitely for another thread to perform a particular action (such as a notification).
5. **Timed Waiting**: A thread is in this state when it is waiting for another thread to perform a particular action for up to a specified waiting time.
6. **Terminated**: A thread is in this state when it has completed its execution or has been explicitly terminated.


Javascript > ThreadLifeCycleExample.java > ThreadLifeCycleExample > main(String[])

```java
class MyThread extends Thread {
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println(i + " " + Thread.currentThread().getName());
                Thread.sleep(millis:1000); // Timed Waiting state
            }
        } catch (InterruptedException e) {
            System.out.println(x:"Thread interrupted.");
        }
    }
}

public class ThreadLifeCycleExample {
    Run | Debug
    public static void main(String[] args) {
        MyThread thread = new MyThread(); // New state
        System.out.println("Thread state: " + thread.getState());
        thread.start(); // Runnable state
        System.out.println("Thread state: " + thread.getState());

        try {
            thread.join(); // Waiting state
        } catch (InterruptedException e) {
            System.out.println(x:"Main thread interrupted.");
        }

        System.out.println("Thread state: " + thread.getState()); // Terminated state
    }
}
```

```
Thread state: RUNNABLE
1 Thread-0
2 Thread-0
3 Thread-0
4 Thread-0
5 Thread-0
Thread state: TERMINATED
PS C:\Users\inn904\Documents\Javascript>
```

Fig3: Thread Life Cycle

Explanation

1. **New State**:
   - The thread thread is created but not yet started.
2. **Runnable State**:
   - The thread enters the Runnable state when start() is called.
3. **Running State**:
   - The thread scheduler moves the thread to the Running state to execute the run() method.
4. **Timed Waiting State**:
   - The thread enters the Timed Waiting state during the Thread.sleep(1000) call.
5. **Waiting State**:
   - The main thread calls join() on the thread, making it wait for the thread to finish execution.
6. **Terminated State**:
   - After completing its run() method, the thread moves to the Terminated state.

Multithreading in Java

Multithreading is a feature of Java that allows concurrent execution of two or more threads. It is essential for performing multiple tasks simultaneously, which can improve the performance of applications, especially when running on multi-core processors.

- **Multithreading** allows concurrent execution of two or more threads for maximum utilization of CPU.

Synchronization:

Synchronization in Java is a capability to control the access of multiple threads to shared resources. It is essential to prevent thread interference and consistency problems.

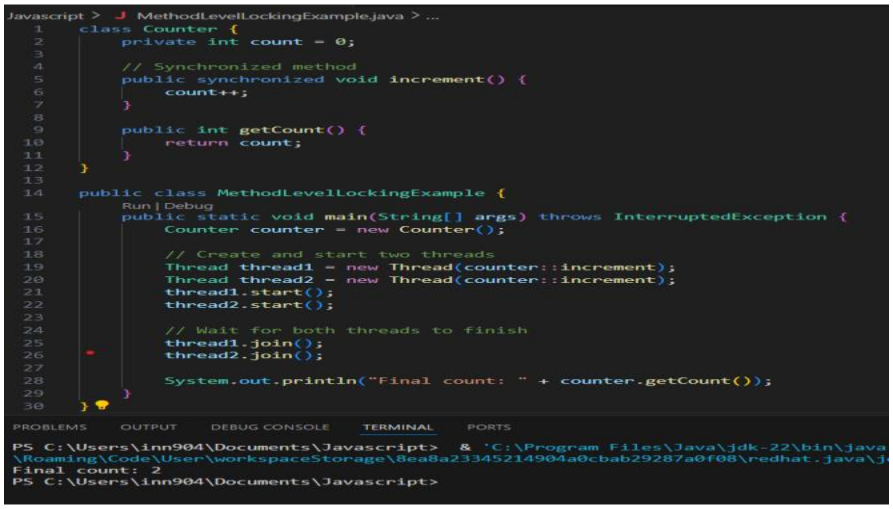
- The synchronized keyword is a modifier that locks a method so that only one thread can use it at a time. This prevents problems that arise from race conditions between threads.
- **Synchronization** ensures that shared resources are accessed by only one thread at a time to prevent data inconsistency and thread interference.
- Use synchronized methods or blocks to control the access to shared resources.
- **Method**: Locks the entire method.
- **Block**: Locks a specific block of code, providing more fine-grained control.

Method Level Locking

Method level locking, also known as synchronized method, synchronizes the entire method. When a thread invokes a synchronized method, it acquires the lock for the object and no other thread can execute any synchronized method on the same object until the lock is released.

Syntax:

```
public synchronized void someMethod() {

   // Entire method is synchronized

}
```



Javascript > MethodLevelLockingExample.java > ...

```java
class Counter {
    private int count = 0;

    // Synchronized method
    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class MethodLevelLockingExample {
    Run | Debug
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Create and start two threads
        Thread thread1 = new Thread(counter::increment);
        Thread thread2 = new Thread(counter::increment);
        thread1.start();
        thread2.start();

        // Wait for both threads to finish
        thread1.join();
        thread2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java
\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab29287a0f08\redhat.java\j
Final count: 2
PS C:\Users\inn904\Documents\Javascript>
```

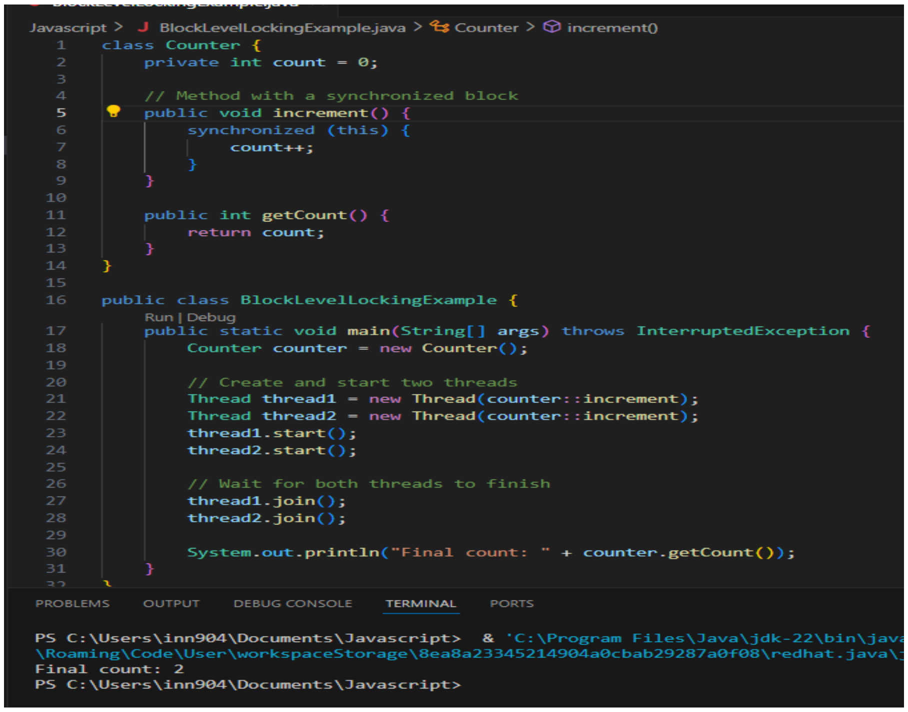
Fig4: Method Level Locking

Block Level Locking

Block level locking, also known as synchronized block, is a finer-grained synchronization mechanism. It allows you to synchronize only a specific section of code within a method, rather than the entire method. This can improve performance by reducing the scope of the lock, allowing other parts of the method to be executed by other threads.

Syntax:

```
public void someMethod() {
    // Some non-synchronized code
    synchronized (lockObject) {
        // Synchronized code block
    }
    // Some more non-synchronized code
}
```



Javascript > BlockLevelLockingExample.java > Counter > increment()

```java
class Counter {
    private int count = 0;

    // Method with a synchronized block
    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}

public class BlockLevelLockingExample {
    Run | Debug
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Create and start two threads
        Thread thread1 = new Thread(counter::increment);
        Thread thread2 = new Thread(counter::increment);
        thread1.start();
        thread2.start();

        // Wait for both threads to finish
        thread1.join();
        thread2.join();

        System.out.println("Final count: " + counter.getCount());
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\inn904\Documents\Javascript>  & 'C:\Program Files\Java\jdk-22\bin\java
\Roaming\Code\User\workspaceStorage\8ea8a23345214904a0cbab29287a0f08\redhat.java\
Final count: 2
PS C:\Users\inn904\Documents\Javascript>
```

Fig5: Block Level Locking

Deadlock:

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

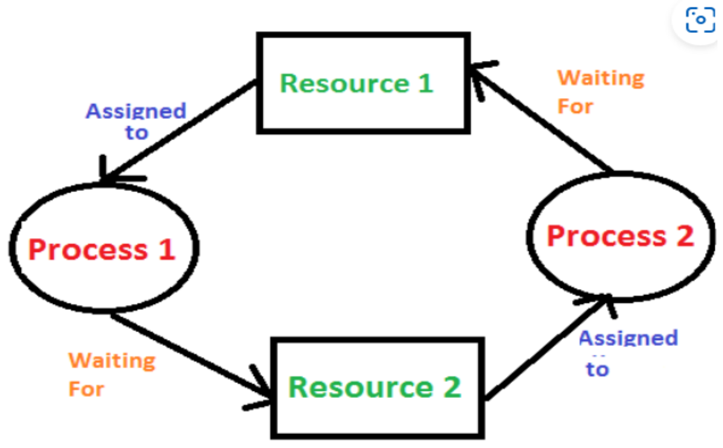


Fig6: Deadlock

Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

Deadlock can arise if the **following four conditions hold simultaneously**
Mutual Exclusion: Two or more resources are non-shareable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Preemption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes waiting for each other in circular form.

Avoiding Deadlocks

To avoid deadlocks, you can use various techniques:

1. **Avoid Nested Locks**:

   - Avoid locking multiple resources if possible. If you must, always lock them in the same order to prevent circular wait.

2. **Timeouts**:

   - Use try-lock mechanisms with timeouts where the thread attempts to acquire the lock and if it fails, it releases any locks it holds and retries after some time.

3. **Deadlock Detection**:

   - Implement deadlock detection algorithms that periodically check for circular dependencies among threads and take corrective actions if a deadlock is detected.

4. **Lock Ordering**:

   - Establish a global order for acquiring locks and ensure that all threads acquire the locks in this predefined order.

Executor Framework in Java

The Executor framework in Java provides a higher-level replacement for working directly with threads. It is part of the java.util.concurrent package and simplifies the management of thread execution. The framework decouples task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc.

Benefits of Framework

- **Simplified Thread Management**: Developers do not need to create and manage threads explicitly. The framework handles the creation, reuse, and termination of threads.
- **Thread Pooling**: Using thread pools can improve performance by reusing existing threads and reducing the overhead of thread creation.
- **Task Scheduling**:The framework provides powerful scheduling capabilities, such as scheduling tasks to run after a delay or periodically.
- **Graceful Shutdown**: The framework supports graceful shutdown of executor services, allowing for orderly termination of tasks.
- **Asynchronous Task Execution**: The framework provides support for asynchronous task execution and result handling via the Future interface.

Key Components of the Executor Framework

1. **Executor Interface**:

   - The root interface of the framework. It has a single method, execute(Runnable command), which is used to submit a task for execution.

2. **ExecutorService Interface**:

   - An extension of the Executor interface that adds methods for lifecycle management and methods to submit tasks and get their results.

- Key methods: submit(), invokeAll(), invokeAny(), shutdown(), and awaitTermination().

3. **Executors Class**:

   - A utility class that provides factory methods for creating different types of executor services, such as newFixedThreadPool(), newCachedThreadPool(), newSingleThreadExecutor(), and newScheduledThreadPool().

4. **Future Interface**:

   - Represents the result of an asynchronous computation. Methods include isDone(), get(), cancel(), and isCancelled().