DSEU

Crafting Excellence

HARSH KANDPAL
41523035

Delhi Skill and entrepreneurship University
Computer Science and Engineering
Java Programming
BT-CS-ES501
Okhla 2
Vth
3rd

Mr Manjeet Singh Pangtey

2025

Table of Contents

	containing the return statement.			
2.3	Create a class to compute the area of square, rectangle, and triangle (use the method overloading concept).		26 September 2025	
2.4	Add a constructor to your box class.		26 September 2025	
2.5	Show constructor overloading using the Box class.		26 September 2025	
2.6	Write a program to show the use of `this` keyword.		26 September 2025	
2.7	Write class arithmetic for calculation of addition, subtraction, multiplication, and division of two numbers		26 September 2025	
2.8	**WAP to demonstrate** call by value and call by reference.		26 September 2025	
2.9	Write a program to show the use of static members and static blocks.		26 September 2025	
3.1	Create a base class shape. It contains 2 methods get () and print () to accept and display parameters of shape respectively. Create a subclass Rectangle. It contains a method to display the length and breadth of a rectangle (Method overriding).		03 October 2025	
3.2	Use Box class. Create a subclass ColoredBox with one parameter String color. Override print method of Box class.		03 October 2025	
3.3	Write a program to show the Dynamic method dispatch concept.		03 October 2025	

3.4	WAP to implement stack using arrays in java		03 October 2025	
3.5	Write an abstract class Employee with three variables name, sal and Grosssal, suitable constructors, print method and two abstract methods calculate_gross_salary() and annual_increment(). Create a manager sub class of employee with Hra as member variable and write implementation of the abstract method. Also create a subclass of manager as sales manger with commission as member variable and override the calculate_gross_salary() method		03 October 2025	
3.6	Write a program to show the use of Interfaces		03 October 2025	

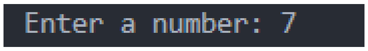
1.1

AIM: WAP to find the factorial of given number

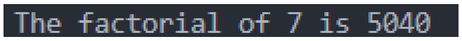
Logic: The factorial of a non-negative integer 'n', denoted by n!, is the product of all positive integers less than or equal to n. We can calculate this by initializing a variable `factorial` to 1 and then using a loop to multiply it by each number from 1 up to the given number. The factorial of 0 is defined as 1.

Code:

```
import java.util.Scanner;

public class Factorial {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int num = scanner.nextInt();
        long factorial = 1;

        if (num < 0) {
            System.out.println("Factorial is not defined for negative numbers.");
        } else {
            for (int i = 1; i <= num; ++i) {
                factorial *= i;
            }
            System.out.printf("The factorial of %d is %d\n", num, factorial);
        }
        scanner.close();
    }
}
```

INPUT 

```
Enter a number: 7
```

OUTPUT 

```
The factorial of 7 is 5040
```

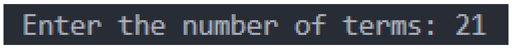
1.2

AIM: WAP to print Fibonacci series for a given number

Logic: The Fibonacci series starts with 0 and 1. Each subsequent number is the sum of the two preceding ones. To generate the series up to 'n' terms, we first print the initial two terms (0 and 1). Then, we loop 'n-2' times, calculating the next term by summing the previous two, printing it, and then updating the values of the two preceding terms for the next iteration.

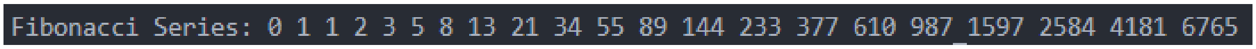
Code:

```
import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of terms: ");
        int n = scanner.nextInt();

        int t1 = 0, t2 = 1;
        System.out.print("Fibonacci Series: ");

        for (int i = 1; i <= n; ++i) {
            System.out.print(t1 + " ");
            int sum = t1 + t2;
            t1 = t2;
            t2 = sum;
        }
        scanner.close();
    }
}
```

INPUT 

```
Enter the number of terms: 21
```

OUTPUT



```
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

1.3

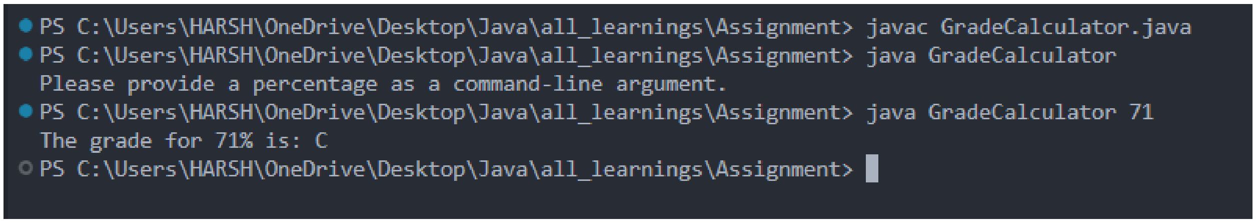
AIM: WAP to print grade for input percentage from command line using nested if else

Logic: This program reads a number (percentage) from the command-line arguments. The `args` array in the `main` method holds these values as strings. We parse the first argument `args[0]` into an integer. Then, a series of `if-else if` statements checks the percentage against predefined grade boundaries to determine and print the correct grade.

Code:

```
Import java.util.scanner;

public class GradeCalculator {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Please provide a percentage as a
command-line argument.");
            return;
        }

        int percentage = Integer.parseInt(args[0]);
        char grade;
        if (percentage >= 90) {
            grade = 'A';
        } else if (percentage >= 80) {
            grade = 'B';
        } else if (percentage >= 70) {
            grade = 'C';
        } else if (percentage >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("The grade for " + percentage + "% is: " + grade);
    }
}
```

RESULT



```
● PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> javac GradeCalculator.java
● PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> java GradeCalculator
  Please provide a percentage as a command-line argument.
● PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> java GradeCalculator 71
  The grade for 71% is: C
○ PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> 
```

1.4

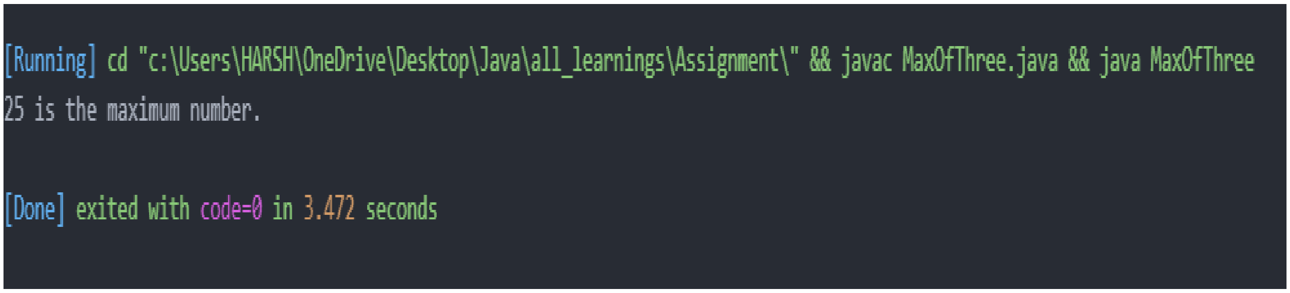
AIM: WAP to find the maximum of three numbers.

Logic: To find the largest of three numbers (n1, n2, n3), we compare them. First, check if n1 is greater than or equal to both n2 and n3. If it is, n1 is the maximum. If not, we then check if n2 is greater than or equal to both n1 and n3. If so, n2 is the maximum. If neither of these conditions is true, then n3 must be the maximum.

Code:

```
public class MaxOfThree {
    public static void main(String[] args) {
        int n1 = 10, n2 = 25, n3 = 15;

        if (n1 >= n2 && n1 >= n3) {
            System.out.println(n1 + " is the maximum number.");
        } else if (n2 >= n1 && n2 >= n3) {
            System.out.println(n2 + " is the maximum number.");
        } else {
            System.out.println(n3 + " is the maximum number.");
        }
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment\" && javac MaxOfThree.java && java MaxOfThree
25 is the maximum number.


[Done] exited with code=0 in 3.472 seconds
```

1.5

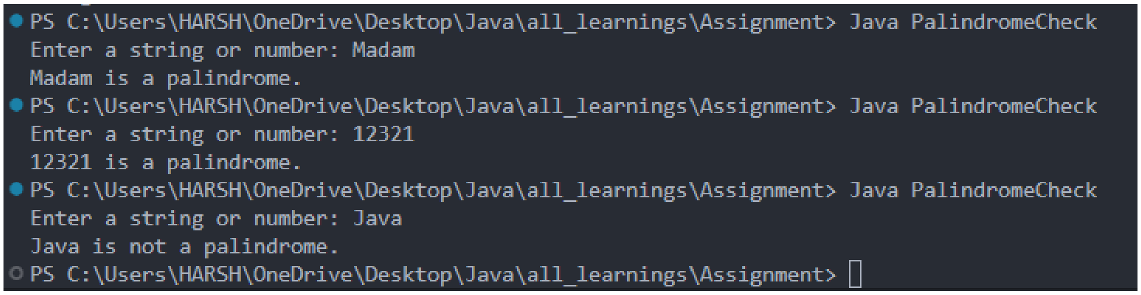
AIM: WAP to check whether a number or string is a palindrome.

Logic: A string is a palindrome if it reads the same forwards and backwards. We can check this by creating a new, reversed version of the string and comparing it to the original. A simple way to reverse the string is to loop through it from the last character to the first, appending each character to a new string.

Code:

```
import java.util.Scanner;
public class PalindromeCheck {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string or number: ");
        String original = scanner.nextLine();
        String reversed = "";
        for (int i = original.length() - 1; i >= 0; i--) {
            reversed += original.charAt(i);
        }
        if (original.equalsIgnoreCase(reversed)) {
            System.out.println(original + " is a palindrome.");
        } else {
            System.out.println(original + " is not a palindrome.");
        }
        scanner.close();
    }
}
```

RESULT



```
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> Java PalindromeCheck
Enter a string or number: Madam
Madam is a palindrome.
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> Java PalindromeCheck
Enter a string or number: 12321
12321 is a palindrome.
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> Java PalindromeCheck
Enter a string or number: Java
Java is not a palindrome.
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> 
```

1.6

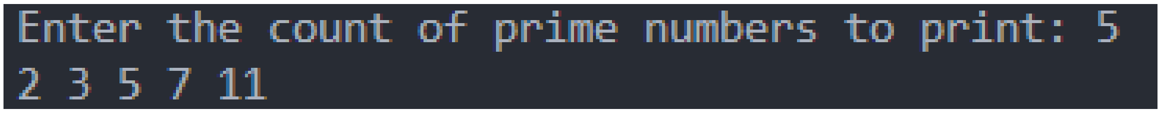
AIM: WAP to print first n prime numbers.

Logic: A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. To find the *first n* prime numbers, we start checking numbers (2, 3, 4, ...) one by one. For each number, we determine if it's prime. If it is, we print it and increment a counter. We continue this process until the counter reaches 'n'. A number is tested for primality by checking for divisibility by numbers from 2 up to its square root.

Code:

```
import java.util.Scanner;

public class FirstNPrimes {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the count of prime numbers to print: ");
        int n = scanner.nextInt();

        int count = 0;
        int num = 2;

        while(count < n) {
            if(isPrime(num)) {
                System.out.print(num + " ");
                count++;
            }
            num++;
        }
        scanner.close();
    }

    public static boolean isPrime(int number) {
        if (number <= 1) return false;
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) return false;
        }
        return true;
    }
}
```

RESULT



```
Enter the count of prime numbers to print: 5
2 3 5 7 11
```

1.7

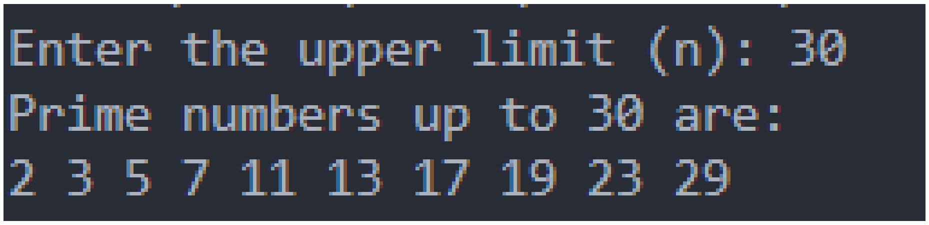
AIM: WAP to print a series of prime numbers up to n.

Logic: This is slightly different from the previous problem. Here, we need to print all prime numbers that exist in the range from 2 *up to* a given number 'n'. We use a `for` loop to iterate through every number from 2 to 'n'. In each iteration, we use the same `isPrime` helper function to check if the current number is prime. If it is, we print it.

Code:

```
import java.util.Scanner;

public class PrimesUpToN {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the upper limit (n): ");
        int n = scanner.nextInt();

        System.out.println("Prime numbers up to " + n + " are:");
        for(int i = 2; i <= n; i++) {
            if(isPrime(i)) {
                System.out.print(i + " ");
            }
        }
        scanner.close();
    }

    public static boolean isPrime(int number) {
        if (number <= 1) return false;
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) return false;
        }
        return true;
    }
}
```

RESULT



```
Enter the upper limit (n): 30
Prime numbers up to 30 are:
2 3 5 7 11 13 17 19 23 29
```

1.8

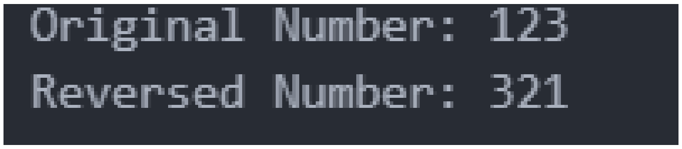
AIM: WAP to print reverse of a given digit of a given number.

Logic: To reverse a number, we can extract its digits one by one from the end. We use a `while` loop that continues as long as the number is not zero. In each iteration, we get the last digit using the modulo operator (`% 10`). We then build the reversed number by multiplying the current reversed value by 10 and adding the extracted digit. Finally, we remove the last digit from the original number using integer division (`/ 10`).

Code:

```
public class ReverseNumber {
    public static void main(String[] args) {
        int num = 123, reversed = 0;
        System.out.println("Original Number: " + num);

        while(num != 0) {
            int digit = num % 10; // get last digit
            reversed = reversed * 10 + digit;
            num /= 10; // remove last digit
        }
        System.out.println("Reversed Number: " + reversed);
    }
}
```

RESULT



```
Original Number: 123
Reversed Number: 321
```

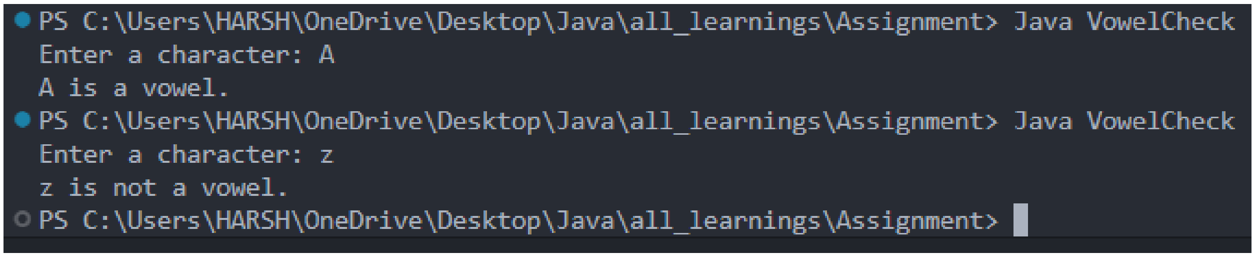
1.9
AIM: WAP to find whether a given char is a vowel using switch case.

Logic: This program checks if a given character is a vowel ('a', 'e', 'i', 'o', 'u'). As required, it uses a `switch` statement. The character is passed to the `switch`. We create `case` labels for both lowercase and uppercase vowels. Since they all perform the same action, we let them "fall through" to a single print statement. The `default` case handles all other characters (consonants).

Code:

```
public class VowelCheck {
    public static void main(String[] args) {
        char ch = 'e';

        switch (ch) {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U':
                System.out.println(ch + " is a vowel.");
                break;
            default:
                System.out.println(ch + " is a consonant.");
        }
    }
}
```

RESULT



```
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> Java VowelCheck
Enter a character: A
A is a vowel.
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> Java VowelCheck
Enter a character: z
z is not a vowel.
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment>
```

1.10

AIM: WAP to print pattern.

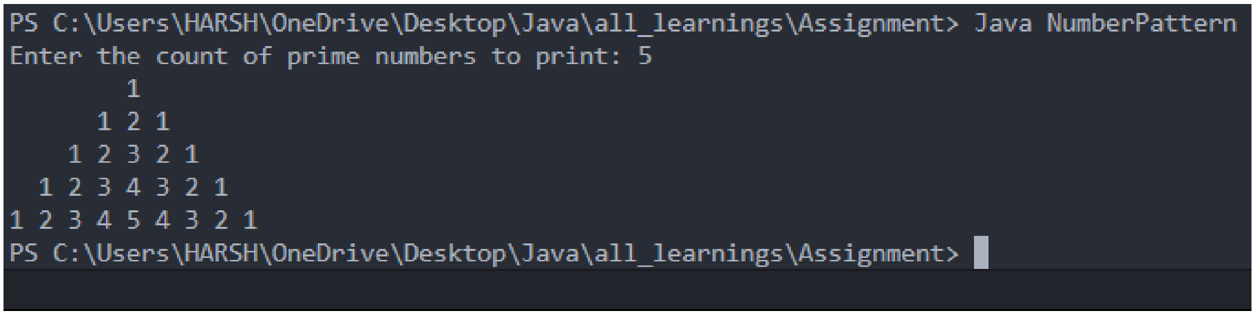
Logic: This pattern is a number pyramid. It requires nested loops. The outer loop controls the rows. For each row `i`, we need to print:

1. **Spaces:** A certain number of leading spaces to align the pyramid correctly. For `n` total rows, row `i` has `n-i` spaces.
2. **Increasing Numbers:** A loop that prints numbers from 1 up to the current row number `i`.
3. **Decreasing Numbers:** Another loop that prints numbers from `i-1` back down to 1.

Code:

```
public class NumberPattern {
    public static void main(String[] args) {
        int rows = 3;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= i; j++) {
                System.out.print(j + " ");
            }
            for (int k = i - 1; k >= 1; k--) {
                System.out.print(k + " ");
            }
            System.out.println();
        }
    }
}
```

RESULT



```
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment> Java NumberPattern
Enter the count of prime numbers to print: 5
        1
      1 2 1
    1 2 3 2 1
  1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment>
```

2.1

AIM: WAP to create a Simple Box class with method `void volume()` and compute volume for two instances

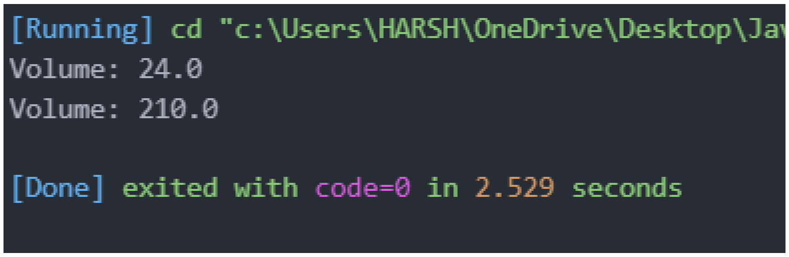
Logic:

We define a class `Box` with three instance variables: `width`, `height`, and `depth`. The `volume()` method calculates and displays the volume. Two instances are created.

Code:

```
class Box {
    double width, height, depth;

    void volume() {
        double vol = width * height * depth;
        System.out.println("Volume: " + vol);
    }
}

public class Main {
    public static void main(String[] args) {
        Box box1 = new Box();
        box1.width = 2;
        box1.height = 3;
        box1.depth = 4;

        Box box2 = new Box();
        box2.width = 5;
        box2.height = 6;
        box2.depth = 7;

        box1.volume();
        box2.volume();
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\Desktop\Jav
Volume: 24.0
Volume: 210.0

[Done] exited with code=0 in 2.529 seconds
```

2.2

AIM: Rewrite program 1 to modify the volume method containing the return statement.

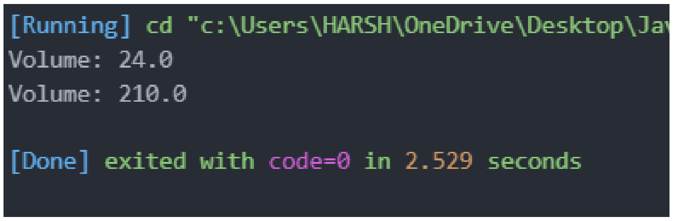
Logic:

Instead of printing inside the method, we return the volume from the method and print it in `main()`.

Code:

```
class Box {
    double width, height, depth;
    double volume() {
        return width * height * depth;
    }
}
public class Main {
    public static void main(String[] args) {
        Box box1 = new Box();
        box1.width = 2;
        box1.height = 3;
        box1.depth = 4;
        Box box2 = new Box();
        box2.width = 5;
        box2.height = 6;
        box2.depth = 7;
        System.out.println("Volume 1: " + box1.volume());
        System.out.println("Volume 2: " + box2.volume());
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\Desktop\Ja
Volume: 24.0
Volume: 210.0

[Done] exited with code=0 in 2.529 seconds
```

2.3

AIM: Create a class to compute the area of square, rectangle and triangle.

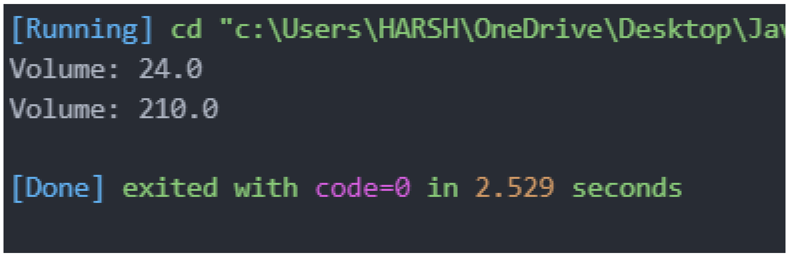
Logic:

We use the concept of **method overloading** to calculate areas of square, rectangle, and triangle with the same method name `area()`.

Code:

```
class AreaCalculator {
    void area(int side) {
        System.out.println("Area of Square: " + (side * side));
    }
    void area(int length, int breadth) {
        System.out.println("Area of Rectangle: " + (length * breadth));
    }
    void area(double base, double height) {
        System.out.println("Area of Triangle: " + (0.5 * base * height));
    }
}
public class Main {
    public static void main(String[] args) {
        AreaCalculator obj = new AreaCalculator();
        obj.area(4);            // Square
        obj.area(5, 3);         // Rectangle
        obj.area(6.0, 2.0);     // Triangle
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\Desktop\Ja
Volume: 24.0
Volume: 210.0

[Done] exited with code=0 in 2.529 seconds
```

2.4

AIM: Add constructor to box.

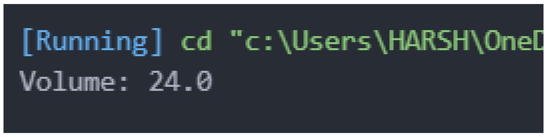
Logic:

We use a constructor to initialize box dimensions instead of assigning them manually.

Code:

```
class Box {
    double width, height, depth;

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
    }
}

public class Main {
    public static void main(String[] args) {
        Box box = new Box(2, 3, 4);
        System.out.println("Volume: " + box.volume());
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneD
Volume: 24.0
```

2.5

AIM: Show constructor overloading using Box Class.

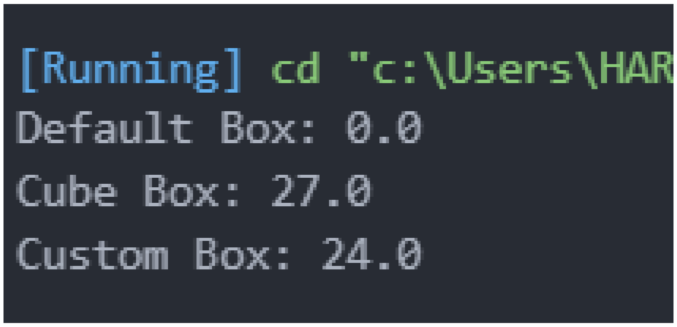
Logic:

We create multiple constructors: default, cube, and parameterized to demonstrate constructor overloading.

Code:

```
class Box {
    double width, height, depth;

    Box() {
        width = height = depth = 0;
    }

    Box(double side) {
        width = height = depth = side;
    }

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
    }
}

public class Main {
    public static void main(String[] args) {
        Box b1 = new Box();           // Default
        Box b2 = new Box(3);          // Cube
        Box b3 = new Box(2, 3, 4);    // Custom

        System.out.println("Default Box: " + b1.volume());
        System.out.println("Cube Box: " + b2.volume());
        System.out.println("Custom Box: " + b3.volume());
    }
}
```

RESULT



```
[Running] cd "c:\Users\HAR
Default Box: 0.0
Cube Box: 27.0
Custom Box: 24.0
```

2.6

AIM: Write a program to show use of this keyword.

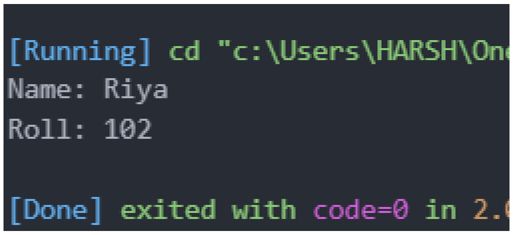
Logic:

`this` keyword refers to the current object and is used to differentiate instance variables from parameters.

Code:

```
class Student {
    String name;
    int roll;
    Student(String name, int roll) {
        this.name = name;
        this.roll = roll;
    }

    void display() {
        System.out.println("Name: " + this.name);
        System.out.println("Roll: " + this.roll);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Riya", 102);
        s.display();
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\On
Name: Riya
Roll: 102

[Done] exited with code=0 in 2.
```

2.7

AIM: write class arithmetic for calculation of addition and subtraction,multiplication and division of two number.

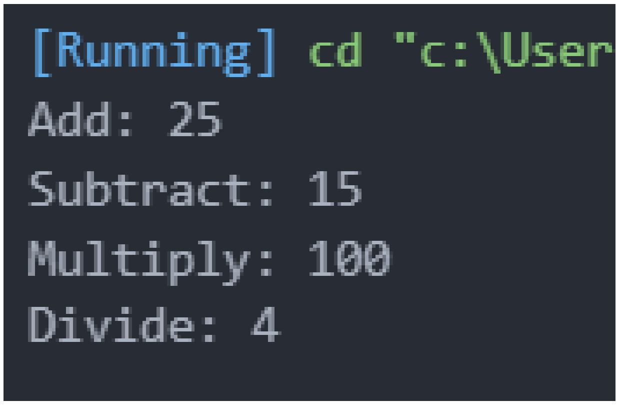
Logic:

Create an `Arithmetic` class with methods for each operation.

Code:

```
class Arithmetic {
    void add(int a, int b) {
        System.out.println("Add: " + (a + b));
    }

    void subtract(int a, int b) {
        System.out.println("Subtract: " + (a - b));
    }

    void multiply(int a, int b) {
        System.out.println("Multiply: " + (a * b));
    }

    void divide(int a, int b) {
        if (b != 0)
            System.out.println("Divide: " + (a / b));
        else
            System.out.println("Division by zero error");
    }
}

public class Main {
    public static void main(String[] args) {
        Arithmetic obj = new Arithmetic();
        obj.add(20, 5);
        obj.subtract(20, 5);
        obj.multiply(20, 5);
        obj.divide(20, 5);
    }
}
```

RESULT



```
[Running] cd "c:\User
Add: 25
Subtract: 15
Multiply: 100
Divide: 4
```

2.8

AIM: WAP to demonstrate call by value and call by reference.

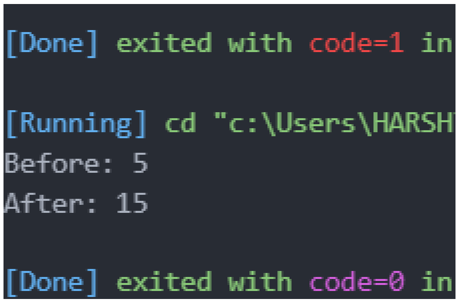
Logic:

Java supports **call by value**, but objects can mimic **call by reference**.

Code:

```
class Data {
    int x;

    void modify(Data d) {
        d.x = d.x + 10;
    }
}

public class Main {
    public static void main(String[] args) {
        Data d = new Data();
        d.x = 5;

        System.out.println("Before: " + d.x);
        d.modify(d);
        System.out.println("After: " + d.x);
    }
}
```

RESULT



```
[Done] exited with code=1 in

[Running] cd "c:\Users\HARSH
Before: 5
After: 15

[Done] exited with code=0 in
```

2.9

AIM: Wap to show the use of static member and static block.

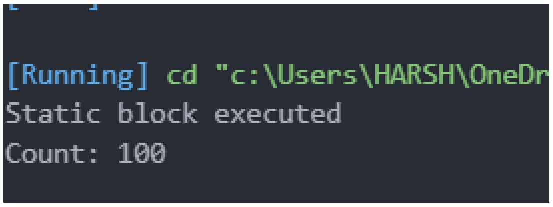
Logic:

Static block is executed once when class loads. Static methods and variables belong to the class, not objects.

Code:

```
class Demo {
    static int count;

    static {
        count = 100;
        System.out.println("Static block executed");
    }

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

public class Main {
    public static void main(String[] args) {
        Demo.show();
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDr
Static block executed
Count: 100
```

3.1

AIM: Create a base class shape. It contains 2 methods get () and print () to accept and display parameters of shape respectively. Create a subclass Rectangle. It contains a method to display the length and breadth of a rectangle (Method overriding).

Logic:

This problem demonstrates **Inheritance** and **Method Overriding**. The `Rectangle` subclass will provide its own specific versions of the methods it inherits from the `Shape` class.

1. **Base Class (`Shape`):** Create a class named `Shape` with two methods, `get()` and `print()`. These can have generic placeholder actions, like printing "Getting shape parameters."
2. **Subclass (`Rectangle`):** Create a class `Rectangle` that `extends Shape`. Add member variables for `length` and `breadth`.
3. **Override Methods:** In the `Rectangle` class, you must **override** both `get()` and `print()`.
   - The new `get()` method should specifically ask the user for the length and breadth of the rectangle.
   - The new `print()` method should display the stored length and breadth values.
4. **Demonstration:** In your `main` method, create an object of the `Rectangle` class. When you call `get()` and `print()` on this object, Java will automatically use the specialized versions from `Rectangle`, not the generic ones from `Shape`.

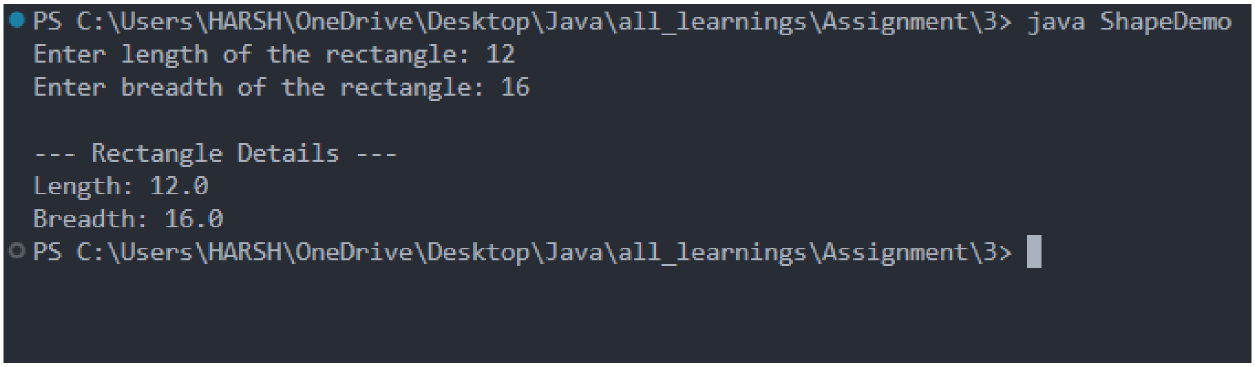
Code:

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

```
class Shape {
    public void get() {
        System.out.println("This is the get() method of the Shape class.");
    }

    public void print() {
        System.out.println("This is the print() method of the Shape class.");
    }
}


class Rectangle extends Shape {
    private double length;
    private double breadth;


    @Override
    public void get() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter length of the rectangle: ");
        this.length = scanner.nextDouble();
        System.out.print("Enter breadth of the rectangle: ");
        this.breadth = scanner.nextDouble();
    }


    @Override
    public void print() {
        System.out.println("\n--- Rectangle Details ---");
        System.out.println("Length: " + this.length);
        System.out.println("Breadth: " + this.breadth);
    }
}
```

```
public class ShapeDemo {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
        rect.get();  // Calls the overridden get() method in Rectangle
        rect.print(); // Calls the overridden print() method in Rectangle
    }
}
```

RESULT



```
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment\3> java ShapeDemo
Enter length of the rectangle: 12
Enter breadth of the rectangle: 16

--- Rectangle Details ---
Length: 12.0
Breadth: 16.0
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignment\3>
```

3.2

AIM: Use Box class. Create a subclass ColoredBox with one parameter String color. Override print method of Box class.

Logic:

This is another exercise in **Inheritance** and **Method Overriding**, this time involving a constructor in the subclass.

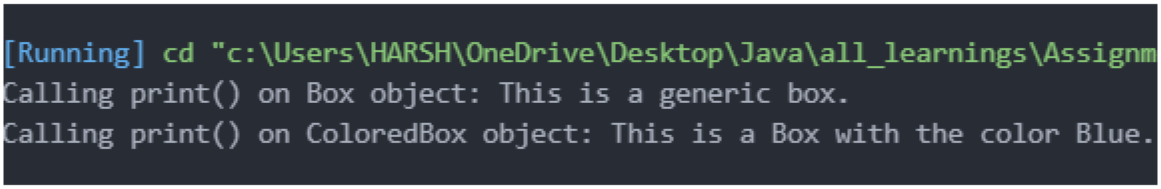
1. **Base Class (`Box`):** Create a simple `Box` class with a `print()` method that has a generic output, like "This is a generic box."
2. **Subclass (`ColoredBox`):** Create a class `ColoredBox` that `extends Box`.
3. **Add State and Constructor:** Add a `String color` member variable. Create a constructor for `ColoredBox` that accepts a string (e.g., `public ColoredBox(String color)`) and sets the `color` variable.
4. **Override `print()`:** In `ColoredBox`, **override** the `print()` method. Its new implementation should include the color, for example, "This is a box of [color] color."
5. **Demonstration:** In `main`, create a `ColoredBox` object by passing a color to its constructor (e.g., `new ColoredBox("Red")`) and then call its `print()` method to show the overridden behavior.

Code:

```
class Box {
    public void print() {
        System.out.println("This is a generic box.");
    }
}
```

```
class ColoredBox extends Box {
    private String color;

    public ColoredBox(String color) {
        this.color = color;
    }

    @Override
    public void print() {
        System.out.println("This is a Box with the color " + this.color + ".");
    }
}

public class BoxDemo {
    public static void main(String[] args) {
        Box genericBox = new Box();
        ColoredBox blueBox = new ColoredBox("Blue");

        System.out.print("Calling print() on Box object: ");
        genericBox.print();

        System.out.print("Calling print() on ColoredBox object: ");
        blueBox.print(); // Calls the overridden method
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\Desktop\Java\all_learnings\Assignm
Calling print() on Box object: This is a generic box.
Calling print() on ColoredBox object: This is a Box with the color Blue.
```

3.3

AIM: Write a program to show Dynamic method dispatch concept.

Logic:

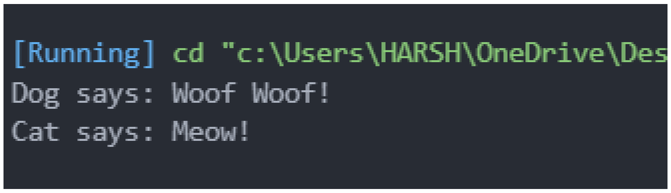
This concept is also known as **Runtime Polymorphism**. It's about how the method that gets executed is determined at runtime based on the object's type, not the reference variable's type.

1. **Superclass:** Create a base class, like `Animal`, with a method `makeSound()` that prints a generic sound.
2. **Subclasses:** Create at least two subclasses, like `Dog` and `Cat`, that `extend Animal`.
3. **Override:** In both `Dog` and `Cat`, **override** the `makeSound()` method to print a specific sound ("Woof" or "Meow").
4. **Demonstrate Dispatch:**
   - In `main`, declare a reference variable of the **superclass type**: `Animal myAnimal;`.
   - Assign a `Dog` object to it: `myAnimal = new Dog();`. When you call `myAnimal.makeSound()`, it will execute the `Dog`'s version and print "Woof".
   - Now, assign a `Cat` object to the **same reference**: `myAnimal = new Cat();`. Calling `myAnimal.makeSound()` now will execute the `Cat`'s version and print "Meow". The key is that the exact same line of code (`myAnimal.makeSound();`) behaves differently depending on the object it points to at runtime.

Code:

```
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a generic sound.");
    }
}
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog says: Woof Woof!");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat says: Meow!");
    }
}

public class DispatchDemo {
    public static void main(String[] args) {
        Animal myAnimal;
        myAnimal = new Dog();
        myAnimal.makeSound();
        myAnimal = new Cat();
        myAnimal.makeSound();
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\Des
Dog says: Woof Woof!
Cat says: Meow!
```

3.4

AIM: WAP to implement stack using arrays in java

Logic:

A stack is a **Last-In, First-Out (LIFO)** data structure. You'll implement this behavior using a simple array.

1. **Stack Class:** Create a class for your stack (e.g., `ArrayStack`). It should contain an integer array to hold the data, and an integer `top` to track the index of the last element added. Initialize `top` to **-1** to indicate the stack is empty.
2. **Push Operation (`push(item)`):**
   - Check for **stack overflow** (if `top` is at the last index of the array). If it is, you can't add more.
   - If there's space, increment `top` by one, then add the `item` to the array at the `top` index.
3. **Pop Operation (`pop()`):**
   - Check for **stack underflow** (if `top` is -1). If it is, the stack is empty and there's nothing to remove.
   - If it's not empty, retrieve the element at the `top` index, then decrement `top` by one.
4. **Helper Methods:** It's also good practice to include an `isEmpty()` method (returns true if `top == -1`) and a `peek()` method (returns the top element without removing it).

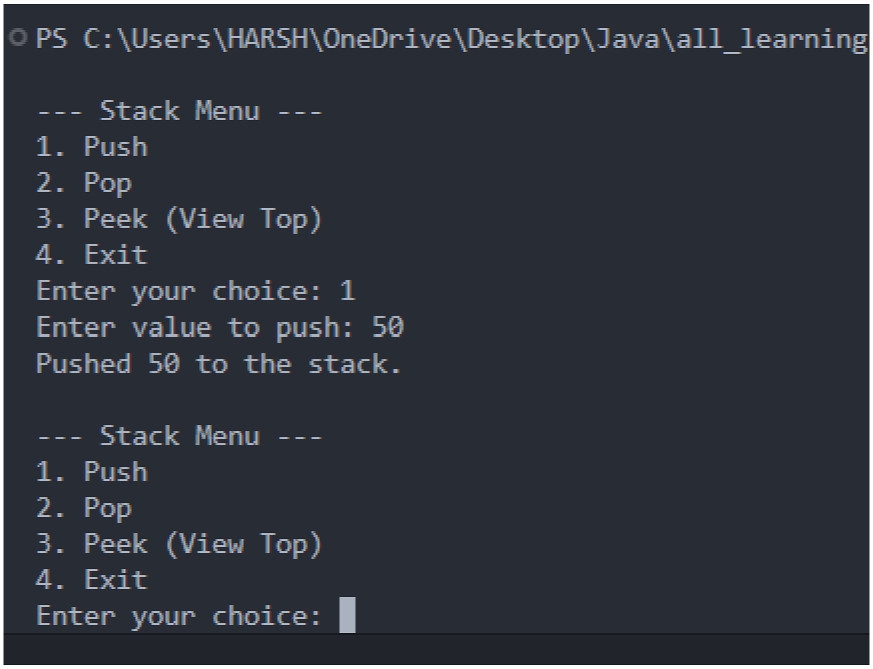
Code:

```
import java.util.Scanner;
class ArrayStack {
    private int maxSize;
    private int[] stackArray;
    private int top;
```

```
public ArrayStack(int size) {
    this.maxSize = size;
    this.stackArray = new int[maxSize];
    this.top = -1; // Indicates stack is empty
}

public void push(int item) {
    if (isFull()) {
        System.out.println("Stack Overflow! Cannot push " + item);
    } else {
        top++;
        stackArray[top] = item;
        System.out.println("Pushed " + item + " to the stack.");
    }
}

public int pop() {
    if (isEmpty()) {
        System.out.println("Stack Underflow! Stack is empty.");
        return -1; // Return a sentinel value
    } else {
        int item = stackArray[top];
        top--;
        return item;
    }
}
public int peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty.");
        return -1; // Return a sentinel value
    } else {
        return stackArray[top];
    }
}
```

```
    public boolean isEmpty() {
        return (top == -1);
    }
    public boolean isFull() {
        return (top == maxSize - 1);
    }
}

public class StackDemo {
    public static void main(String[] args) {
        ArrayStack myStack = new ArrayStack(5);
        Scanner scanner = new Scanner(System.in);
        int choice, value;

        while (true) {
            System.out.println("\n--- Stack Menu ---");
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Peek (View Top)");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter value to push: ");
                    value = scanner.nextInt();
                    myStack.push(value);
                    break;
                case 2:
                    if (!myStack.isEmpty()) {
                        System.out.println("Popped " + myStack.pop() + " from the
stack.");
                    } else {
                        myStack.pop(); // To print underflow message
                    }
                    break;
                case 3:
                     if (!myStack.isEmpty()) {
                        System.out.println("Top element is: " + myStack.peek());
                    } else {
                        myStack.peek(); // To print empty message
                    }
                    break;
                case 4:
                    System.out.println("Exiting...");
                    scanner.close();
                    return;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

RESULT



```
PS C:\Users\HARSH\OneDrive\Desktop\Java\all_learning

--- Stack Menu ---
1. Push
2. Pop
3. Peek (View Top)
4. Exit
Enter your choice: 1
Enter value to push: 50
Pushed 50 to the stack.

--- Stack Menu ---
1. Push
2. Pop
3. Peek (View Top)
4. Exit
Enter your choice: 
```

3.5

AIM: Write an abstract class Employee with three variables name, sal and Grosssal, suitable constructors, print method and two abstract methods calculate_gross_salary() and annual_increment(). Create a manager sub class of employee with Hra as member variable and write implementation of the abstract method. Also create a subclass of manager as sales manager with commission as member variable and override the calculate_gross_salary() method.

Logic:

This question uses **Abstraction** to create a template (`Employee`) that forces its concrete subclasses (`Manager`, `SalesManager`) to implement specific behaviors.

1. **Abstract Base Class (`Employee`):**
    - Declare the class as `abstract`.
    - Include variables `name`, `sal`, `Grosssal` and a constructor to initialize them.
    - Declare two `abstract` methods: `abstract void calculate_gross_salary();` and `abstract void annual_increment();`. These methods have no body.
2. **First Subclass (`Manager`):**
    - Create a class `Manager` that `extends Employee`.
    - Add a member variable `Hra`.
    - You **must provide a concrete implementation** for both `calculate_gross_salary()` (e.g., `Grosssal = sal + Hra`) and `annual_increment()`.
3. **Second Subclass (`SalesManager`):**
    - Create a class `SalesManager` that `extends Manager`.
    - Add a member variable `commission`.

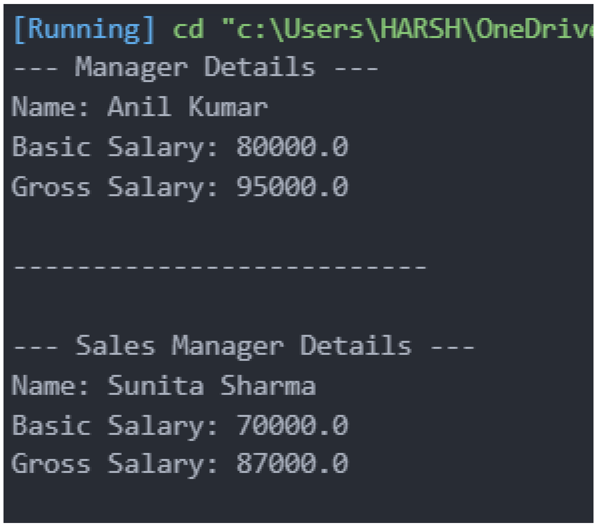
    - **Override** the `calculate_gross_salary()` method again. This new logic should also include the `commission` in its calculation (e.g., `Grosssal = sal + Hra + commission`).

Code:

```
import java.util.Scanner;

// Abstract Base Class
abstract class Employee {
    String name;
    double sal;
    double grossSal;

    public Employee(String name, double sal) {
        this.name = name;
        this.sal = sal;
    }

    public void printDetails() {
        System.out.println("Name: " + name);
        System.out.println("Basic Salary: " + sal);
        System.out.println("Gross Salary: " + grossSal);
    }

    // Abstract methods to be implemented by subclasses
    public abstract void calculate_gross_salary();
    public abstract void annual_increment();
}

// Concrete Subclass 1
class Manager extends Employee {
    double Hra;
    public Manager(String name, double sal, double Hra) {
        super(name, sal);
        this.Hra = Hra;
    }

    @Override
    public void calculate_gross_salary() {
        this.grossSal = this.sal + this.Hra;
    }

    @Override
    public void annual_increment() {
        // Example increment logic: 10% of basic salary
        this.sal += this.sal * 0.10;
        System.out.println("Salary incremented for Manager.");
    }
}

// Concrete Subclass 2
class SalesManager extends Manager {
    double commission;

    public SalesManager(String name, double sal, double Hra, double
commission) {
        super(name, sal, Hra);
        this.commission = commission;
    }

    // Overriding the method from Manager
    @Override
    public void calculate_gross_salary() {
        this.grossSal = this.sal + this.Hra + this.commission;
    }
}
```

```
public class EmployeeDemo {
    public static void main(String[] args) {
        // Creating a Manager object
        Manager mgr = new Manager("Anil Kumar", 80000, 15000);
        mgr.calculate_gross_salary();
        System.out.println("--- Manager Details ---");
        mgr.printDetails();

        System.out.println("\n--------------------------\n");

        // Creating a SalesManager object
        SalesManager smgr = new SalesManager("Sunita Sharma", 70000,
12000, 5000);
        smgr.calculate_gross_salary();
        System.out.println("--- Sales Manager Details ---");
        smgr.printDetails();
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive
--- Manager Details ---
Name: Anil Kumar
Basic Salary: 80000.0
Gross Salary: 95000.0

--------------------------

--- Sales Manager Details ---
Name: Sunita Sharma
Basic Salary: 70000.0
Gross Salary: 87000.0
```

3.6

AIM: Write a program to show the use of Interfaces

Logic:

Java doesn't allow a class to extend multiple other classes, but it achieves **multiple inheritance of behavior** by allowing a class to `implement` multiple **Interfaces**.

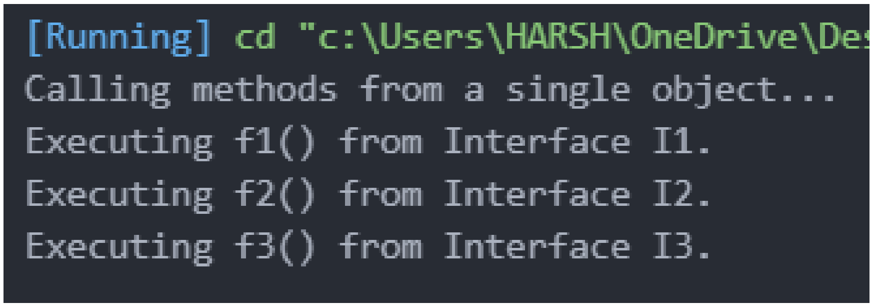
1. **Define Interfaces:** Based on the diagram, create three separate interfaces, for example, `I1`, `I2`, and `I3`.
2. **Declare Methods:** Inside each interface, declare a single abstract method signature: `void f1();` in `I1`, `void f2();` in `I2`, and `void f3();` in `I3`.
3. **Implement in a Class:** Create a single class (e.g., `MyClass`) that implements all three interfaces: `class MyClass implements I1, I2, I3`.
4. **Provide Implementations:** Because `MyClass` promises to implement these interfaces, it is now **required** to provide a body for all three methods: `f1()`, `f2()`, and `f3()`. Inside each method, you can simply print a message like "Method f1 was called."
5. **Demonstration:** In `main`, create an object of `MyClass`. You can then call `.f1()`, `.f2()`, and `.f3()` on that single object, proving it has combined behaviors from three different sources.

Code:

```
interface I1 {
    void f1(); // abstract by default
}
interface I2 {
    void f2(); // abstract by default
}
```

```
interface I3 {
    void f3(); // abstract by default
}
class MyClass implements I1, I2, I3 {
    @Override
    public void f1() {
        System.out.println("Executing f1() from Interface I1.");
    }
    @Override
    public void f2() {
        System.out.println("Executing f2() from Interface I2.");
    }
    @Override
    public void f3() {
        System.out.println("Executing f3() from Interface I3.");
    }
}
public class InterfaceDemo {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        System.out.println("Calling methods from a single object...");
        obj.f1();
        obj.f2();
        obj.f3();
    }
}
```

RESULT



```
[Running] cd "c:\Users\HARSH\OneDrive\De
Calling methods from a single object...
Executing f1() from Interface I1.
Executing f2() from Interface I2.
Executing f3() from Interface I3.
```

Sr. No.	Assignment Name	Submission Date	Due date	Remarks

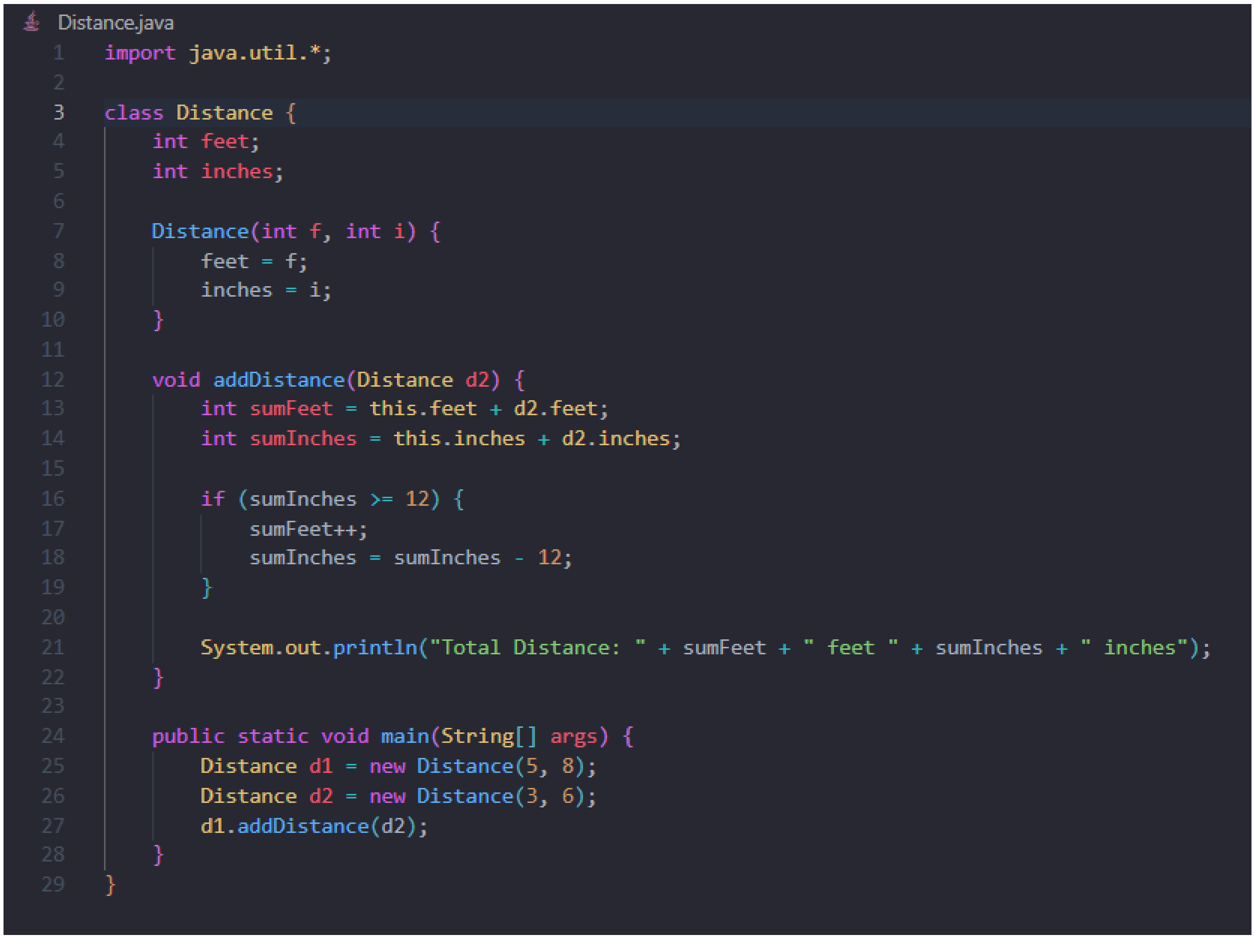
LAB ASSIGNMENT 4

4.1

AIM: Create a class distance and write a method to add two distances.

Logic: We create a class named `Distance` at line 3 with integer variables for `feet` and `inches`. We define a constructor to initialize these values. We define a method named `addDistance` that takes another `Distance` object as an argument. Inside this method, we sum the feet and inches of the current object and the passed object. We use a logic check: if inches are greater than 12, we increment the feet count and adjust the inches. Finally, in the main method, we create two objects, call the add method, and print the result.

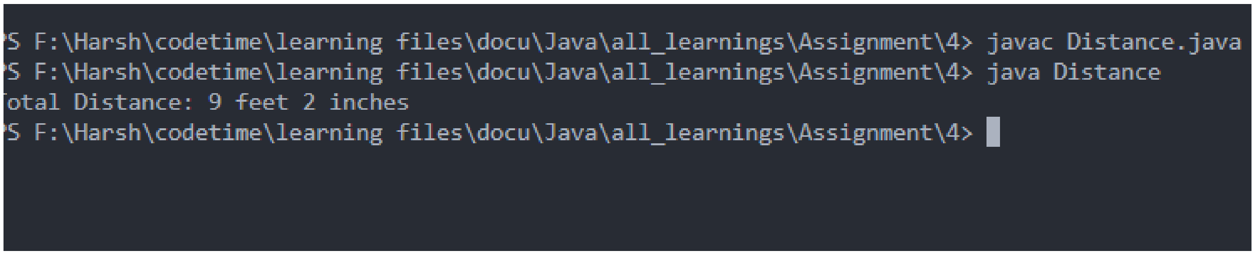
Code: Fig 4.1



Distance.java

```
1   import java.util.*;
2
3   class Distance {
4       int feet;
5       int inches;
6
7       Distance(int f, int i) {
8           feet = f;
9           inches = i;
10      }
11
12      void addDistance(Distance d2) {
13          int sumFeet = this.feet + d2.feet;
14          int sumInches = this.inches + d2.inches;
15
16          if (sumInches >= 12) {
17              sumFeet++;
18              sumInches = sumInches - 12;
19          }
20
21          System.out.println("Total Distance: " + sumFeet + " feet " + sumInches + " inches");
22      }
23
24      public static void main(String[] args) {
25          Distance d1 = new Distance(5, 8);
26          Distance d2 = new Distance(3, 6);
27          d1.addDistance(d2);
28      }
29  }
```

Result: Fig 4.2



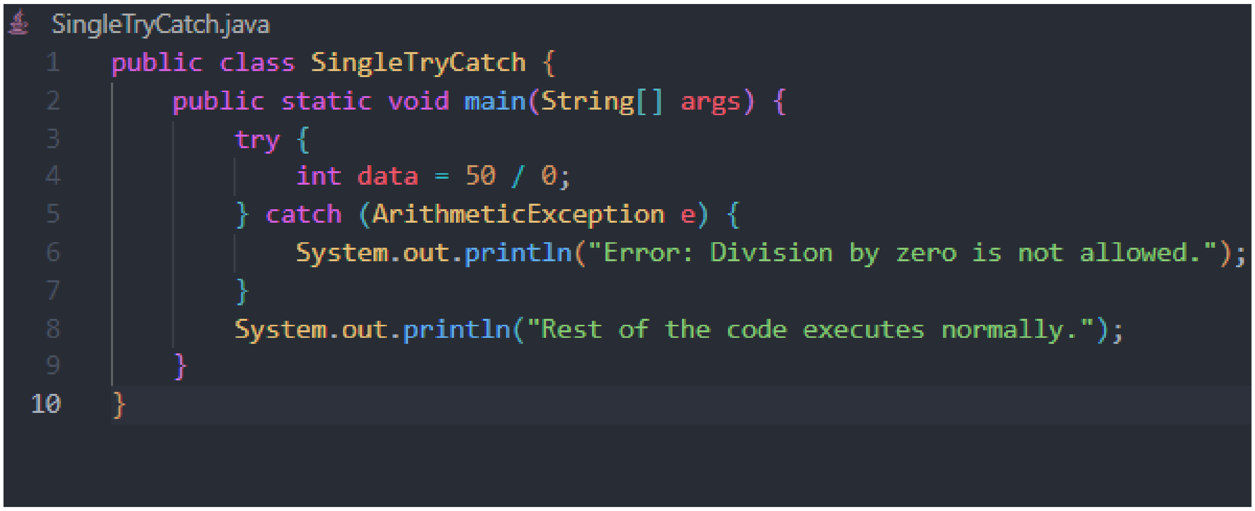
```
S F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\4> javac Distance.java
S F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\4> java Distance
otal Distance: 9 feet 2 inches
S F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\4>
```

4.2

AIM: Demonstrate Exception handling using single try catch block

Logic: We initiate a `main` class. Inside the `main` method, we start a `try` block. Inside the `try` block, we attempt a mathematical operation that is impossible (dividing a number by zero). This will generate an `ArithmeticException`. Immediately following the `try` block, we write a `catch` block that looks for `ArithmeticException`. Inside the catch block, we use `System.out.println` to print a custom message indicating that an error occurred, preventing the program from crashing abruptly.

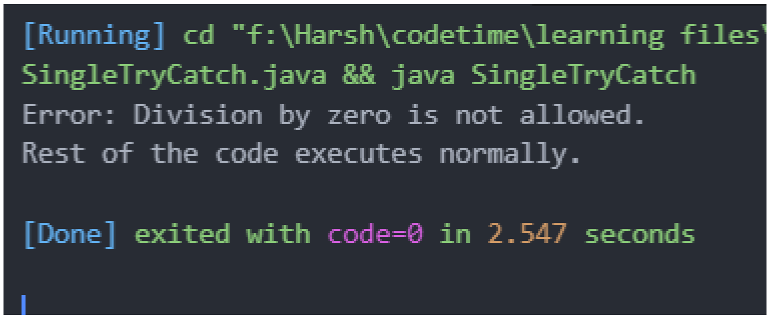
Code: Fig 4.3



SingleTryCatch.java

```java
public class SingleTryCatch {
    public static void main(String[] args) {
        try {
            int data = 50 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Error: Division by zero is not allowed.");
        }
        System.out.println("Rest of the code executes normally.");
    }
}
```

Result: Fig 4.4



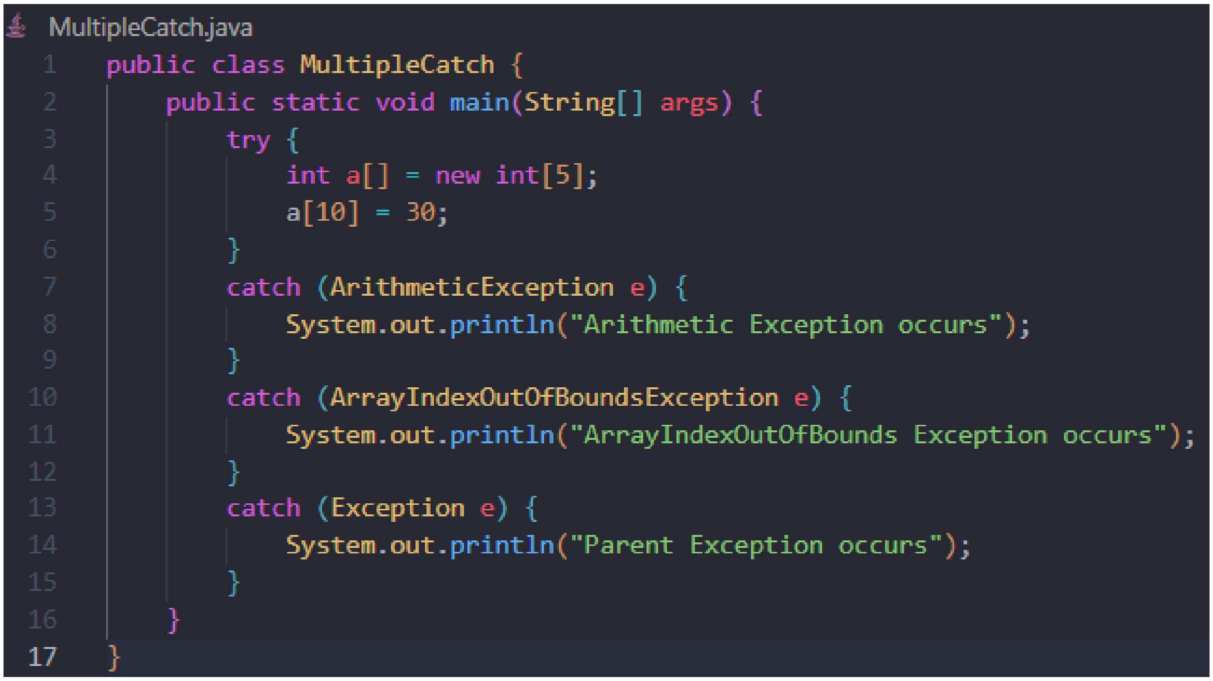
```
[Running] cd "f:\Harsh\codetime\learning files\
SingleTryCatch.java && java SingleTryCatch
Error: Division by zero is not allowed.
Rest of the code executes normally.

[Done] exited with code=0 in 2.547 seconds
```

4.3

AIM: Show use of multiple catch block

Logic: We create a `main` class and open a `try` block. Inside the `try` block, we create an array of size 5 and try to assign a value to index 10. This is invalid and will cause an exception. We then write multiple `catch` blocks sequentially. The first `catch` block looks for `ArithmeticException`, the second looks for `ArrayIndexOutOfBoundsException`, and the third looks for a generic `Exception`. The program will check them in order and execute only the block that matches the specific error (in this case, the Array index error).

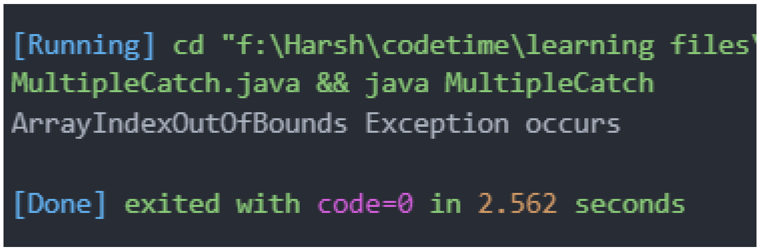
Code: Fig 4.5



MultipleCatch.java

```java
public class MultipleCatch {
    public static void main(String[] args) {
        try {
            int a[] = new int[5];
            a[10] = 30;
        }
        catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception occurs");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
        }
        catch (Exception e) {
            System.out.println("Parent Exception occurs");
        }
    }
}
```

Result: Fig 4.6



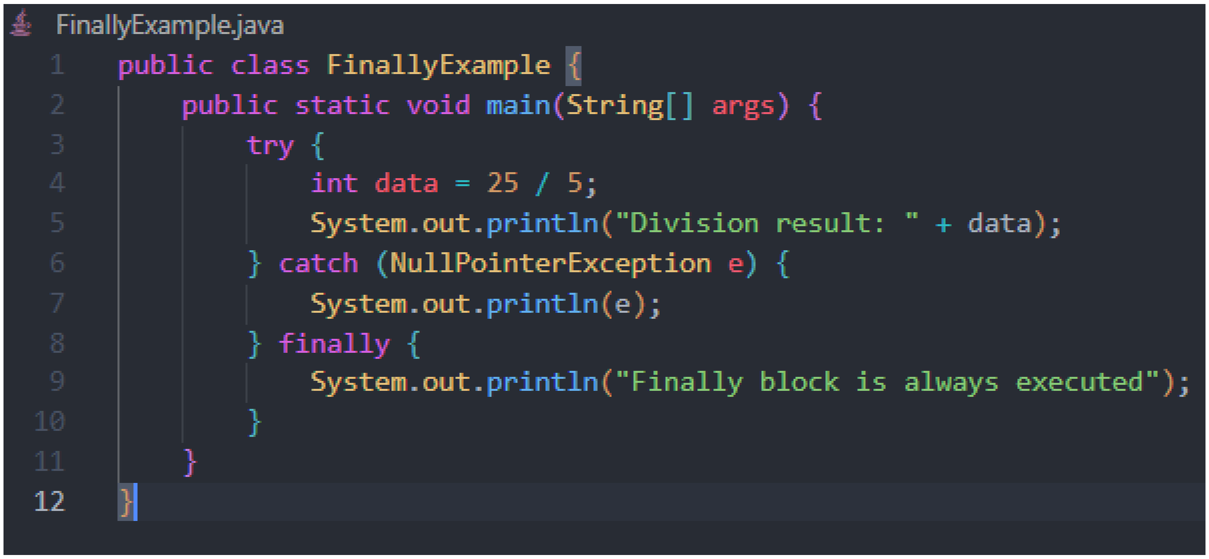
```
[Running] cd "f:\Harsh\codetime\learning files
MultipleCatch.java && java MultipleCatch
ArrayIndexOutOfBounds Exception occurs

[Done] exited with code=0 in 2.562 seconds
```

4.4

AIM: Show use of finally block

Logic: We create a class containing a `main` method. We start a `try` block where we perform a division operation. We add a `catch` block to handle potential division errors. After the catch block, we add a `finally` block. The logic here is that the code inside the `finally` block (a print statement saying "Finally block is always executed") will run regardless of whether an exception occurred in the try block or not. It ensures cleanup code or final messages are always processed.

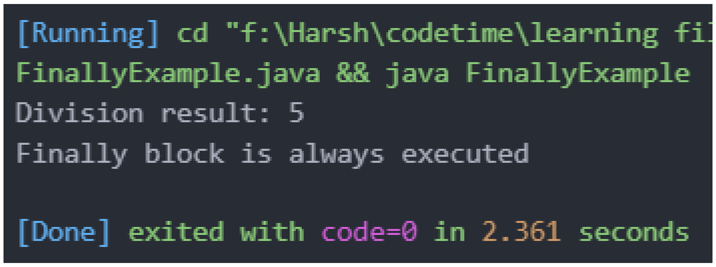
Code: Fig 4.7



FinallyExample.java

```java
public class FinallyExample {
    public static void main(String[] args) {
        try {
            int data = 25 / 5;
            System.out.println("Division result: " + data);
        } catch (NullPointerException e) {
            System.out.println(e);
        } finally {
            System.out.println("Finally block is always executed");
        }
    }
}
```

Result: Fig 4.8



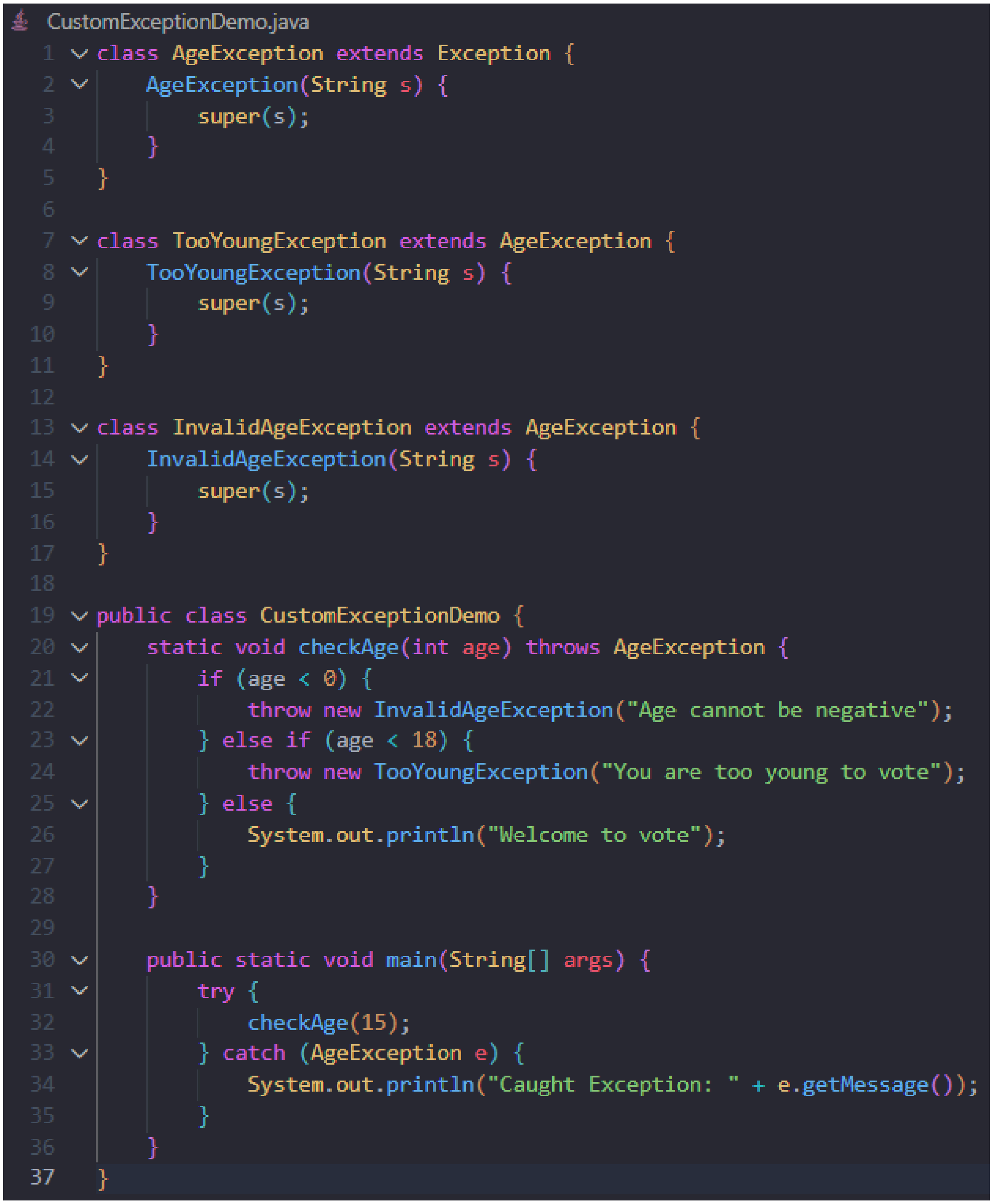
```
[Running] cd "f:\Harsh\codetime\learning fi
FinallyExample.java && java FinallyExample
Division result: 5
Finally block is always executed

[Done] exited with code=0 in 2.361 seconds
```

4.5

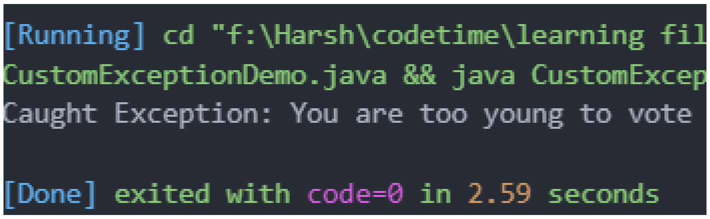
AIM: Write a program to show use of throw and throw keywords. Create your own exception class AgeException. Also create two subclasses TooYoungException and InvalidAgeException of AgeException as discussed in class.

Logic: First, we define a custom exception class `AgeException` that extends the standard `Exception` class. Then we create two subclasses: `TooYoungException` and `InvalidAgeException`, both extending `AgeException`. In the main class, we define a static method named `checkAge` that accepts an integer. We use the `throws` keyword in the method signature to declare that this method might throw an `AgeException`. Inside the method, we use `if` statements and the `throw` keyword: if age is less than 0, we manually throw `InvalidAgeException`; if age is less than 18, we throw `TooYoungException`. In the `main` method, we call `checkAge` inside a `try-catch` block to handle these custom exceptions.

Code: Fig 4.9



```java
class AgeException extends Exception {
    AgeException(String s) {
        super(s);
    }
}

class TooYoungException extends AgeException {
    TooYoungException(String s) {
        super(s);
    }
}

class InvalidAgeException extends AgeException {
    InvalidAgeException(String s) {
        super(s);
    }
}

public class CustomExceptionDemo {
    static void checkAge(int age) throws AgeException {
        if (age < 0) {
            throw new InvalidAgeException("Age cannot be negative");
        } else if (age < 18) {
            throw new TooYoungException("You are too young to vote");
        } else {
            System.out.println("Welcome to vote");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (AgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

Result: Fig 4.10 

```
[Running] cd "f:\Harsh\codetime\learning fil
CustomExceptionDemo.java && java CustomExcep
Caught Exception: You are too young to vote

[Done] exited with code=0 in 2.59 seconds
```

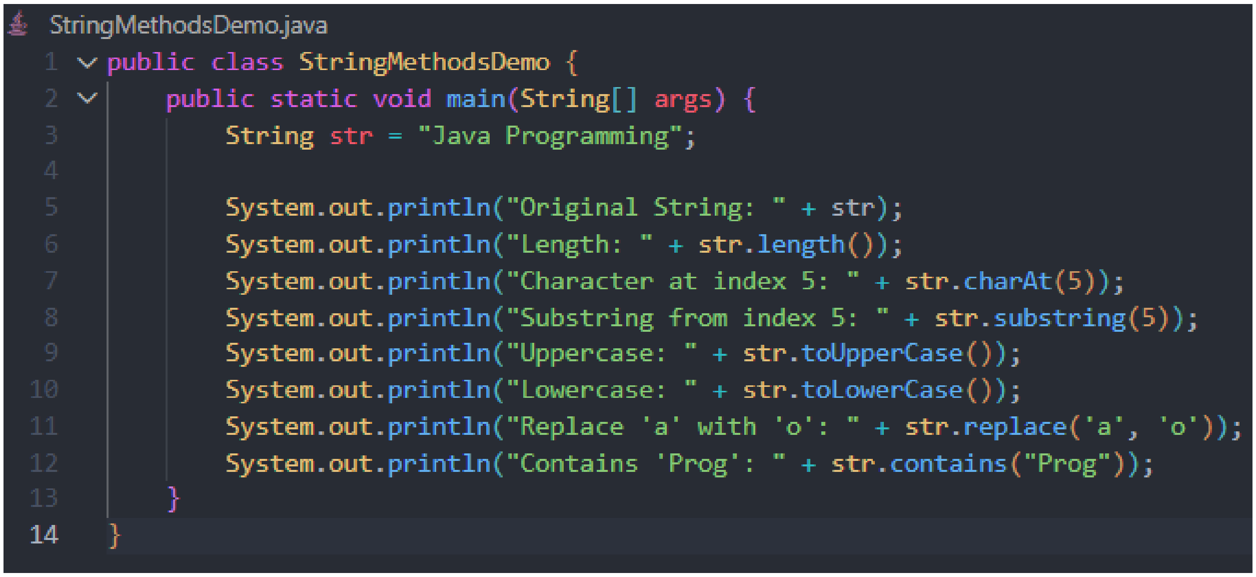
LAB ASSIGNMENT 5

5.1

AIM: Write a program to demonstrate the use of all String class methods

Logic: We create a class with a `main` method. We define a string variable. Then we sequentially call various common methods of the String class. We use `length()` to get the size, `charAt()` to access specific characters, `substring()` to extract parts of the string, `toUpperCase()` and `toLowerCase()` for case conversion, and `replace()` to swap characters. We print the result of each method directly to the console to demonstrate how they work.

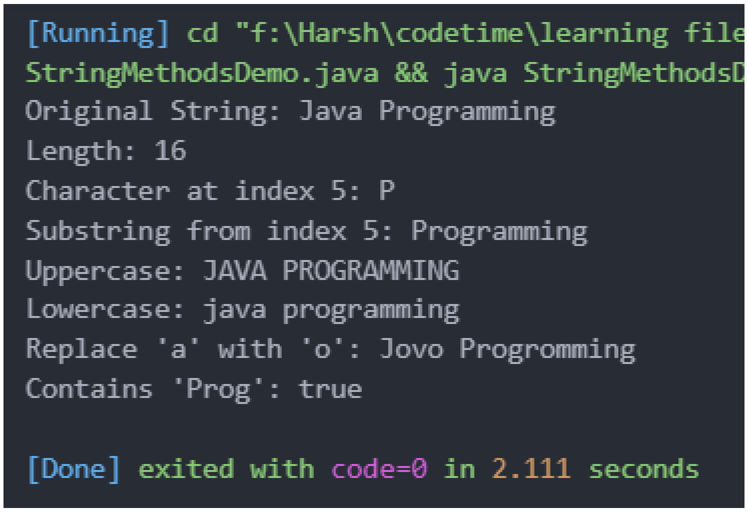
Code: Fig 5.1



StringMethodsDemo.java

```java
public class StringMethodsDemo {
    public static void main(String[] args) {
        String str = "Java Programming";

        System.out.println("Original String: " + str);
        System.out.println("Length: " + str.length());
        System.out.println("Character at index 5: " + str.charAt(5));
        System.out.println("Substring from index 5: " + str.substring(5));
        System.out.println("Uppercase: " + str.toUpperCase());
        System.out.println("Lowercase: " + str.toLowerCase());
        System.out.println("Replace 'a' with 'o': " + str.replace('a', 'o'));
        System.out.println("Contains 'Prog': " + str.contains("Prog"));
    }
}
```

Result: Fig 5.2



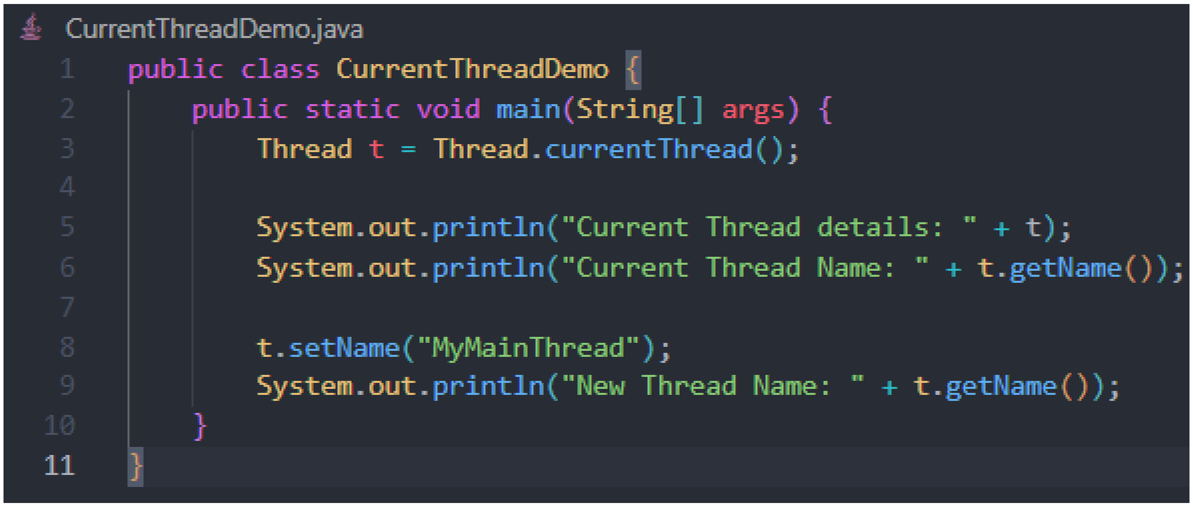
```
[Running] cd "f:\Harsh\codetime\learning file
StringMethodsDemo.java && java StringMethodsD
Original String: Java Programming
Length: 16
Character at index 5: P
Substring from index 5: Programming
Uppercase: JAVA PROGRAMMING
Lowercase: java programming
Replace 'a' with 'o': Jovo Progromming
Contains 'Prog': true

[Done] exited with code=0 in 2.111 seconds
```

5.2

AIM: Write a program to display the current thread

Logic: We define a class containing the `main` method. Inside the main method, we use the `Thread` class's static method `currentThread()` to get a reference to the thread that is currently executing the code. We assign this to a Thread object named `t`. Then, we use `System.out.println` to print the thread object, which displays the thread's name, priority, and group. We also use `t.getName()` to specifically print just the name of the thread.

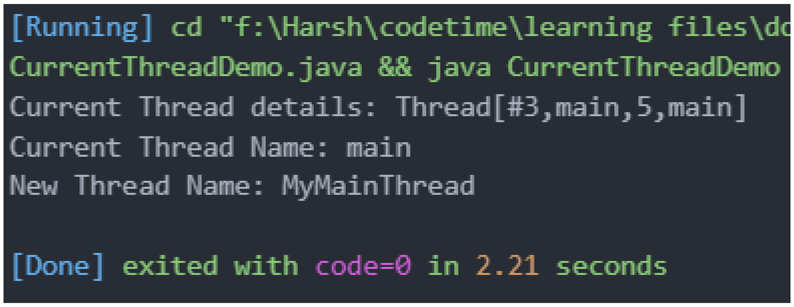
Code: Fig 5.3



CurrentThreadDemo.java

```java
public class CurrentThreadDemo {
    public static void main(String[] args) {
        Thread t = Thread.currentThread();

        System.out.println("Current Thread details: " + t);
        System.out.println("Current Thread Name: " + t.getName());

        t.setName("MyMainThread");
        System.out.println("New Thread Name: " + t.getName());
    }
}
```

Result: Fig 5.4



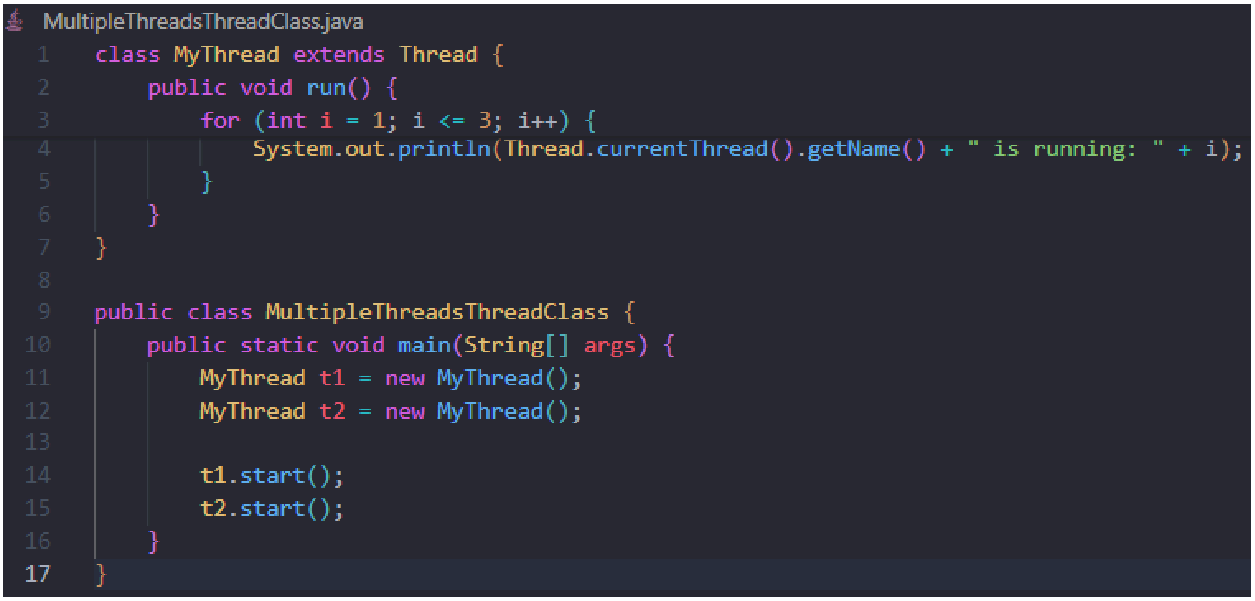
```
[Running] cd "f:\Harsh\codetime\learning files\d
CurrentThreadDemo.java && java CurrentThreadDemo
Current Thread details: Thread[#3,main,5,main]
Current Thread Name: main
New Thread Name: MyMainThread

[Done] exited with code=0 in 2.21 seconds
```

5.3

AIM: Write a program to create multiple threads, use Thread class

Logic: We create a class named `MyThread` that extends the built-in `Thread` class. We override the `run()` method inside this class, which contains the code that the thread will execute (a loop printing numbers). In the `main` class, we create two objects of our `MyThread` class. We call the `start()` method on both objects. This causes both threads to execute the `run()` method concurrently, rather than sequentially.

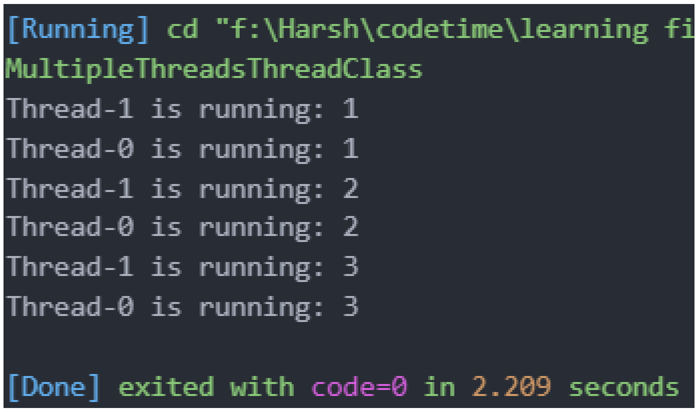
Code: Fig 5.5



MultipleThreadsThreadClass.java

```java
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + " is running: " + i);
        }
    }
}

public class MultipleThreadsThreadClass {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();
    }
}
```

Result: Fig 5.6



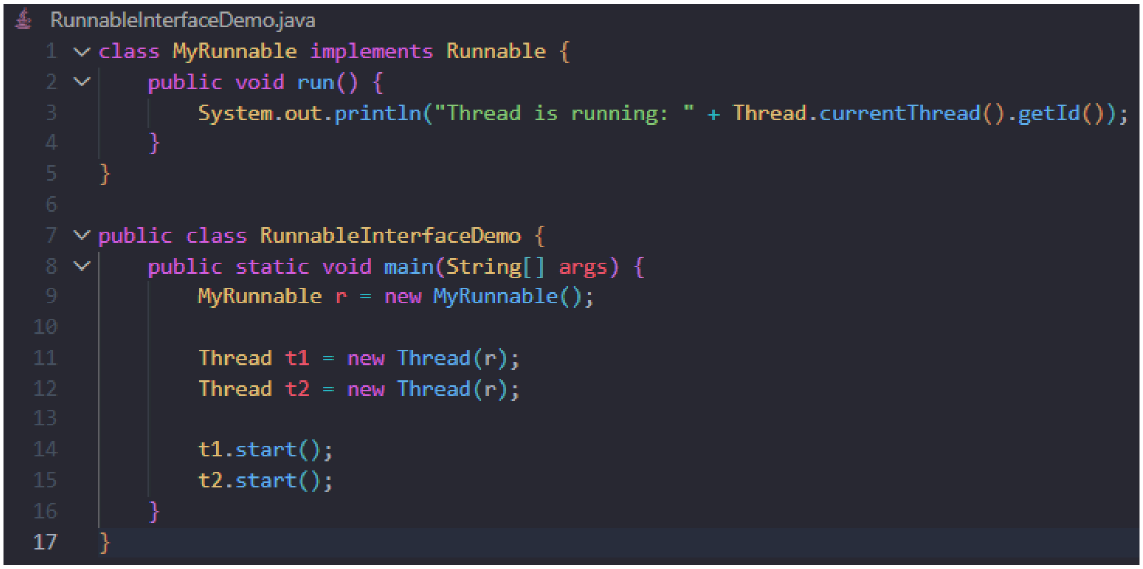
```
[Running] cd "f:\Harsh\codetime\learning fi
MultipleThreadsThreadClass
Thread-1 is running: 1
Thread-0 is running: 1
Thread-1 is running: 2
Thread-0 is running: 2
Thread-1 is running: 3
Thread-0 is running: 3

[Done] exited with code=0 in 2.209 seconds
```

5.4

AIM: Write a program to create multiple threads, implementing Runnable Interface

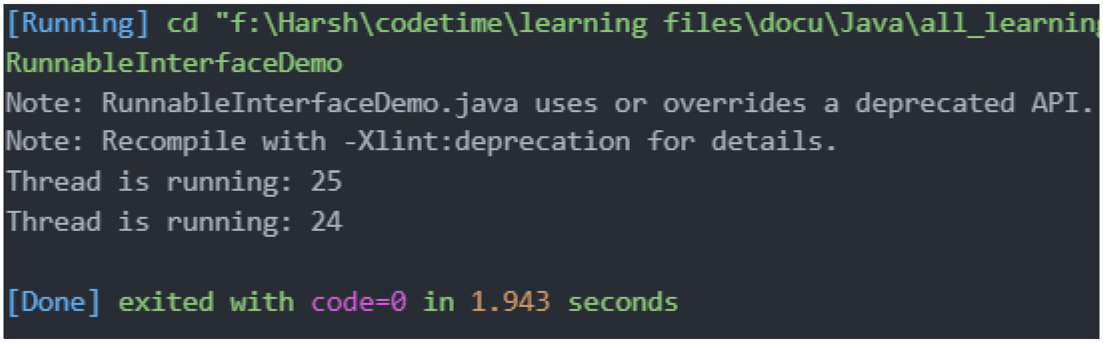
Logic: We create a class named `MyRunnable` that implements the `Runnable` interface. We define the `run()` method inside it. In the `main` class, we create an instance of `MyRunnable`. Then, we create two objects of the `Thread` class and pass the `MyRunnable` instance to their constructors. Finally, we call the `start()` method on these Thread objects to begin execution.

Code: Fig 5.7



```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running: " + Thread.currentThread().getId());
    }
}

public class RunnableInterfaceDemo {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();

        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);

        t1.start();
        t2.start();
    }
}
```

Result: Fig 5.8



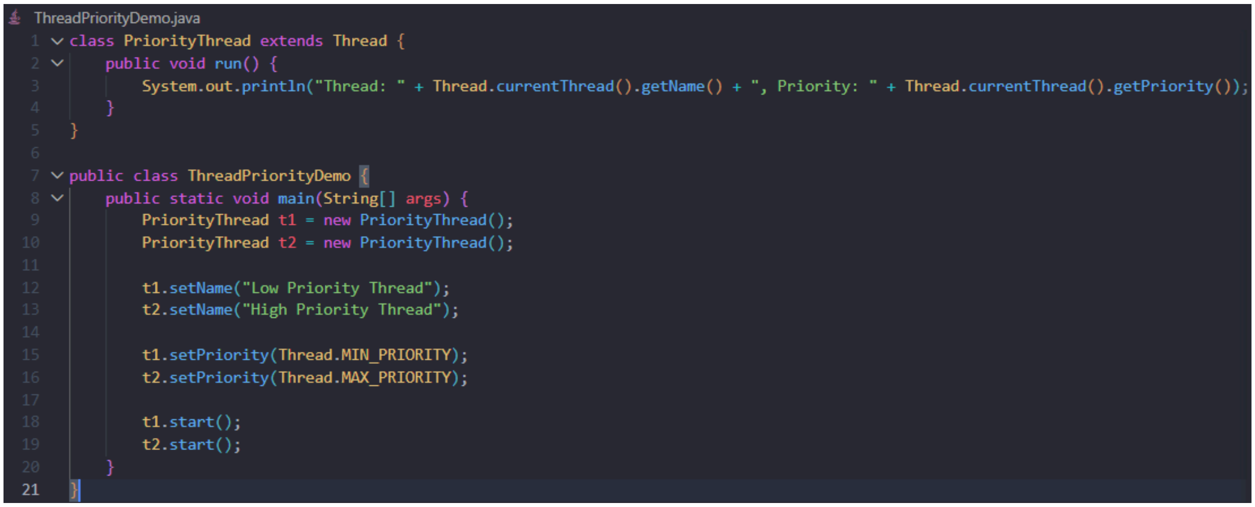
```
[Running] cd "f:\Harsh\codetime\learning files\docu\Java\all_learnin
RunnableInterfaceDemo
Note: RunnableInterfaceDemo.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Thread is running: 25
Thread is running: 24

[Done] exited with code=0 in 1.943 seconds
```

5.5

AIM: Write a program to create multiple threads by making use of thread priorities

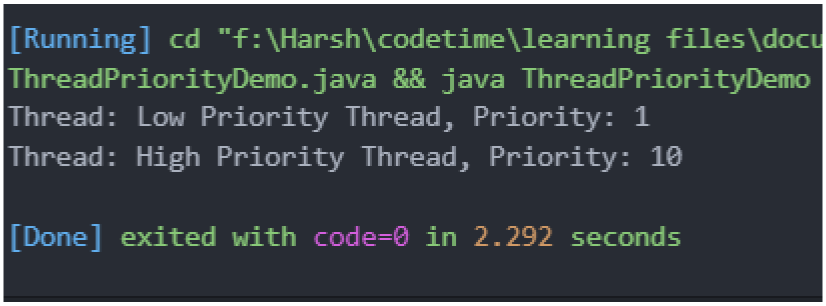
Logic: We create a class that extends `Thread`. In the `main` method, we create two thread objects. We use the `setPriority()` method to assign different priorities to them. We assign `MIN_PRIORITY` (1) to the first thread and `MAX_PRIORITY` (10) to the second thread. Inside the `run()` method, we print the thread name and its priority using `getPriority()` to verify that the settings were applied.

Code: Fig 5.9



```java
class PriorityThread extends Thread {
    public void run() {
        System.out.println("Thread: " + Thread.currentThread().getName() + ", Priority: " + Thread.currentThread().getPriority());
    }
}

public class ThreadPriorityDemo {
    public static void main(String[] args) {
        PriorityThread t1 = new PriorityThread();
        PriorityThread t2 = new PriorityThread();

        t1.setName("Low Priority Thread");
        t2.setName("High Priority Thread");

        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);

        t1.start();
        t2.start();
    }
}
```

Result: Fig 5.10



```
[Running] cd "f:\Harsh\codetime\learning files\docu
ThreadPriorityDemo.java && java ThreadPriorityDemo
Thread: Low Priority Thread, Priority: 1
Thread: High Priority Thread, Priority: 10

[Done] exited with code=0 in 2.292 seconds
```

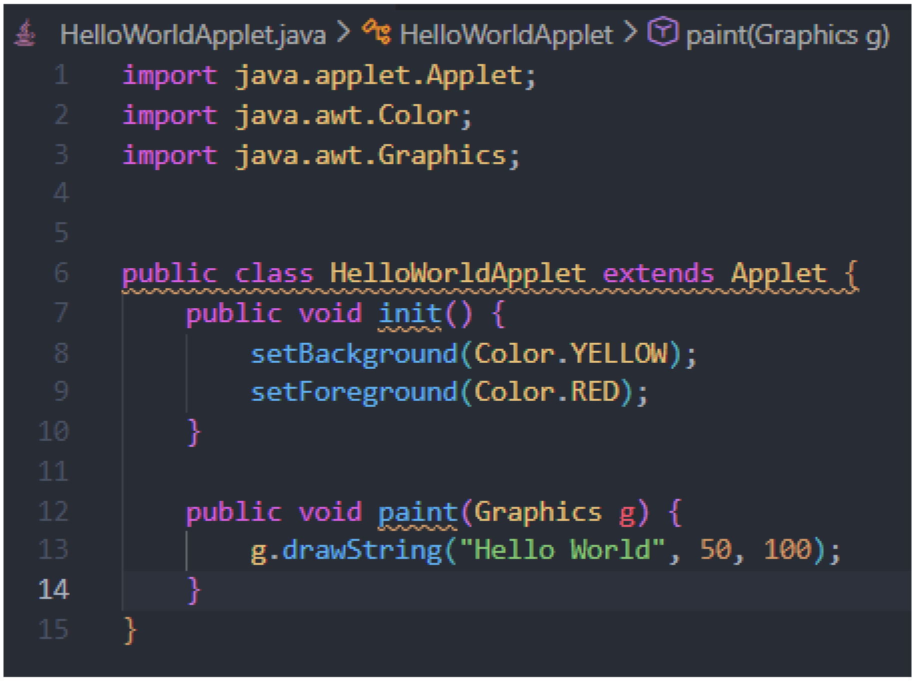
LAB ASSIGNMENT 6

6.1

AIM: Write a program to display a simple applet which displays "Hello World" and background color is yellow and foreground color is red.

Logic: We define a class that extends the `Applet` class. In the `init()` method, which is called when the applet starts, we set the background color to yellow using `setBackground(Color.YELLOW)` and the foreground (text) color to red using `setForeground(Color.RED)`. We then override the `paint` method. Inside `paint`, we use the `drawString` method of the `Graphics` object to display the text "Hello World" at specific coordinates.

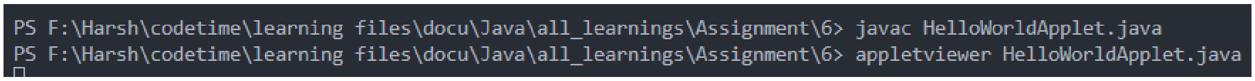
Code: Fig 6.1


HelloWorldApplet.java > HelloWorldApplet > paint(Graphics g)

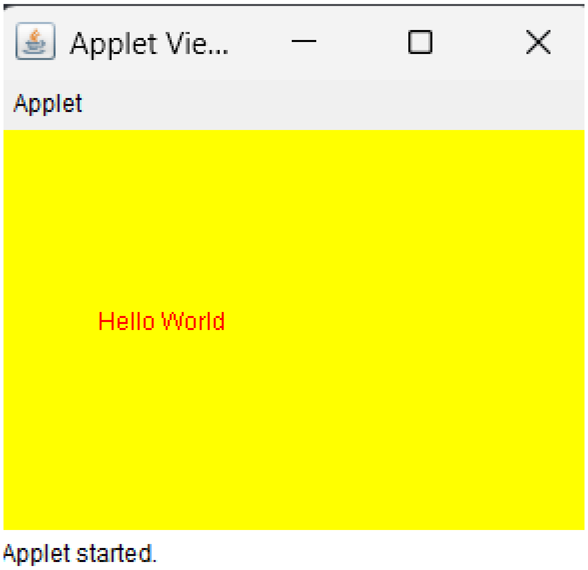
```java
import java.applet.Applet;
import java.awt.Color;
import java.awt.Graphics;


public class HelloWorldApplet extends Applet {
    public void init() {
        setBackground(Color.YELLOW);
        setForeground(Color.RED);
    }

    public void paint(Graphics g) {
        g.drawString("Hello World", 50, 100);
    }
}
```

Result:

Fig 6.2



```
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac HelloWorldApplet.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer HelloWorldApplet.java
```



```
Hello World
```

Applet started.

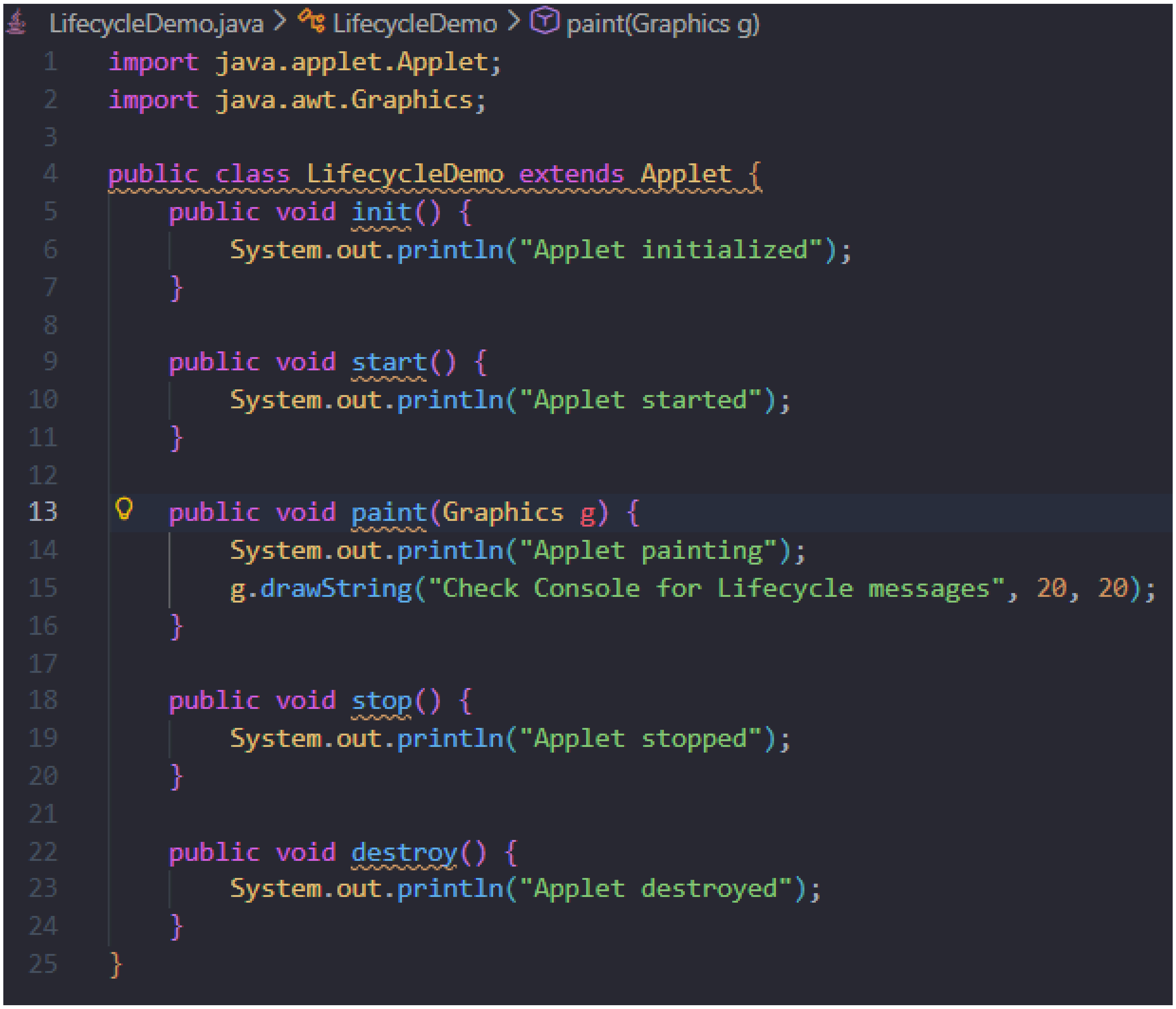
Fig 6.3

6.2

AIM: Write an applet to demonstrate lifecycle methods.

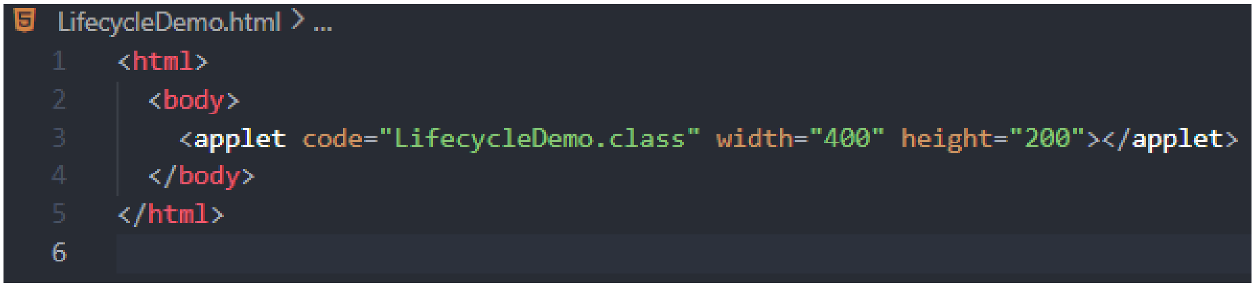
Logic: We create a class extending `Applet`. We override the five main lifecycle methods: `init()`, `start()`, `paint()`, `stop()`, and `destroy()`. Inside each method, we print a simple message to the console using `System.out.println`. This allows us to observe the order of execution in the console window when the applet is loaded, minimized, restored, or closed. We also display a message on the applet window using `paint`, We used a html document to show the applet.

Code: Fig 6.4



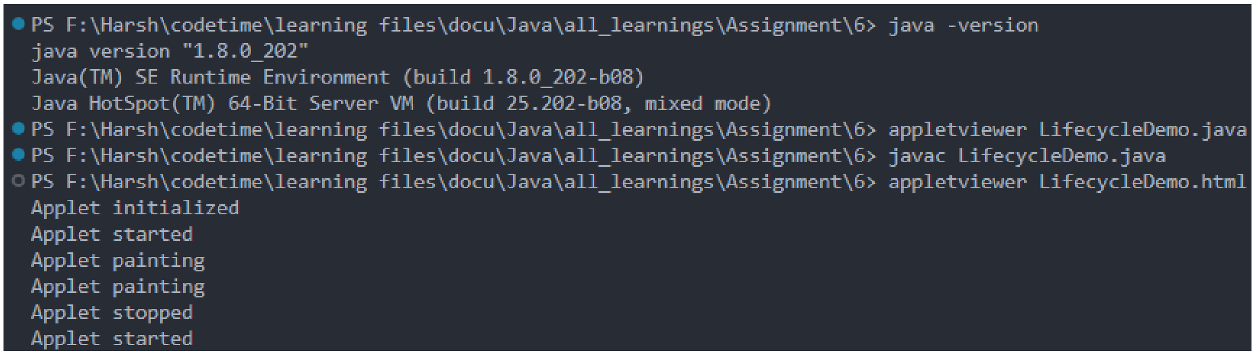
```java
import java.applet.Applet;
import java.awt.Graphics;

public class LifecycleDemo extends Applet {
    public void init() {
        System.out.println("Applet initialized");
    }

    public void start() {
        System.out.println("Applet started");
    }

    public void paint(Graphics g) {
        System.out.println("Applet painting");
        g.drawString("Check Console for Lifecycle messages", 20, 20);
    }

    public void stop() {
        System.out.println("Applet stopped");
    }

    public void destroy() {
        System.out.println("Applet destroyed");
    }
}
```

Fig: 6.5

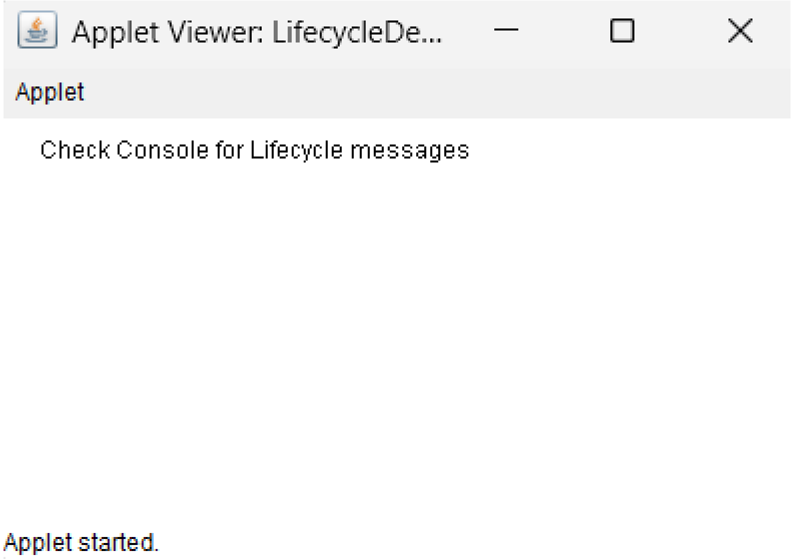


```html
<html>
  <body>
    <applet code="LifecycleDemo.class" width="400" height="200"></applet>
  </body>
</html>

```

Result: Fig 6.6



```
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> java -version
java version "1.8.0_202"
Java(TM) SE Runtime Environment (build 1.8.0_202-b08)
Java HotSpot(TM) 64-Bit Server VM (build 25.202-b08, mixed mode)
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer LifecycleDemo.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac LifecycleDemo.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer LifecycleDemo.html
Applet initialized
Applet started
Applet painting
Applet painting
Applet stopped
Applet started
```



Check Console for Lifecycle messages

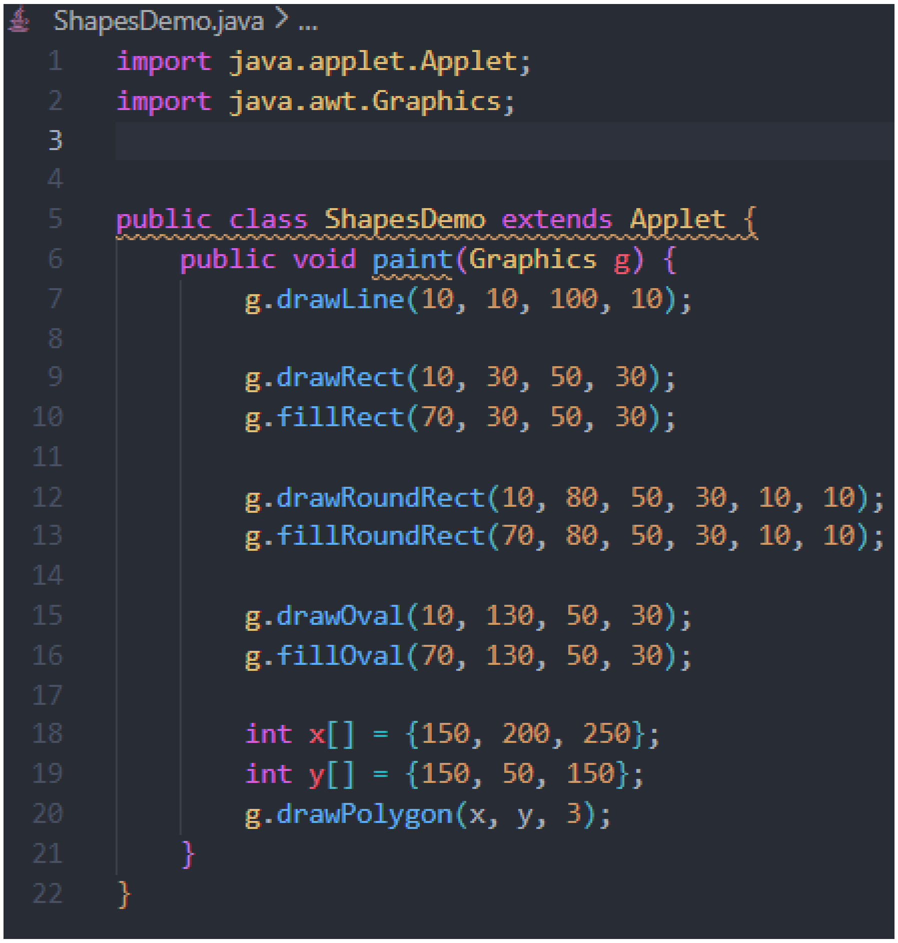
Applet started.

Fig 6.7

6.3

AIM: Write an applet which shows use of following classes: drawLine, drawRect, fillRect, drawRoundRect, fillRoundRect, drawOval, fillOval, drawPolygon.

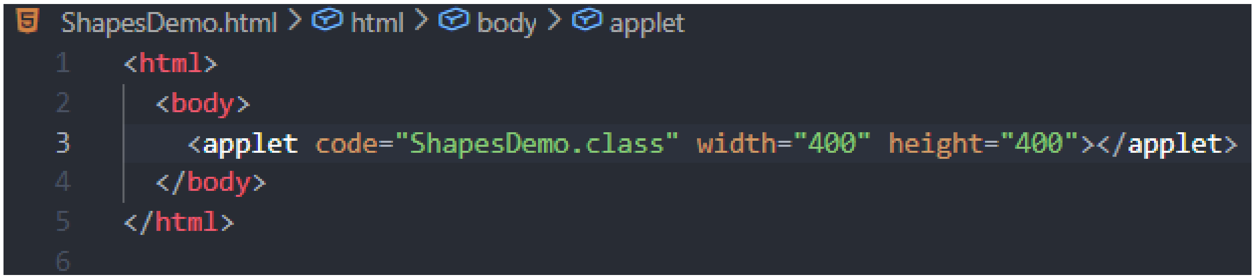
Logic: We create a class extending `Applet` and override the `paint` method. Inside `paint`, we use the `Graphics` object `g` to call various drawing methods. We use `drawLine` to draw a straight line, `drawRect` and `fillRect` for empty and filled rectangles, `drawRoundRect` and `fillRoundRect` for rectangles with curved corners, and `drawOval` and `fillOval` for circles/ellipses. For `drawPolygon`, we create two arrays containing x and y coordinates and pass them to the method along with the number of points, We used a html document to show the applet.



```java
import java.applet.Applet;
import java.awt.Graphics;


public class ShapesDemo extends Applet {
    public void paint(Graphics g) {
        g.drawLine(10, 10, 100, 10);

        g.drawRect(10, 30, 50, 30);
        g.fillRect(70, 30, 50, 30);

        g.drawRoundRect(10, 80, 50, 30, 10, 10);
        g.fillRoundRect(70, 80, 50, 30, 10, 10);

        g.drawOval(10, 130, 50, 30);
        g.fillOval(70, 130, 50, 30);

        int x[] = {150, 200, 250};
        int y[] = {150, 50, 150};
        g.drawPolygon(x, y, 3);
    }
}
```

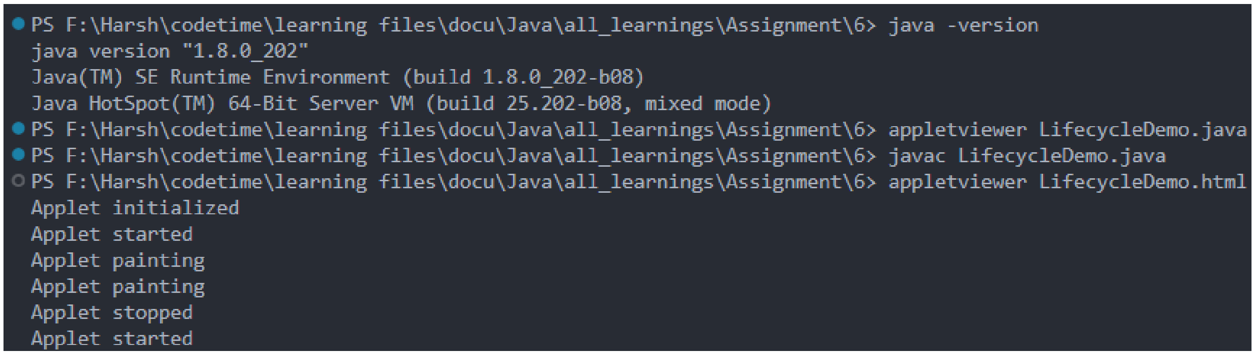
Code: Fig 6.8

Fig 6.9



```html
<html>
  <body>
    <applet code="ShapesDemo.class" width="400" height="400"></applet>
  </body>
</html>

```

Result: Fig 6.10



```
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> java -version
java version "1.8.0_202"
Java(TM) SE Runtime Environment (build 1.8.0_202-b08)
Java HotSpot(TM) 64-Bit Server VM (build 25.202-b08, mixed mode)
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer LifecycleDemo.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac LifecycleDemo.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer LifecycleDemo.html
Applet initialized
Applet started
Applet painting
Applet painting
Applet stopped
Applet started
```

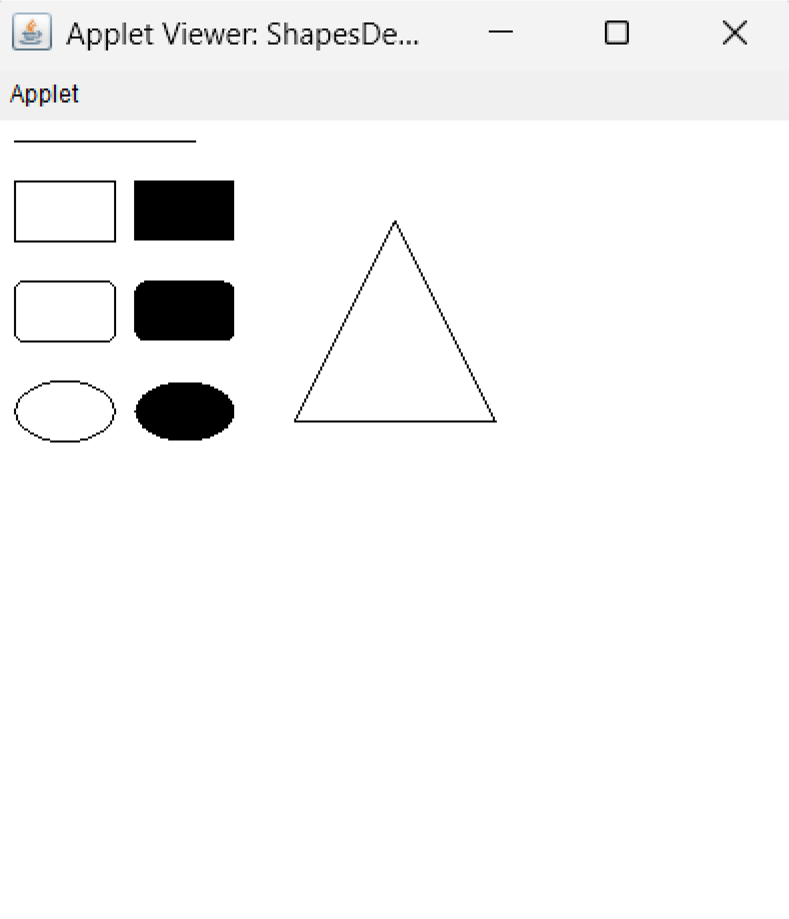


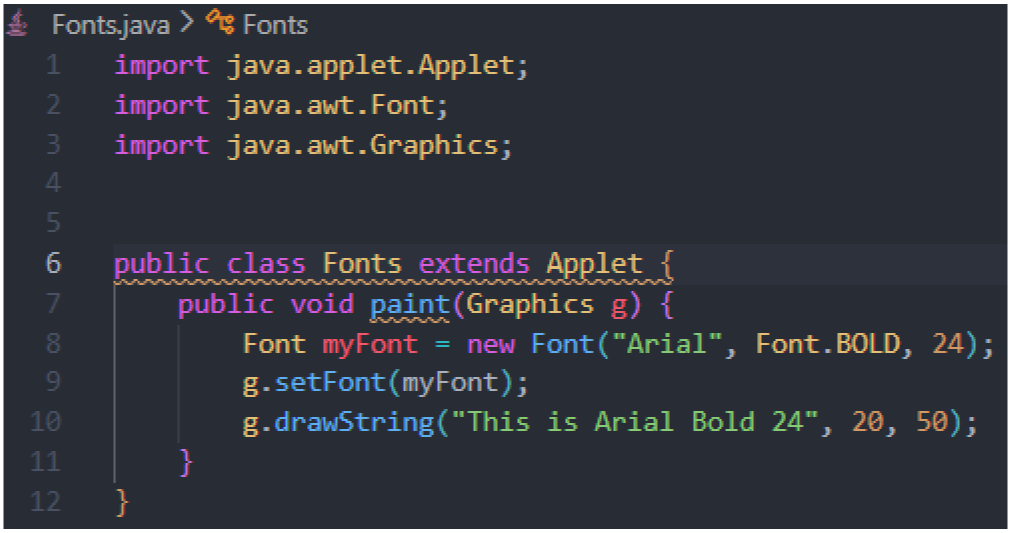
Fig 6.11 Applet started.

6.4

AIM: Write an applet to modify the font of Graphics Object.

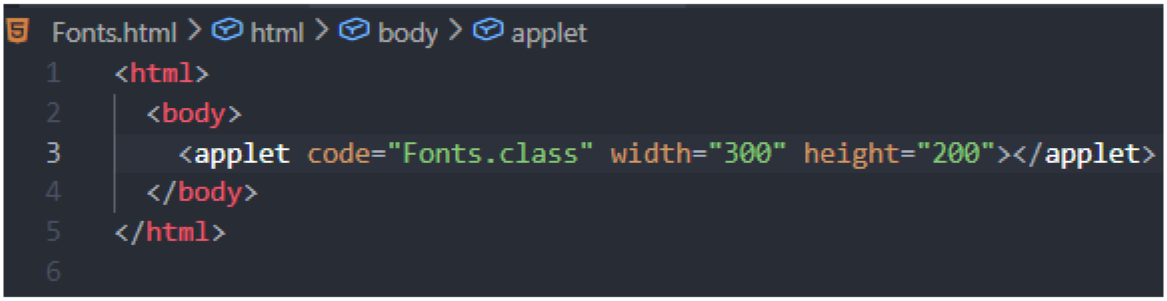
Logic: We create an applet and override the `paint` method. Inside `paint`, we define a new `Font` object with a specific name (e.g., "Arial"), style (e.g., `Font.BOLD`), and size (e.g., 20). We apply this font to the graphics environment using `g.setFont()`. Finally, we draw a string using `g.drawString()`, which will appear in the new custom font style.

Code: Fig 6.12



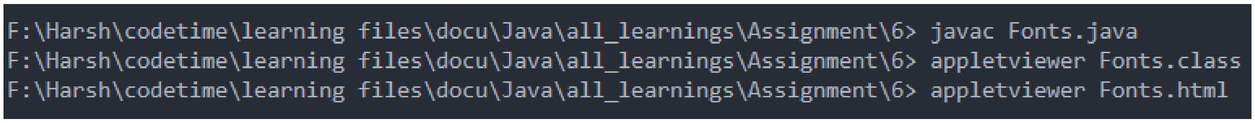
```java
import java.applet.Applet;
import java.awt.Font;
import java.awt.Graphics;


public class Fonts extends Applet {
    public void paint(Graphics g) {
        Font myFont = new Font("Arial", Font.BOLD, 24);
        g.setFont(myFont);
        g.drawString("This is Arial Bold 24", 20, 50);
    }
}
```

Fig 6.13

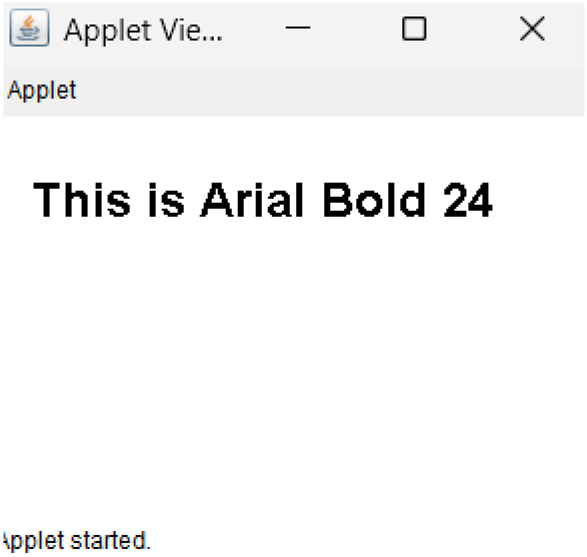


```html
<html>
  <body>
    <applet code="Fonts.class" width="300" height="200"></applet>
  </body>
</html>
```

Result: Fig 6.14



```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac Fonts.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer Fonts.class
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer Fonts.html
```



This is Arial Bold 24

Applet started.

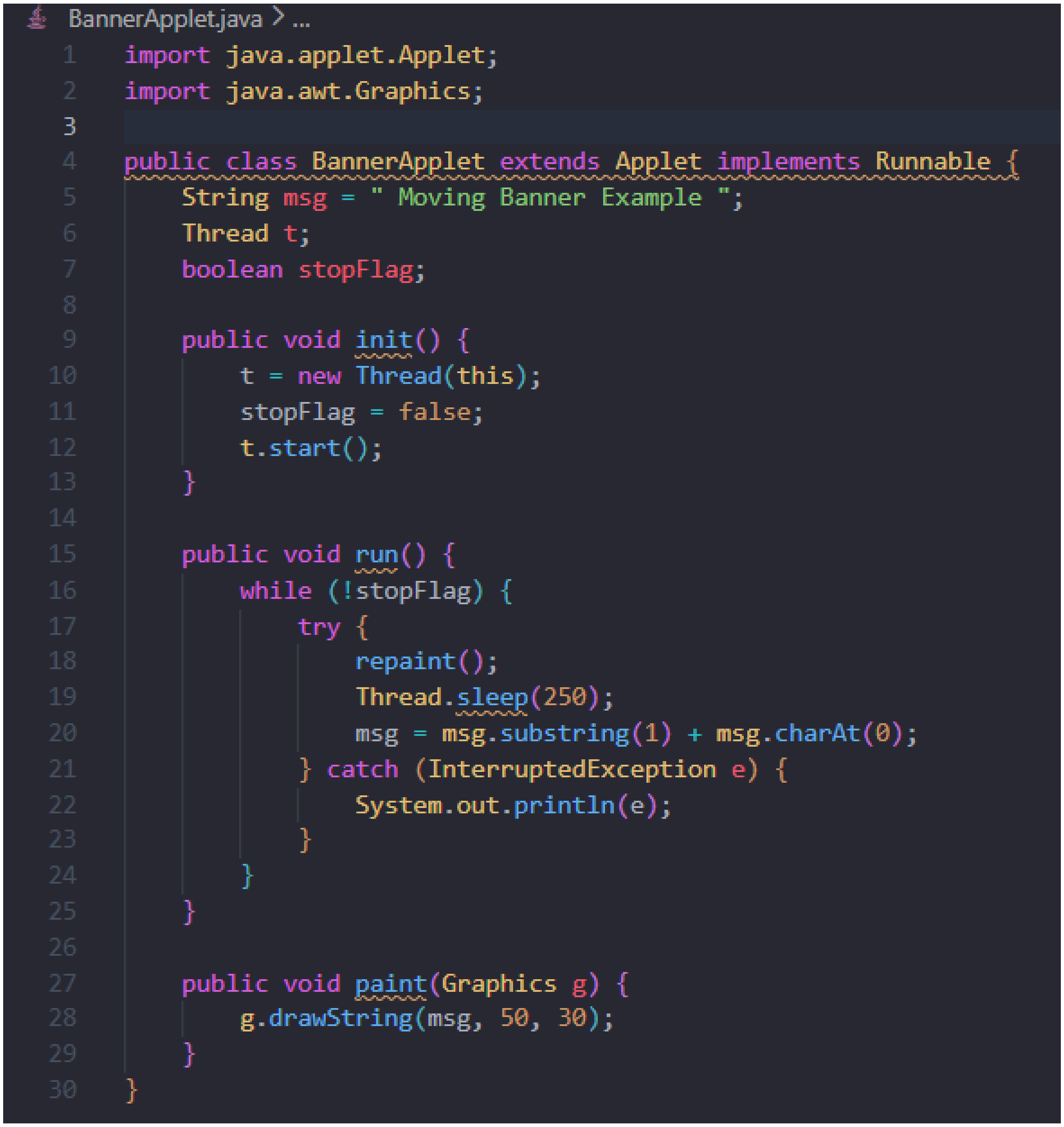
Fig 6.15

6.5

AIM: Write a moving banner program to demonstrate use of multithread and repaint method in applet.

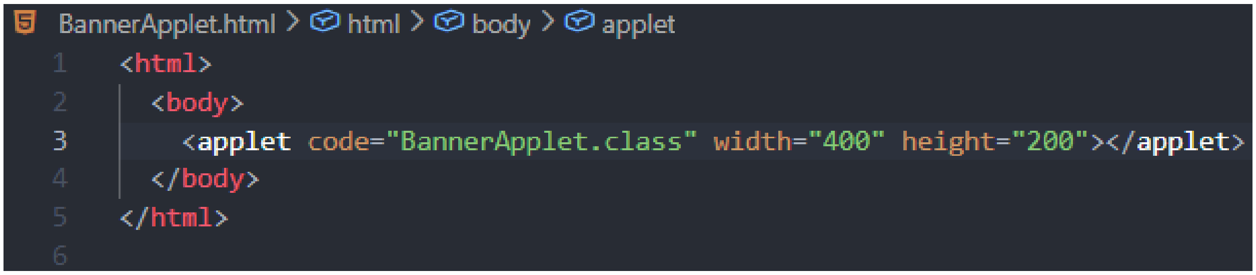
Logic: We create a class that extends `Applet` and implements the `Runnable` interface to support multithreading. We declare a `Thread` object and a String for the banner text. In `init()`, we start the thread. In the `run()` method, we use an infinite loop that repaints the applet, updates the x-coordinate of the text to move it, and then sleeps for a short time using `Thread.sleep`. In `paint()`, we simply draw the string at the current moving coordinates.

Code: Fig 6.16



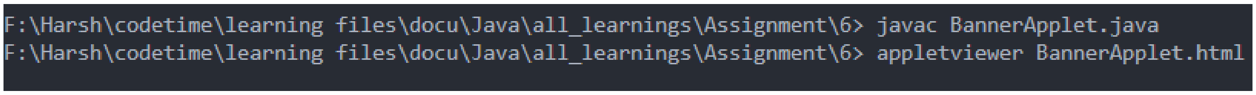
```java
import java.applet.Applet;
import java.awt.Graphics;

public class BannerApplet extends Applet implements Runnable {
    String msg = " Moving Banner Example ";
    Thread t;
    boolean stopFlag;

    public void init() {
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    public void run() {
        while (!stopFlag) {
            try {
                repaint();
                Thread.sleep(250);
                msg = msg.substring(1) + msg.charAt(0);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
        }
    }

    public void paint(Graphics g) {
        g.drawString(msg, 50, 30);
    }
}
```

Fig 6.17

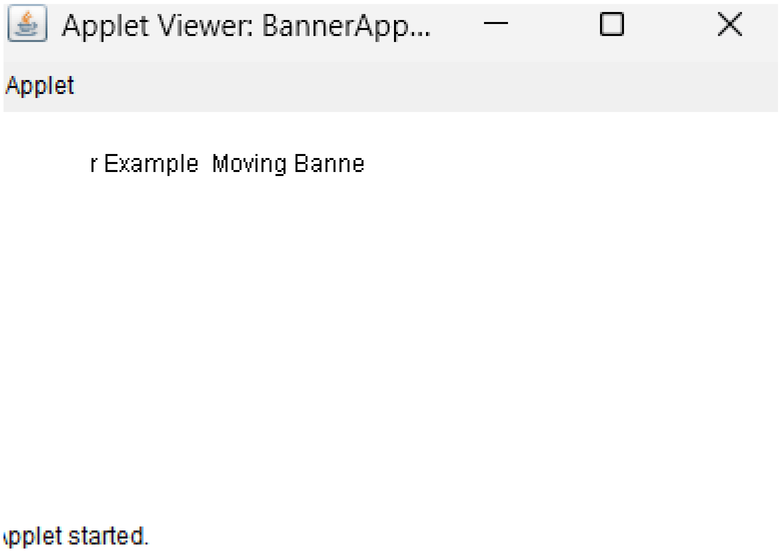


```html
<html>
  <body>
    <applet code="BannerApplet.class" width="400" height="200"></applet>
  </body>
</html>
```

Result: Fig 6.18

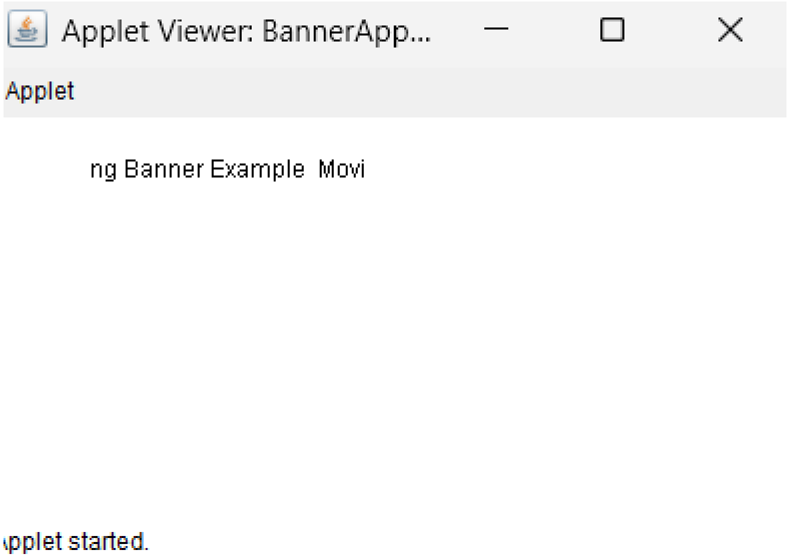


```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac BannerApplet.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer BannerApplet.html
```



```
r Example  Moving Banne
pplet started.
```

Fig 6.19



```
ng Banner Example  Movi
pplet started.
```

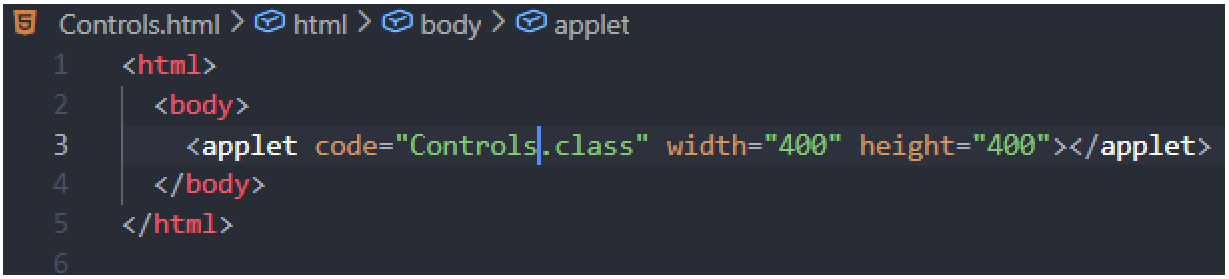
Fig 6.20

6.6

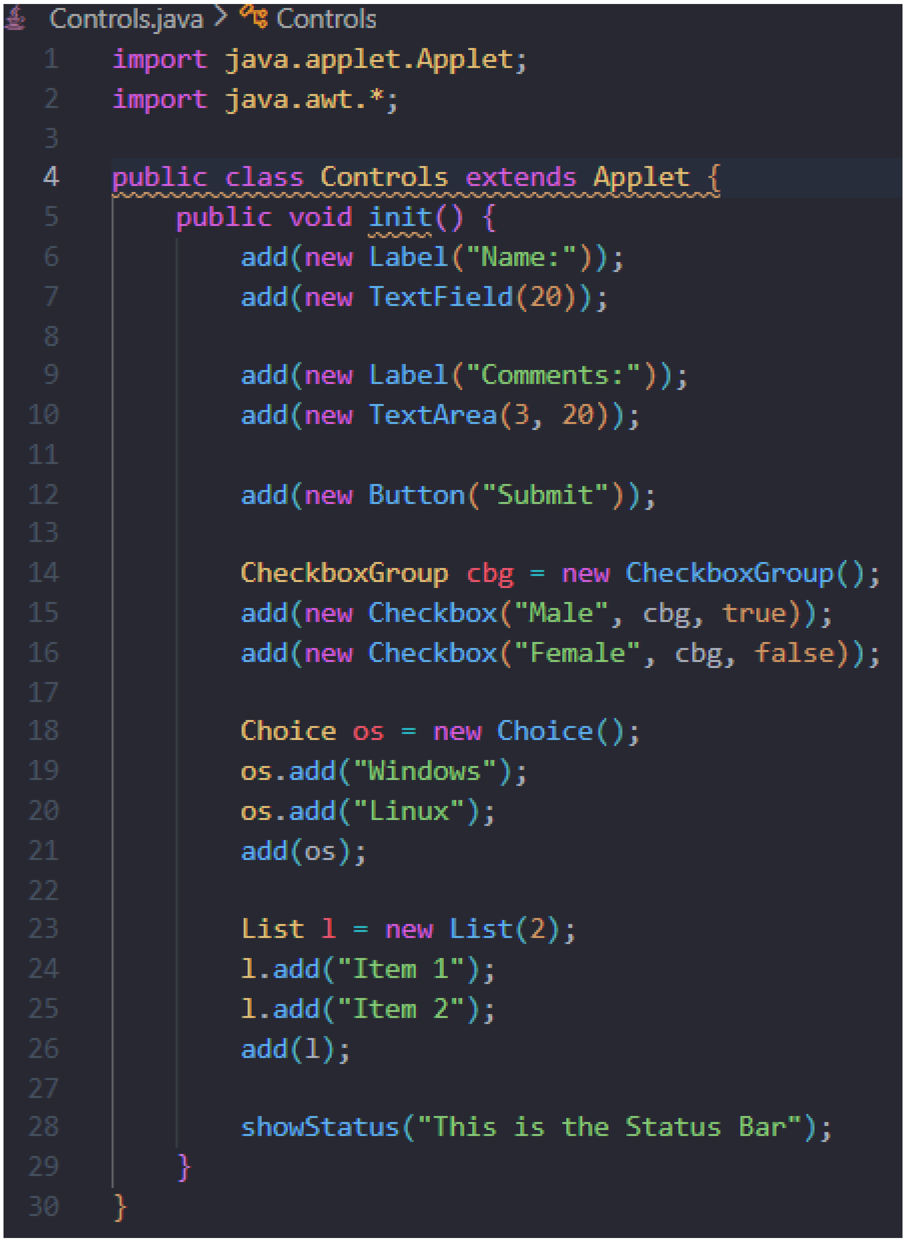
AIM: Write a program to use following control: Checkbox, CheckboxGroup, Choice, Label, Button, List, Status Bar, TextField and TextArea.

Logic: We create an applet and inside the `init()` method, we instantiate various AWT components. We use `Label` for text labels, `Button` for clickable buttons, `Checkbox` for toggle options, and `CheckboxGroup` to group checkboxes into radio buttons. We use `Choice` for a dropdown menu and `List` for a scrollable list of items. `TextField` and `TextArea` are used for single-line and multi-line input respectively. We add all these to the applet using `add()`. Finally, we display a message on the status bar using `showStatus()`.

Code: Fig 6.21



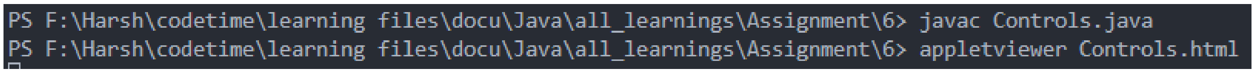
```html
<html>
  <body>
    <applet code="Controls.class" width="400" height="400"></applet>
  </body>
</html>
```



```java
import java.applet.Applet;
import java.awt.*;

public class Controls extends Applet {
    public void init() {
        add(new Label("Name:"));
        add(new TextField(20));

        add(new Label("Comments:"));
        add(new TextArea(3, 20));

        add(new Button("Submit"));

        CheckboxGroup cbg = new CheckboxGroup();
        add(new Checkbox("Male", cbg, true));
        add(new Checkbox("Female", cbg, false));

        Choice os = new Choice();
        os.add("Windows");
        os.add("Linux");
        add(os);

        List l = new List(2);
        l.add("Item 1");
        l.add("Item 2");
        add(l);

        showStatus("This is the Status Bar");
    }
}
```

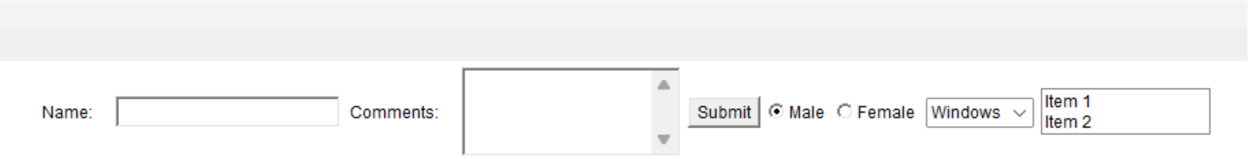
Code: Fig 6.22

Result: Fig 6.23



```
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac Controls.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer Controls.html
```

Fig 2.24

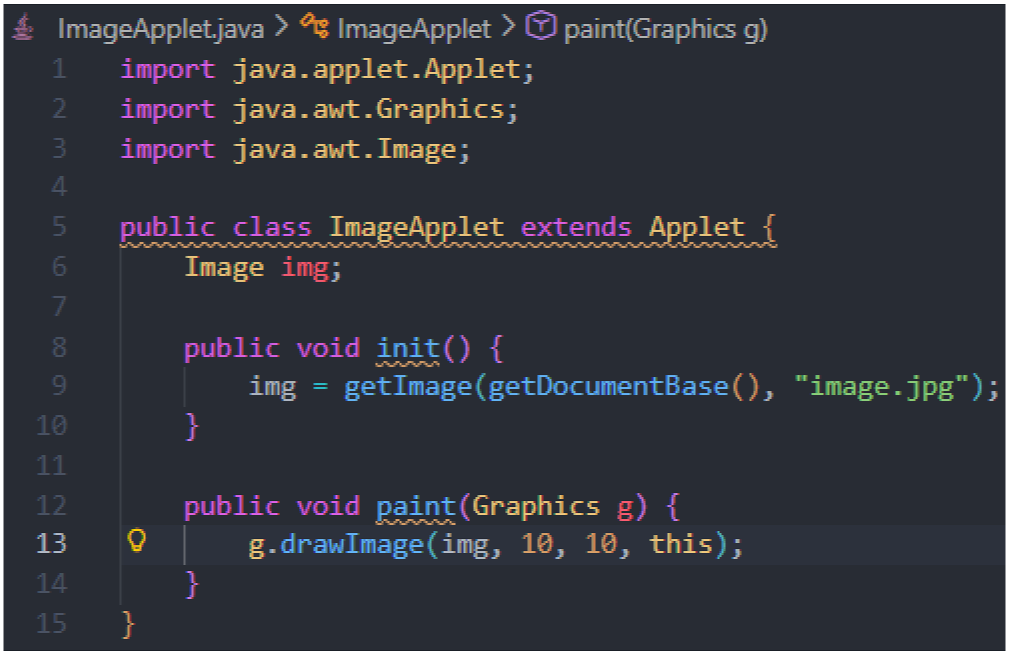


6.7

AIM: Write an applet to which displays image.

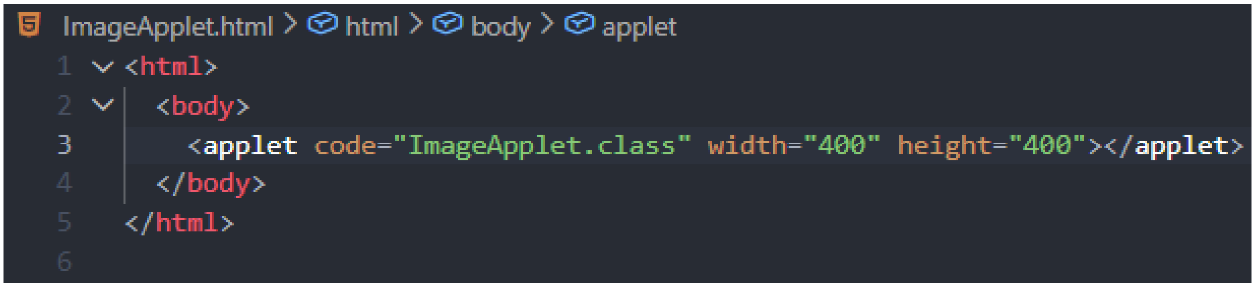
Logic: We create an applet and declare an `Image` object. In the `init()` method, we load the image using `getImage()`, passing the document base URL and the filename (e.g., "picture.jpg"). In the `paint()` method, we use `g.drawImage()` to render the loaded image onto the screen.
Note: You must have an image file named "picture.jpg" in the same folder as your java file for this to work.

Code: Fig 6.25



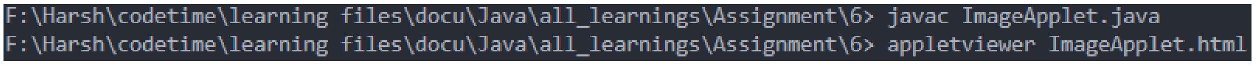
```java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Image;

public class ImageApplet extends Applet {
    Image img;

    public void init() {
        img = getImage(getDocumentBase(), "image.jpg");
    }

    public void paint(Graphics g) {
        g.drawImage(img, 10, 10, this);
    }
}
```

Fig 6.26



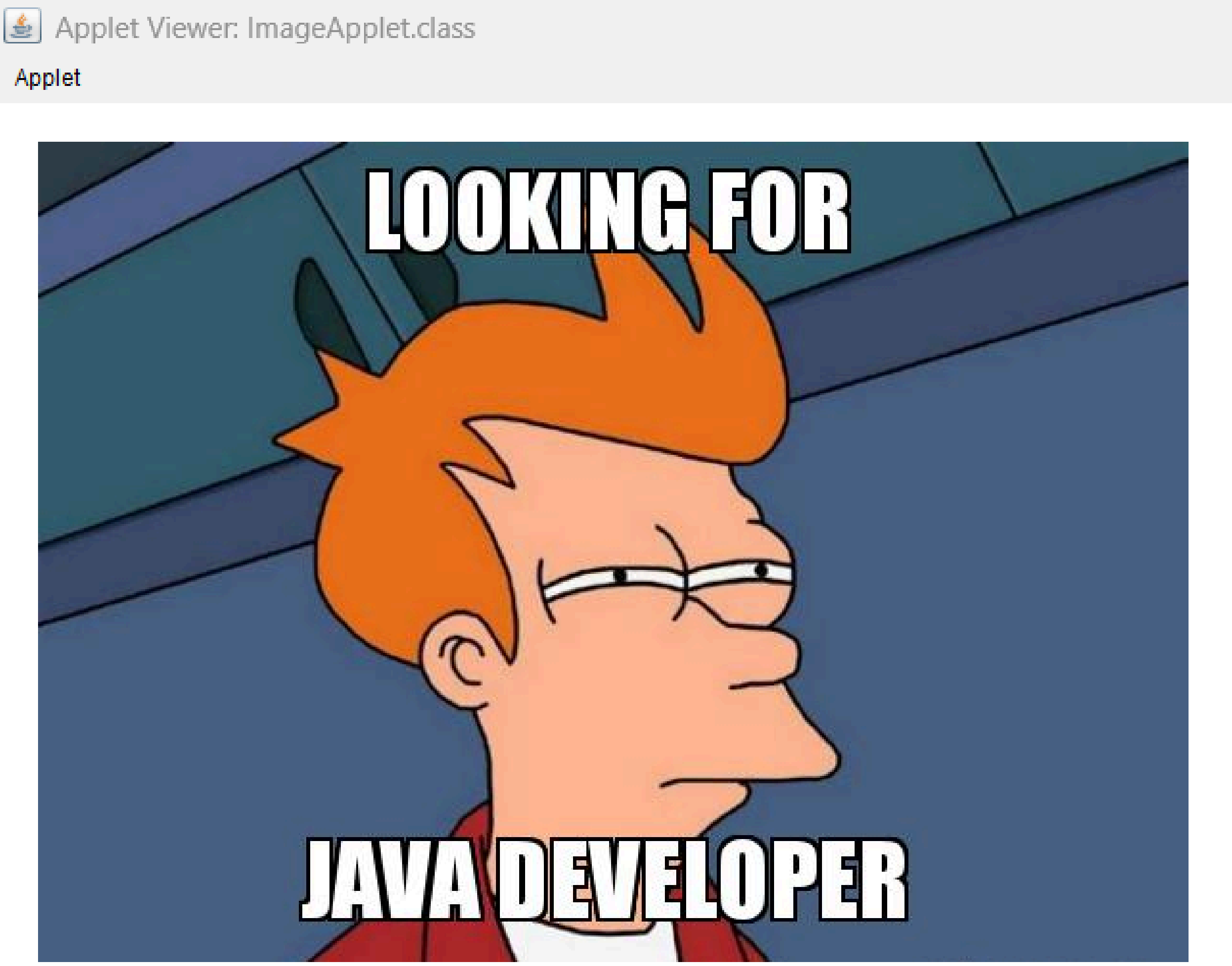
```html
<html>
  <body>
    <applet code="ImageApplet.class" width="400" height="400"></applet>
  </body>
</html>

```

Result: Fig 6.27



```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> javac ImageApplet.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\6> appletviewer ImageApplet.html
```

Fig 6.28



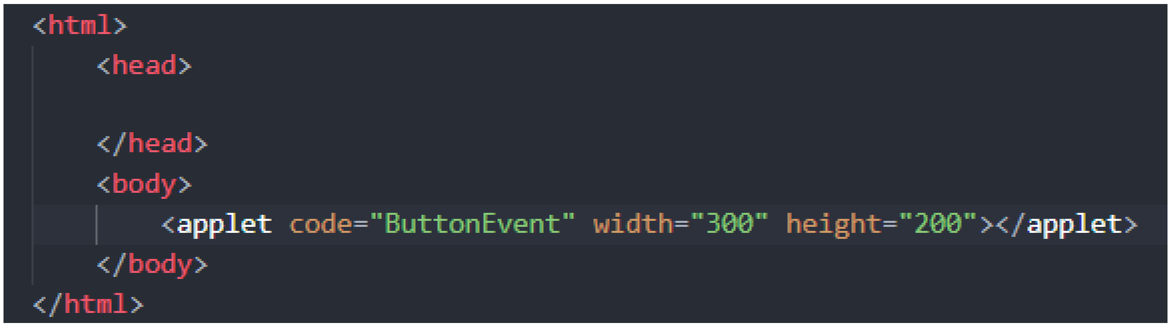
LAB ASSIGNMENT 7

7.1

AIM: Write an applet to demonstrate Button Events using ActionListener interface

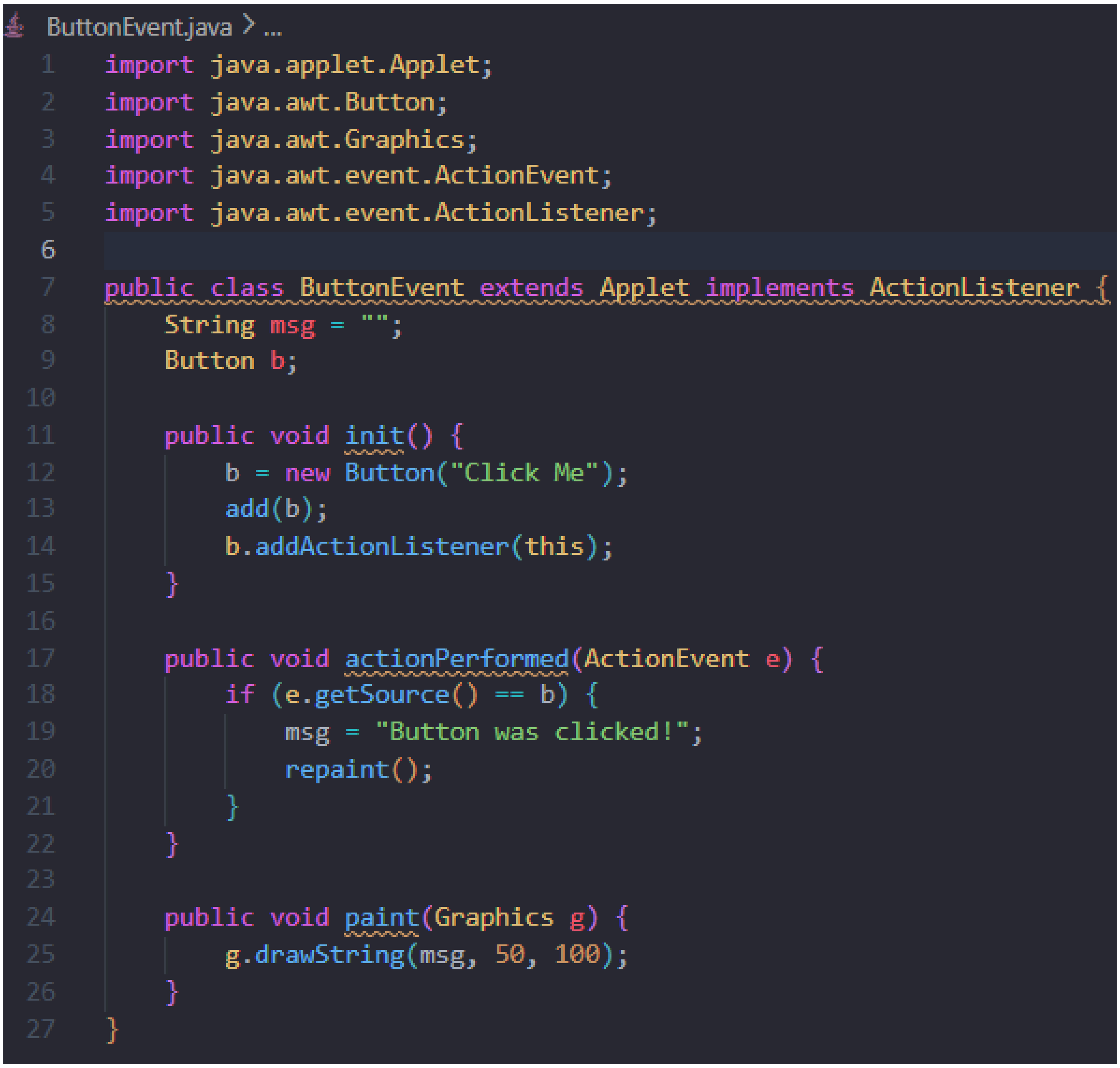
Logic: We create a class that extends `Applet` and implements `ActionListener`. In `init()`, we create a `Button` and add it to the applet. We register the applet as a listener for the button using `addActionListener(this)`. We create a `msg` string. In the `actionPerformed` method, we check if the event source was the button. If so, we change the `msg` string and call `repaint()`. The `paint` method draws the `msg` string.

Code: Fig 7.1



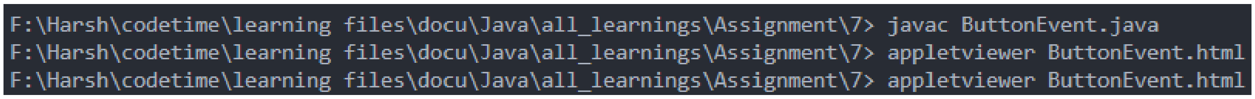
```html
<html>
    <head>

    </head>
    <body>
        <applet code="ButtonEvent" width="300" height="200"></applet>
    </body>
</html>
```

Fig 7.2

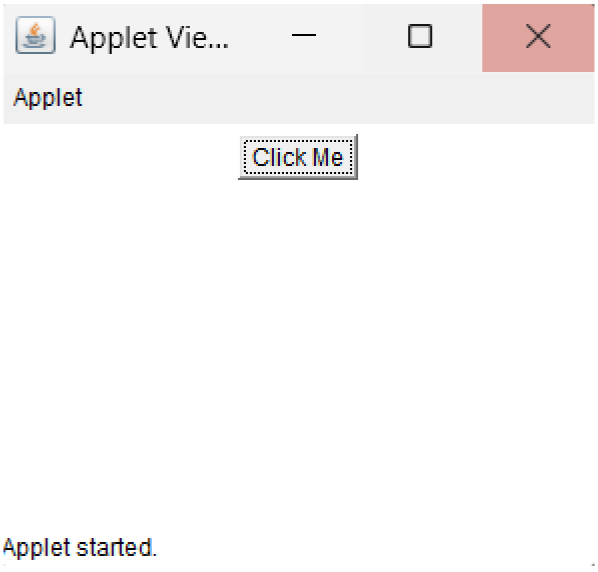


```java
import java.applet.Applet;
import java.awt.Button;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ButtonEvent extends Applet implements ActionListener {
    String msg = "";
    Button b;

    public void init() {
        b = new Button("Click Me");
        add(b);
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == b) {
            msg = "Button was clicked!";
            repaint();
        }
    }

    public void paint(Graphics g) {
        g.drawString(msg, 50, 100);
    }
}
```

Result: Fig 7.3

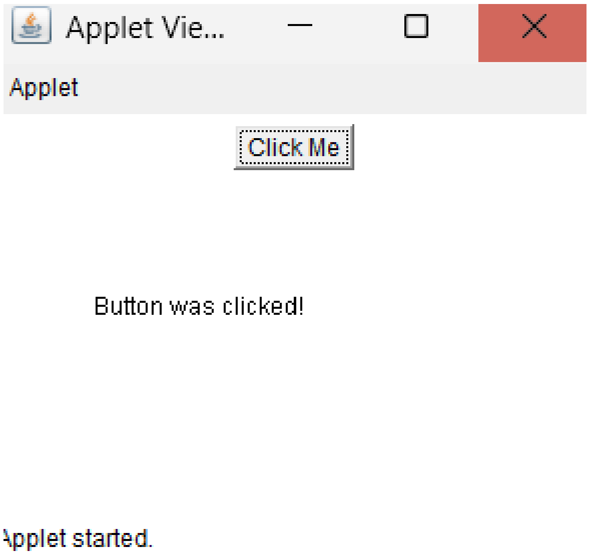


```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac ButtonEvent.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer ButtonEvent.html
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer ButtonEvent.html
```



Applet started.

Fig 7.4



Button was clicked!

Applet started.

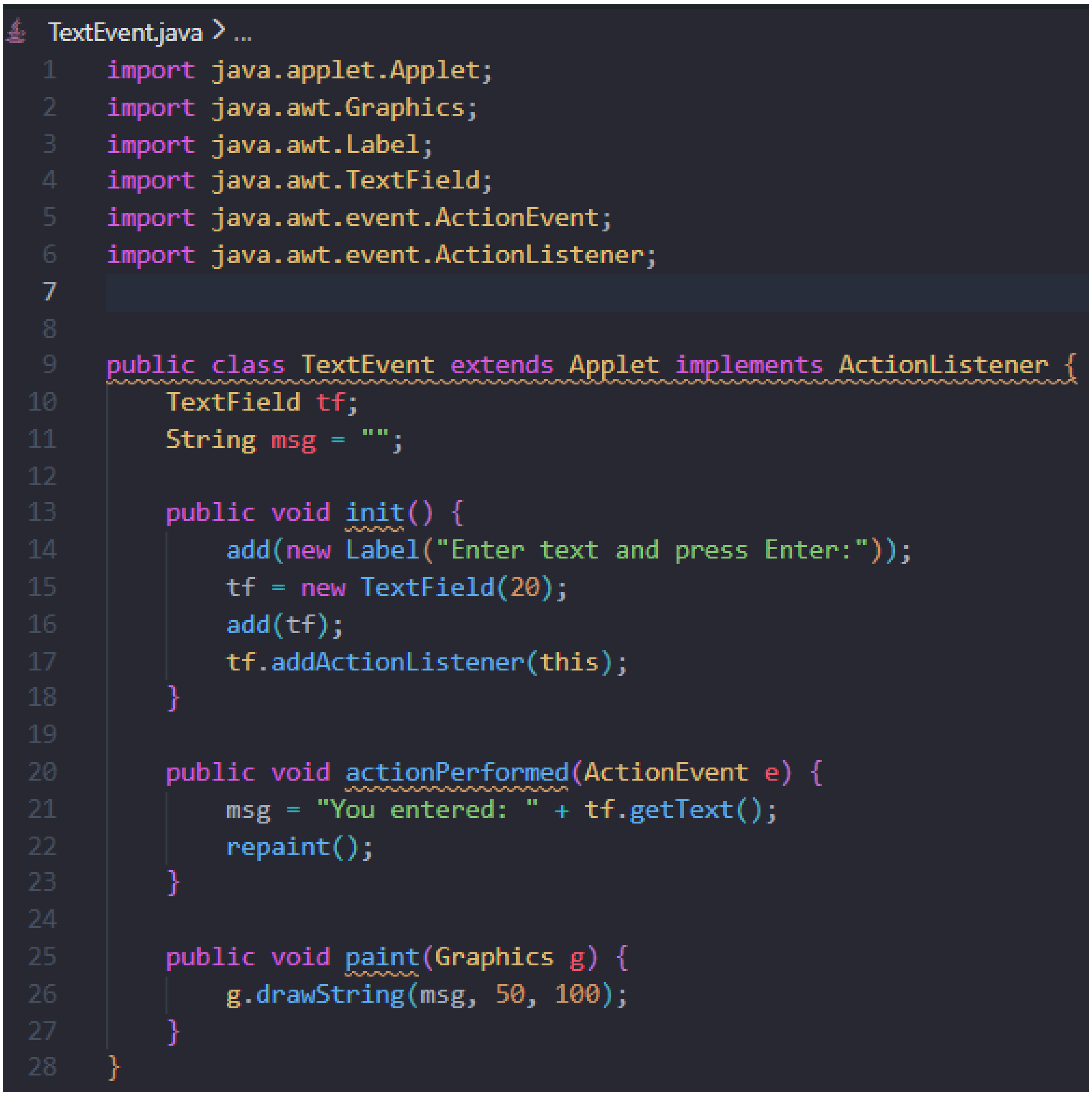
Fig 7.5

7.2

AIM: Write an applet to demonstrate Text Events using ActionListener interface

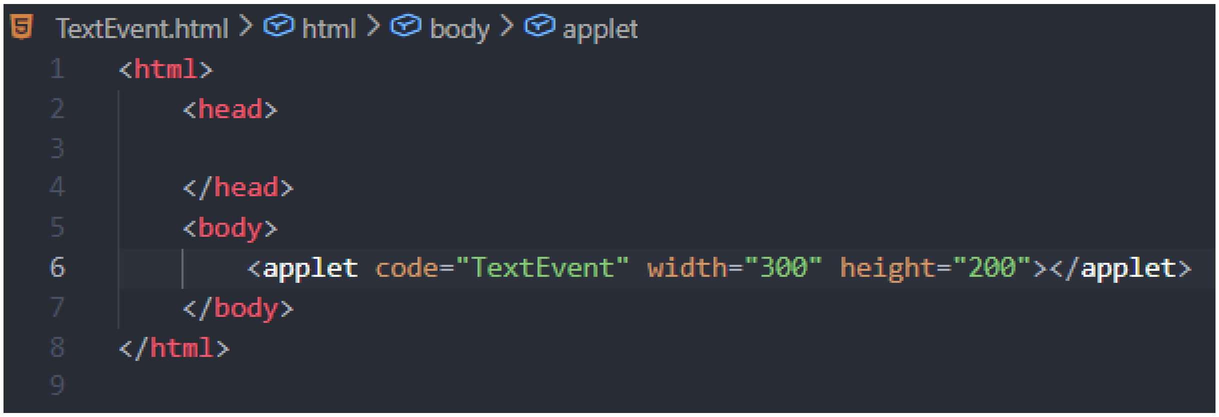
Logic: We create a class extending `Applet` and implementing `ActionListener`. In `init()`, we add a `Label` and a `TextField`. We register an `ActionListener` to the `TextField`. This event fires when the user presses Enter in the text field. In `actionPerformed`, we get the text from the `TextField` using `getText()` and store it. We then call `repaint()` to draw this new text on the applet.

Code: Fig 7.6



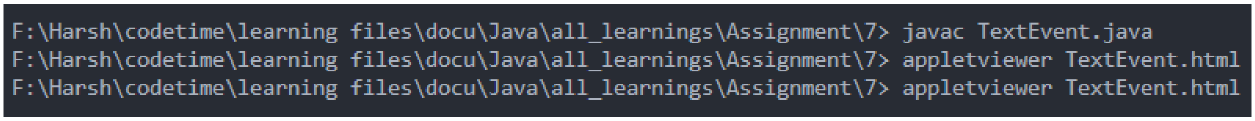
```java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Label;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;


public class TextEvent extends Applet implements ActionListener {
    TextField tf;
    String msg = "";

    public void init() {
        add(new Label("Enter text and press Enter:"));
        tf = new TextField(20);
        add(tf);
        tf.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        msg = "You entered: " + tf.getText();
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 50, 100);
    }
}
```

Fig 7.7

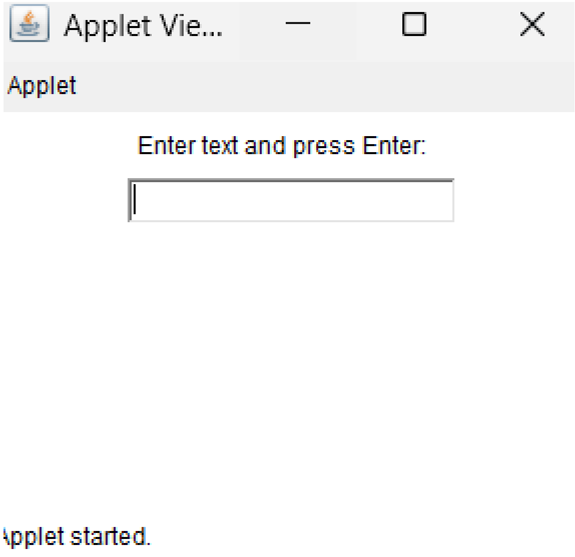


```
<html>
    <head>

    </head>
    <body>
        <applet code="TextEvent" width="300" height="200"></applet>
    </body>
</html>

```

Result: Fig 7.8

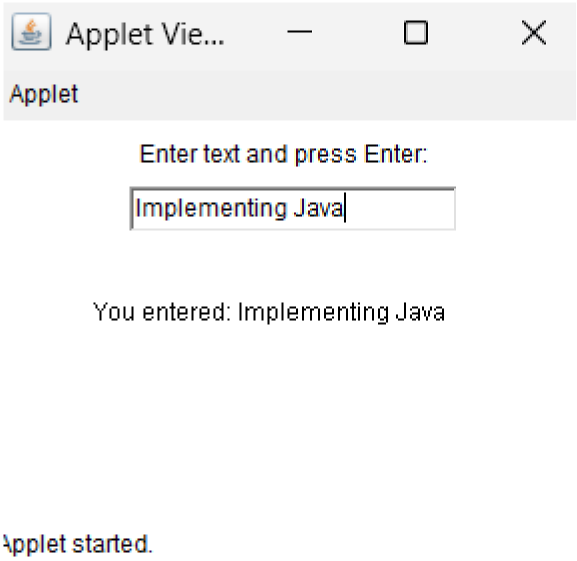


```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac TextEvent.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer TextEvent.html
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer TextEvent.html
```



Enter text and press Enter:

Fig 7.9 Applet started.



Enter text and press Enter:

Implementing Java

You entered: Implementing Java

Applet started.

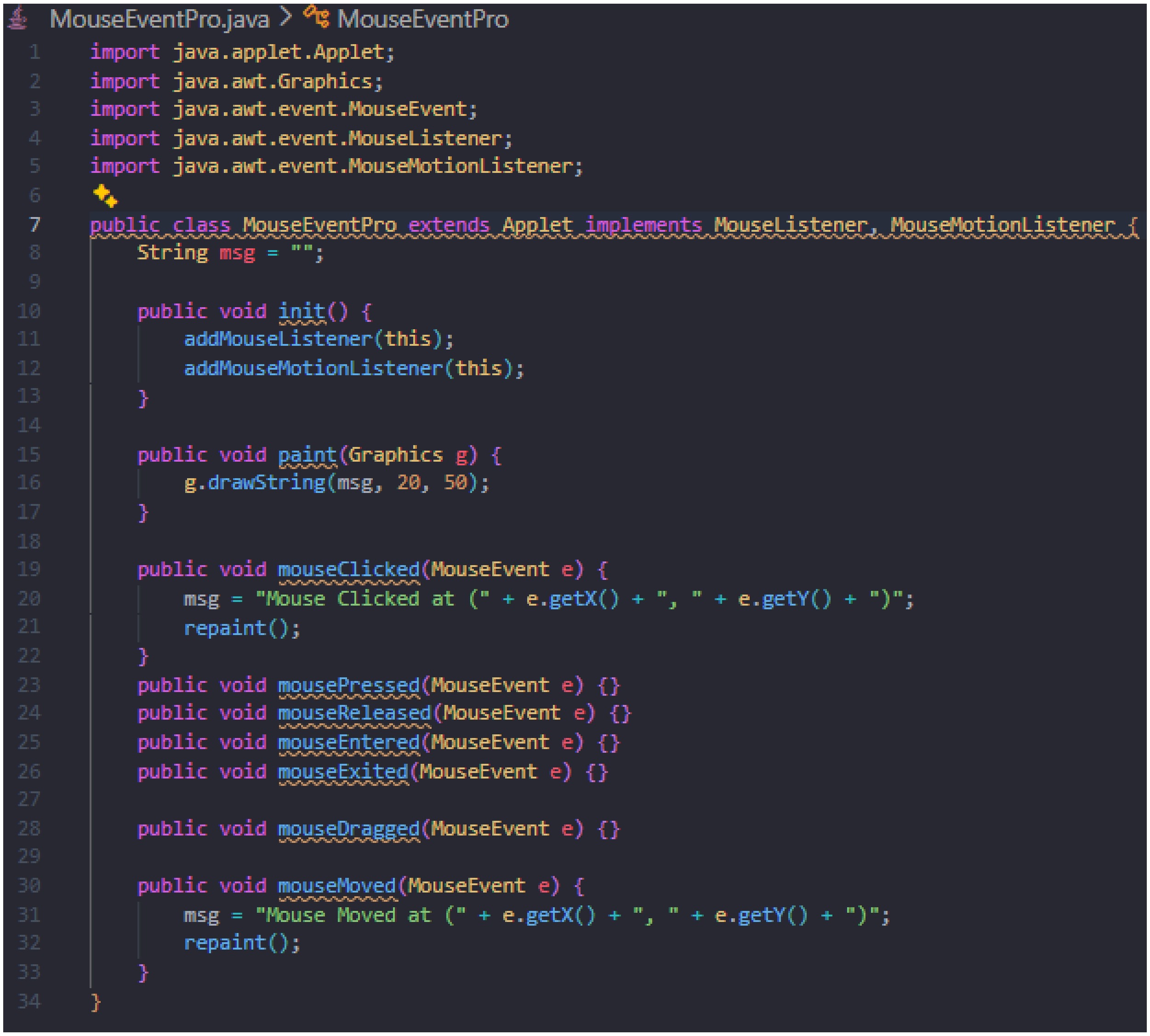
Fig 7.10

7.3

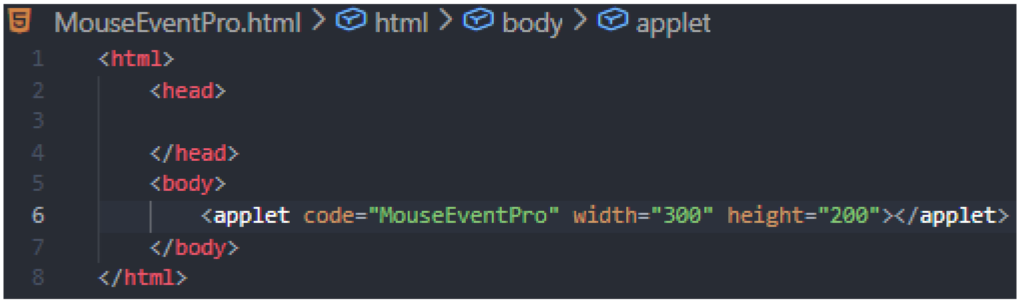
AIM: Write an applet to demonstrate Mouse Events using MouseListener, MouseMotionListener interfaces

Logic: We create an applet that implements both `MouseListener` and `MouseMotionListener`. In `init()`, we register these listeners using `addMouseListener(this)` and `addMouseMotionListener(this)`. We then implement all methods from both interfaces (like `mouseClicked`, `mouseMoved`, etc.). In these methods, we update a string variable `msg` with the event description and the mouse coordinates (from the `MouseEvent` object `e`) and call `repaint()`.

Code: Fig 7.11



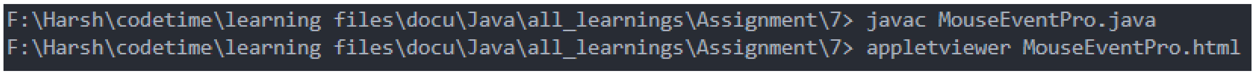
```java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;

public class MouseEventPro extends Applet implements MouseListener, MouseMotionListener {
    String msg = "";

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void paint(Graphics g) {
        g.drawString(msg, 20, 50);
    }

    public void mouseClicked(MouseEvent e) {
        msg = "Mouse Clicked at (" + e.getX() + ", " + e.getY() + ")";
        repaint();
    }
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    public void mouseDragged(MouseEvent e) {}

    public void mouseMoved(MouseEvent e) {
        msg = "Mouse Moved at (" + e.getX() + ", " + e.getY() + ")";
        repaint();
    }
}
```



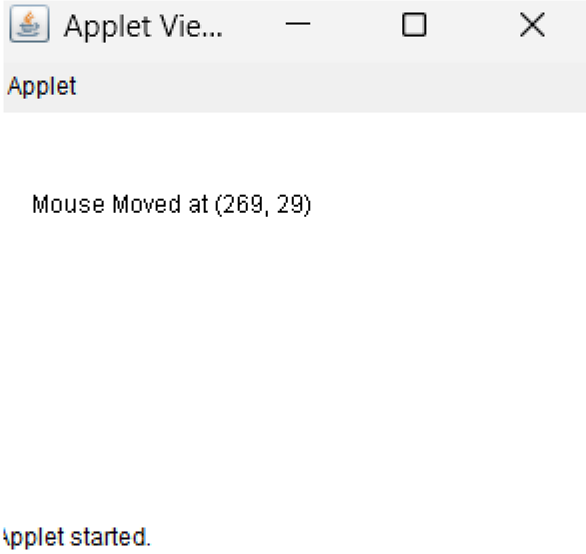
```html
<html>
    <head>

    </head>
    <body>
        <applet code="MouseEventPro" width="300" height="200"></applet>
    </body>
</html>
```

Fig 7.12

Result: Fig 7.13



```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac MouseEventPro.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer MouseEventPro.html
```



```
Mouse Moved at (269, 29)
```

Applet started.

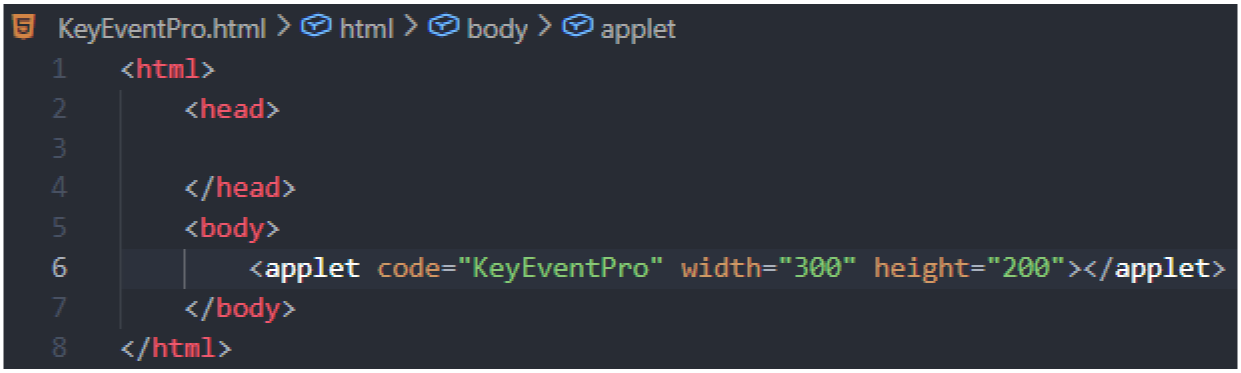
Fig 7.14

7.4

AIM: Write an applet to demonstrate Keyboard Events using KeyListener interface

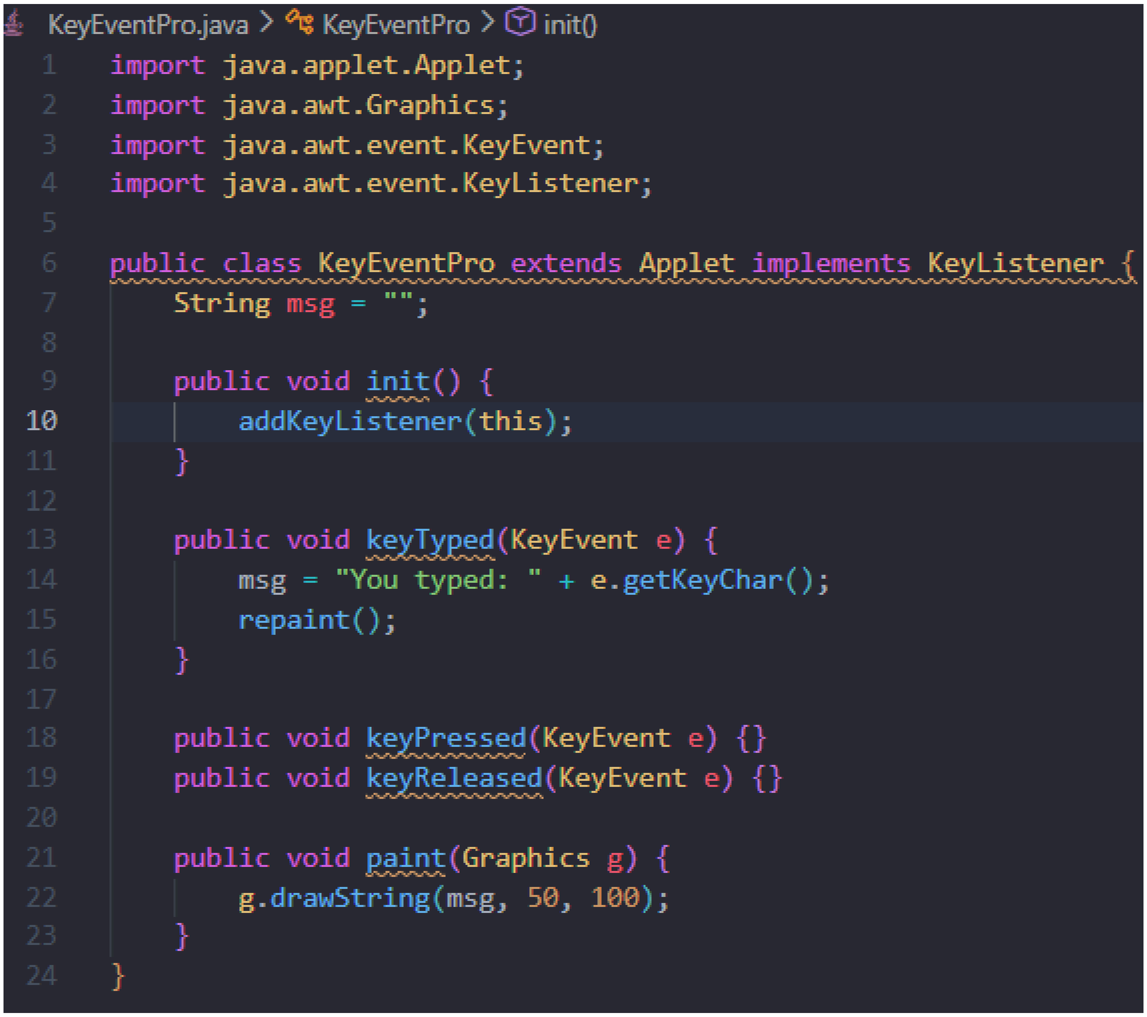
Logic: We create an applet that implements `KeyListener`. In `init()`, we register the listener using `addKeyListener(this)`. We must implement three methods: `keyPressed`, `keyReleased`, and `keyTyped`. In the `keyTyped` method, we get the character that was typed using `e.getKeyChar()` and store it in a string. We call `repaint()` to update the display. The `paint` method draws this string.

Code: Fig 7.15



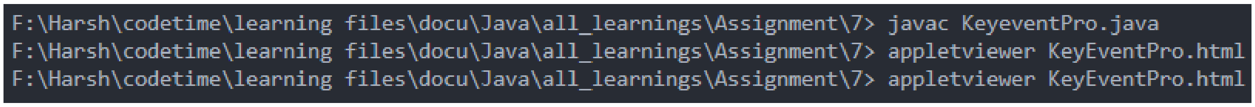
```html
<html>
    <head>

    </head>
    <body>
        <applet code="KeyEventPro" width="300" height="200"></applet>
    </body>
</html>
```

Fig 7.16

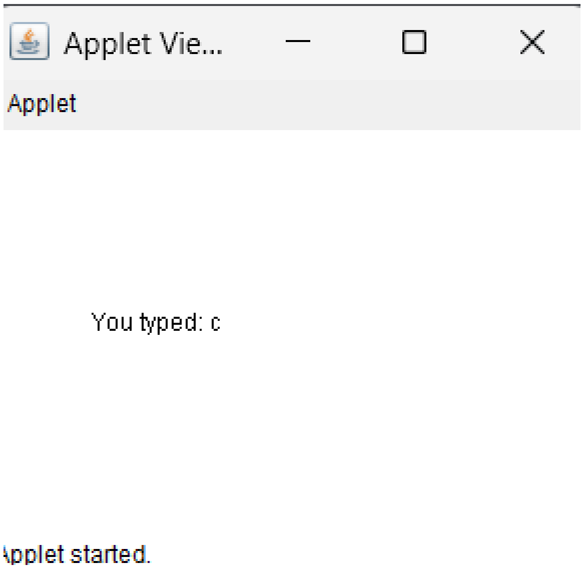


```java
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class KeyEventPro extends Applet implements KeyListener {
    String msg = "";

    public void init() {
        addKeyListener(this);
    }

    public void keyTyped(KeyEvent e) {
        msg = "You typed: " + e.getKeyChar();
        repaint();
    }

    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}

    public void paint(Graphics g) {
        g.drawString(msg, 50, 100);
    }
}
```

Result: Fig 7.17



```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac KeyeventPro.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer KeyEventPro.html
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer KeyEventPro.html
```



```
You typed: c
```

Fig 7.18

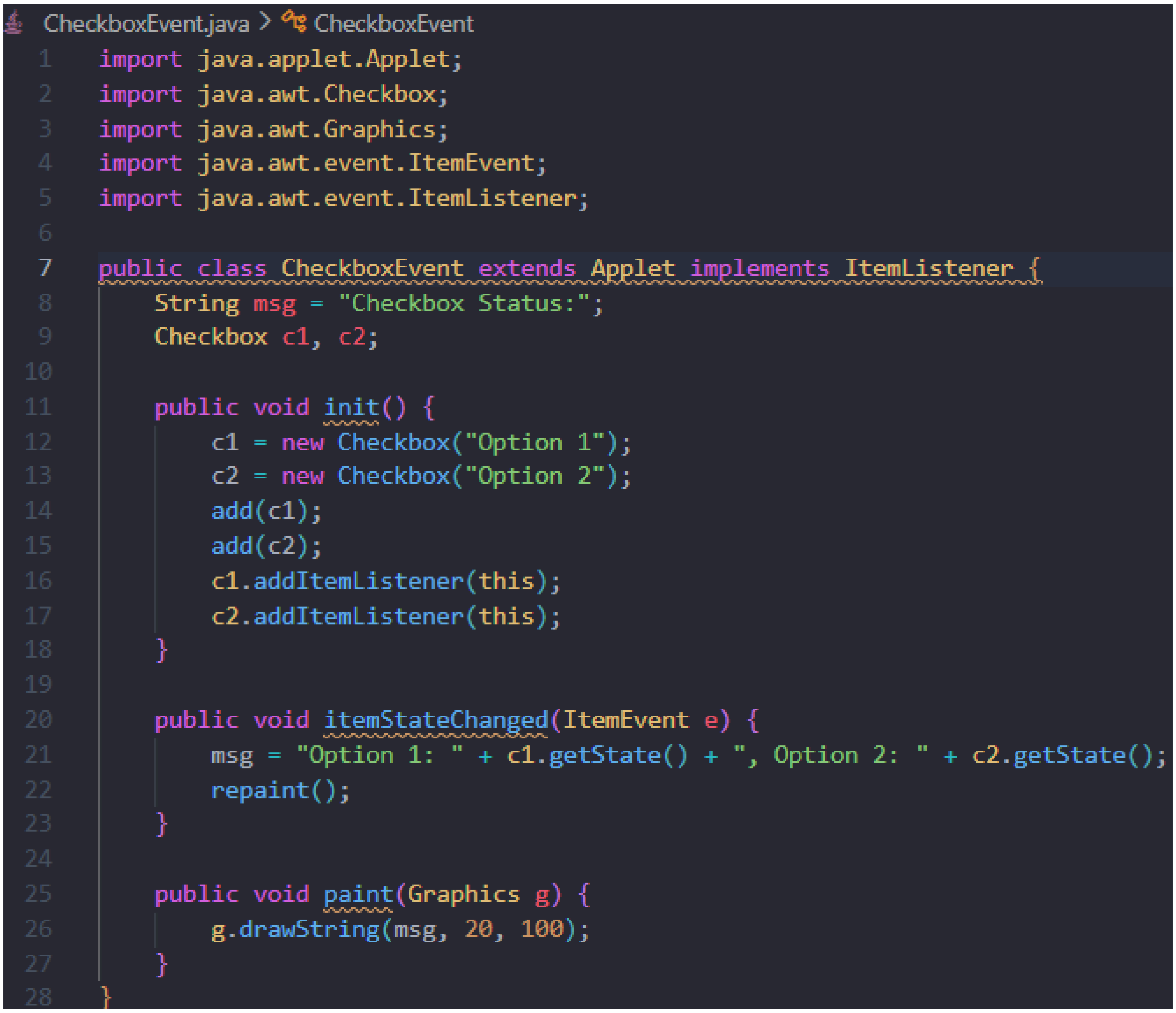
```
\pplet started.
```

7.5

AIM: Write an applet to demonstrate checkbox Events using ItemListener

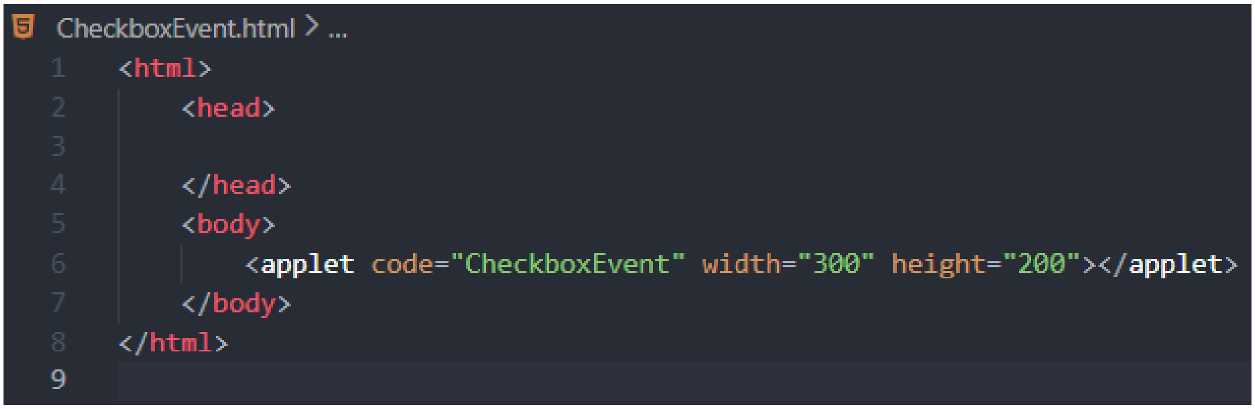
Logic: We create an applet implementing `ItemListener`. In `init()`, we create two `Checkbox` objects and add them. We register the listener for both checkboxes using `addItemListener(this)`. We override the `itemStateChanged` method. Inside this method, we can check which checkbox triggered the event and whether it was `ItemEvent.SELECTED` or `ItemEvent.DESELECTED`, updating a status message accordingly.

Code: Fig 7.19



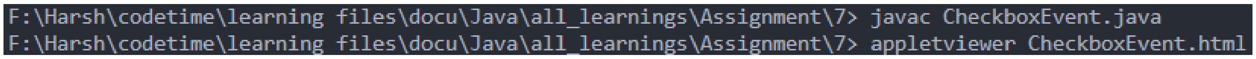
```java
import java.applet.Applet;
import java.awt.Checkbox;
import java.awt.Graphics;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;

public class CheckboxEvent extends Applet implements ItemListener {
    String msg = "Checkbox Status:";
    Checkbox c1, c2;

    public void init() {
        c1 = new Checkbox("Option 1");
        c2 = new Checkbox("Option 2");
        add(c1);
        add(c2);
        c1.addItemListener(this);
        c2.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent e) {
        msg = "Option 1: " + c1.getState() + ", Option 2: " + c2.getState();
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 20, 100);
    }
}
```

Fig 7.20

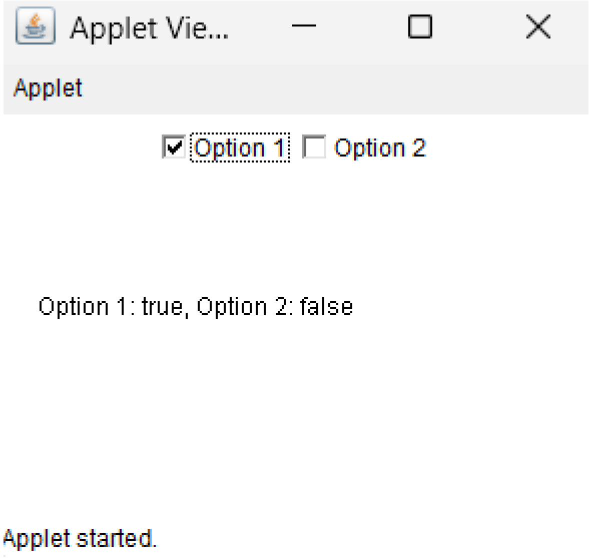


```html
<html>
    <head>

    </head>
    <body>
        <applet code="CheckboxEvent" width="300" height="200"></applet>
    </body>
</html>

```

Result: Fig 7.21



```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac CheckboxEvent.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer CheckboxEvent.html
```



☑ Option 1 ☐ Option 2

```
Option 1: true, Option 2: false
```

```
Applet started.
```

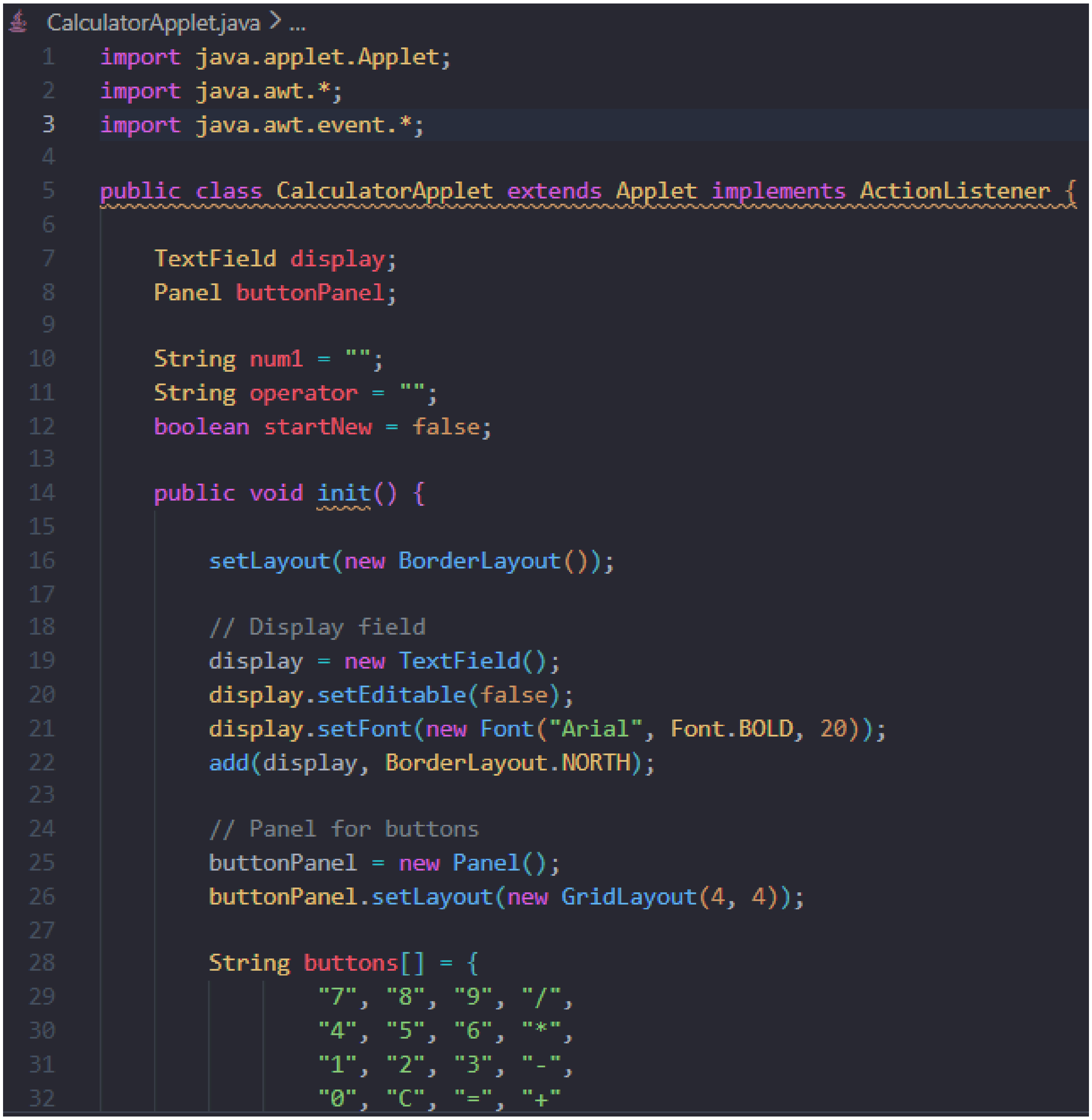
Fig 7.22

7.6

AIM: Write a calculator in java using applets (use Layout Manager)

Logic: We use a `BorderLayout` for the applet. We place a `TextField` in the `NORTH` region for display. We create a `Panel` and set its layout to `GridLayout(4, 4)` for the buttons. We create `Button` objects for numbers (0-9) and operators (+, -, *, /) and add them to the panel. We add the panel to the `CENTER` region of the applet. This code focuses on the layout as requested. Implementing `ActionListener` to handle button clicks.

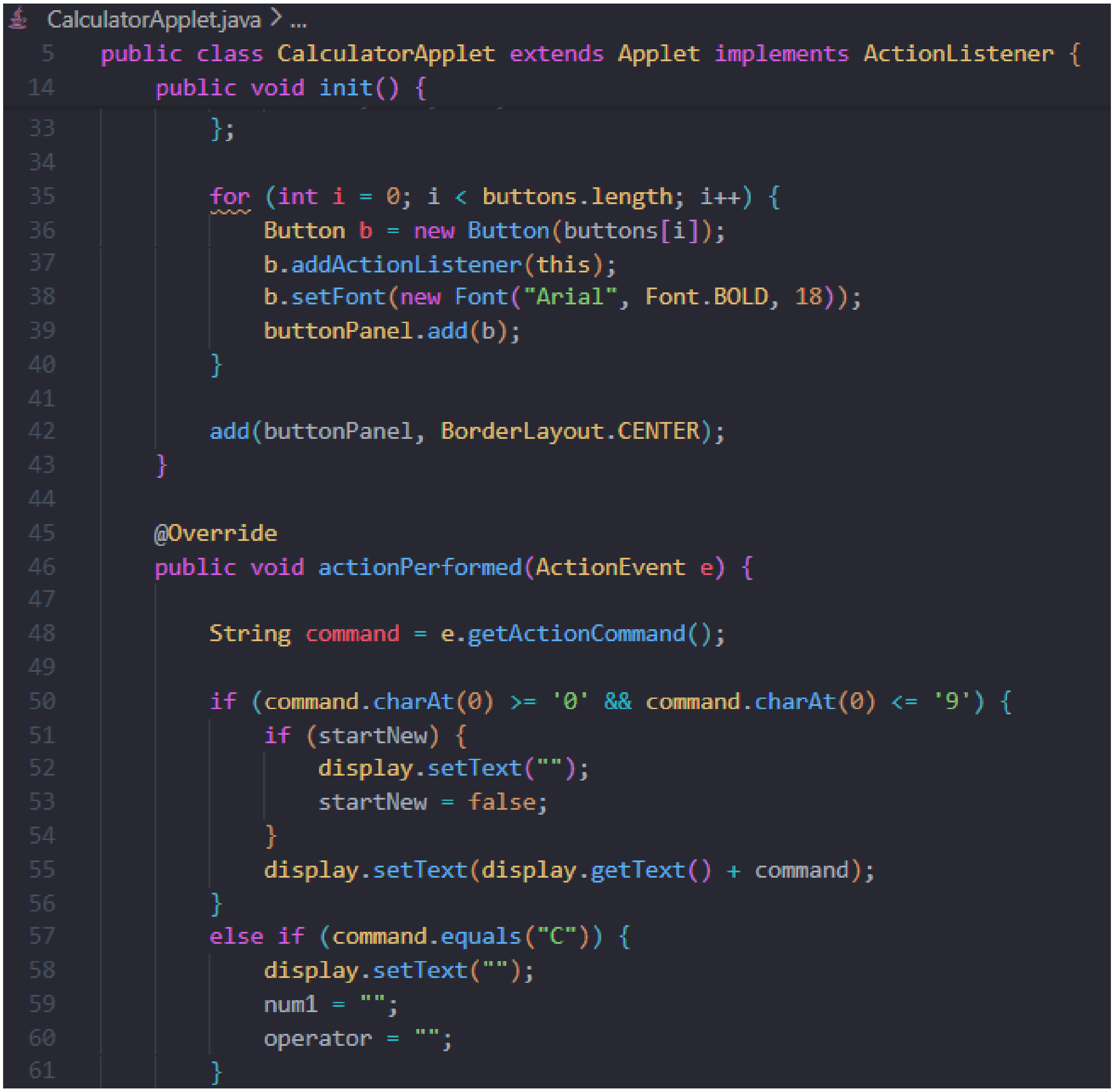
Code: Fig 7.23



CalculatorApplet.java > ...

```java
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class CalculatorApplet extends Applet implements ActionListener {

    TextField display;
    Panel buttonPanel;

    String num1 = "";
    String operator = "";
    boolean startNew = false;

    public void init() {

        setLayout(new BorderLayout());

        // Display field
        display = new TextField();
        display.setEditable(false);
        display.setFont(new Font("Arial", Font.BOLD, 20));
        add(display, BorderLayout.NORTH);

        // Panel for buttons
        buttonPanel = new Panel();
        buttonPanel.setLayout(new GridLayout(4, 4));

        String buttons[] = {
                "7", "8", "9", "/",
                "4", "5", "6", "*",
                "1", "2", "3", "-",
                "0", "C", "=", "+"
```

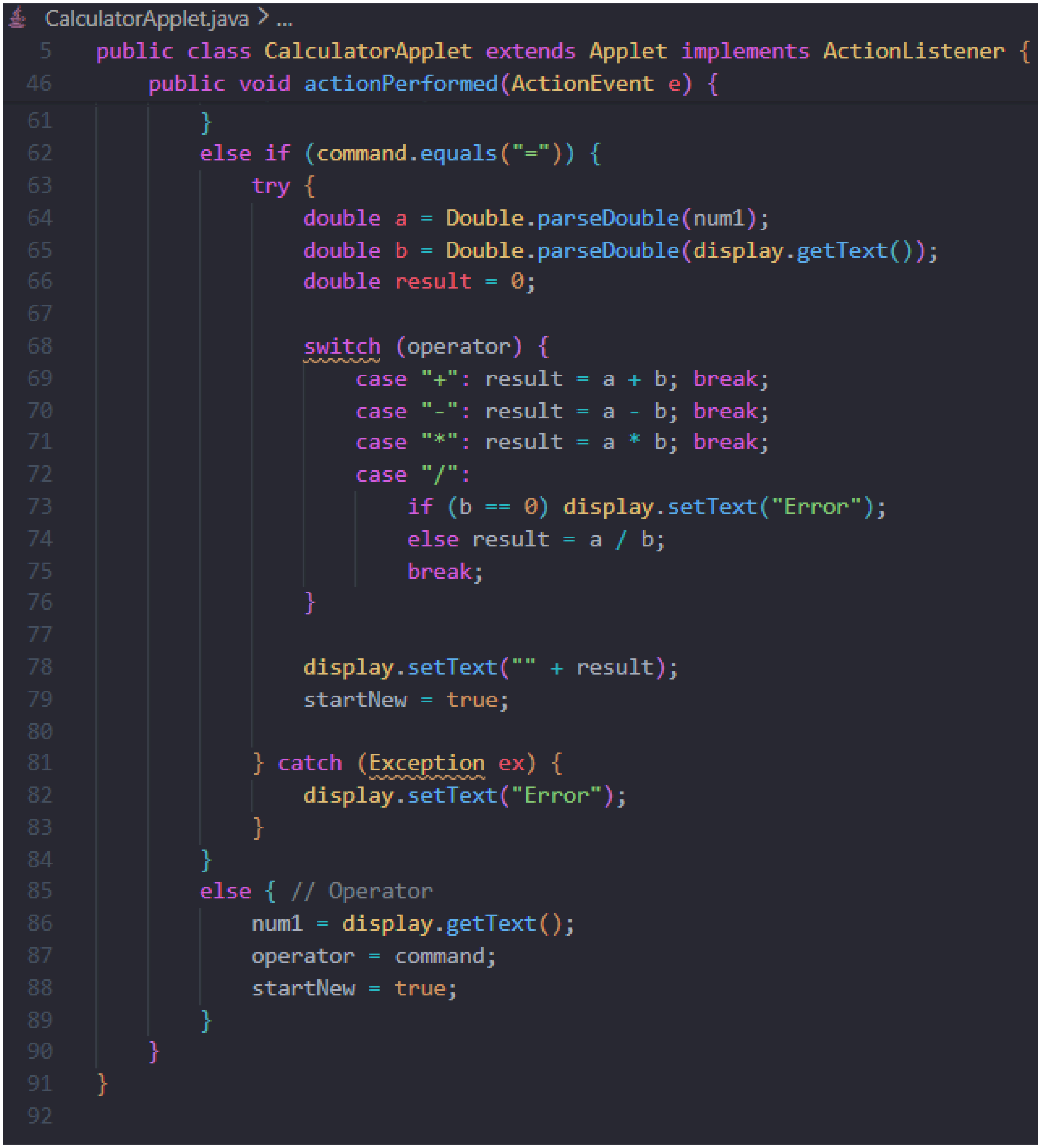
Fig 7.24



CalculatorApplet.java > ...

```java
public class CalculatorApplet extends Applet implements ActionListener {
    public void init() {
        };

        for (int i = 0; i < buttons.length; i++) {
            Button b = new Button(buttons[i]);
            b.addActionListener(this);
            b.setFont(new Font("Arial", Font.BOLD, 18));
            buttonPanel.add(b);
        }

        add(buttonPanel, BorderLayout.CENTER);
    }

    @Override
    public void actionPerformed(ActionEvent e) {

        String command = e.getActionCommand();

        if (command.charAt(0) >= '0' && command.charAt(0) <= '9') {
            if (startNew) {
                display.setText("");
                startNew = false;
            }
            display.setText(display.getText() + command);
        }
        else if (command.equals("C")) {
            display.setText("");
            num1 = "";
            operator = "";
        }
```

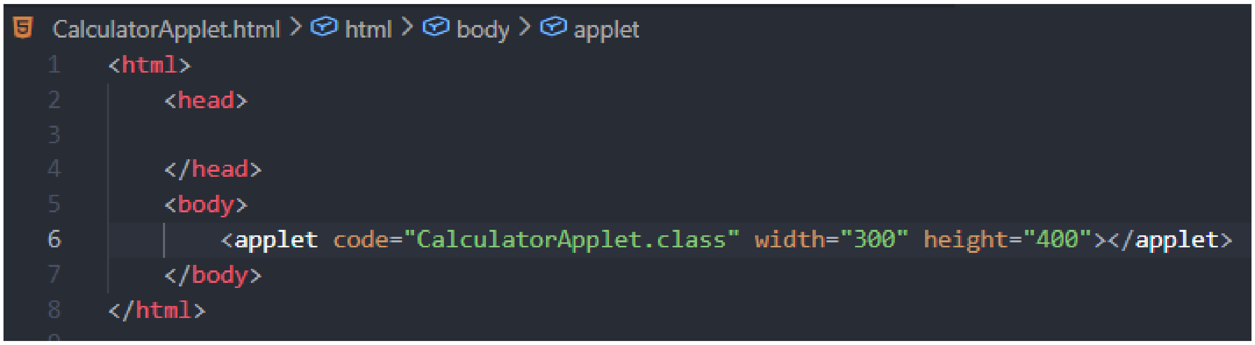
Fig 7.25



CalculatorApplet.java > ...

```
public class CalculatorApplet extends Applet implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        }
        else if (command.equals("=")) {
            try {
                double a = Double.parseDouble(num1);
                double b = Double.parseDouble(display.getText());
                double result = 0;

                switch (operator) {
                    case "+": result = a + b; break;
                    case "-": result = a - b; break;
                    case "*": result = a * b; break;
                    case "/":
                        if (b == 0) display.setText("Error");
                        else result = a / b;
                        break;
                }

                display.setText("" + result);
                startNew = true;

            } catch (Exception ex) {
                display.setText("Error");
            }
        }
        else { // Operator
            num1 = display.getText();
            operator = command;
            startNew = true;
        }
    }
}
```

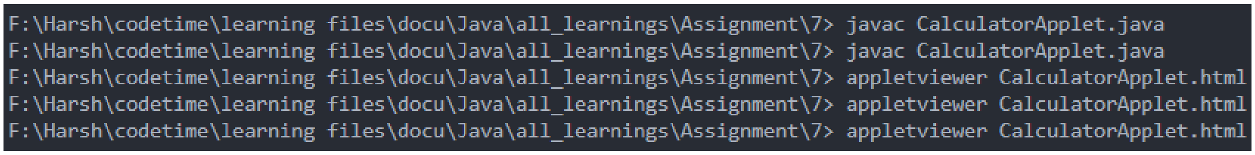
Fig 7.26



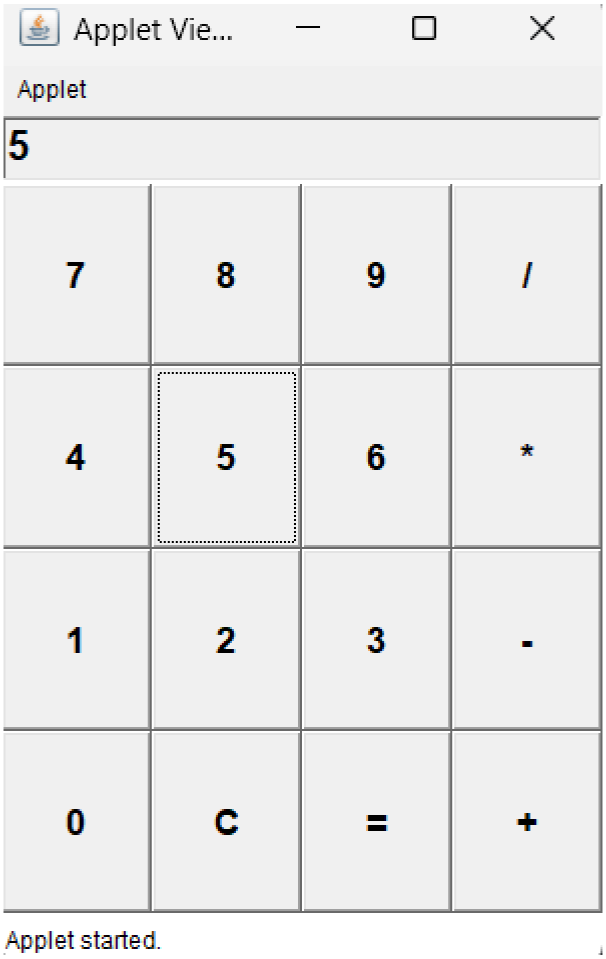
CalculatorApplet.html > html > body > applet

```html
<html>
    <head>

    </head>
    <body>
        <applet code="CalculatorApplet.class" width="300" height="400"></applet>
    </body>
</html>
```

Result: Fig 7.27

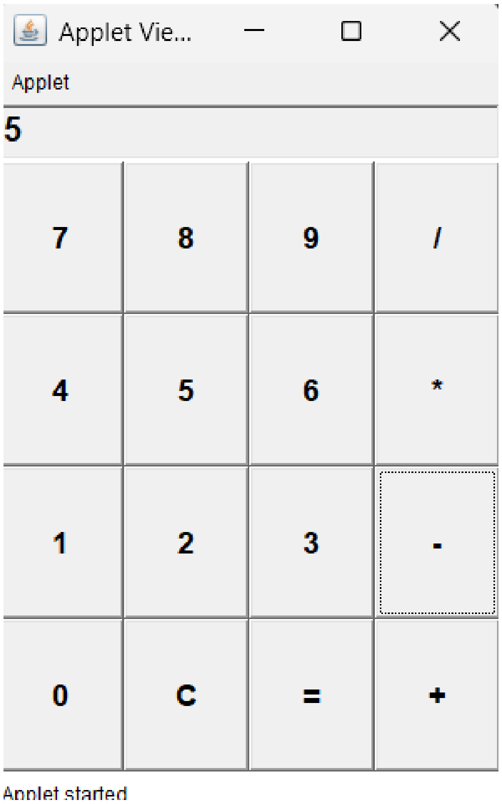


```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac CalculatorApplet.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac CalculatorApplet.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer CalculatorApplet.html
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer CalculatorApplet.html
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> appletviewer CalculatorApplet.html
```



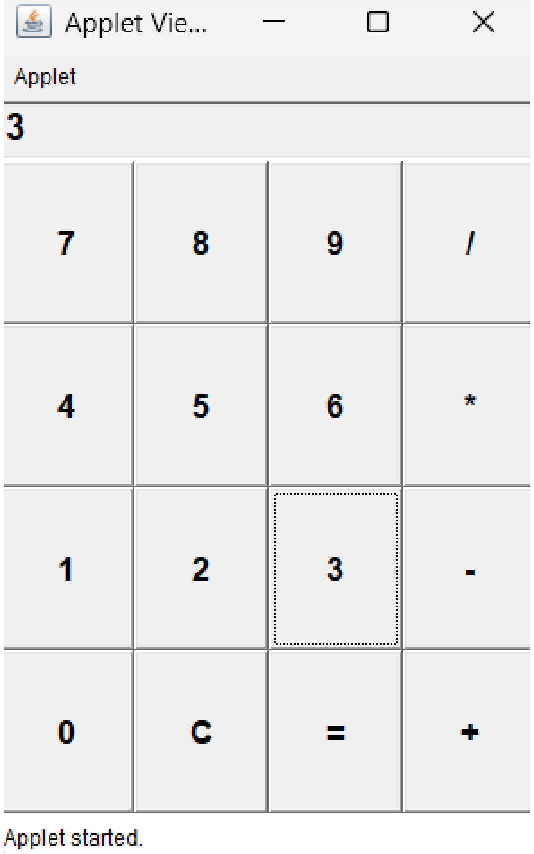
Applet started.

Fig 7.28



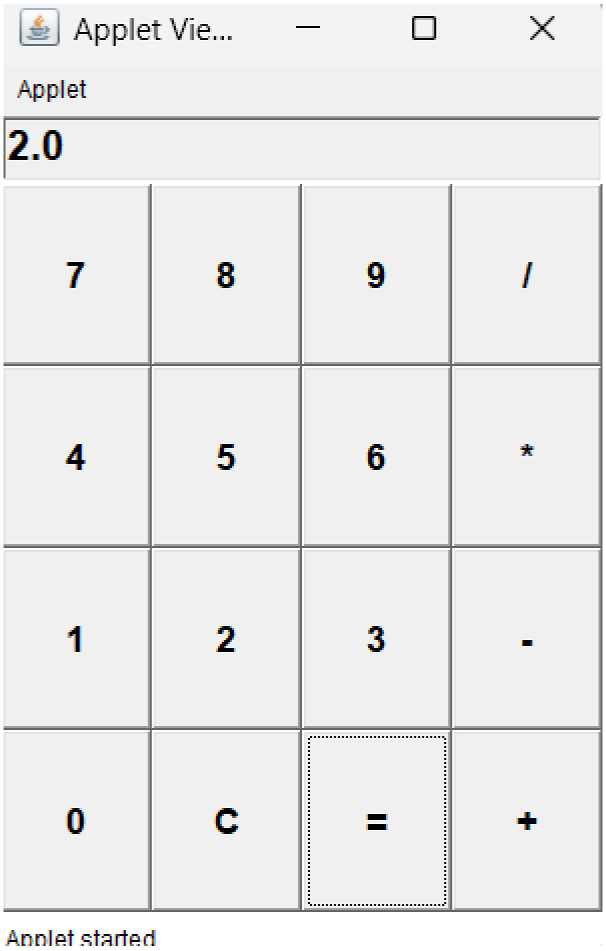
Applet started

Fig 7.29



Applet started.

Fig 7.30



```
2.0
```

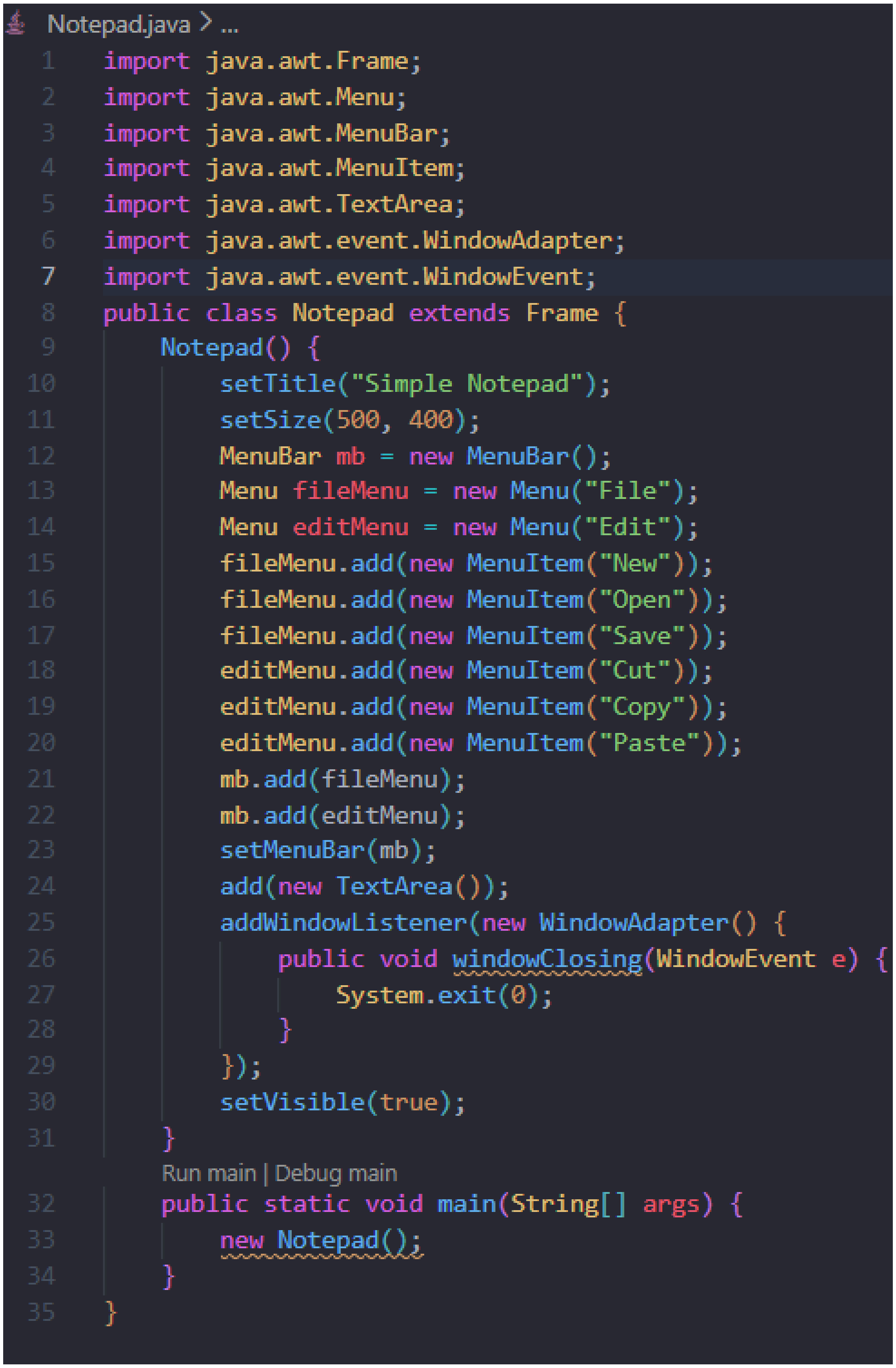
Applet started

Fig 7.31

7.7

AIM: Write a menu driven notepad

Logic: This program uses `Frame` (not `Applet`) to create a window. We create a `MenuBar`, two `Menu` objects ("File", "Edit"), and several `MenuItem` objects ("New", "Open", "Cut", "Copy", "Paste"). We add the `MenuItem`s to their respective `Menu`s, add the `Menu`s to the `MenuBar`, and set the `MenuBar` for the `Frame`. We add a `TextArea` to the `CENTER` of the `Frame`. We also add a `WindowAdapter` to handle the window-closing event.

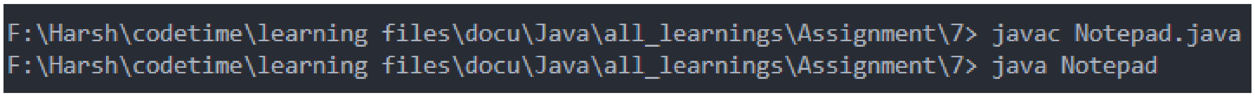


Notepad.java > ...

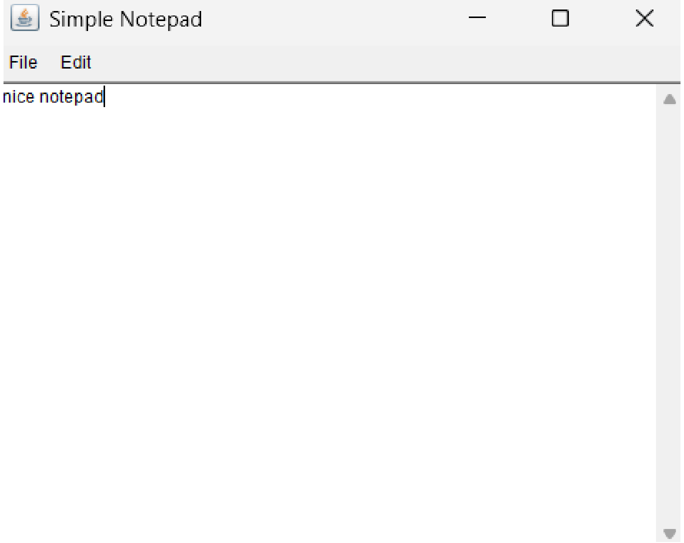
```java
import java.awt.Frame;
import java.awt.Menu;
import java.awt.MenuBar;
import java.awt.MenuItem;
import java.awt.TextArea;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
public class Notepad extends Frame {
    Notepad() {
        setTitle("Simple Notepad");
        setSize(500, 400);
        MenuBar mb = new MenuBar();
        Menu fileMenu = new Menu("File");
        Menu editMenu = new Menu("Edit");
        fileMenu.add(new MenuItem("New"));
        fileMenu.add(new MenuItem("Open"));
        fileMenu.add(new MenuItem("Save"));
        editMenu.add(new MenuItem("Cut"));
        editMenu.add(new MenuItem("Copy"));
        editMenu.add(new MenuItem("Paste"));
        mb.add(fileMenu);
        mb.add(editMenu);
        setMenuBar(mb);
        add(new TextArea());
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setVisible(true);
    }
    Run main | Debug main
    public static void main(String[] args) {
        new Notepad();
    }
}
```

Code: FIg 7.32

Result: Fig 7.33



```
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> javac Notepad.java
F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\7> java Notepad
```



nice notepad

Fig 7.34

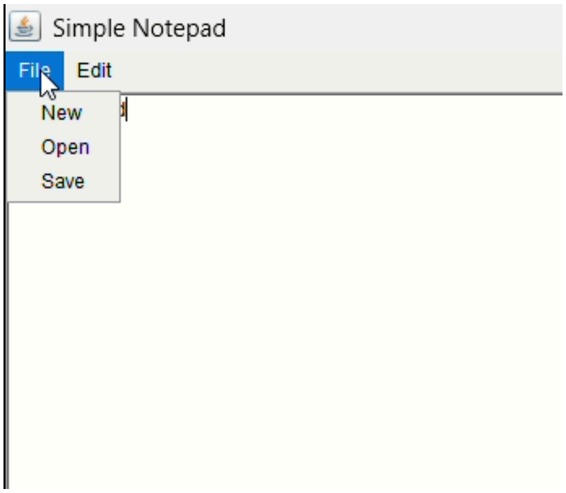
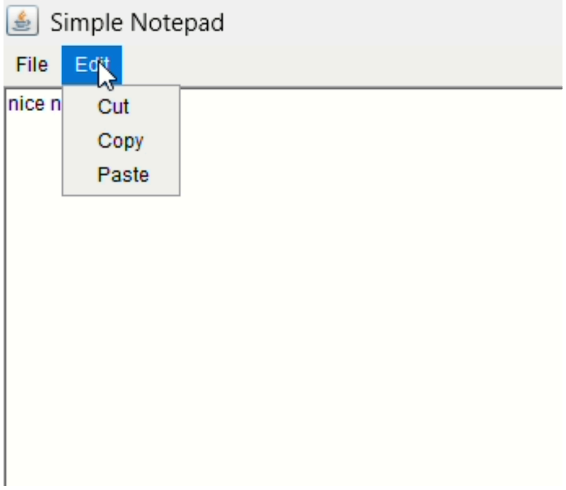


Fig 7.35



nice n

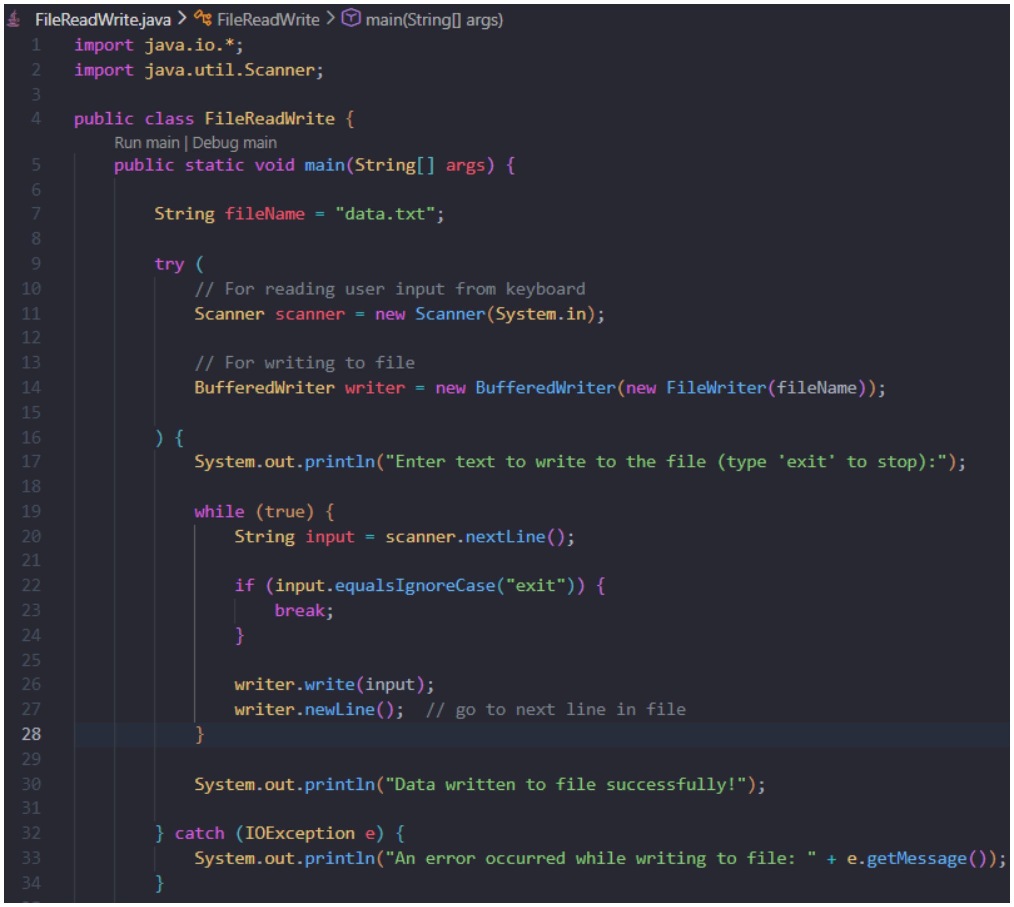
Fig 7.36

LAB ASSIGNMENT 8

8.1

AIM: Write a program to read from a file and write into file using java.io

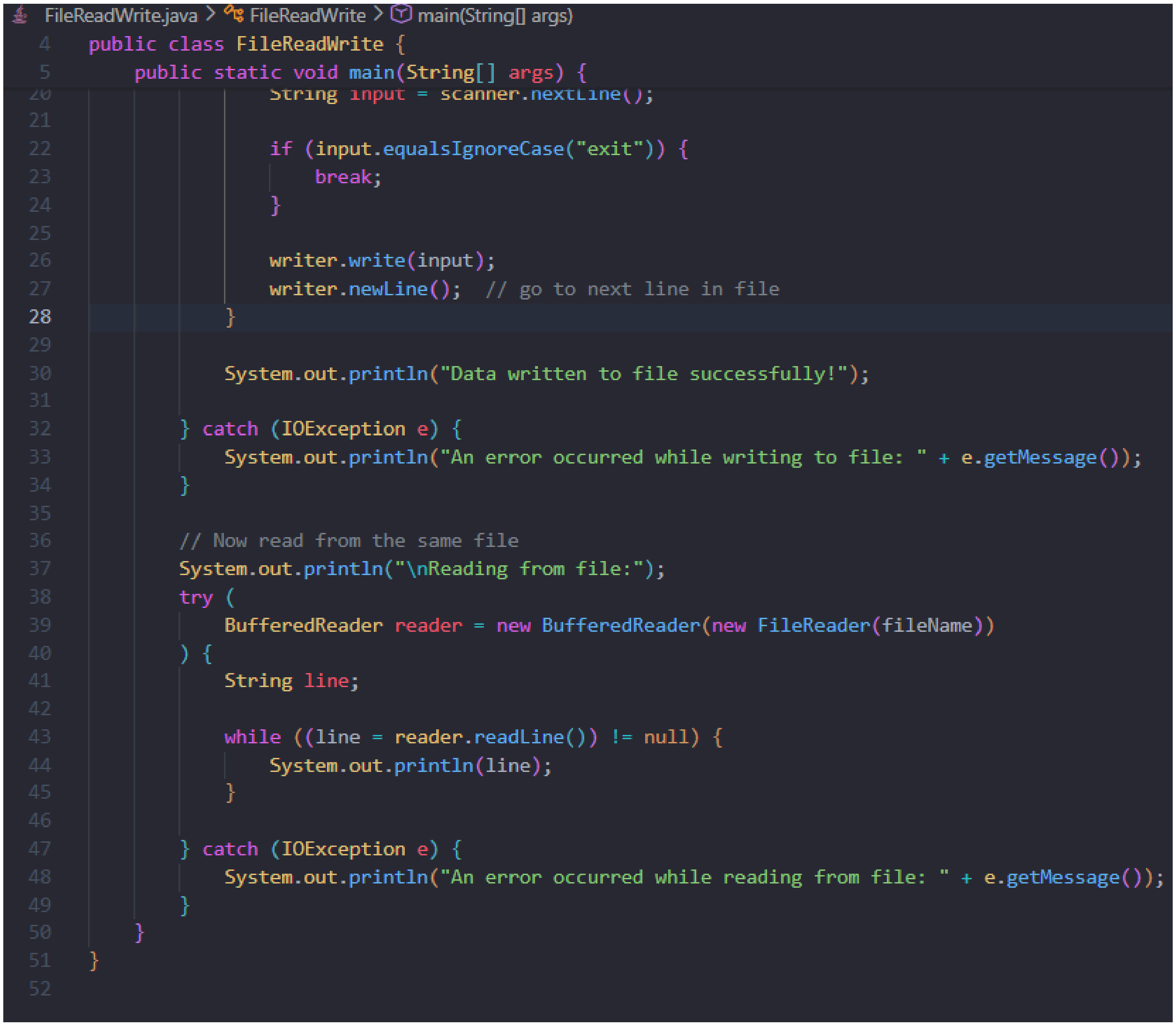
Logic: We use `FileInputStream` to read from an "input.txt" file and `FileOutputStream` to write to an "output.txt" file. We use a `try-catch` block to handle file-related exceptions. Inside `try`, we create a `while` loop that reads one byte at a time from the input stream using `read()`. The loop continues as long as `read()` does not return -1 (end of file). Each byte read is immediately written to the output stream using `write()`. Finally, we close both streams.



```java
import java.io.*;
import java.util.Scanner;

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

        String fileName = "data.txt";

        try (
            // For reading user input from keyboard
            Scanner scanner = new Scanner(System.in);

            // For writing to file
            BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));

        ) {
            System.out.println("Enter text to write to the file (type 'exit' to stop):");

            while (true) {
                String input = scanner.nextLine();

                if (input.equalsIgnoreCase("exit")) {
                    break;
                }

                writer.write(input);
                writer.newLine();  // go to next line in file
            }

            System.out.println("Data written to file successfully!");

        } catch (IOException e) {
            System.out.println("An error occurred while writing to file: " + e.getMessage());
        }
```

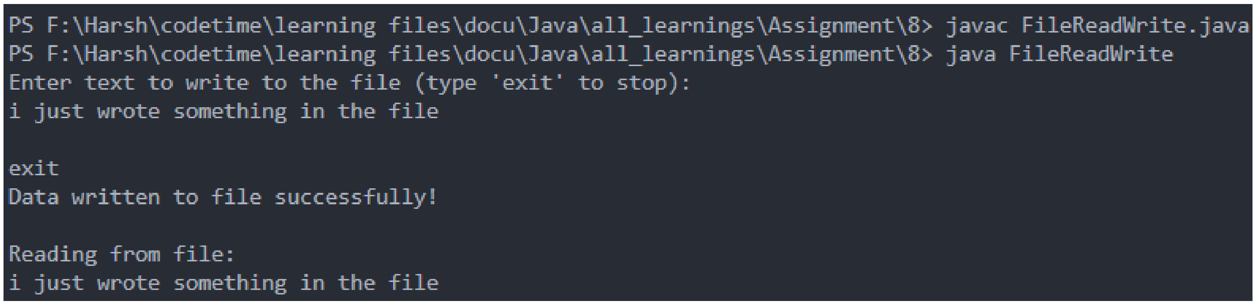
Code: Fig 8.1

Fig 8.2

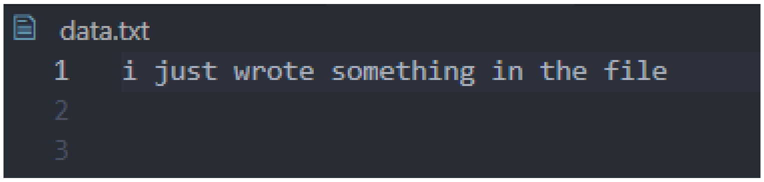


```java
public class FileReadWrite {
    public static void main(String[] args) {
                String input = scanner.nextLine();

                if (input.equalsIgnoreCase("exit")) {
                    break;
                }

                writer.write(input);
                writer.newLine();  // go to next line in file
            }

            System.out.println("Data written to file successfully!");

        } catch (IOException e) {
            System.out.println("An error occurred while writing to file: " + e.getMessage());
        }

        // Now read from the same file
        System.out.println("\nReading from file:");
        try (
            BufferedReader reader = new BufferedReader(new FileReader(fileName))
        ) {
            String line;

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }

        } catch (IOException e) {
            System.out.println("An error occurred while reading from file: " + e.getMessage());
        }
    }
}
```

Result: Fig 8.3



```
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\8> javac FileReadWrite.java
PS F:\Harsh\codetime\learning files\docu\Java\all_learnings\Assignment\8> java FileReadWrite
Enter text to write to the file (type 'exit' to stop):
i just wrote something in the file

exit
Data written to file successfully!

Reading from file:
i just wrote something in the file
```

Fig 8.4 

data.txt

```
i just wrote something in the file


```

8.2

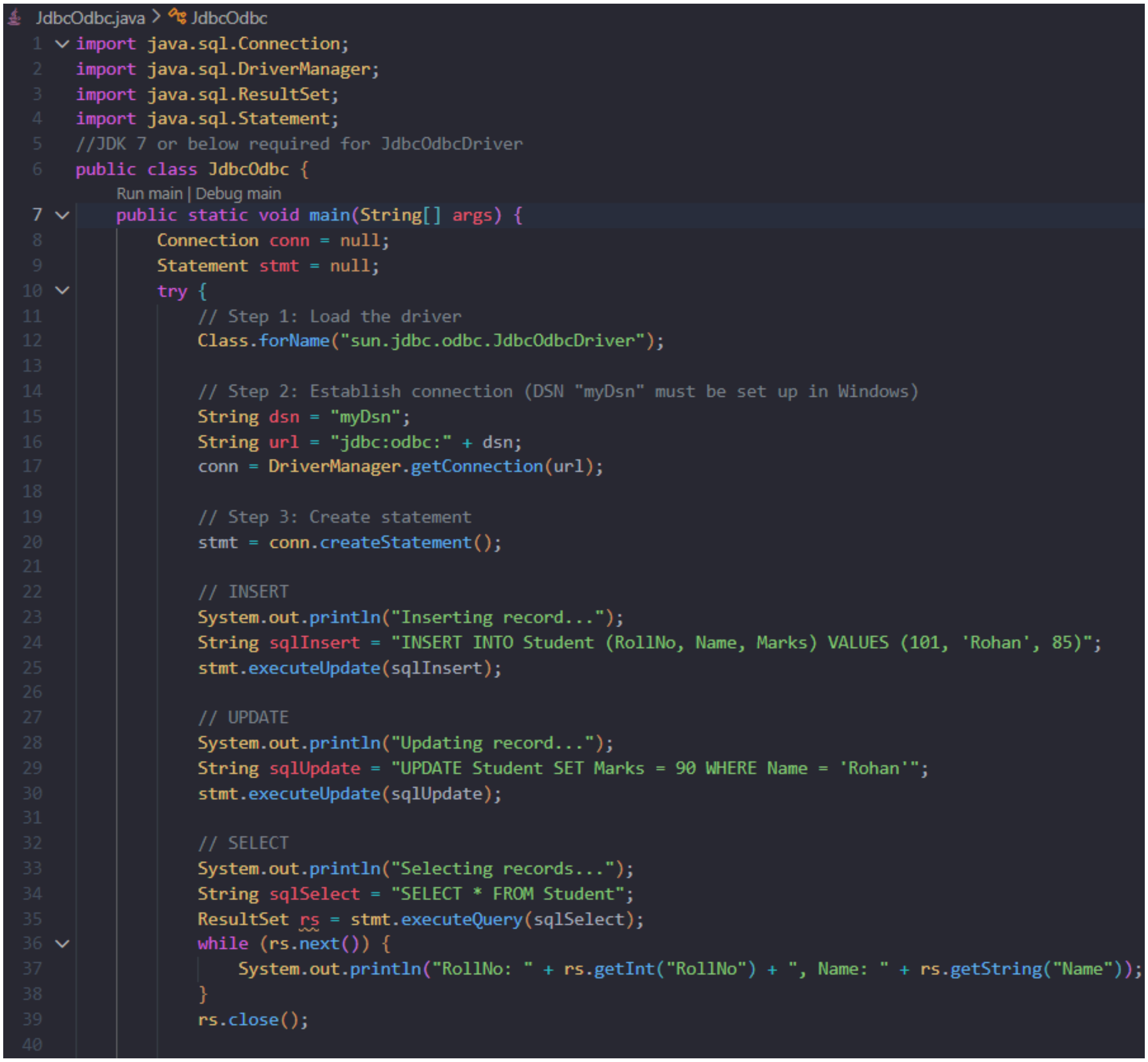
AIM: Write a program to establish connection with MS Access and manipulate it using select, insert, delete and update statement. (Write a program in Java to represent 'JDBC-ODBC' connectivity by inserting data in a table 'Student'.)

Logic: Important Note: The JDBC-ODBC Bridge driver (`sun.jdbc.odbc.JdbcOdbcDriver`) was removed in Java 8 (2014) and is highly deprecated. This code will only work with JDK 7 or older, and requires a 32-bit ODBC DSN (Data Source Name) to be pre-configured in the Windows "ODBC Data Source Administrator".

The logic is:

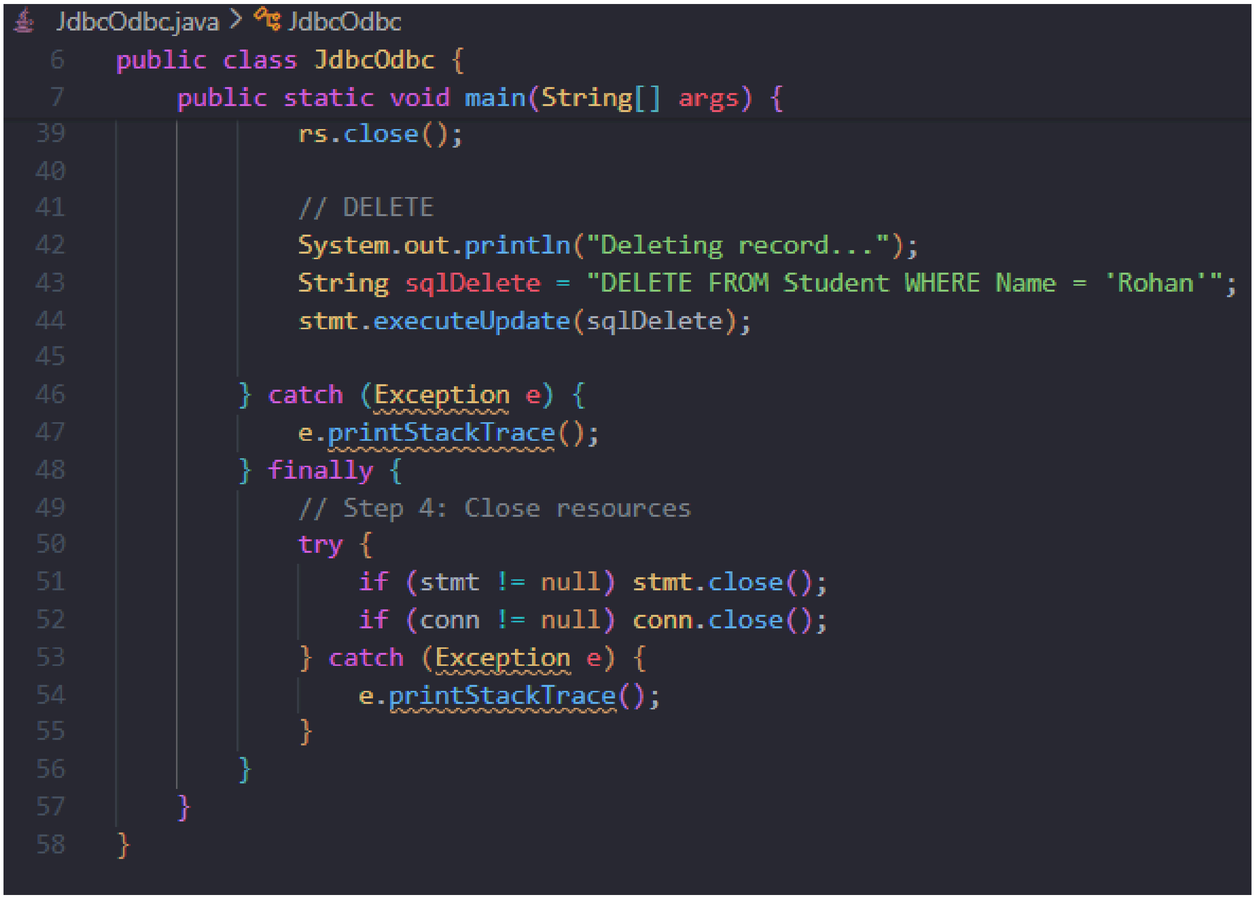
1. Load the JDBC-ODBC driver using `Class.forName()`.
2. Establish a connection using `DriverManager.getConnection()`, referencing the DSN (e.g., "myAccessDb").
3. Create a `Statement` object.
4. Execute SQL queries: `executeUpdate()` for INSERT, UPDATE, DELETE, and `executeQuery()` for SELECT.
5. When selecting, iterate over the `ResultSet` to print the data.
6. Close all resources (`ResultSet`, `Statement`, `Connection`) in a `finally` block.

Code: Fig 8.5



```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
//JDK 7 or below required for JdbcOdbcDriver
public class JdbcOdbc {
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        try {
            // Step 1: Load the driver
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

            // Step 2: Establish connection (DSN "myDsn" must be set up in Windows)
            String dsn = "myDsn";
            String url = "jdbc:odbc:" + dsn;
            conn = DriverManager.getConnection(url);

            // Step 3: Create statement
            stmt = conn.createStatement();

            // INSERT
            System.out.println("Inserting record...");
            String sqlInsert = "INSERT INTO Student (RollNo, Name, Marks) VALUES (101, 'Rohan', 85)";
            stmt.executeUpdate(sqlInsert);

            // UPDATE
            System.out.println("Updating record...");
            String sqlUpdate = "UPDATE Student SET Marks = 90 WHERE Name = 'Rohan'";
            stmt.executeUpdate(sqlUpdate);

            // SELECT
            System.out.println("Selecting records...");
            String sqlSelect = "SELECT * FROM Student";
            ResultSet rs = stmt.executeQuery(sqlSelect);
            while (rs.next()) {
                System.out.println("RollNo: " + rs.getInt("RollNo") + ", Name: " + rs.getString("Name"));
            }
            rs.close();

```

Fig 8.6



```java
public class JdbcOdbc {
    public static void main(String[] args) {
            rs.close();

            // DELETE
            System.out.println("Deleting record...");
            String sqlDelete = "DELETE FROM Student WHERE Name = 'Rohan'";
            stmt.executeUpdate(sqlDelete);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // Step 4: Close resources
            try {
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

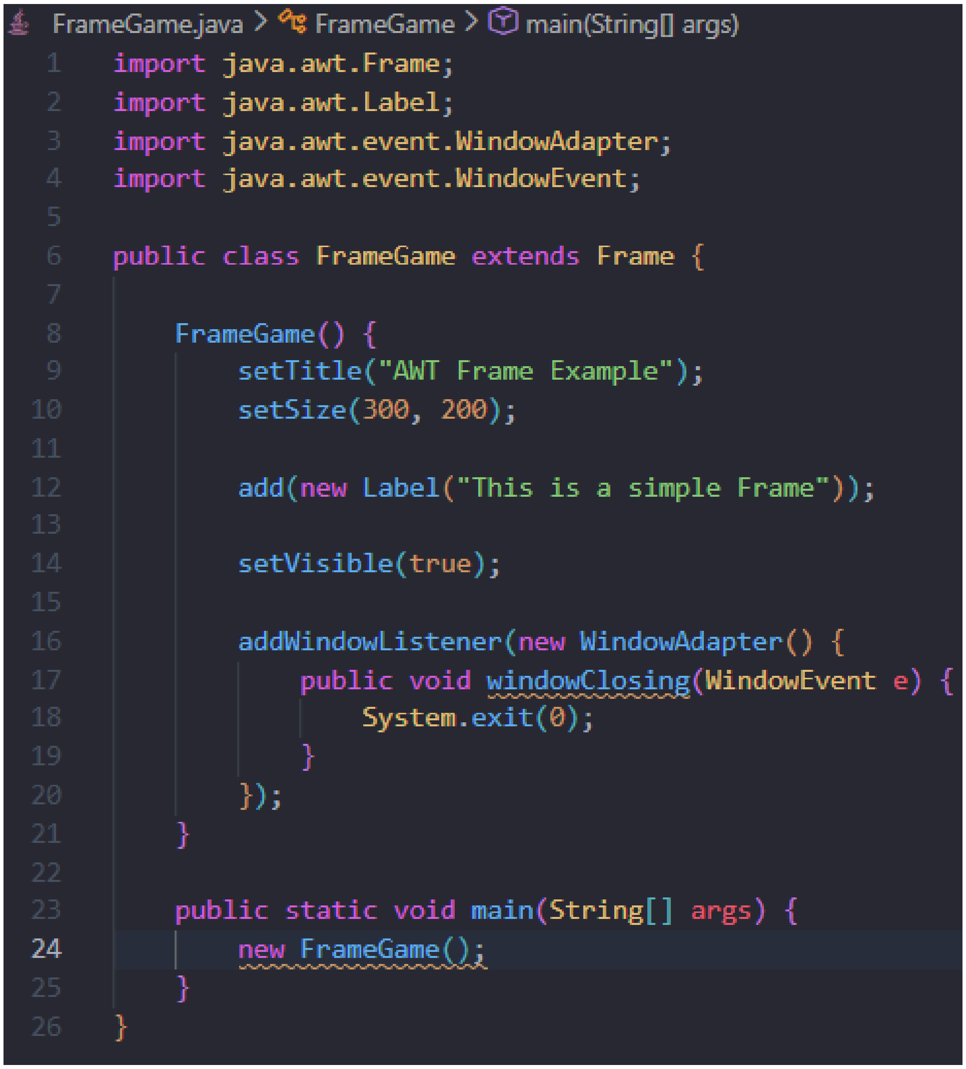
LAB ASSIGNMENT 9

9.1

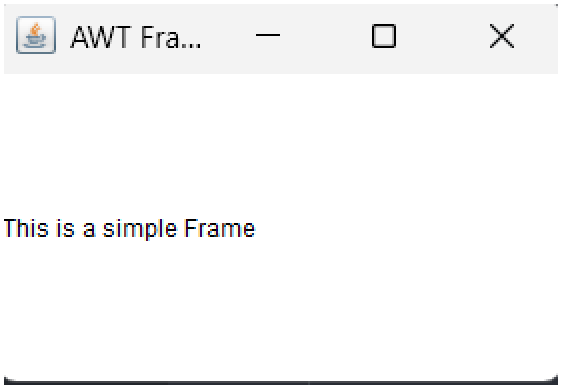
AIM: Write a program to use Frame Class

Logic: We create a class that extends the `java.awt.Frame` class. In the constructor, we set the window's title using `setTitle()`, its size using `setSize()`, and its layout using `setLayout()`. We make it visible using `setVisible(true)`. We also add a `WindowAdapter` and override the `windowClosing()` method to ensure the program exits (using `System.exit(0)`) when the window's close button is clicked.

Code: Fig 9.1



```java
import java.awt.Frame;
import java.awt.Label;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class FrameGame extends Frame {

    FrameGame() {
        setTitle("AWT Frame Example");
        setSize(300, 200);

        add(new Label("This is a simple Frame"));

        setVisible(true);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    public static void main(String[] args) {
        new FrameGame();
    }
}
```



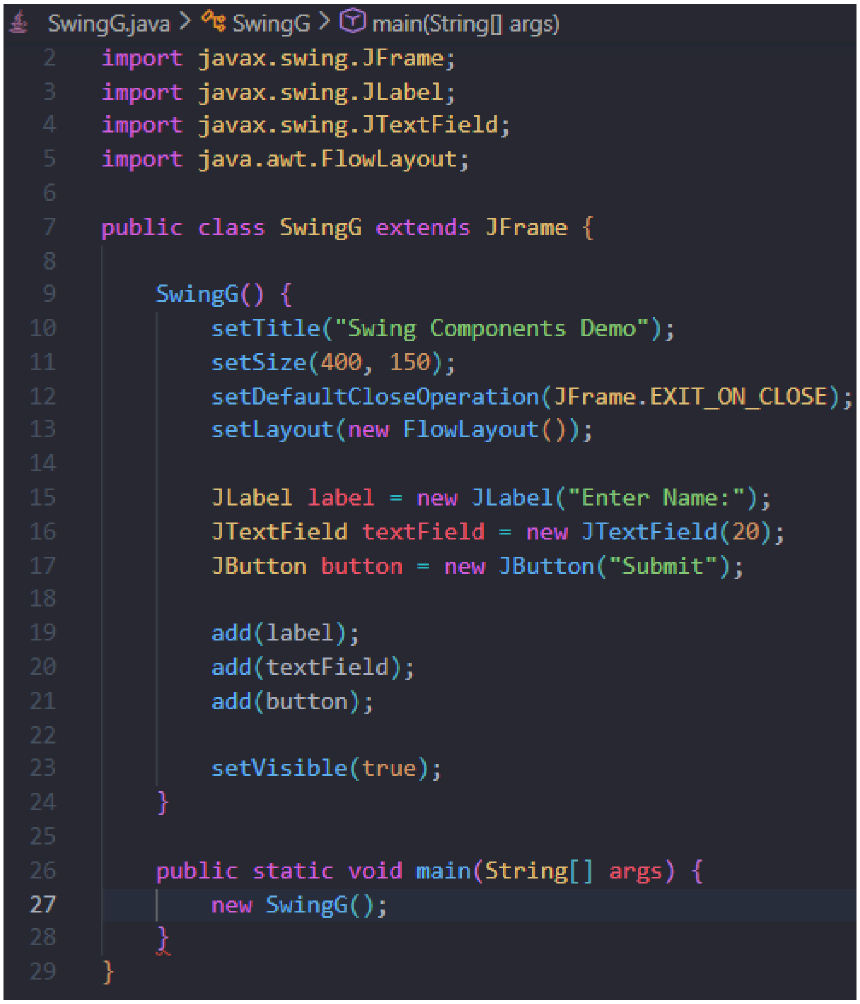
This is a simple Frame

Result: Fig 9.2

9.2

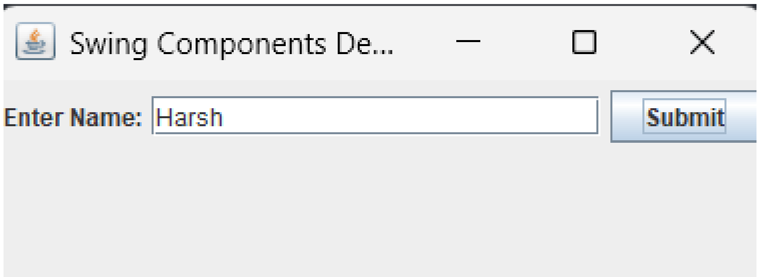
AIM: Write a program to create swing and use JButton and JTextField etc

Logic: We create a class that extends `javax.swing.JFrame`. In the constructor, we set the title and size. We create a `JTextField` for text input and a `JButton`. We add these components to the frame's content pane (which uses `FlowLayout` by default). We use `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` so the program terminates when the window is closed, and finally set the frame to be visible.



```java
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import java.awt.FlowLayout;

public class SwingG extends JFrame {

    SwingG() {
        setTitle("Swing Components Demo");
        setSize(400, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());

        JLabel label = new JLabel("Enter Name:");
        JTextField textField = new JTextField(20);
        JButton button = new JButton("Submit");

        add(label);
        add(textField);
        add(button);

        setVisible(true);
    }

    public static void main(String[] args) {
        new SwingG();
    }
}
```

Code: Fig 9.3



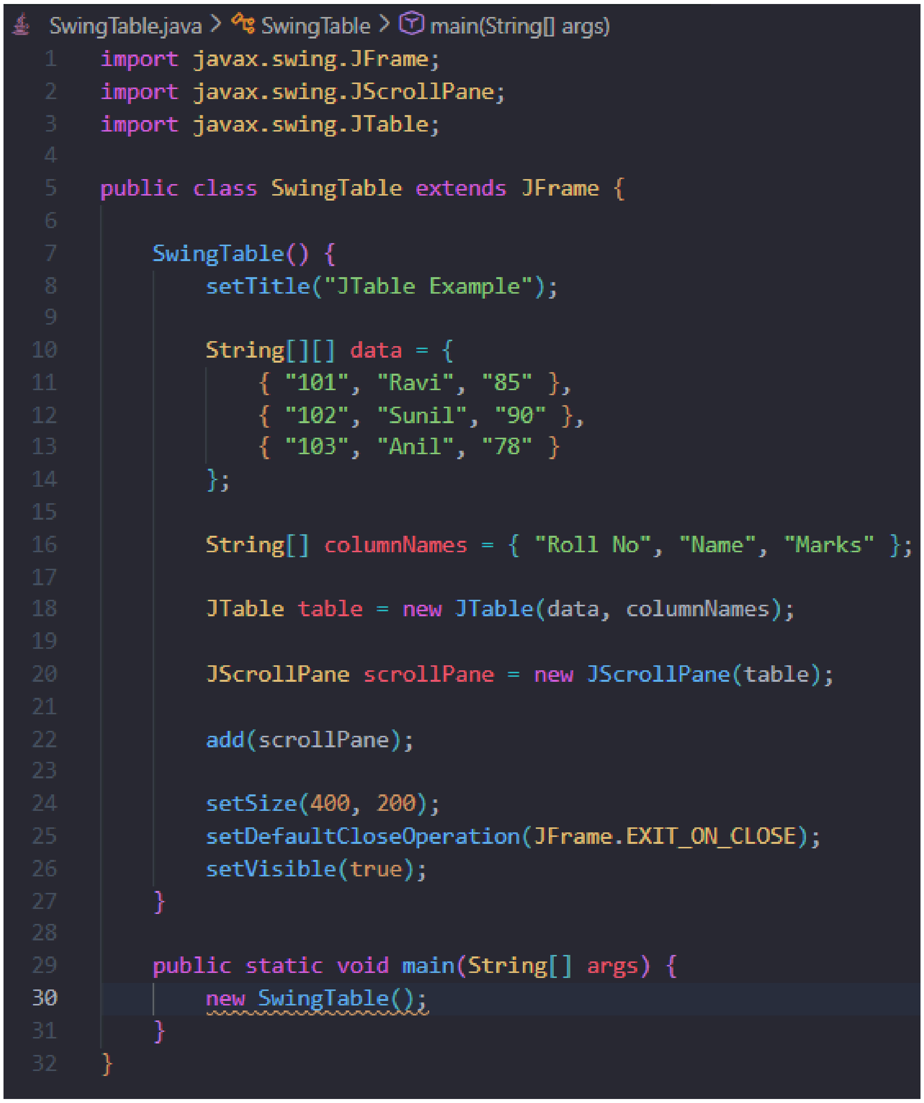
Enter Name: Harsh

Result: Fig 9.4

9.3

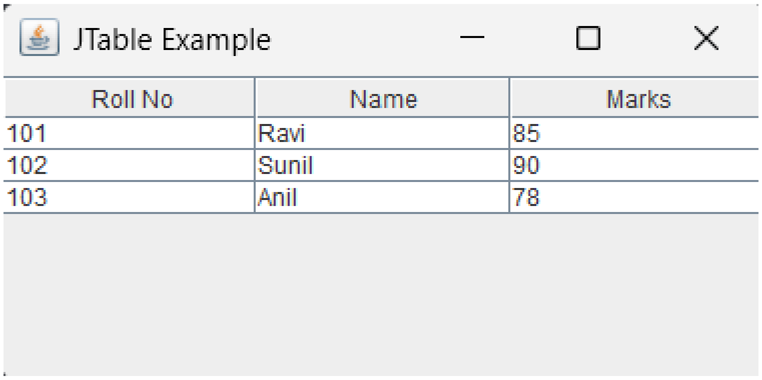
AIM: Write a Swing to create a table.

Logic: We create a class extending `JFrame`. We define a `String` array for the column headers and a 2D `String` array for the table's data. We create a `JTable` object, passing the data and headers to its constructor. To make the table scrollable and display the headers, we place the `JTable` inside a `JScrollPane`. We add this scroll pane to the `JFrame` and set the frame's size and close operation.



```java
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class SwingTable extends JFrame {

    SwingTable() {
        setTitle("JTable Example");

        String[][] data = {
            { "101", "Ravi", "85" },
            { "102", "Sunil", "90" },
            { "103", "Anil", "78" }
        };

        String[] columnNames = { "Roll No", "Name", "Marks" };

        JTable table = new JTable(data, columnNames);

        JScrollPane scrollPane = new JScrollPane(table);

        add(scrollPane);

        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        new SwingTable();
    }
}
```

Code: Fig 9.5



Roll No	Name	Marks
101	Ravi	85
102	Sunil	90
103	Anil	78

Result: Fig 9.6.

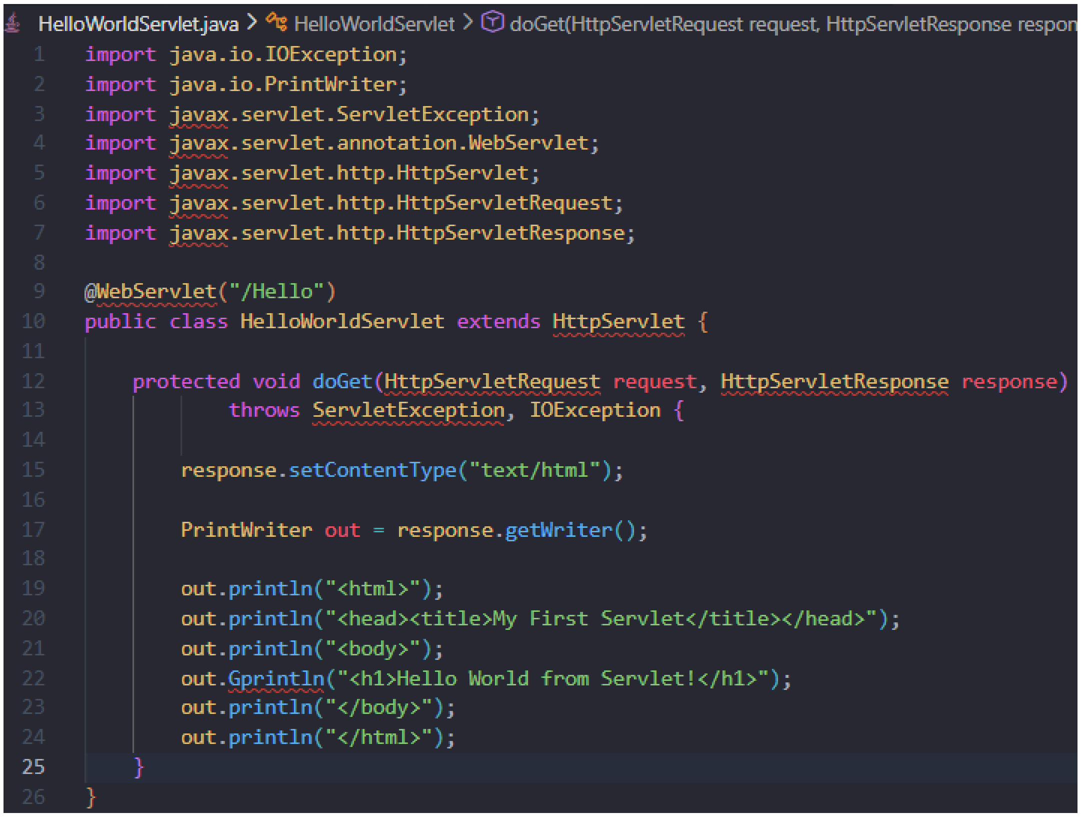
LAB ASSIGNMENT 10

10.1

AIM: Write a Servlet in java

Logic: We create a class that extends `HttpServlet`. We use the `@WebServlet("/Hello")` annotation to map this servlet to the URL "/Hello", which avoids needing a `web.xml` file. We override the `doGet` method, which handles HTTP GET requests. Inside `doGet`, we set the response content type to "text/html" using `response.setContentType()`. We get a `PrintWriter` object from the response and use its `println()` method to write basic HTML code, including a "Hello World" message, to the web page.

Code: Fig 10.1



HelloWorldServlet.java > HelloWorldServlet > doGet(HttpServletRequest request, HttpServletResponse respon

```java
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Hello")
public class HelloWorldServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head><title>My First Servlet</title></head>");
        out.println("<body>");
        out.Gprintln("<h1>Hello World from Servlet!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```