

Program – 1

Aim – Create a class Box that uses a parameterized constructor to initialize the dimensions of a box. The dimensions of the Box are width, height, depth. The class should have a method that can return the volume of the box. Create an object of the Box class and test the functionalities.

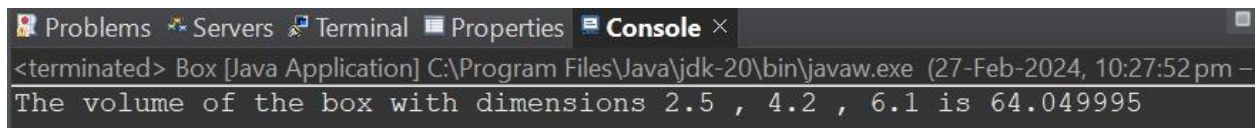
Theory –

In Java, a Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

Input –

```
public class box {  
    float height, width, depth;  
    box(float h, float w, float d){  
        height = h;  
        width = w;  
        depth = d;  
    }  
    private float getvolume() {  
        return height*width*depth;  
    }  
    public static void main(String[] args) {  
        box b=new box(2.4f,5.3f,5.2f);  
        System.out.println("The Volume of the box is --> " + b.getvolume() + " units");  
    }  
}
```

Output –



```
Problems Servers Terminal Properties Console ×  
<terminated> Box [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (27-Feb-2024, 10:27:52 pm -  
The volume of the box with dimensions 2.5 , 4.2 , 6.1 is 64.049995
```

Program – 2

Aim – Create a base class Fruit which has name, taste and size as its attributes. A method called eat() is created which describes the name of the fruit and its taste. Inherit the same in 2 other class Apple and Orange and override the eat() method to represent each fruit taste. (Method overriding)

Theory –

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Input –

```
package java;
```

```
class fruit{
```

```
    String name, taste;
```

```
    int size;
```

```
    fruit(){};
```

```
    fruit(String name, String taste){
```

```
        this.name = name;
```

```
        this.taste = taste;
```

```
    }
```

```

        void eat() {
            System.out.println("The fruit is " +name + " and it tastes " + taste);
        }
    }

class Apple extends fruit{
    void eat() {
        System.out.println("The fruit is apple and it tastes sweet");
    }
}

class Orange extends fruit{
    void eat() {
        System.out.println("The fruit is orange and it tastes sour");
    }
}

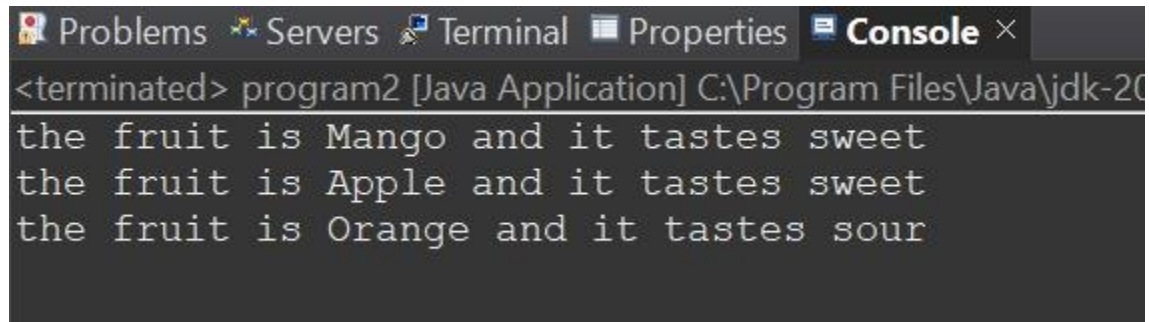
class Mango extends fruit{
    void eat() {
        System.out.println("The fruit is mango and it tastes sweet");
    }
}

public class inheritance {
    public static void main(String[] args) {
        fruit mango = new fruit("Mango", "Sweet");
        mango.eat();
        Apple a = new Apple();
        a.eat();
        Orange o = new Orange();
        o.eat();
    }
}

```

}

Output –

A screenshot of an IDE's console window. The title bar shows tabs for 'Problems', 'Servers', 'Terminal', 'Properties', and 'Console'. The console text shows the program has terminated and displays three lines of output: 'the fruit is Mango and it tastes sweet', 'the fruit is Apple and it tastes sweet', and 'the fruit is Orange and it tastes sour'.

```
<terminated> program2 [Java Application] C:\Program Files\Java\jdk-20  
the fruit is Mango and it tastes sweet  
the fruit is Apple and it tastes sweet  
the fruit is Orange and it tastes sour
```

Program – 3

Aim – Write a program to create a class named shape. It should contain 2 methods- draw() and erase() which should print “Drawing Shape” and “Erasing Shape” respectively. For this class we have three subclasses- Circle, Triangle and Square and each class override the parent class functions- draw () and erase(). The draw() method should print “Drawing Circle”, “Drawing Triangle”, “Drawing Square” respectively. The erase() method should print “Erasing Circle”, “Erasing Triangle”, “Erasing Square” respectively. Create objects of Circle, Triangle and Square in the following way and observe the polymorphic nature of the class by calling draw() and erase() method using each object. Shape c=new Circle(); Shape t=new Triangle(); Shape s=new Square(); (Polymorphism)

Theory –

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. Method overriding is one of the ways by which Java achieves Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

Input –

```
package java;

class Shape {
    void draw() {
        System.out.println("Drawing Shape");
    }
    void erase() {
        System.out.println("Erasing Shape");
    }
}

class Circle extends Shape {
    void draw() {
```

```
        System.out.println("Drawing Circle");
    }
    void erase() {
        System.out.println("Erasing Circle");
    }
}

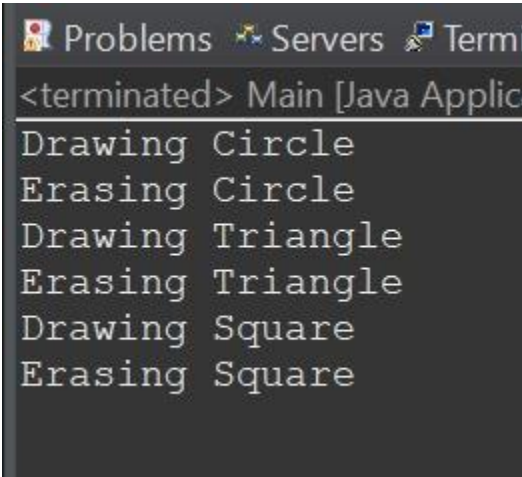
class Triangle extends Shape {
    void draw() {
        System.out.println("Drawing Triangle");
    }
    void erase() {
        System.out.println("Erasing Triangle");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Drawing Square");
    }
    void erase() {
        System.out.println("Erasing Square");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape c = new Circle();
        Shape t = new Triangle();
        Shape s = new Square();
        c.draw();
    }
}
```

```
        c.erase();  
        t.draw();  
        t.erase();  
        s.draw();  
        s.erase();  
    }  
}
```

Output –



The screenshot shows a terminal window from an IDE. The title bar includes icons for 'Problems', 'Servers', and 'Terminal'. The terminal text shows the program has terminated and then lists six actions: 'Drawing Circle', 'Erasing Circle', 'Drawing Triangle', 'Erasing Triangle', 'Drawing Square', and 'Erasing Square'.

```
<terminated> Main [Java Applic  
Drawing Circle  
Erasing Circle  
Drawing Triangle  
Erasing Triangle  
Drawing Square  
Erasing Square
```

Program – 4

Aim – Write a Program to take care of Number Format Exception if user enters values other than integer for calculating average marks of 2 students. The name of the students and marks in 3 subjects are taken from the user while executing the program. In the same Program write your own Exception classes to take care of Negative values and values out of range (i.e. other than in the range of 0-100)

Theory –

An exception is an issue (run time error) that occurred during the execution of a program. When an exception occurred the program gets terminated abruptly and, the code past the line that generated the exception never gets executed. Java exceptions cover almost all the general types of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are a few of the reasons to use custom exceptions -

To catch and provide specific treatment to a subset of existing Java exceptions.

Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

Input –

```
package java;
import java.util.Scanner;
class NegativeValueException extends Exception {
    public NegativeValueException(String s) {
        super(s);
    }
}
class ValueOutOfRangeException extends Exception {
    public ValueOutOfRangeException(String s) {
        super(s);
    }
}
public class prog4 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        for (int i = 0; i < 2; i++) {
            try {
```



```

        System.out.println("Enter the name of the student:");
        String name = sc.next();
        System.out.println("Enter the marks of the student in 3 subjects:");
        int sum = 0;
        for (int j = 0; j < 3; j++) {
            int mark = Integer.parseInt(sc.next());
            if (mark < 0) throw new NegativeValueException("Negative value");
            if (mark > 100) throw new ValueOutOfRangeException("Value out of range");
            sum += mark;
        }
        System.out.println("The average marks of " + name + " is " + sum / 3.0);
    } catch (NumberFormatException e) {
        System.out.println("Number format exception occurred");
    } catch (NegativeValueException e) {
        System.out.println(e.getMessage());
    } catch (ValueOutOfRangeException e) {
        System.out.println(e.getMessage());
    }
}
}
sc.close();
}
}

```

Output –

```

Enter the name of the student:
harsh
Enter the marks of the student in 3 subjects:
90
85
75
The average marks of harsh is 83.33333333333333
Enter the name of the student:
john
Enter the marks of the student in 3 subjects:
80
75
95
The average marks of john is 83.33333333333333

```

Program – 5

Aim – Write a program that takes as input the size of the array and the elements in the array. The program then asks the user to enter a particular index and prints the element at that index. Index starts from zero. This program may generate Array Index Out Of Bounds Exception or NumberFormatException. Use exception handling mechanisms to handle this exception.

Theory –

An exception is an issue (run time error) that occurred during the execution of a program. When an exception occurred the program gets terminated abruptly and, the code past the line that generated the exception never gets executed

Array-Index-Out-Of-Bounds Exception -

This exception is thrown when an attempt is made to access an array element with an index that is outside the array's bounds.

In the second program, it's handled when the user enters an index that is not within the valid range for the array.

Input –

```
package java;

import java.util.Scanner;

public class prog5 {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        try {

            System.out.print("Enter the size of the array: ");

            int size = Integer.parseInt(scanner.nextLine());

            int[] arr = new int[size];

            System.out.println("Enter the elements of the array:");

            for (int i = 0; i < size; i++) {

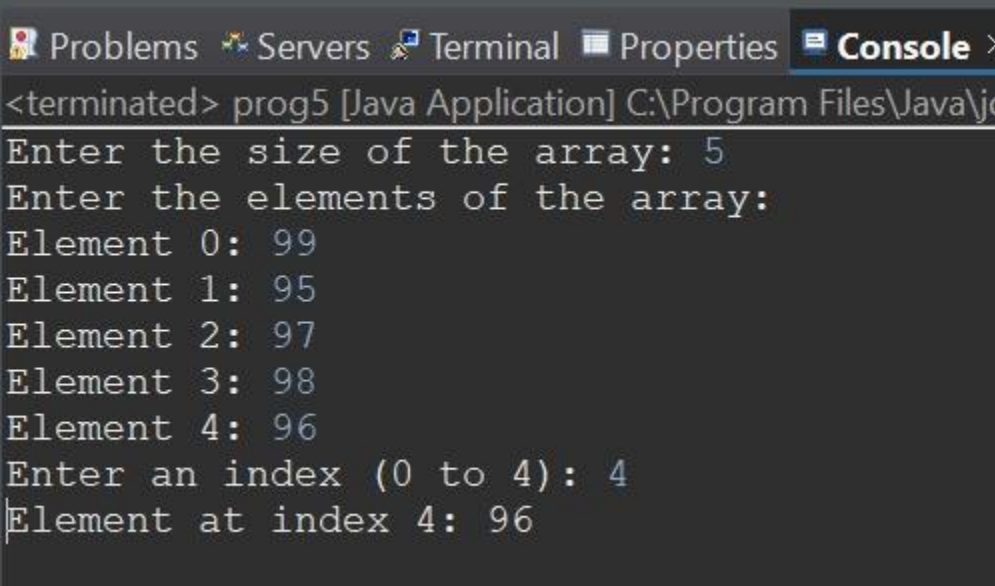
                System.out.print("Element " + i + ": ");
```

```

        arr[i] = scanner.nextInt();
    }
    System.out.print("Enter an index (0 to " + (size - 1) + "): ");
    int index = scanner.nextInt();
    System.out.println("Element at index " + index + ": " + arr[index]);
} catch (NumberFormatException e) {
    System.err.println("Invalid input. Please enter valid integers.");
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Index out of bounds. Please enter a valid index.");
}
}
}

```

Output –



```

Problems Servers Terminal Properties Console >
<terminated> prog5 [Java Application] C:\Program Files\Java\j
Enter the size of the array: 5
Enter the elements of the array:
Element 0: 99
Element 1: 95
Element 2: 97
Element 3: 98
Element 4: 96
Enter an index (0 to 4): 4
Element at index 4: 96

```

Program – 6

Aim – Write a Java program to demonstrate the concept of socket programming.

Theory –

Socket programming is a way of enabling communication between two nodes on a network. It allows processes to communicate with each other, whether they are on the same machine or on different machines across a network.

In socket programming, a socket is one endpoint of a two-way communication link between two programs running on the network. Each socket is associated with a port number, which helps in routing the data to the appropriate application or service running on a system.

There are two types of sockets:

1. Server Socket: This socket waits for incoming connections. When a client connects to the server, the server creates a new socket for that particular connection and communicates with the client through that socket.
2. Client Socket: This socket initiates a connection to the server. Once the connection is established, the client and server can exchange data through their respective sockets.

Input –

Server.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {
    public static void main(String[] args) {
        final int PORT = 12345;
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
```

```

System.out.println("Server started. Listening on port " + PORT);
while (true) {
    Socket clientSocket = serverSocket.accept();
    System.out.println("Client connected: " + clientSocket);
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket));
    String inputLine;
    while ((inputLine = in.readLine()) != null) {
        System.out.println("Received from client: " + inputLine);
        System.out.println("Server received: " + inputLine);
    }
    in.close();
    out.close();
    clientSocket.close();
    System.out.println("Client disconnected: " + clientSocket);
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

Client.java

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
public class Client {

```

```

public static void main(String[] args) {
    final String SERVER_IP = "localhost";
    final int SERVER_PORT = 12345;
    try (Socket socket = new Socket(SERVER_IP, SERVER_PORT);
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        BufferedReader userInput = new BufferedReader(new InputStreamReader(System.in)))
    {
        System.out.println("Connected to server. Type a message and press Enter to send.");
        String userInputLine;
        while ((userInputLine = userInput.readLine()) != null) {
            out.println(userInputLine);
            System.out.println("Server response: " + in.readLine());
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Output –

```

Server started. Listening on port 12345
Client connected: Socket[addr=/127.0.0.1,port=55852,localport=12345]
Received from client: Hello from client!
Client disconnected: Socket[addr=/127.0.0.1,port=55852,localport=12345]

```

```

Connected to server. Type a message and press Enter to send.
Server response: Server received: Hello from client!

```

Program – 7

Aim – Implement Datagram UDP socket programming in java.

Theory –

UDP (User Datagram Protocol) in Java refers to the implementation of UDP socket programming using Java programming language. UDP is a connectionless protocol that operates at the transport layer of the OSI model. In Java, UDP communication is facilitated using classes provided by the java.net package, primarily the DatagramSocket and DatagramPacket classes.

1. DatagramSocket Class:

- i) The DatagramSocket class represents a socket for sending and receiving UDP packets (datagrams).
- ii) It provides methods for creating and binding sockets, sending and receiving datagrams, setting socket options, etc.

2. DatagramPacket Class:

- i) The DatagramPacket class represents a packet (datagram) of data to be sent or received by a DatagramSocket.
- ii) It encapsulates the data to be sent or received, as well as the destination address and port for outgoing packets, or the sender's address and port for incoming packets.

Input –

```
import java.io.*;
import java.net.*;

public class UDPServer {
    public static void main(String[] args) {
        int port = 12345;
        try {
            DatagramSocket serverSocket = new DatagramSocket(port);
            byte[] receiveData = new byte[1024];
            byte[] sendData;

            System.out.println("Server is running and listening on port " + port);

            while (true) {
                DatagramPacket receivePacket = new DatagramPacket(receiveData,
                    receiveData.length);
```

```

serverSocket.receive(receivePacket);

String sentence = new String(receivePacket.getData(), 0, receivePacket.getLength());

System.out.println("Received from client: " + sentence);

String capitalizedSentence = sentence.toUpperCase();

InetAddress IPAddress = receivePacket.getAddress();

int clientPort = receivePacket.getPort();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
IPAddress, clientPort);

serverSocket.send(sendPacket);

System.out.println("Sent to client: " + capitalizedSentence);
}
} catch (IOException e) {
    System.err.println("IOException: " + e.getMessage());
}
}
}

```

Output –

```

Server is running and listening on port 12345
Received from client: Hello UDP Server!
Sent to client: HELLO UDP SERVER!

```


Program – 8

Aim – Implement Socket programming for TCP in Java Server and Client Sockets.

Theory –

1. Server Socket:

ServerSocket class is used to create a server-side socket that listens for incoming client connections.

The server socket is bound to a specific port number using the ServerSocket(int port) constructor.

The accept() method of ServerSocket waits for a client to connect to the server. Once a connection is established, it returns a Socket object representing the client connection.

2. Client Socket:

Socket class is used to create a client-side socket that connects to a server.

The Socket(String host, int port) constructor is used to specify the hostname/IP address and port number of the server to connect to.

Communication with the server is performed using input and output streams obtained from the Socket object (getInputStream() and getOutputStream() methods).

3. Input and Output Streams:

InputStream and OutputStream classes are used to read from and write to sockets, respectively.

For reading text-based data, BufferedReader can be used to read from an InputStream.

For writing text-based data, DataOutputStream can be used to write to an OutputStream.

4. Sending and Receiving Data:

After establishing a connection, the server and client can exchange data by sending and receiving messages.

The server can read data sent by the client using input streams, and it can write data back to the client using output streams.

Similarly, the client can read data sent by the server and write data to the server.

5. Closing the Connection:

After communication is complete, both the server and client should close their sockets to release system resources.

This is typically done by calling the close() method on the Socket and ServerSocket objects.

Input –

TCP Server

```
import java.io.*;
```

```
import java.net.*;
```

```
public class TCPServer {
```

```

public static void main(String[] args) {
    try {
        ServerSocket serverSocket = new ServerSocket(5000);

        System.out.println("Server is waiting for client...");

        Socket clientSocket = serverSocket.accept(); // Wait for a client connection

        System.out.println("Client connected: " + clientSocket);

        BufferedReader inFromClient = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

        DataOutputStream outToClient = new
DataOutputStream(clientSocket.getOutputStream());

        String clientMessage = inFromClient.readLine();

        System.out.println("Received from client: " + clientMessage);

        outToClient.writeBytes(clientMessage.toUpperCase() + '\n');

        clientSocket.close();

        serverSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

TCP Client

```

import java.io.*;
import java.net.*;

public class TCPClient {

    public static void main(String[] args) {

        try {

            Socket clientSocket = new Socket("localhost", 5000);

            System.out.println("Connected to server.");

```

```

        BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

        DataOutputStream outToServer = new
DataOutputStream(clientSocket.getOutputStream());

        String messageToSend = "Hello, Server!";
        outToServer.writeBytes(messageToSend + '\n');
        System.out.println("Sent to server: " + messageToSend);
        String serverResponse = inFromServer.readLine();
        System.out.println("Received from server: " + serverResponse);
        clientSocket.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Output –

```

Server is waiting for client...
Client connected: Socket[addr=/127.0.0.1,port=58550,localport=5000]
Received from client: Hello, Server!

```

```

Connected to server.
Sent to server: Hello, Server!
Received from server: HELLO, SERVER!

```