Lesson 04 Demo 08

Implementing throw and throws in a Banking Application

Objective: To implement throw and throws in a banking application in Java

Tools required: Eclipse IDE

Prerequisites: None

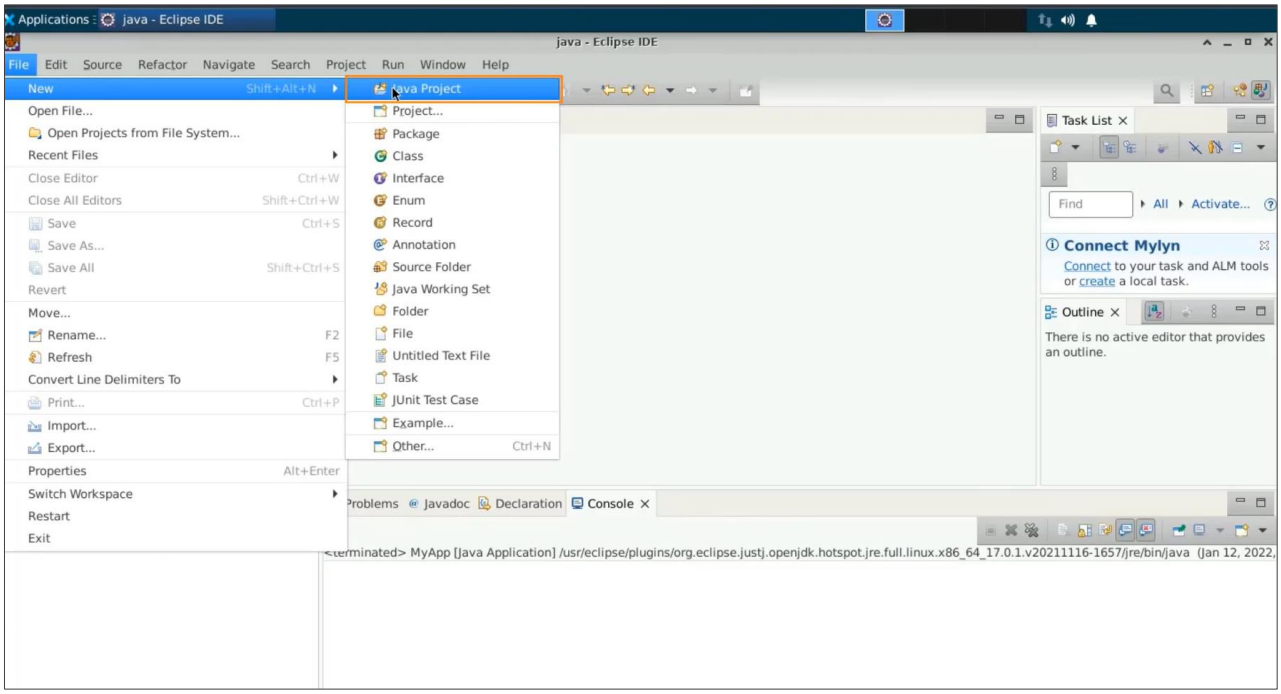
Steps to be followed:

1. Open IDE and create a new project
2. Create a class with an executable method
3. Use a reference variable with the default constructor
4. Execute the code with example data
5. Add the code in the try catch

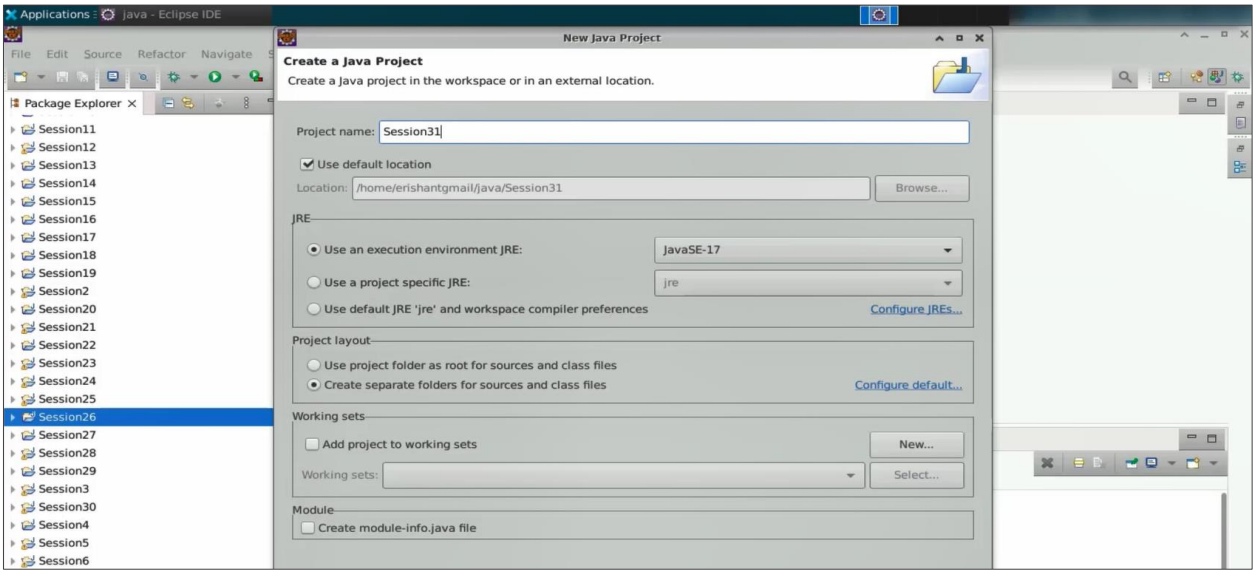
Step 1: Open IDE and create a new project

1.1 Open the **Eclipse IDE**

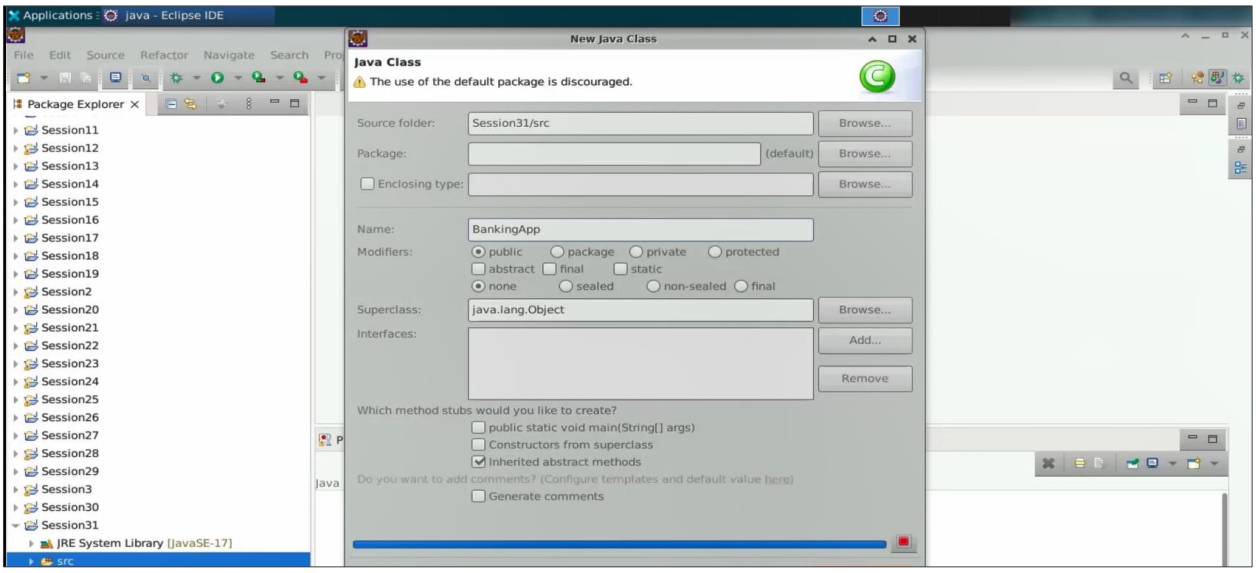
1.2 Select **File**, then **New,** and then **Java project**



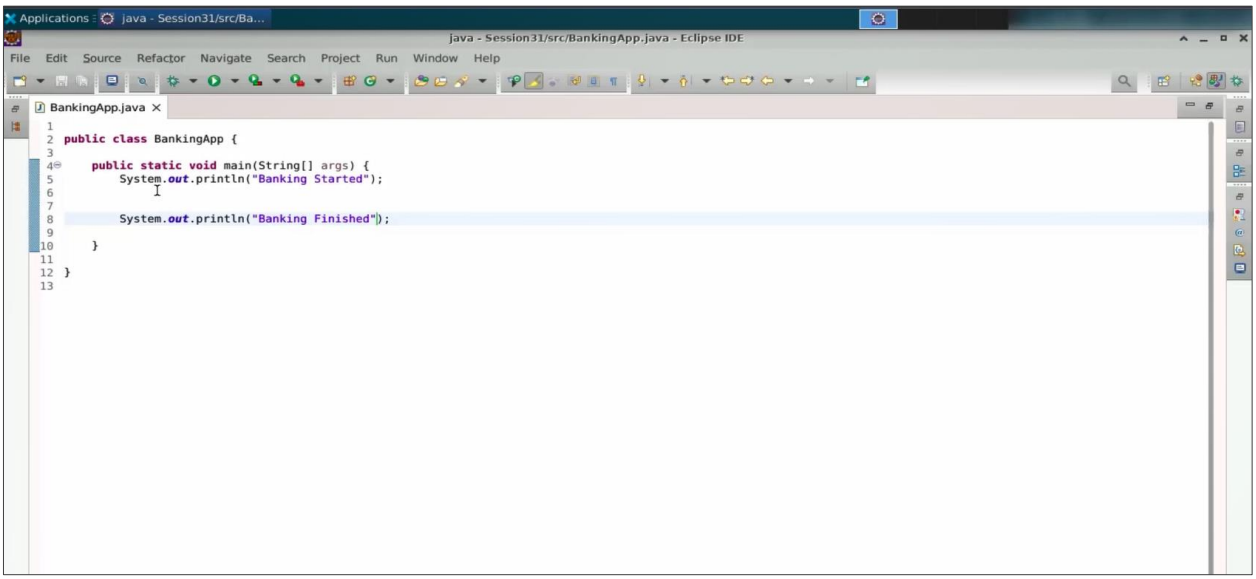
1.3 Name the project **“Session31”,** uncheck **“Create a module info.Java file”**, and press **Finish**



1.4 With Session31 in the src, do a right-click and create a new class. Name this class as BankingApp, then select the main method, and then select finish.

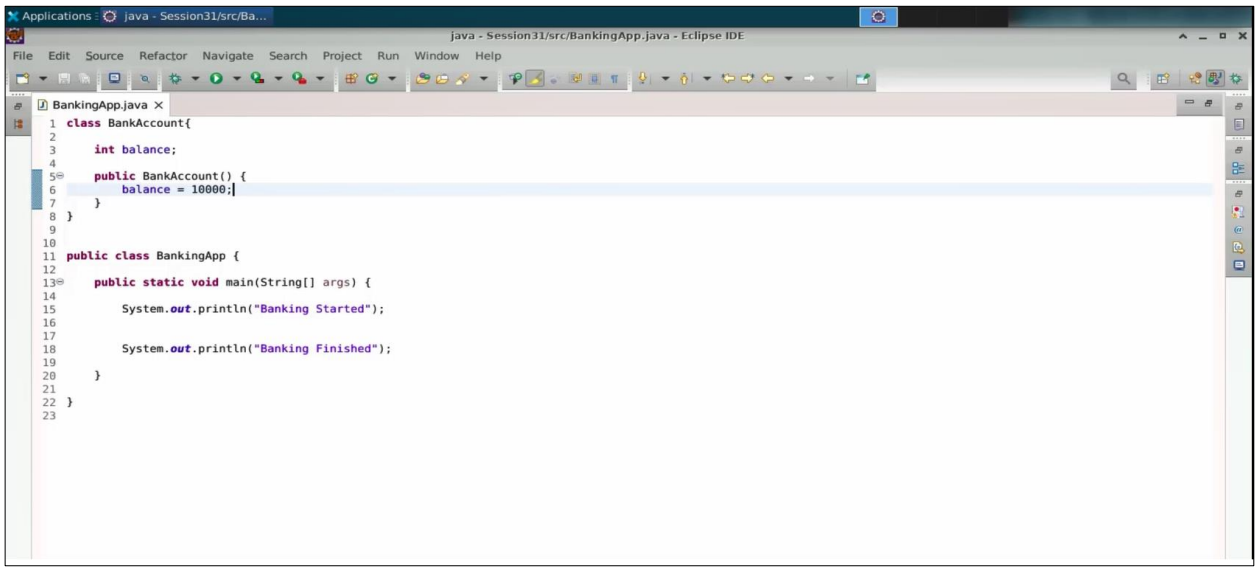


1.5 In the banking app, let the first statement be as banking started. And the last statement goes as banking finished. Thus, here you are with two lines of code with the main method.

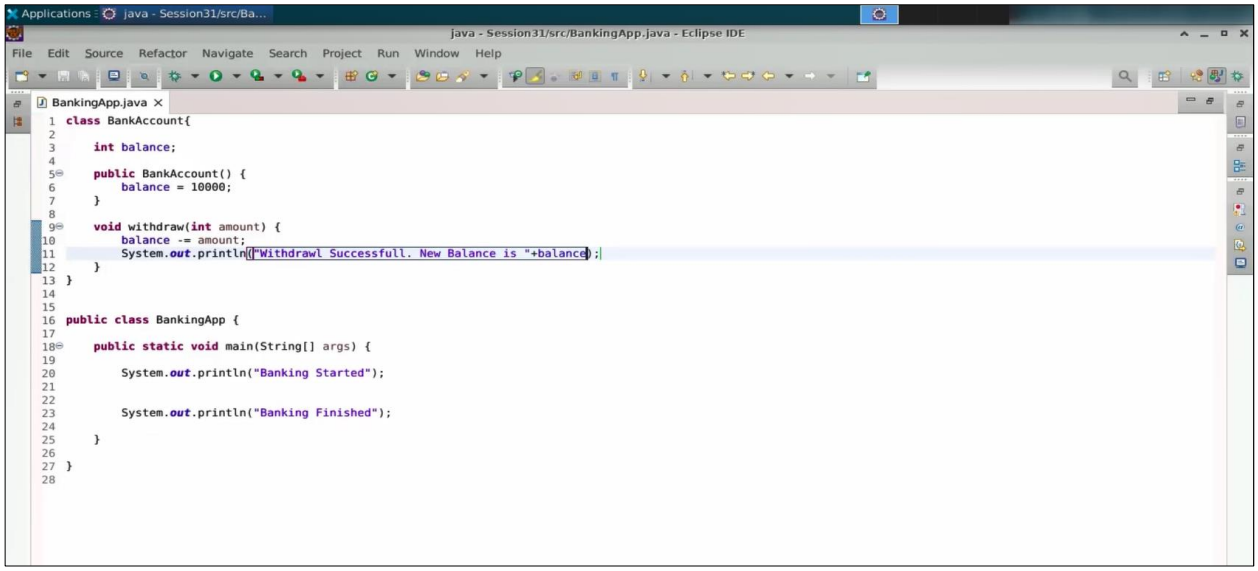


Step 2: Create a class with an executable method

2.1 Come here and create a class called bank account. For the bank account, you have the balance. In the bank account, the moment you create the object of the bank account, you will give a minimum balance of 10,000.

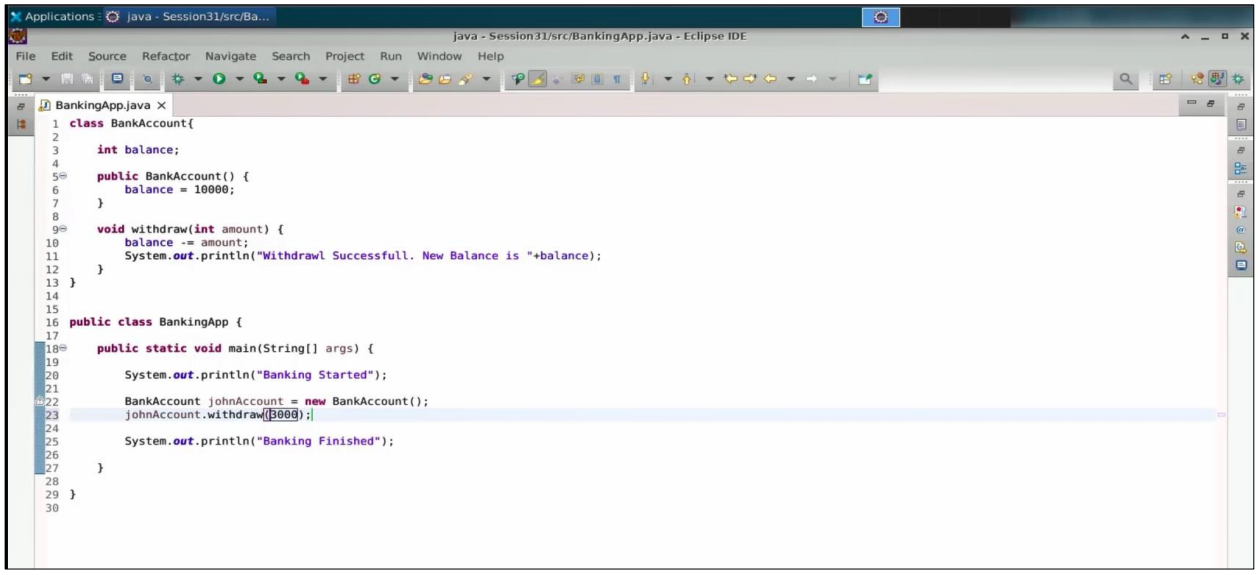


2.2 For the bank account, you can implement a method called withdraw that accepts an amount as a parameter. When a withdrawal is made, this method will update the balance by subtracting the specified amount. After updating the balance, it will indicate that the withdrawal was successful and display the new balance.

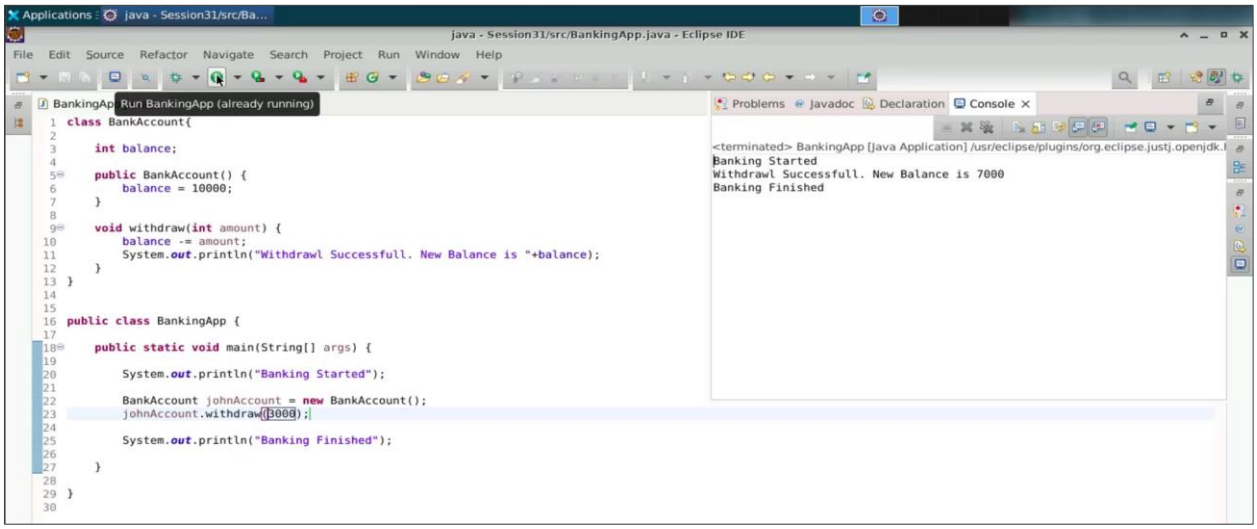


Step 3: Use a reference variable with the default constructor

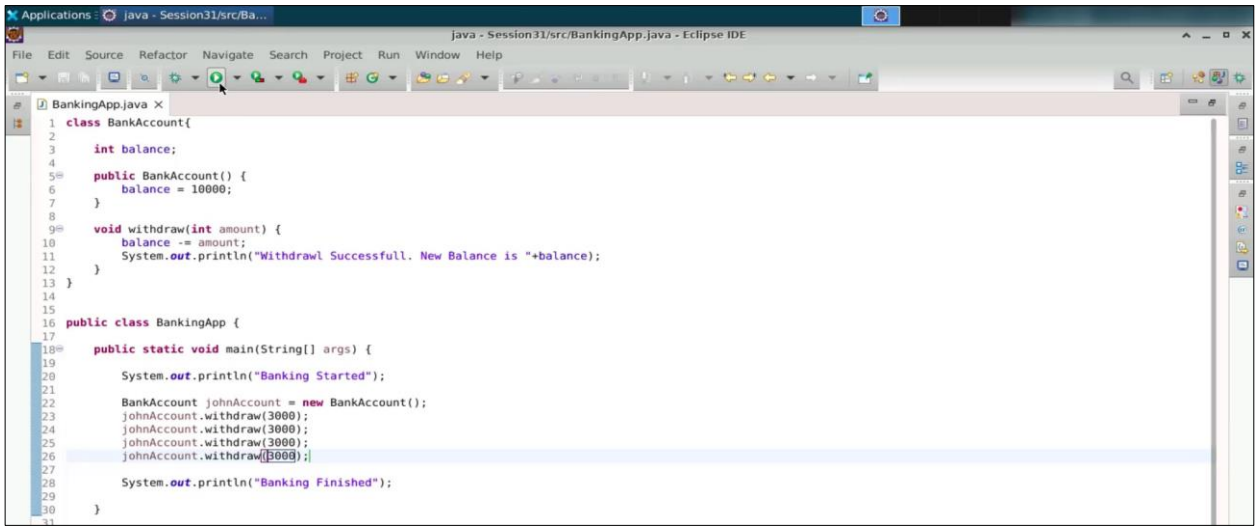
3.1 Now, let us open a bank account for John. We will create a reference variable called johnAccount as a new instance of the bank account. With the default constructor, the account balance will be set to 10,000. On johnAccount, we will then execute the withdraw function to withdraw 3,000.



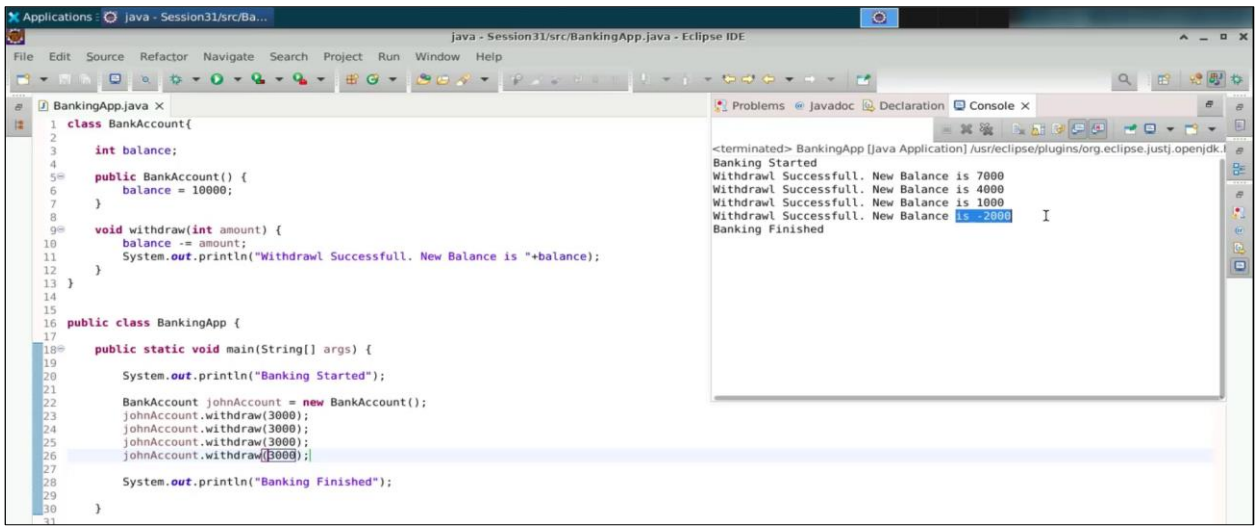
3.2 Let us run the code and see what happens. The banking process has started, the withdrawal is successful, and the new balance is 7,000.



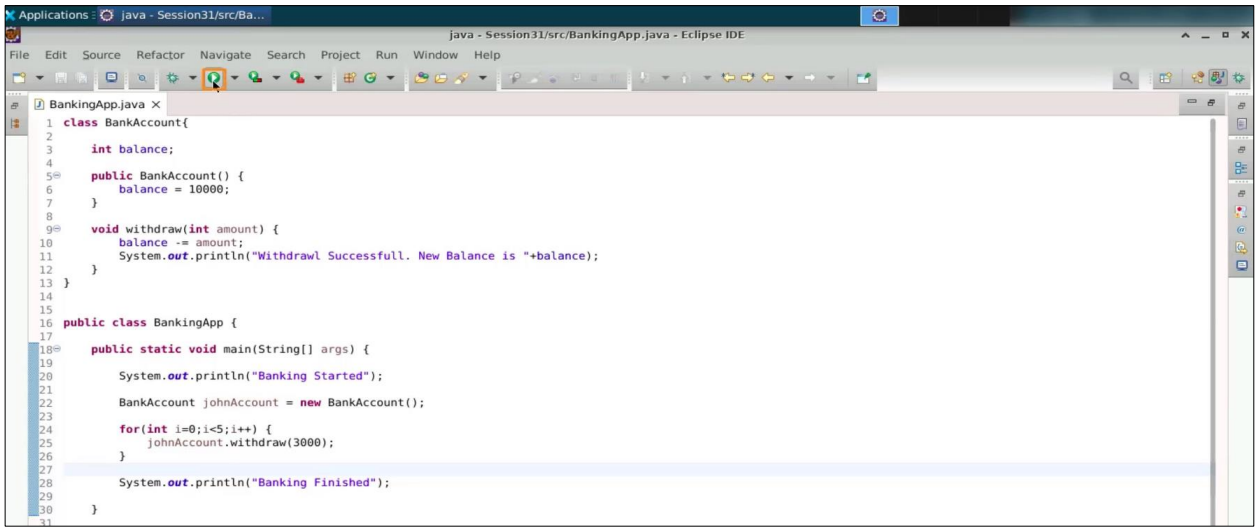
3.3 What if John tries to make multiple withdrawals? Write johnAccount.withdraw(3000) here. Then, John would like to do another withdrawal of 3000, and one more withdrawal of 3000.



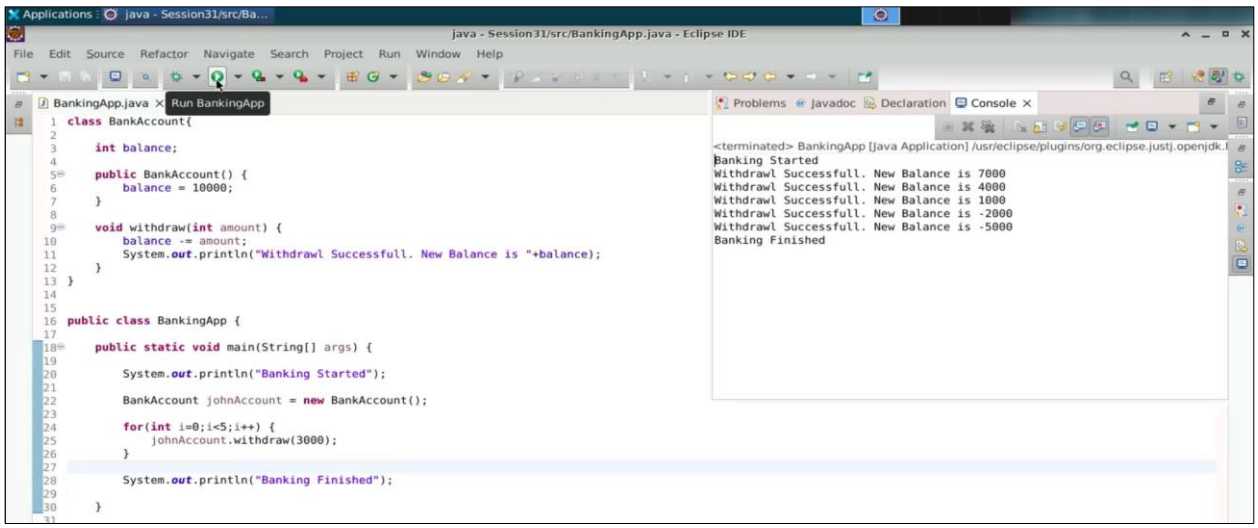
3.4 Now, when you run the code, it shows that the balance has gone negative. This is not an ideal use case for the banking withdrawal process. Therefore, you need to ensure that certain conditions are checked in the withdraw function so that a suitable message can be given to the user.



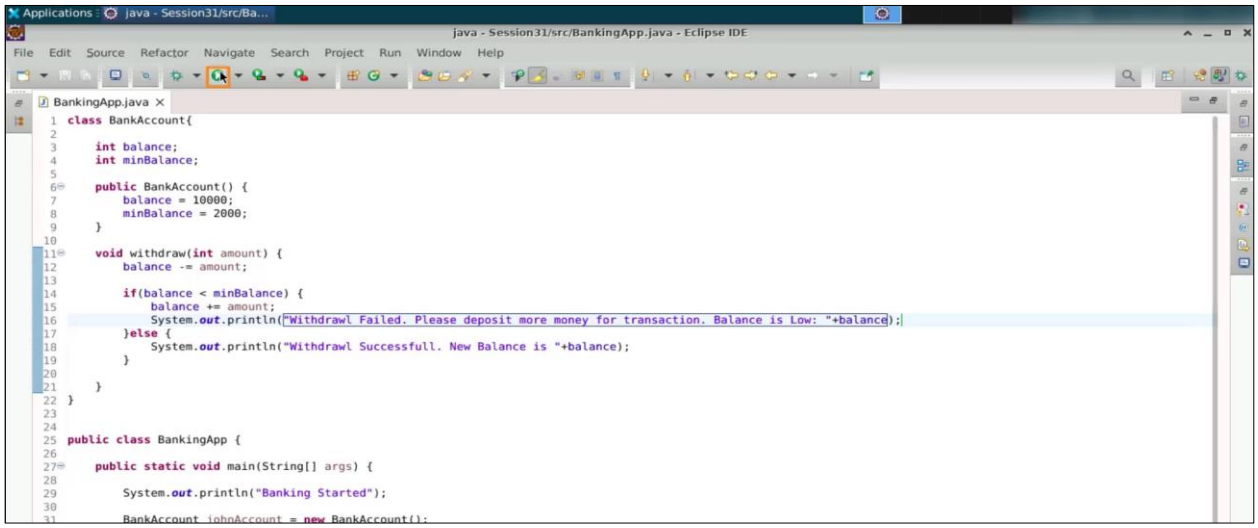
3.5 Making it simpler, you have this for loop. And these are the attempts which John is trying to make. John is trying to make certain attempts known as five different attempts for the withdrawal part.



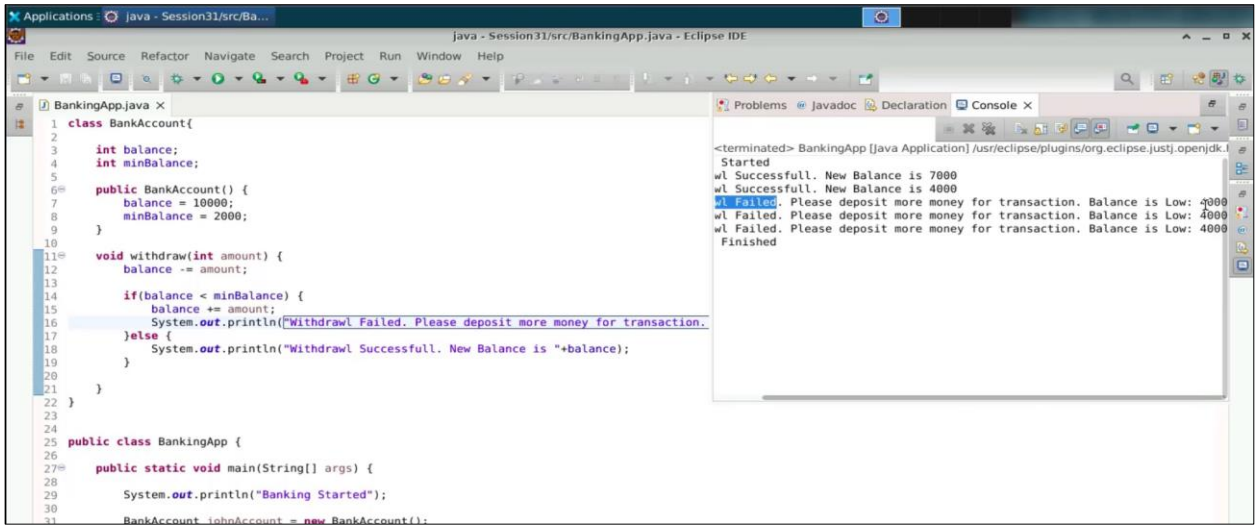
3.6 Run the code and this goes minus 5000. This is not a doable or this should not be supposed to be there.



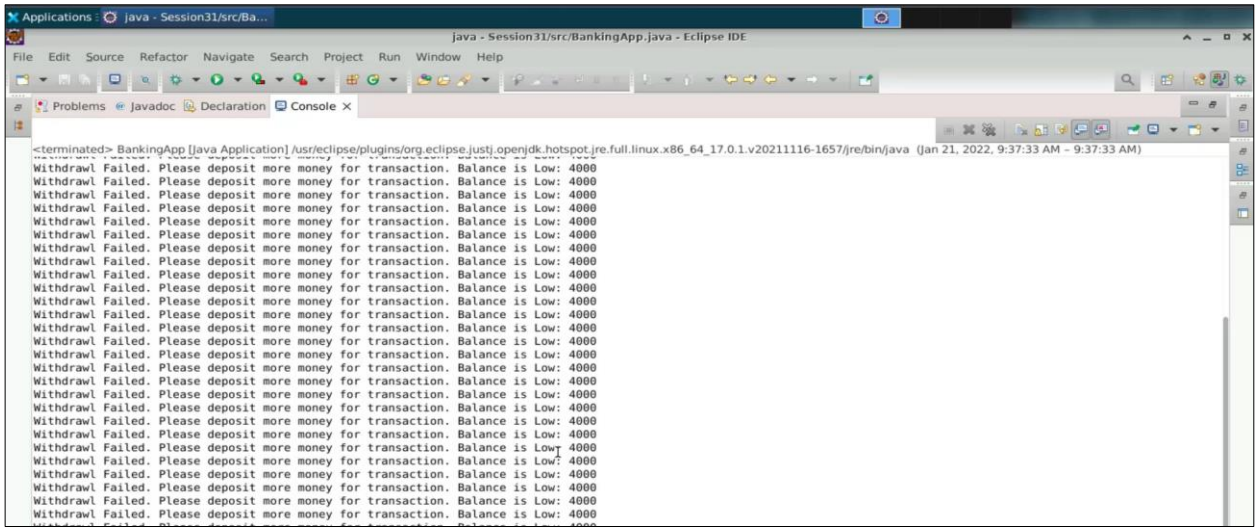
3.7 Add a check to ensure the balance doesn't go negative. Alternatively, set a minimum balance of 2000. Ensure that users cannot withdraw funds if it would result in a balance below this minimum. If a withdrawal causes the balance to drop below the minimum, roll back the transaction, and display a message: "Withdrawal failed, please deposit more money. Balance is low," followed by the current balance.



3.8 Let us run the code again. As you can see, the withdrawal failed and the balance is low, currently at 4000. Subtracting 3000 from 4000 would result in a balance of 1000, which is below the minimum balance requirement. This check prevents the user from completing the transaction.

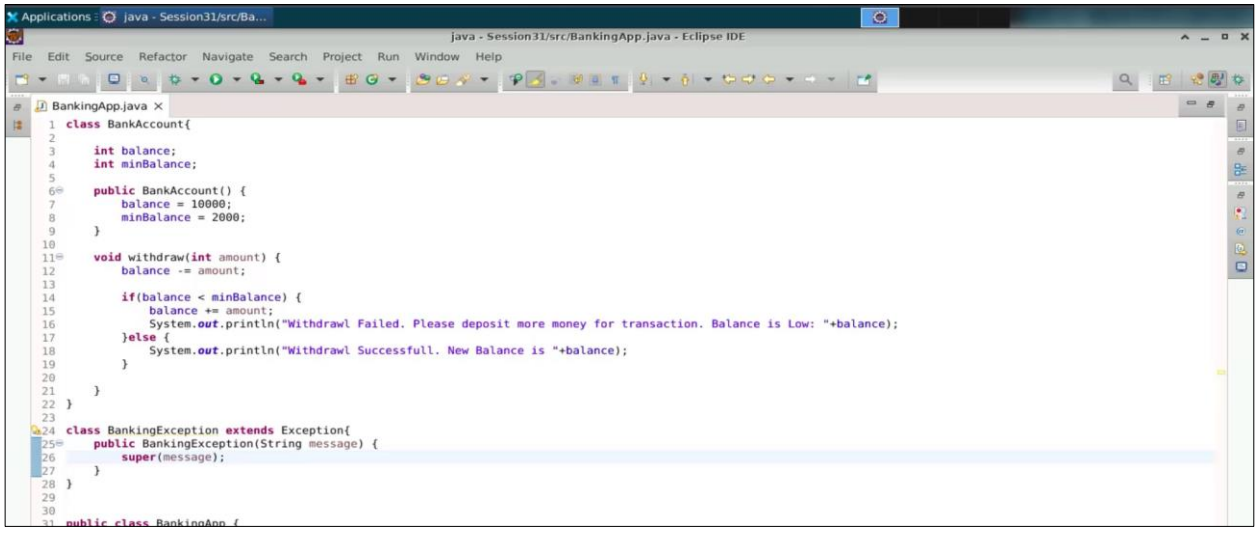


3.9 Consider John wants to try this 50 times. This means the loop will run 50 times, but due to the balance check, each attempt will result in the same message: "Your balance is low." The system will not allow the transaction to proceed.

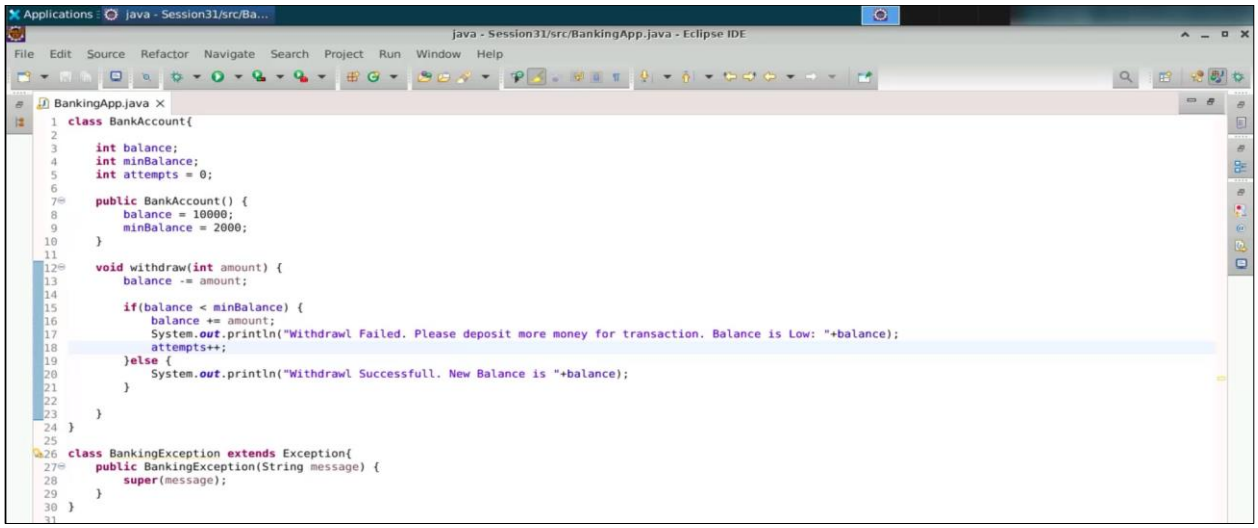


Step 4: Execute the code with example data

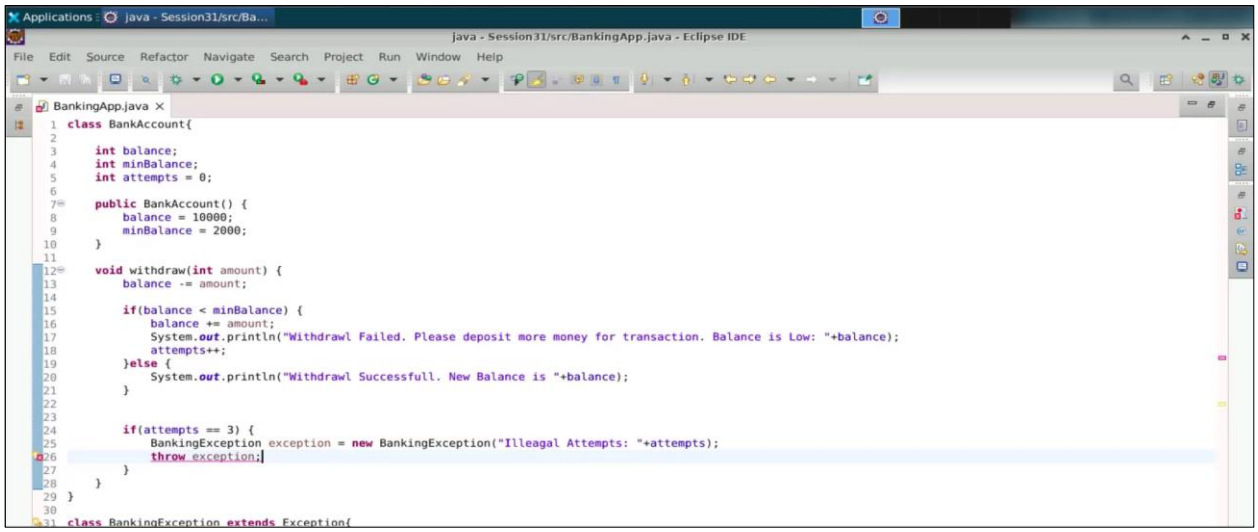
4.1 Let us say your bank server or other resources are being used unnecessarily. As a developer, you should throw an error and crash the program in such cases. Do this, implement a class called BankingException that extends Exception to create a checked exception. Then, take a message as input and pass it to the parent class using super(message).



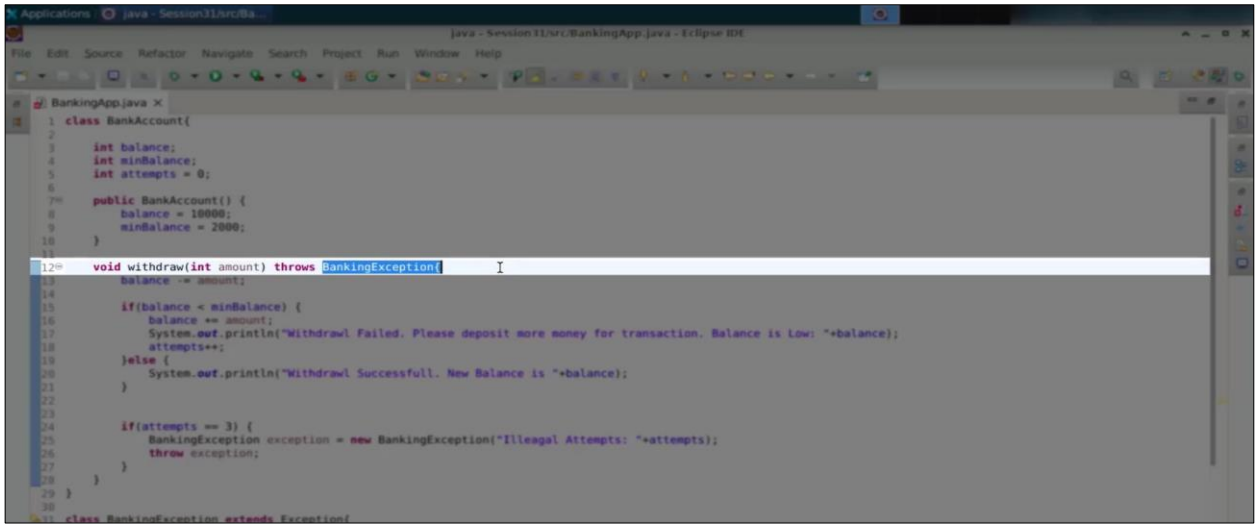
4.2 Create a variable called attempts initialized to zero. If a negative attempt is made, increment the value of attempts by one. This means if you try to withdraw beyond the threshold, you will increase the value of attempts.



4.3 In the withdraw function, if your attempts reach three, create a `BankingException` object. You can write `BankingException exception = new BankingException("Illegal attempts");`. Then, throw the exception. This will trigger the compiler to act.

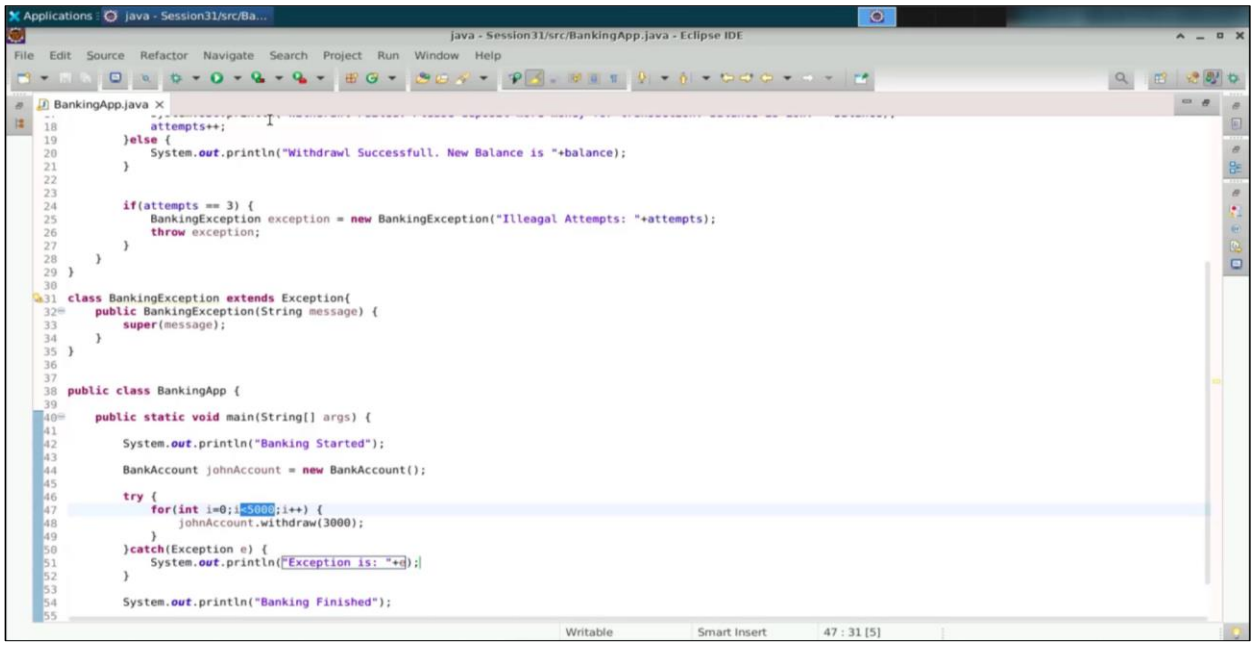


4.4 The compiler knows you are throwing a checked exception, so it enforces that you add a `throws` keyword to you withdraw method signature. This indicates that the method can throw a `BankingException` when executed.

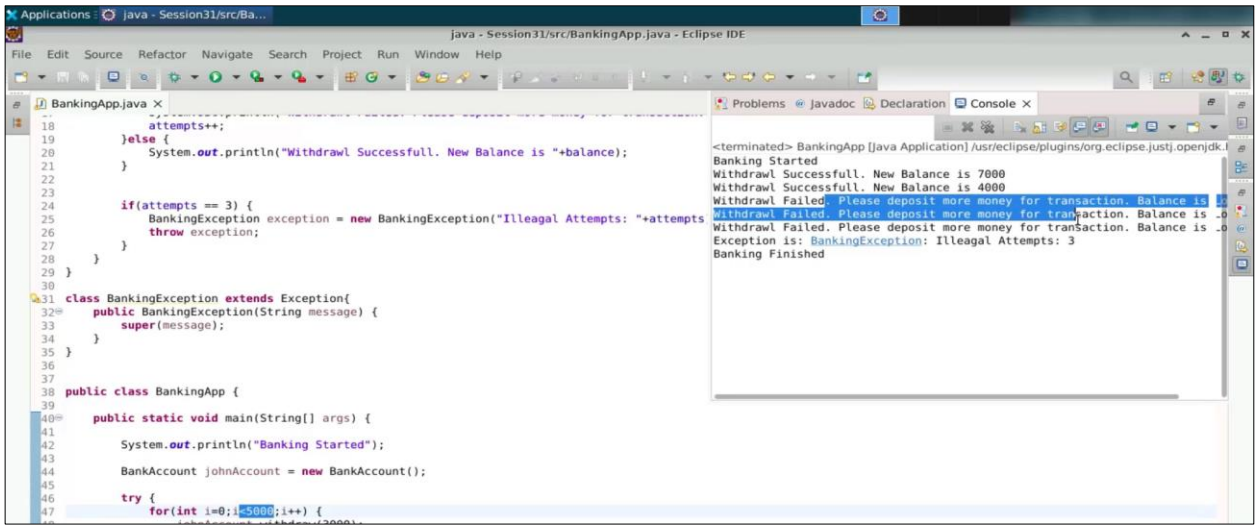


Step 5: Add the code in the try catch

5.1 Let's add this code in a try-catch block. For checked exceptions, the compiler enforces the use of exception handling techniques. You can use a generic exception `e`, meaning any kind of exception will be managed by the parent class. Hence, write `Exception: " + e` to display the exception message.



5.2 If John tries 5000 times, after three illegal attempts, an exception is thrown with the message "illegal attempts," ending that part of the execution. The banking process finishes, meaning the program did not just crash, but the crash was handled, and the program terminated normally.



By using the above steps, you have successfully implemented `throw` and `throws` in your banking application in Java, ensuring robust error handling and proper method declarations for managing exceptional conditions.