

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:

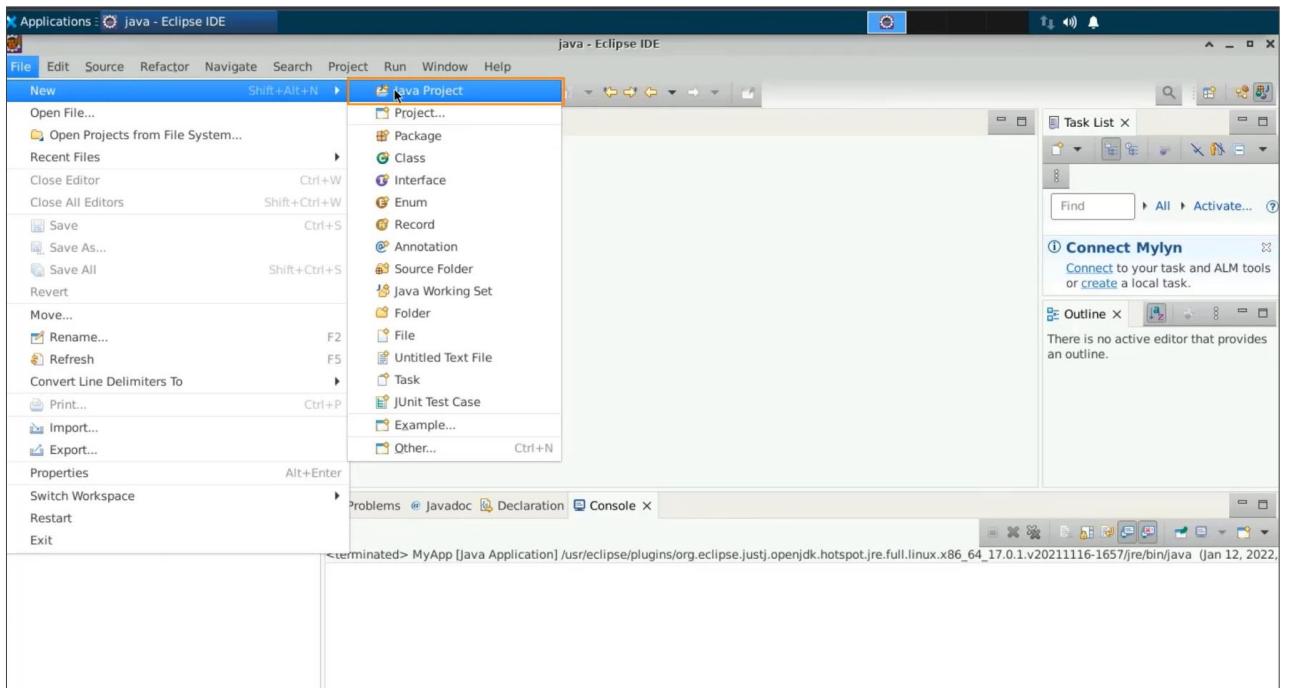
1. Open IDE and create a new project
2. Create a class with an executable method
3. Use a reference variable with the default constructor
4. Execute the code with example data
5. Add the code in the try catch

Step 1: Open IDE and create a new project

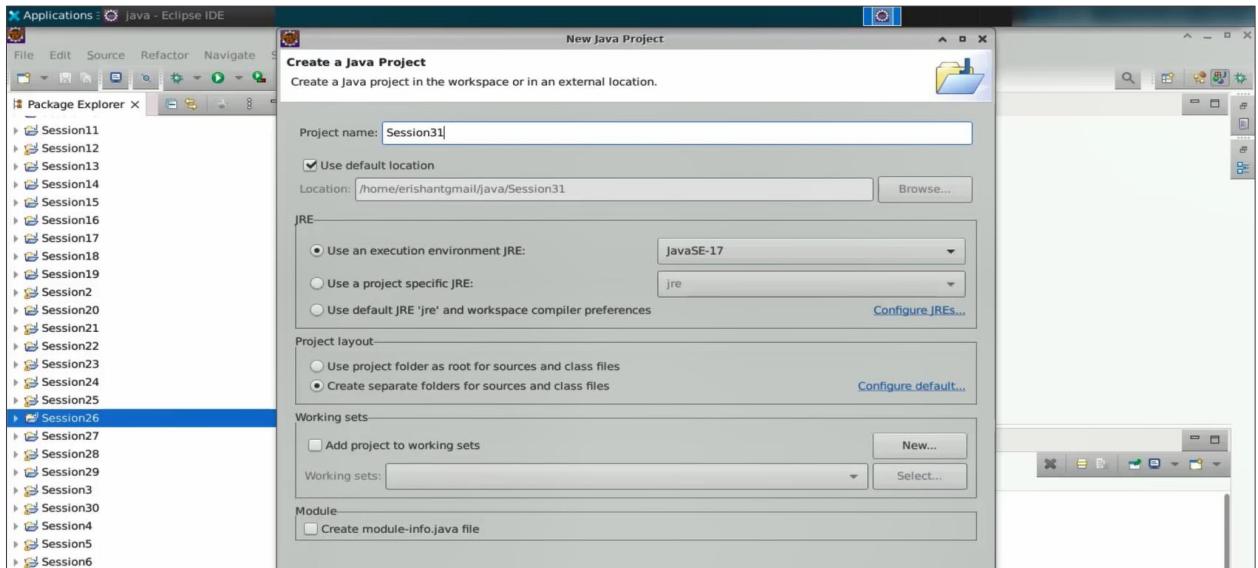
1.1 Open the Eclipse IDE



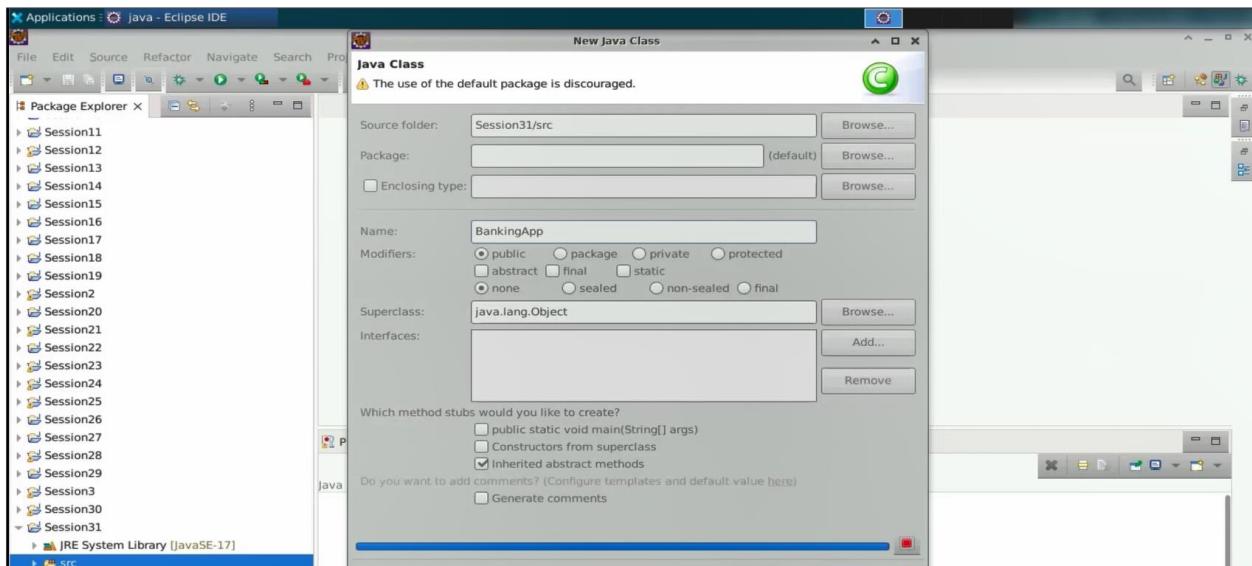
1.2 Select File, then New, and then Java project



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.

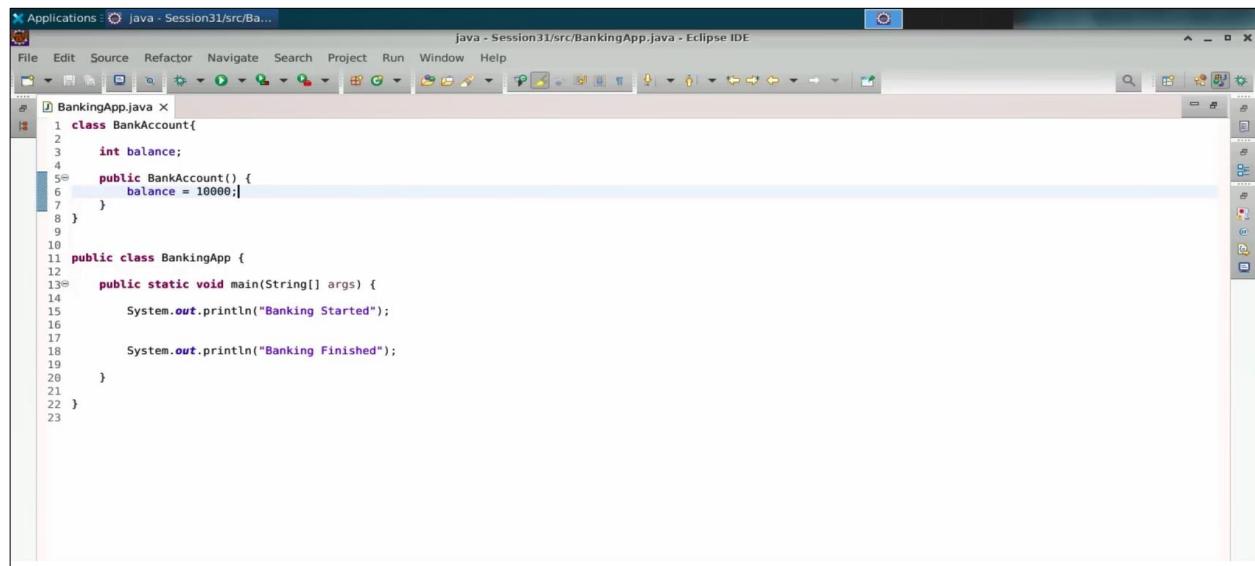
```

1  public class BankingApp {
2
3     public static void main(String[] args) {
4         System.out.println("Banking Started");
5         System.out.println("Banking Finished");
6     }
7
8 }
9
10
11
12
13

```

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.

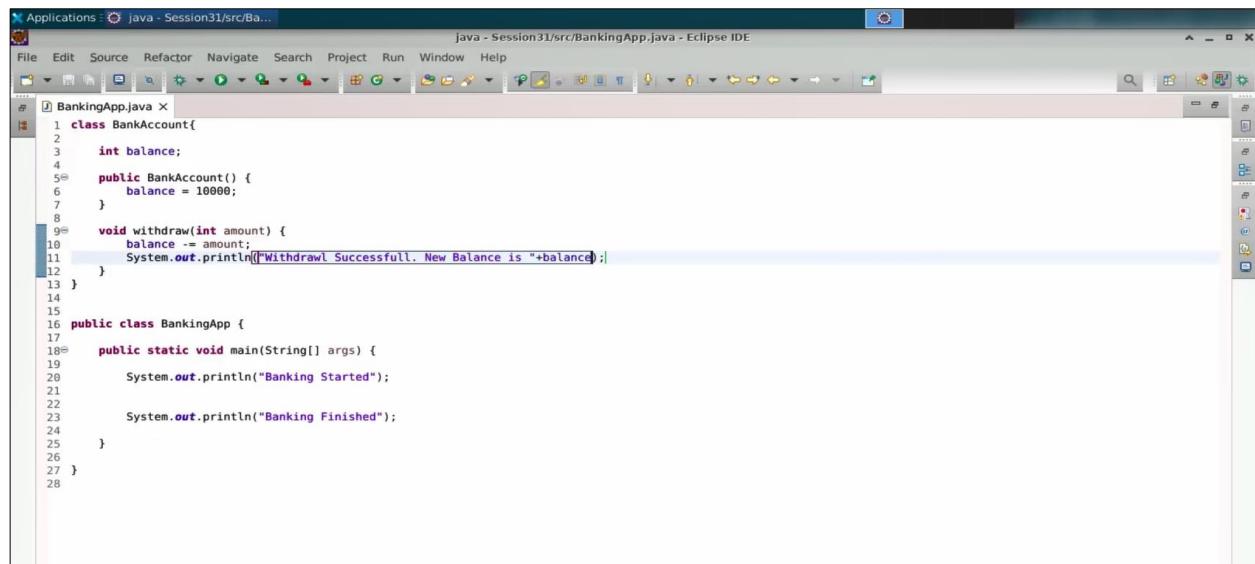


```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1 class BankAccount{
2
3     int balance;
4
5     public BankAccount() {
6         balance = 10000;
7     }
8 }
9
10 public class BankingApp {
11
12     public static void main(String[] args) {
13
14         System.out.println("Banking Started");
15
16
17         System.out.println("Banking Finished");
18     }
19 }
20
21
22 }
23
24
25
26
27
28

```

- 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.



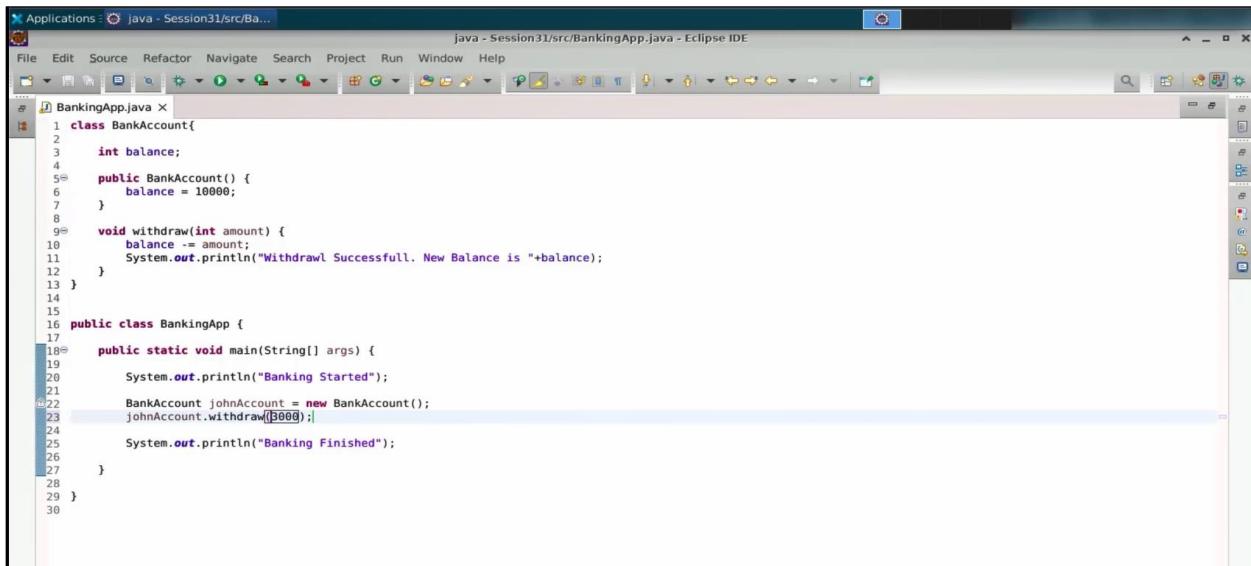
```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1 class BankAccount{
2
3     int balance;
4
5     public BankAccount() {
6         balance = 10000;
7     }
8
9     void withdraw(int amount) {
10         balance -= amount;
11         System.out.println(["Withdraw Successfull. New Balance is "+balance]);
12     }
13 }
14
15
16 public class BankingApp {
17
18     public static void main(String[] args) {
19
20         System.out.println("Banking Started");
21
22
23         System.out.println("Banking Finished");
24     }
25 }
26
27
28

```

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.

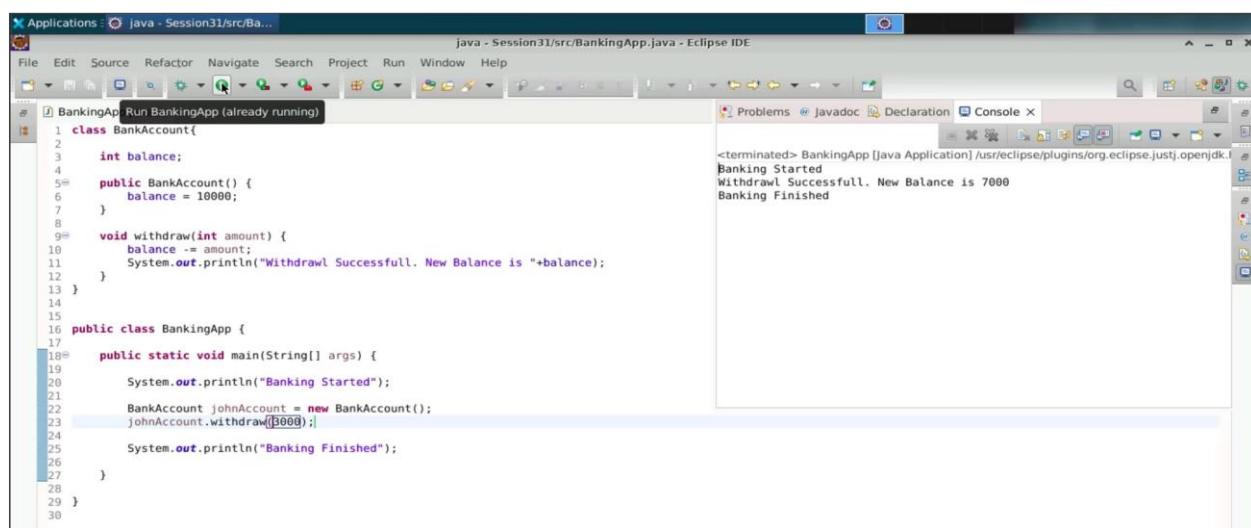


```

1 Applications : Java - Session31/src/Ba...
2 java - Session31/src/BankingApp.java - Eclipse IDE
3
4 File Edit Source Refactor Navigate Search Project Run Window Help
5
6 BankingApp.java X
7
8 1 class BankAccount{
9 2
10 3     int balance;
11 4
12 5     public BankAccount() {
13 6         balance = 10000;
14 7     }
15 8
16 9     void withdraw(int amount) {
17 10         balance -= amount;
18 11         System.out.println("Withdraw Successfull. New Balance is "+balance);
19 12     }
20 13 }
21 14
22 15 public class BankingApp {
23 16
24 17     public static void main(String[] args) {
25 18         System.out.println("Banking Started");
26 19         BankAccount johnAccount = new BankAccount();
27 20         johnAccount.withdraw(3000);
28 21         System.out.println("Banking Finished");
29 22     }
30 23 }
31 24
32 25
33 26
34 27
35 28
36 29
37 30

```

- 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.



```

1 Applications : Java - Session31/src/Ba...
2 java - Session31/src/BankingApp.java - Eclipse IDE
3
4 File Edit Source Refactor Navigate Search Project Run Window Help
5
6 BankingApp Run BankingApp (already running)
7
8 1 class BankAccount{
9 2
10 3     int balance;
11 4
12 5     public BankAccount() {
13 6         balance = 10000;
14 7     }
15 8
16 9     void withdraw(int amount) {
17 10         balance -= amount;
18 11         System.out.println("Withdraw Successfull. New Balance is "+balance);
19 12     }
20 13 }
21 14
22 15 public class BankingApp {
23 16
24 17     public static void main(String[] args) {
25 18         System.out.println("Banking Started");
26 19         BankAccount johnAccount = new BankAccount();
27 20         johnAccount.withdraw(3000);
28 21         System.out.println("Banking Finished");
29 22     }
30 23 }
31 24
32 25
33 26
34 27
35 28
36 29
37 30

```

Console Output:

```

<terminated> BankingApp [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.l
Banking Started
Withdraw Successfull. New Balance is 7000
Banking Finished

```

- 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.

```

Applications : Java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1 class BankAccount{
2
3     int balance;
4
5     public BankAccount() {
6         balance = 10000;
7     }
8
9     void withdraw(int amount) {
10        balance -= amount;
11        System.out.println("Withdraw Successfull. New Balance is "+balance);
12    }
13 }
14
15 public class BankingApp {
16
17     public static void main(String[] args) {
18
19         System.out.println("Banking Started");
20
21         BankAccount johnAccount = new BankAccount();
22         johnAccount.withdraw(3000);
23         johnAccount.withdraw(3000);
24         johnAccount.withdraw(3000);
25         johnAccount.withdraw(3000);
26         johnAccount.withdraw(3000);
27
28         System.out.println("Banking Finished");
29
30     }
31 }
```

- 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.

```

Applications : Java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1 class BankAccount{
2
3     int balance;
4
5     public BankAccount() {
6         balance = 10000;
7     }
8
9     void withdraw(int amount) {
10        balance -= amount;
11        System.out.println("Withdraw Successfull. New Balance is "+balance);
12    }
13 }
14
15 public class BankingApp {
16
17     public static void main(String[] args) {
18
19         System.out.println("Banking Started");
20
21         BankAccount johnAccount = new BankAccount();
22         johnAccount.withdraw(3000);
23         johnAccount.withdraw(3000);
24         johnAccount.withdraw(3000);
25         johnAccount.withdraw(3000);
26         johnAccount.withdraw(3000);
27
28         System.out.println("Banking Finished");
29
30     }
31 }
```

Console Output:

```

<terminated> BankingApp [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.l
Banking Started
Withdraw Successfull. New Balance is 7000
Withdraw Successfull. New Balance is 4000
Withdraw Successfull. New Balance is 1000
Withdraw Successfull. New Balance is -2000
Banking Finished
```

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.

```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1  class BankAccount{
2
3      int balance;
4
5      public BankAccount() {
6          balance = 10000;
7      }
8
9      void withdraw(int amount) {
10         balance -= amount;
11         System.out.println("Withdraw Successfull. New Balance is "+balance);
12     }
13 }
14
15
16 public class BankingApp {
17
18     public static void main(String[] args) {
19
20         System.out.println("Banking Started");
21
22         BankAccount johnAccount = new BankAccount();
23
24         for(int i=0;i<5;i++) {
25             johnAccount.withdraw(3000);
26         }
27
28         System.out.println("Banking Finished");
29     }
30 }
31

```

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

```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X Run BankingApp
1  class BankAccount{
2
3      int balance;
4
5      public BankAccount() {
6          balance = 10000;
7      }
8
9      void withdraw(int amount) {
10         balance -= amount;
11         System.out.println("Withdraw Successfull. New Balance is "+balance);
12     }
13 }
14
15
16 public class BankingApp {
17
18     public static void main(String[] args) {
19
20         System.out.println("Banking Started");
21
22         BankAccount johnAccount = new BankAccount();
23
24         for(int i=0;i<5;i++) {
25             johnAccount.withdraw(3000);
26         }
27
28         System.out.println("Banking Finished");
29     }
30 }
31

```

Console Output:

```

<terminated> BankingApp [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.l
Banking Started
Withdraw Successfull. New Balance is 7000
Withdraw Successfull. New Balance is 4000
Withdraw Successfull. New Balance is 1000
Withdraw Successfull. New Balance is -2000
Withdraw Successfull. New Balance is -5000
Banking Finished

```

- 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.

```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1  class BankAccount{
2
3      int balance;
4      int minBalance;
5
6      public BankAccount() {
7          balance = 10000;
8          minBalance = 2000;
9      }
10
11     void withdraw(int amount) {
12         balance -= amount;
13
14         if(balance < minBalance) {
15             balance += amount;
16             System.out.println("Withdraw Failed. Please deposit more money for transaction. Balance is Low: "+balance);
17         }else {
18             System.out.println("Withdraw Successfull. New Balance is "+balance);
19         }
20     }
21 }
22
23
24
25 public class BankingApp {
26
27     public static void main(String[] args) {
28
29         System.out.println("Banking Started");
30
31         BankAccount johnAccount = new BankAccount();

```

- 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.

```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1  class BankAccount{
2
3      int balance;
4      int minBalance;
5
6      public BankAccount() {
7          balance = 10000;
8          minBalance = 2000;
9      }
10
11     void withdraw(int amount) {
12         balance -= amount;
13
14         if(balance < minBalance) {
15             balance += amount;
16             System.out.println("Withdraw Failed. Please deposit more money for transaction.");
17         }else {
18             System.out.println("Withdraw Successfull. New Balance is "+balance);
19         }
20     }
21 }
22
23
24
25 public class BankingApp {
26
27     public static void main(String[] args) {
28
29         System.out.println("Banking Started");
30
31         BankAccount johnAccount = new BankAccount();

```

Console Output:

```

<terminated> BankingApp [Java Application] /usr/eclipse/plugins/org.eclipse.justj.openjdk.I
Started
w| Successfull. New Balance is 7000
w| Successfull. New Balance is 4000
w| Failed. Please deposit more money for transaction. Balance is Low: 4000
w| Failed. Please deposit more money for transaction. Balance is Low: 4000
w| Failed. Please deposit more money for transaction. Balance is Low: 4000
Finished

```

- 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).

The screenshot shows the Eclipse IDE interface with two open files:

- BankingApp.java**: A Java class with a constructor setting initial balance to 100000 and minimum balance to 20000. It contains a withdraw method that prints an error message if the balance falls below the minimum. The withdraw method is annotated with `@Override`.
- BankingException.java**: A Java class extending `Exception`, with a constructor that takes a `String` message and calls `super(message)`.

```
File Edit Source Refactor Navigate Search Project Run Window Help

BankingApp.java X
1  class BankAccount{
2
3      int balance;
4      int minBalance;
5
6@     public BankAccount() {
7         balance = 100000;
8         minBalance = 20000;
9     }
10
11@    void withdraw(int amount) {
12        balance -= amount;
13
14        if(balance < minBalance) {
15            balance += amount;
16            System.out.println("Withdraw Failed. Please deposit more money for transaction. Balance is Low: "+balance);
17        }else {
18            System.out.println("Withdraw Successfull. New Balance is "+balance);
19        }
20
21    }
22 }
23
24 class BankingException extends Exception{
25@     public BankingException(String message) {
26         super(message);
27     }
28 }
29
30
31 /**
32  * @author Deepak
33  */

java - Session31/src/BankingApp.java - Eclipse IDE
```

- 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.

```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1 class BankAccount{
2
3     int balance;
4     int minBalance;
5     int attempts = 0;
6
7     public BankAccount() {
8         balance = 10000;
9         minBalance = 2000;
10    }
11
12    void withdraw(int amount) {
13        balance -= amount;
14
15        if(balance < minBalance) {
16            balance += amount;
17            System.out.println("Withdraw Failed. Please deposit more money for transaction. Balance is Low: "+balance);
18            attempts++;
19        }else {
20            System.out.println("Withdraw Successfull. New Balance is "+balance);
21        }
22    }
23
24 }
25
26 class BankingException extends Exception{
27     public BankingException(String message) {
28         super(message);
29     }
30 }
31

```

- 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.

```

Applications : java - Session31/src/Ba...
File Edit Source Refactor Navigate Search Project Run Window Help
BankingApp.java X
1 class BankAccount{
2
3     int balance;
4     int minBalance;
5     int attempts = 0;
6
7     public BankAccount() {
8         balance = 10000;
9         minBalance = 2000;
10    }
11
12    void withdraw(int amount) {
13        balance -= amount;
14
15        if(balance < minBalance) {
16            balance += amount;
17            System.out.println("Withdraw Failed. Please deposit more money for transaction. Balance is Low: "+balance);
18            attempts++;
19        }else {
20            System.out.println("Withdraw Successfull. New Balance is "+balance);
21        }
22    }
23
24    if(attempts == 3) {
25        BankingException exception = new BankingException("Illegal Attempts: "+attempts);
26        throw exception;
27    }
28 }
29
30
31 class BankingException extends Exception{

```

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

```

1 class BankAccount{
2     int balance;
3     int minBalance;
4     int attempts = 0;
5
6     public BankAccount() {
7         balance = 10000;
8         minBalance = 2000;
9     }
10
11    void withdraw(int amount) throws BankingException {
12        balance -= amount;
13
14        if(balance < minBalance) {
15            System.out.println("Withdraw Failed. Please deposit more money for transaction. Balance is Low: "+balance);
16            attempts++;
17        }else {
18            System.out.println("Withdraw Successfull. New Balance is "+balance);
19        }
20
21        if(attempts == 3) {
22            BankingException exception = new BankingException("Illegal Attempts: "+attempts);
23            throw exception;
24        }
25    }
26
27 }
28
29
30
31 class BankingException extends Exception{
32     public BankingException(String message) {
33         super(message);
34     }
35 }
36
37
38 public class BankingApp {
39
40     public static void main(String[] args) {
41         System.out.println("Banking Started");
42
43         BankAccount johnAccount = new BankAccount();
44
45         try {
46             for(int i=0;i<5000;i++) {
47                 johnAccount.withdraw(3000);
48             }
49         }catch(Exception e) {
50             System.out.println("Exception is: "+e);
51         }
52
53     }
54
55     System.out.println("Banking Finished");
56 }

```

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.

```

1 class BankAccount{
2     int balance;
3     int minBalance;
4     int attempts = 0;
5
6     public BankAccount() {
7         balance = 10000;
8         minBalance = 2000;
9     }
10
11    void withdraw(int amount) throws BankingException {
12        balance -= amount;
13
14        if(balance < minBalance) {
15            System.out.println("Withdraw Failed. Please deposit more money for transaction. Balance is Low: "+balance);
16            attempts++;
17        }else {
18            System.out.println("Withdraw Successfull. New Balance is "+balance);
19        }
20
21        if(attempts == 3) {
22            BankingException exception = new BankingException("Illegal Attempts: "+attempts);
23            throw exception;
24        }
25    }
26
27 }
28
29
30
31 class BankingException extends Exception{
32     public BankingException(String message) {
33         super(message);
34     }
35 }
36
37
38 public class BankingApp {
39
40     public static void main(String[] args) {
41         System.out.println("Banking Started");
42
43         BankAccount johnAccount = new BankAccount();
44
45         try {
46             for(int i=0;i<5000;i++) {
47                 johnAccount.withdraw(3000);
48             }
49         }catch(Exception e) {
50             System.out.println("Exception is: "+e);
51         }
52
53     }
54
55     System.out.println("Banking Finished");
56 }

```

- 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.

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** Applications - java - Session31/src/BankingApp.java - Eclipse IDE
- Left Panel (Outline View):** Shows the project structure with a file named BankingApp.java.
- Center Panel (Editor View):** Displays the Java code for BankingApp.java. The code includes a main method that prints "Banking Started", creates a BankAccount, and enters a try block. Inside the try block, it has a for loop that attempts 5000 withdrawals. It prints successful withdraws and fails at attempt 3 with a BankingException. The exception message is "Illegal Attempts: 3".
- Right Panel (Console Tab):** Shows the execution output. It starts with "Banking Started", followed by several successful withdrawal messages ("Withdraw Successfull. New Balance is ..."). At attempt 3, it prints "Withdraw Failed. Please deposit more money for transaction. Balance is ...". This pattern repeats until the end of the loop. Finally, it prints "Banking Finished".

```

1 package com.simplilearn;
2
3 public class BankingApp {
4     public static void main(String[] args) {
5         System.out.println("Banking Started");
6         BankAccount johnAccount = new BankAccount();
7         try {
8             for(int i=0;i<5000;i++) {
9                 attempts++;
10                }else {
11                    System.out.println("Withdraw Successfull. New Balance is "+balance);
12                }
13
14                if(attempts == 3) {
15                    BankingException exception = new BankingException("Illegal Attempts: "+attempts);
16                    throw exception;
17                }
18            }
19        }
20
21        class BankingException extends Exception{
22            public BankingException(String message) {
23                super(message);
24            }
25        }
26
27        public class BankAccount {
28            public void withdraw(double amount) {
29                balance -= amount;
30                System.out.println("Withdraw Successfull. New Balance is "+balance);
31            }
32            public void deposit(double amount) {
33                balance += amount;
34            }
35            public double getBalance() {
36                return balance;
37            }
38        }
39    }
40
41
42
43
44
45
46
47

```

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.