Lesson 05 Demo 01

Implementing Wrapper Classes

Objective: To implement Java wrapper classes to represent primitive data types as objects, enabling utility methods and use in collections

Tools Required: Eclipse IDE

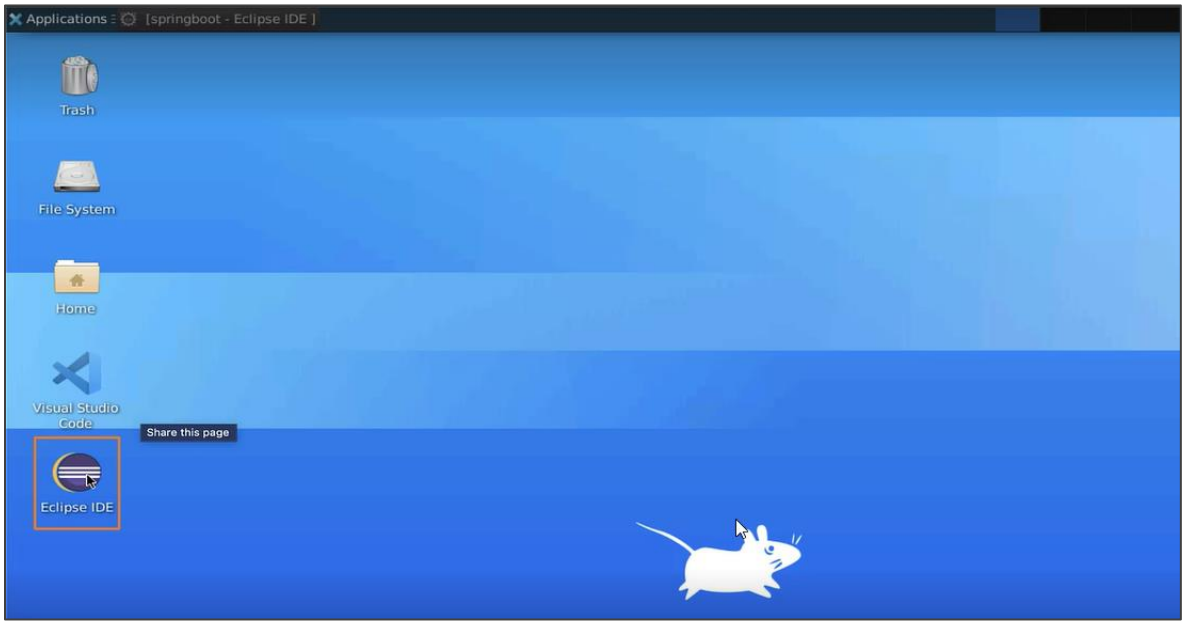
Prerequisites: None

Steps to be followed:

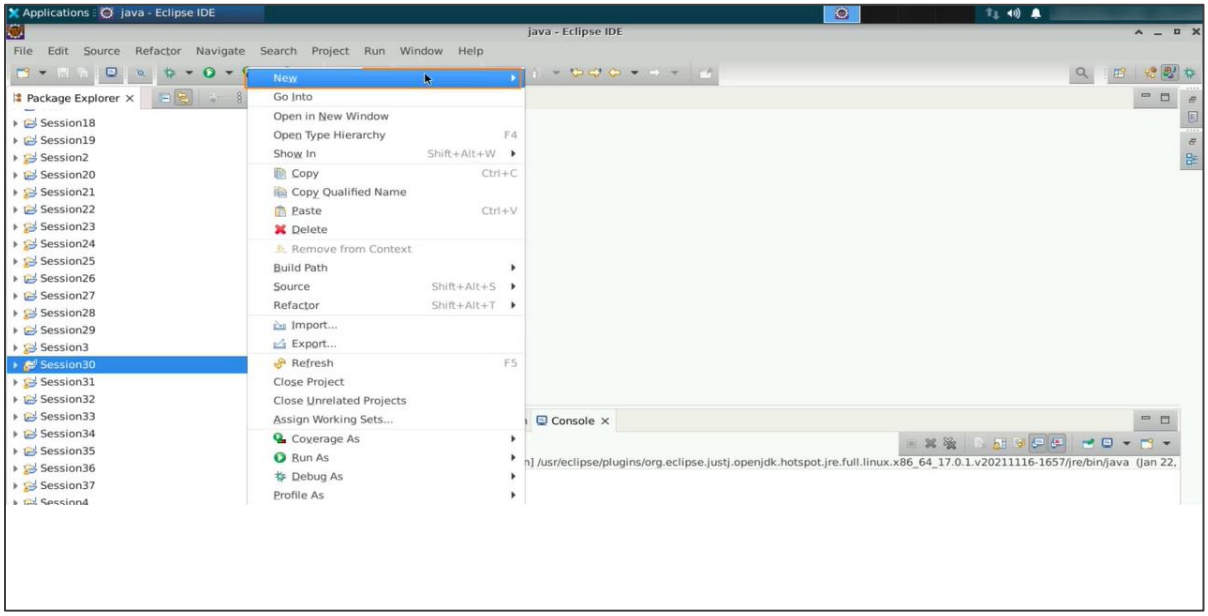
1. Create a Java project
2. Create primitive variables
3. Add methods on the wrapper classes

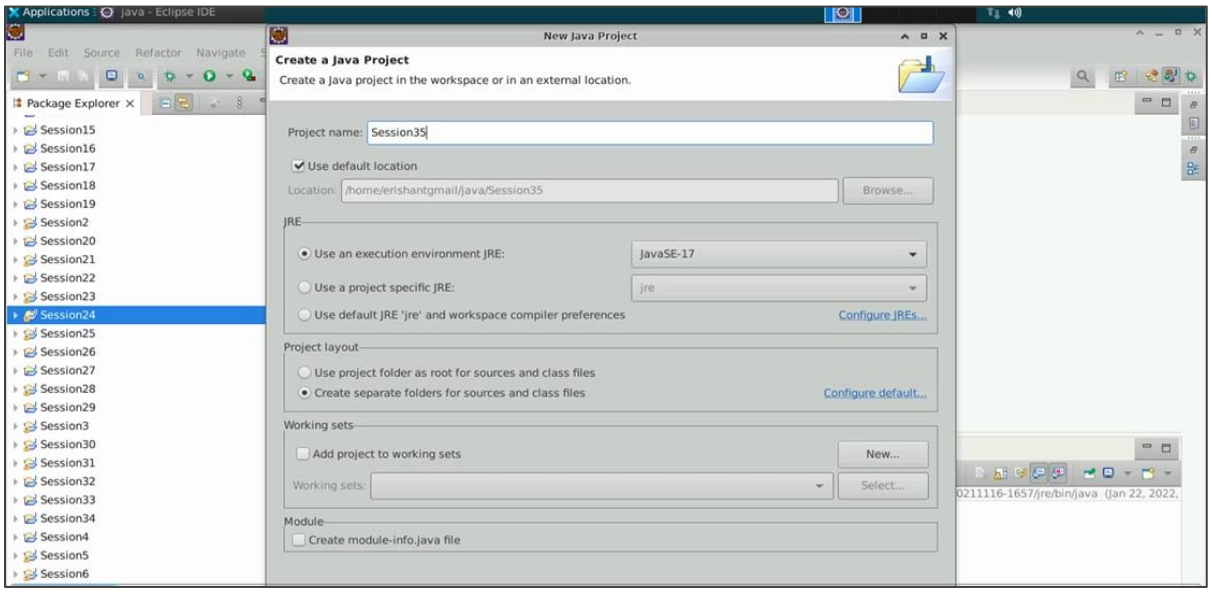
Step 1: Create a Java project

1.1 Open the **Eclipse IDE**

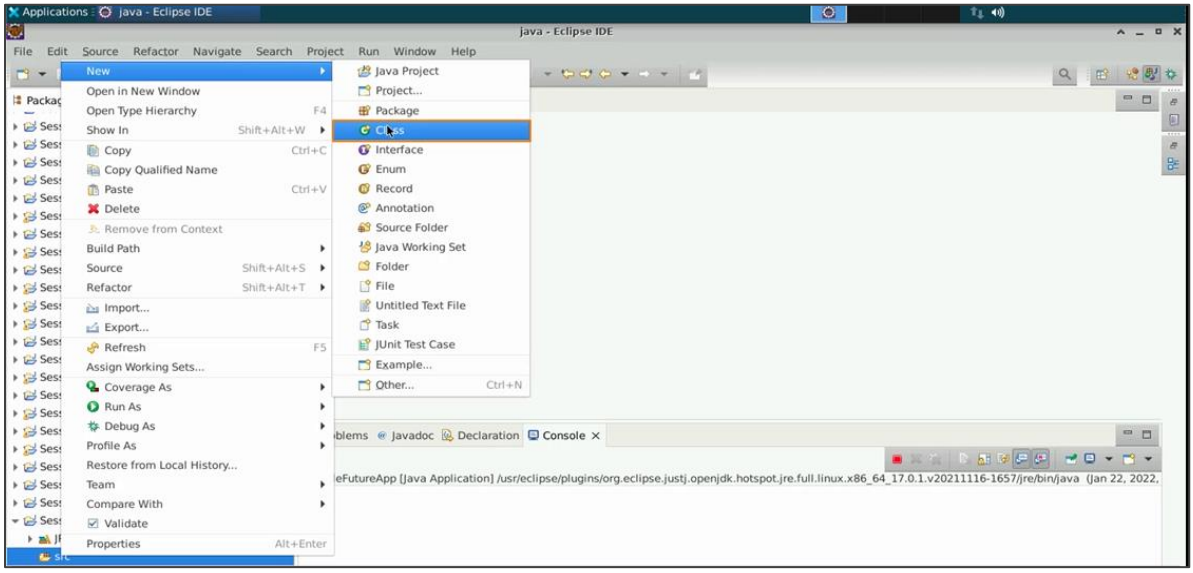


1.2 Create a new Java project called **Session35**

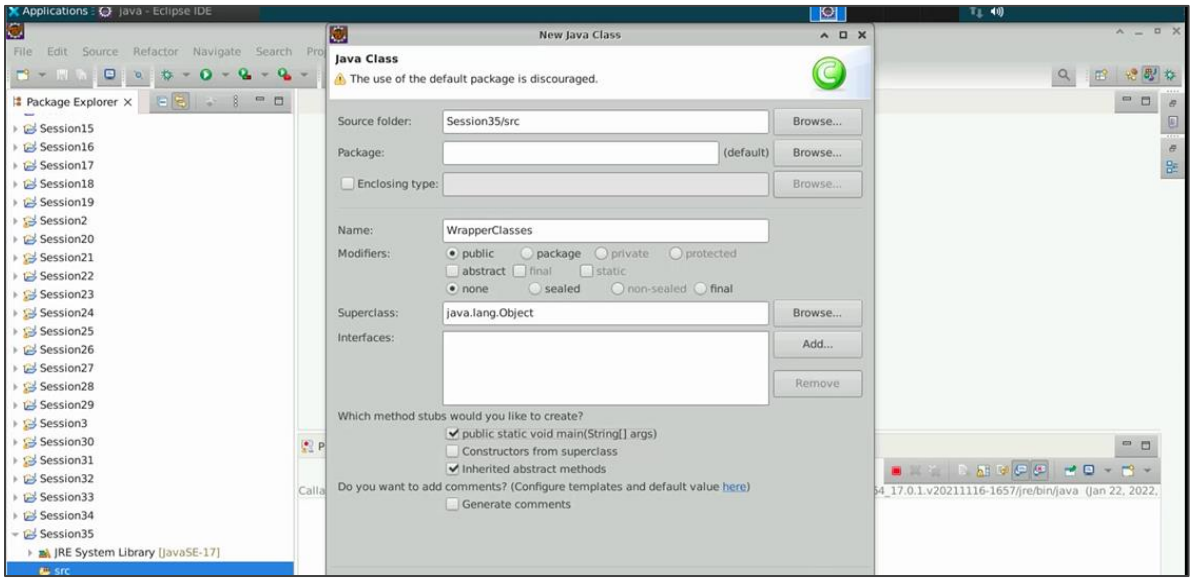




1.3 Right-click on the **src** folder, click on **New**, and then select **Class**



1.4 Name the class **WrapperClasses** and make sure it contains the main method



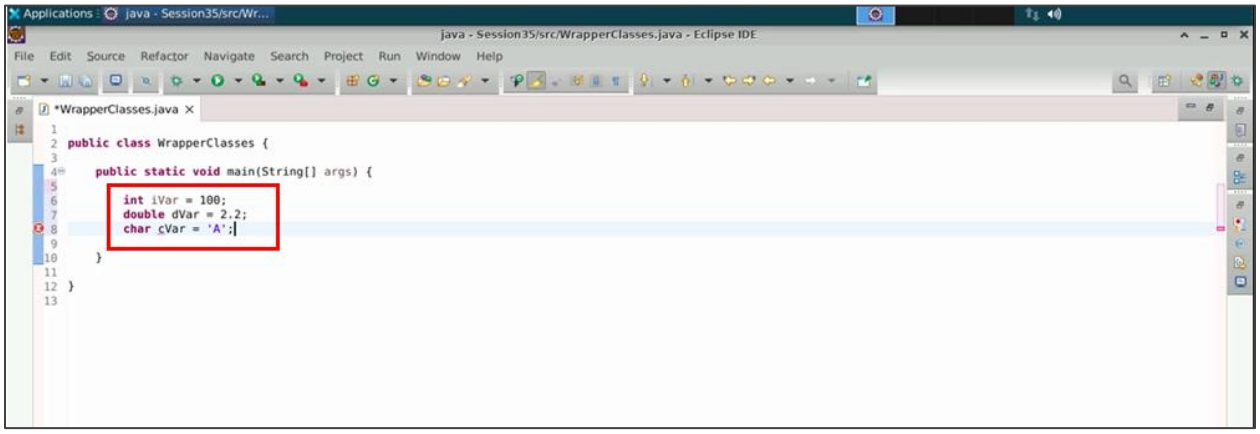
Step 2: Create primitive variables

2.1 Inside the **WrapperClasses** class, declare the following primitive variables:

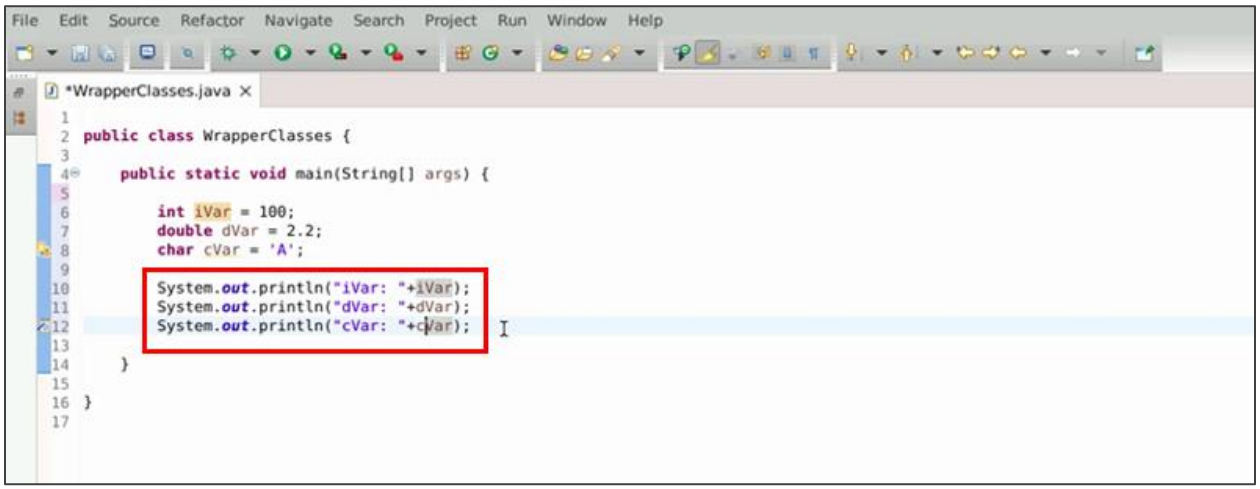
int iVar = 100;

double dVar = 2.2;

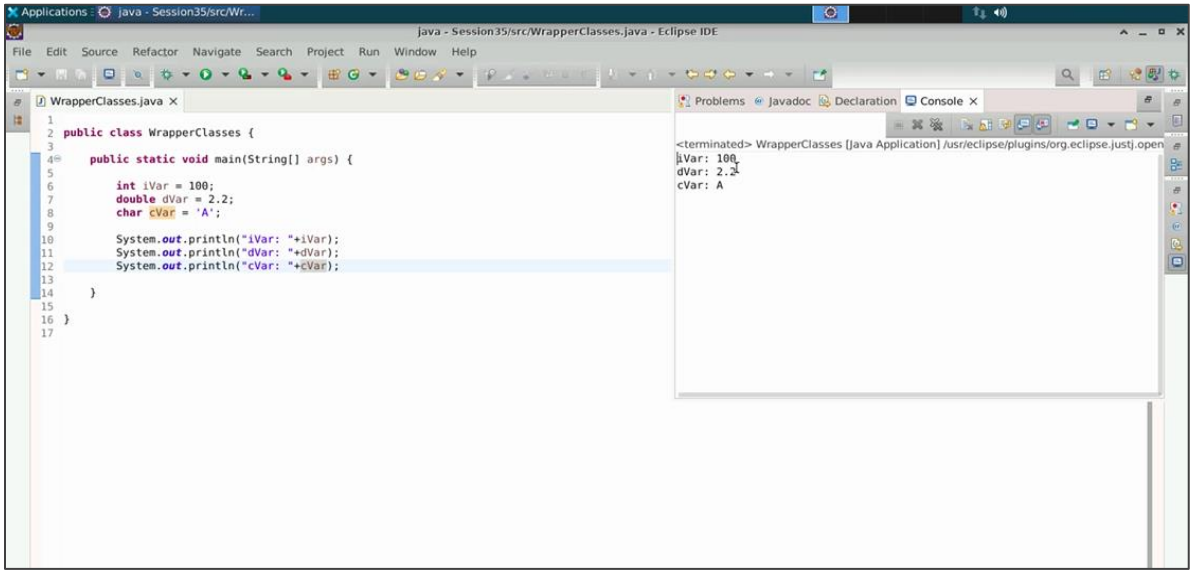
char cVar = 'A';



2.2 Use the **System.out.println()** method to print the values of the primitive variables



2.3 Run the project

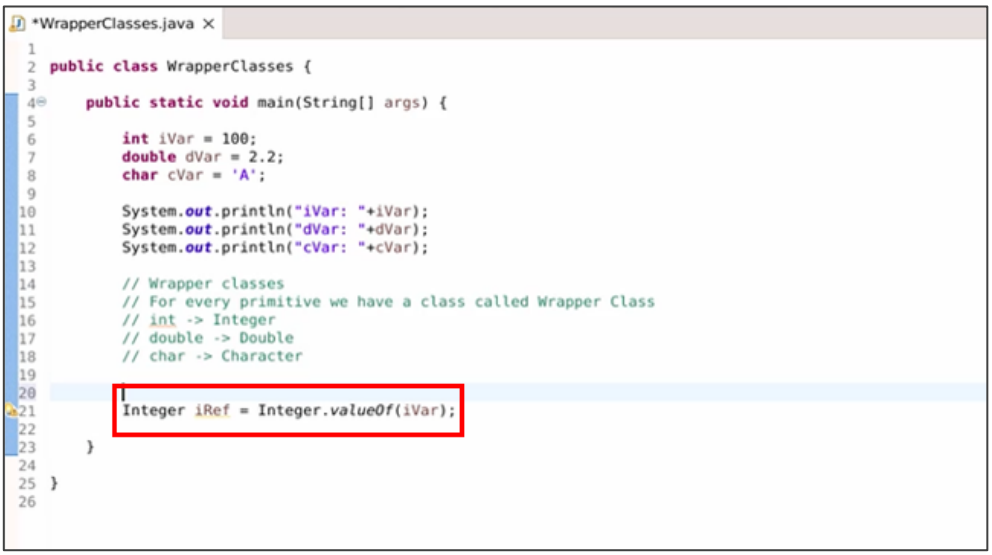


You will see the values in the output as **100**, **2.2**, and **A**.

Step 3: Add methods on the wrapper classes

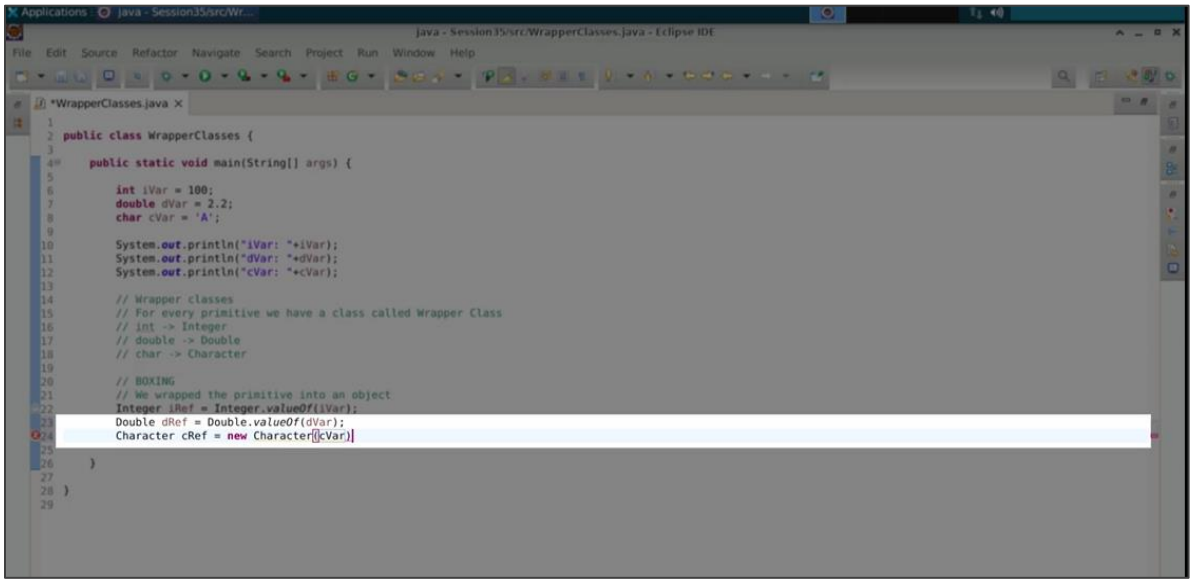
3.1 Create an Integer object to wrap the int primitive using the **valueOf()** method:

Integer iRef = Integer.valueOf(iVar)

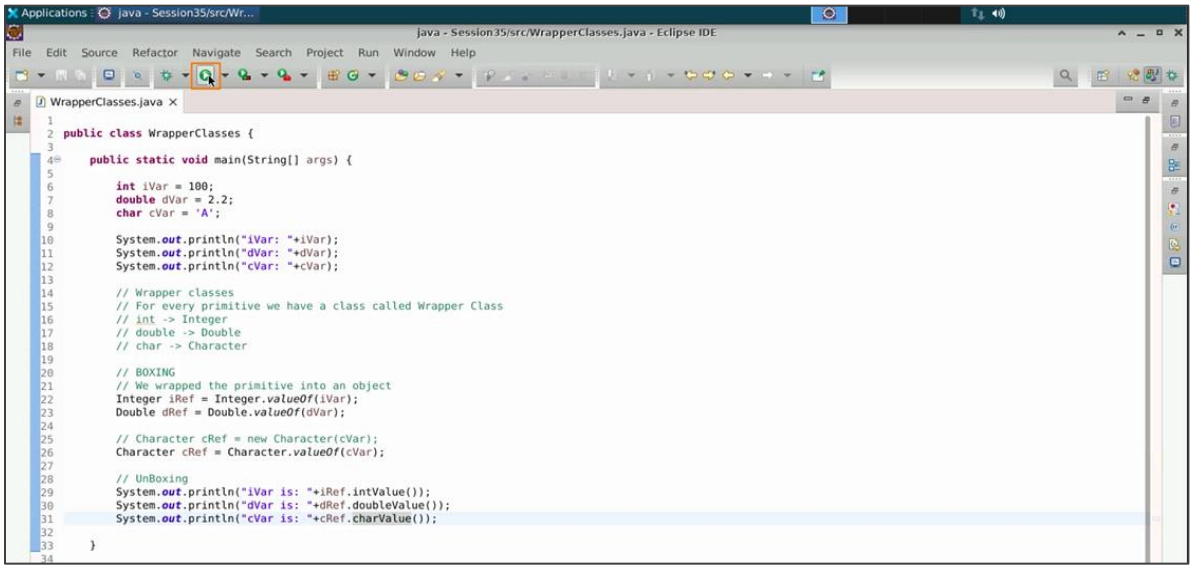


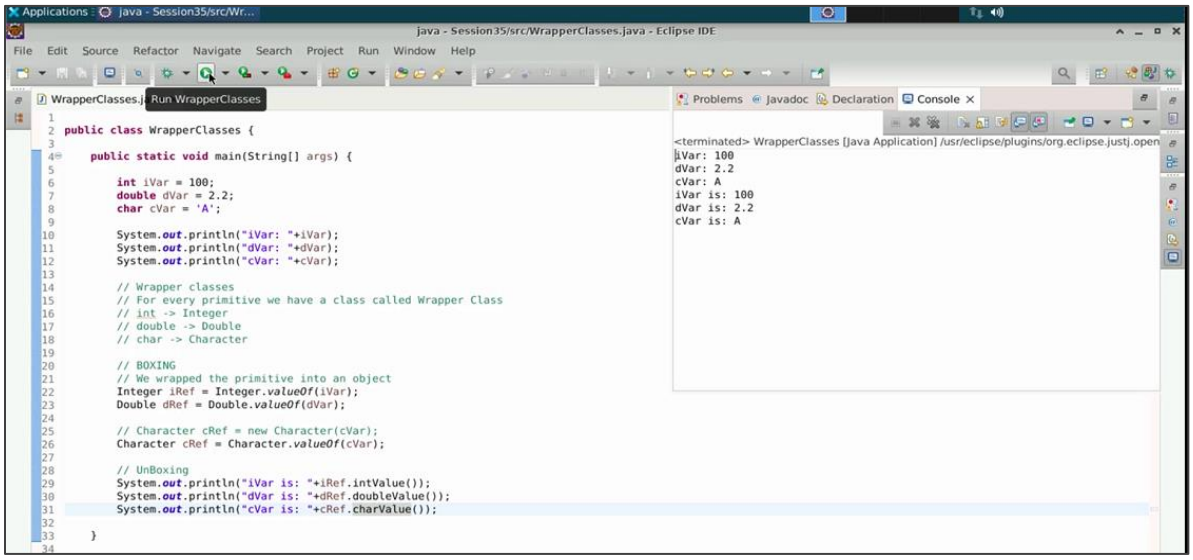
This process of wrapping a primitive value into an object is called boxing.

3.2 Repeat this step for double and char variables, creating Double and Character objects, respectively



3.3 Extract and print the primitive values from the wrapper objects using methods like **intValue()**, **doubleValue(),** and **charValue()**. Run the project by clicking on green play button.

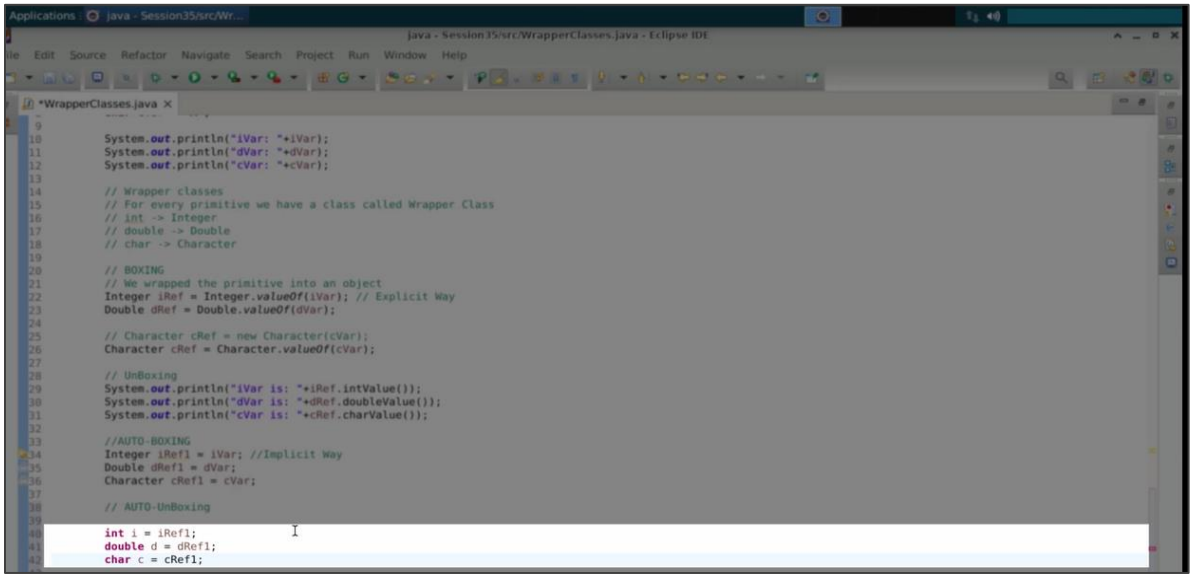





```
public class WrapperClasses {

    public static void main(String[] args) {

        int iVar = 100;
        double dVar = 2.2;
        char cVar = 'A';

        System.out.println("iVar: "+iVar);
        System.out.println("dVar: "+dVar);
        System.out.println("cVar: "+cVar);

        // Wrapper classes
        // For every primitive we have a class called Wrapper Class
        // int -> Integer
        // double -> Double
        // char -> Character

        // BOXING
        // We wrapped the primitive into an object
        Integer iRef = Integer.valueOf(iVar);
        Double dRef = Double.valueOf(dVar);

        // Character cRef = new Character(cVar);
        Character cRef = Character.valueOf(cVar);

        // UnBoxing
        System.out.println("iVar is: "+iRef.intValue());
        System.out.println("dVar is: "+dRef.doubleValue());
        System.out.println("cVar is: "+cRef.charValue());

    }
```

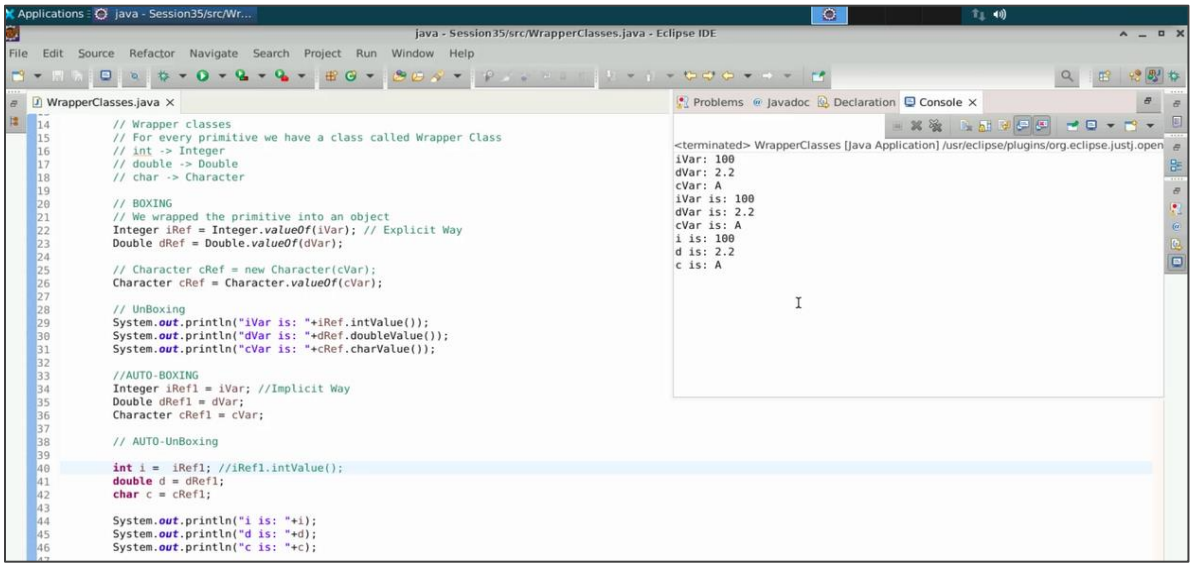
You can see the same values are printed twice.

3.4 Create primitive variables and assign the values from the wrapper class objects directly without using value methods




```
        System.out.println("iVar: "+iVar);
        System.out.println("dVar: "+dVar);
        System.out.println("cVar: "+cVar);

        // Wrapper classes
        // For every primitive we have a class called Wrapper Class
        // int -> Integer
        // double -> Double
        // char -> Character

        // BOXING
        // We wrapped the primitive into an object
        Integer iRef = Integer.valueOf(iVar); // Explicit Way
        Double dRef = Double.valueOf(dVar);

        // Character cRef = new Character(cVar);
        Character cRef = Character.valueOf(cVar);

        // UnBoxing
        System.out.println("iVar is: "+iRef.intValue());
        System.out.println("dVar is: "+dRef.doubleValue());
        System.out.println("cVar is: "+cRef.charValue());

        //AUTO-BOXING
        Integer iRef1 = iVar; //Implicit Way
        Double dRef1 = dVar;
        Character cRef1 = cVar;

        // AUTO-UnBoxing

        int i = iRef1;
        double d = dRef1;
        char c = cRef1;
```

3.5 Run the project and observe the output, which will display the values of the primitive variables as well as the extracted values from the wrapper objects



In conclusion, the demo illustrates the usage of Java wrapper classes to represent primitive data types as references, providing additional functionality and flexibility when working with objects in Java programs.