

FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture30

Lecture 30: 'finally' keyword in Java


Welcome to Lecture 30: Exception Handling. So, till now, we have seen The blocks like try, catch, throw, and throws also. Right. So now, in this lecture, I will talk about the finally block.




What do you mean by finally in the exception? So when I am talking about the finally block. Right. So, this finally block is used to execute the code. Alright.

So, Java code. So, suppose you have opening and closing the connection. Right. So mainly, the closing connections are streamed. We are using the finally block, right? So, this finally block will be executed whether you have an exception or not, right? Whether you have handled the exception or not, we are going to see an example. So, this finally block will be automatically executed, right? So, whatever you are writing in the cout or System.out.println, that will be printed. So, it will be executed. We are going to see. So, the last point suggests when you look at the finally block.

Java finally block

- The **Java finally block** is used to execute crucial code, such as closing connections or streams.
- The **finally** block is always executed, regardless of whether an exception is handled.
- In Java, the **finally** block must follow a **try** or **catch** block.





So, this must follow a try or catch block. So, this must follow a try or catch block. Let us consider an example. So, here you have—I mean, it is a regular program. There is no exception caught.

So, in that case, let us see what is happening. So, we have a class called 'TestFinallyBlock,' right? And you have a main program. So, try: `int data = 25 / 5`. So, we know that this has to give 5, right?

And then you are printing: `System.out.println(data)`. So, therefore, there is no exception here. In that case, let us see what is happening. So, after catch, finally is executed. So, if you look at the third point, we had seen that the finally block must follow a try or catch block.

Correct? So, in that case, line number 7, you have catch and then you have finally. So, here what will happen? You are expecting `system.out.println data`. So, that will be printed.

So, since this is not throwing any exception, line number 7 will not be executed because it will not go to catch. There is no exception. So, there is no point of catch block. So, the compiler will go to line number 8 finally. So, you have `system.out.println` finally block is always executed.

So, this is what we suggested in the second point. The finally block is always executed regardless whether an exception is handled, right? So, here there is no exception, but still this will be printed, right? The line number 8 will be printed. So, your data is 5, right?


So, your data is 5, and if you look at the finally block, it is always executed, and that will also be printed. So now, if I compile and run the code in the terminal, I am getting the value 5, right? And you can see the finally block is executed. So both are getting printed, right? I hope it is clear now.

Usage of Java finally

```
1 class TestFinallyBlock{
2     public static void main(String args[]){
3         try{
4             int data=25/5;
5             System.out.println(data);
6         }
7         catch(NullPointerException e){System.out.println(e);}
8         finally(System.out.println("finally block is always executed"));
9         System.out.println("rest of the code...");
10    }
11 }
```

Terminal

```
sh-4.3$ javac TestFinallyBlock.java
sh-4.3$ java TestFinallyBlock
5
finally block is always executed
rest of the code...
```



In the case of an exception not being thrown, This is an example where the exception is not thrown. So now, you may ask the question: what will happen if there is an exception, right? So, an exception occurs, and we are not handling it, right?

An exception occurs and is not handled. So, in that case, what will happen? Same code, similar code. So, 25 divided by 0, right?


So, here you go, 25 by 0, right? Data is equal to 25 by 0. So, we know the exception will happen, right? What is the exception? Arithmetic exception.

exception occurs and not handled

```
1 class TestFinallyBlock1{
2     public static void main(String args[]){
3         try{
4             int data=25/0;
5             System.out.println(data);
6         }
7         catch(NullPointerException e){System.out.println(e);}
8         finally(System.out.println("finally block is always executed"));
9         System.out.println("rest of the code...");
10    }
11 }
```

Terminal

```
sh-4.3$ javac TestFinallyBlock1.java
sh-4.3$ java TestFinallyBlock1
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestFinallyBlock1.main(TestFinallyBlock1.java:4)
sh-4.3$
```



But unfortunately, cache does not have any arithmetic exception. That is why, if you look at the title, exception occurs and is not handled, right? So, In the standard Java exception, that will be printed. So, what is it printing?

Java.lang.ArithmeticException: divided by 0, right? So, this is the inbuilt call, right? The standard that, in fact, we have seen some of the standard C++

exceptions, right? Similarly, I mean, this is the case of Java, 25 by 0 when you put Assume that you do not know anything about exceptions, you know you will get a runtime error, right?

That runtime error is nothing but exception error, okay? So, this is exception error in Java and we are not handling. As a programmer, we are not handling, correct? So, this will be thrown and more importantly look at, right? So, we have finally, finally is being executed, right?

So, you can see the output here. You can see the output here. Finally, block is always executed. Correct? Finally, block is always executed.

But line number 9, we have one print statement. Right? Rest of the code, it will not go. Right? So, this is the difference.

Line number 8 is being printed, but line number 9 is not being printed. Right? So, line number 4 has an exception: 25 divided by 0. We know that is an arithmetic exception, but it is not being handled. If you look at line number 7, I have used a null pointer exception, right?

But a null pointer exception is not occurring. So, this exception is not being caught by any catch block, right? Therefore, the system itself is automatically throwing the exception, which you call an arithmetic exception, right? So, that is being thrown by the system. You can see that, right?

Exception in the thread, right? So, which is nothing but the arithmetic exception, which is nothing but division by 0, right? That is a division by 0 exception occurring at line number 4. But if you look at the 'finally' keyword in line number 8, that particular block is executed. And if you look at the output, the 'finally' block is always executed, meaning it appears in the output.

Whereas line number 9, rest of the code is not there. So I hope you understood. So finally will always be executed irrespective of exception is handled or not. So now assume that we are handling this exception. We are handling this exception.

The previous example is suppose we are not handling the exception. So in this case assume that we are handling the exception 25 by 0. Line number 4 you have 25 by 0. And you are trying to print the data. So that is arithmetic exception.

So the arithmetic exception. Occurs at line number 4. And if you look at my catch block. We have handled. Arithmetic exception.

System.out.println. The message will be printed. So, whatever the message ArithmeticException is having. That will be printed. So, this is nothing but.

So, this is the one. Where the system already. So, what the system already has java.lang.ArithmeticException slash that means divide by 0, right. So, in this case, you can see that, right. So, this will be printed, right.

So, now line number 8, yes, even the previous example we had seen line number 8 was printed, right. So, system.out.println, finally block is always executed. Line number 8, system.out.println, finally block is always executed. And then finally, your system.out.println, you have the rest of the code. So, you have the rest of the code.

So, that is also being printed here. Whereas in the previous example we saw, we did not find this line number 9. It is not going to line number 9. There is an exception; it is going out. That is it.

But the only difference is whenever you have the finally block, that is always executed. So, you can think of it as we are going to see a case study soon, right? So, whatever the case may be—whether the student is admitted or not, passed or failed—you can always put some System.out.println and print something, right? Even if it is exiting. Alright? Not only for ArithmeticException but also for any other exception like NullPointerException or any other exception we have studied, if it occurs, but you put the finally keyword right, the finally block is always executed, right. So, obviously, the finally block is executed and will be printed. Apart from that, the rest of the code, right, the rest of the code is also being printed, right. So, I hope it is clear now how the finally block works.


So, now we use this, right, all exception handling blocks like try, throw, catch, Also, let us consider the finally block as well, right. So, let us see this particular case study. Suppose you want to divide two numbers, right, entered by the user. If the second number is 0, an exception will be thrown, right.

Since division by 0 is not allowed, right, we know division by 0 is not allowed. So, regardless of the outcome, you want to always print a message: 'Operation

completed.' All right. So, when you get this kind of question, whatever the case may be, what will happen? Assume that I don't know exceptions.

Case Study on Java finally

Suppose you want to divide two numbers entered by the user. If the second number is zero, an exception will be thrown (since division by zero is not allowed). Regardless of the outcome, you want to always print a message saying, "Operation completed."



So, A divided by 0 or A by B, C is equal to A by B, and my user input B is equal to 0. So, what will happen? An exception, and I do not think you can print 'Operation completed' even with a `System.out.println` statement, only `System.out.println`, right. So, regardless of the outcome, you want to always print a message saying that the operation is completed, right. So, only when you use 'finally'.

Right, the Java 'finally'. So, then we can handle this problem, right, or we can handle this kind of problem. So, let us consider, right. So, I have imported all the utilities: lang and input-output. You have the

Driver class division example, I have the main program right. So, I am going to take some input from the user, all right. So, let us consider num1 and num2 and result, all right. So, it is basically a division, all right. So, take two numbers and the corresponding result.

So now, let us try `System.out.printl` and enter the first number. So, you are taking the input from the user. Similarly, I am taking the second input from the user, first input and second input. Stored as number 1 and number 2, and here result is equal to num1 divided by num2, right. So, there is always a possibility that it will throw an exception if num2 is equal to 0. It will throw the exception. We will come to that. Then, we are printing if it is not 0, it will print, right.

So, if there is an error, it will be caught by this exception, right? I mean to say, if there is an exception, it will be caught by this exception, and then it will print, 'Error: cannot divide by'. 0, right? And more importantly, we use finally, right?

Finally, operation completed. So, let us assume I am giving two inputs. One is, let us say, 25 and 5. Assume that I am giving 25 and 5. So, tell me what will happen. So, if I give 25, num1 is 25 and num2 is 5. Right, num1 is 25 and num2 is 5.

So this will go here because I have taken 2 input num1 by num2 alright 25 by 5 result will be 5 correct and yeah obviously this will not go over here right. So obviously this will not go over here and in line number 23 you are printing result. So 5 will be printed 25 by 5 5 alright. So there is no exception thrown So it will not go to catch, but as we know finally will be always executed, right?

Finally will be always executed. So system.out.println and operation completed. This will be printed, right? So this will be printed. So this you know, right?


So you may ask the question here, why you are putting finally, right? Even if I put only system.out.println and it should work. Yes, you are right. It will work, but we are going to see another input. So let us assume...

I am giving this particular input. Assume that I am giving this input. 200 is num1 and 0 as num2. Right? So, when I give this as a input, right?

So, here I will give num1 is 200 and num2 is 0. Right? So, num1 by num2, 200 by 0, there will be an exception. You are not throwing anything, but it will be automatically thrown. Num1 by num2 automatically throw the exception and this exception will be caught over here, catch block.

```
13  try {
14      // Taking input from user
15      System.out.println("Enter first number: ");
16      num1 = scanner.nextInt();
17
18      System.out.println("Enter second number: ");
19      num2 = scanner.nextInt();
20
21      // Attempting division
22      result = num1 / num2;
23      System.out.println("Result: " + result);
24  }
```

Handwritten annotations on the code: An arrow points from the number 200 to line 16. Another arrow points from the number 0 to line 19. A third arrow points from the division operation `num1 / num2` on line 22 to the catch block area. A small diagram shows 200 divided by 5 equals 5.



Arithmetic exception, it is being thrown and then you are printing error cannot divide by 0, right. So, cannot divide by 0. In the last test cases, when I gave 25 and 5, I said if you simply put system.out.print and operation completed will be

executed. But in this case, as we already seen in an example, right? So, if I do not put finally, it will not go to the `system.out.println` block at all.

So, my program is saying that whatever the case, I have to print 'operation completed,' right? This is the reason we use the 'finally' keyword. I hope now you understand why we are using 'finally.' So, whatever the exception, This will be printed.

So, now an exception occurs. So, that means we are going to see the output: 'Error: cannot divide by 0.' This is the first output we are going to get, and then it will go to the 'finally' block, and in the 'finally' block, 'operation completed' will be printed, correct? So, 'operation completed' will be printed, right? So, these are all the outputs we are getting.

So, 'Error: cannot divide by 0,' correct? So, this is being printed. You are right. So, next one: 'operation completed,' right? So, this is one.

So, if num2 is non-zero the case that we had studied 25 by 5 right. So, whatever be the case whether it is throwing an exception or not it is coming to the finally block right. So, that is the beauty of this finally. So, there is an exception still finally will work if there is no exception even that case also works. finally should work right.

So, this is how we can handle. So, even what you can do all right. So, you can think of let us say any other exception let us say null pointer exception right. So, then what do you do you throw the message assume that you are creating a nodes millions of nodes and at one point of time you may face there is no memory. So when there is no memory, the pointer will point to null.

Yesterday we had seen. It is uncontrollable. So in that case, what you can do, you write the code saying that you are throwing the exception. Automatically you throw the exception. And even here also you can throw the exception.

Num2. If num2 is equal to 0, throw the exception. Just modify the code. Now we have seen several examples. You have to write the program on your own.

If num2 is equal to 0, you throw the exception, correct. So when you are throwing the exception, we know that it will be caught over here, right. So even in this case, since num2 is equal to 0, the exception is automatically thrown. So there

are cases in data structures where you have to write, alright? When you are not able to create a further node, the exception will be automatically thrown. So whenever

It is automatically thrown, or you are writing, and then you are getting, right, the throw, right? Throw is being called. So then it will be caught over here, right. So this will be caught over here, and then it is printing this. So in this case, when I give num2 as equal to 0, right? When I see num2 is equal to 0, I am a little bit careful. So what is happening?

Here in this case, I know the exception will be thrown, right. Anyway, this is in the try block. So, catch error: cannot divide by zero, printing it, and then I am also getting 'operation completed' right because, whatever be the case, I have been told in the program 'operation completed' has to be printed. So, we will see one more case study, right. So, let us develop a banking system, right, where a user can withdraw money from their account.

If the withdrawal amount exceeds the available balance, right, this particular example we have seen for several cases. But we can see this also, right? If the withdrawal amount exceeds the available balance, an exception will be thrown, right? The exception is thrown. However, regardless of whether the withdrawal succeeds or fails,

You want to ensure that the transaction is logged properly. Right. So that means they are asking to print that the transaction is logged properly. Right. If it is exceeding, we know what is the error we have to throw or what is the exception we have to throw.


But whatever the case may be. Right. Whether the withdrawal is successful or there is a failure. I mean, you have to ensure that your program ensures the transaction is logged properly. So that message has to be displayed.

Right. So in the previous case, the exception was automatically thrown. And here. So let us see how we are going to throw the exception. So let us consider this particular program.

We have an insufficient funds exception. This is extending exception. Right. Which is the inbuilt base class or inbuilt superclass. And I have a constructor, insufficient funds exception constructor.

Right. So there is a message whose data type is string. And then it will call the superclass constructor. Right. By passing the message.

```
1  /* package whatever; // don't place package name! */
2
3  import java.util.*;
4  import java.lang.*;
5  import java.io.*;
6
7  class InsufficientFundsException extends Exception {
8      public InsufficientFundsException(String message) {
9          super(message);
10     }
11 }
12
13 class BankAccount {
14     private double balance;
15
16     public BankAccount(double initialBalance) {
17         this.balance = initialBalance;
18     }
19 }
```



Fine. Now you have a class BankAccount. Right. So you have a class BankAccount. It has a private member data.

Double balance. Right. And you have The constructor BankAccount, passing one argument, making it a parameterized constructor. And InsufficientFundsException, this is also a parameterized constructor, right?


And the initial balance is assigned to this.balance, right? The initial balance is assigned to this.balance, where balance is the private member data. So, the next method is void withdraw, right? The method, because the previous one is the special method called the constructor. And here you have void withdraw.

You have double amount, right? So, amount is the parameter whose data type is double. It throws an insufficient funds exception, right? So, we are declaring this exception, right? So, you call it an insufficient funds exception.

So, that means we have already seen this, right? In the previous programs or similar programs, we have seen this, right? If the amount is greater than the balance. Obviously, if the amount is greater than the balance. So, then we have to throw this insufficient funds exception.

Alright. And then you have to print the message: 'Insufficient balance, cannot withdraw.' And whatever amount. Alright. Otherwise, if the other way around, the amount is less than or equal to the balance.

```
19.
20.     public void withdraw(double amount) throws InsufficientFundsException {
21.         if (amount > balance) {
22.             throw new InsufficientFundsException("Insufficient balance. Cannot withdraw " + amount);
23.         }
24.         balance -= amount;
25.         System.out.println("Withdrawal successful. Remaining balance: " + balance);
26.     }
27.
28.     public double getBalance() {
29.         return balance;
30.     }
31. }
32.
```



Then, balance is equal to balance minus amount. Right. Balance is equal to balance minus amount. I hope it is clear for everyone. So, now.

If this happens, right, when the code goes here or the amount is less than or equal to the balance, then you have to print: 'Withdrawal successful, remaining balance is balance.' Okay. And then we will have one getter function, getBalance, right? No arguments, and you have a return type double. So, that will return balance, right. I hope it is clear up to here. Return balance. So, now I have a class Transaction.


Class BankTransaction, right? Which is a driver class. You have the main method. I have one object account under BankAccount, right? So what I am doing is dynamically allocating memory with the constructor. You are passing the value 1000, right? You are passing the value 1000. So when I am passing the value 1000 to BankAccount, right? So here you go, the BankAccount correct class. Constructor. So initial balance, that means 1000, I am passing, right? So this dot balance will be 1000, right? So 1000 I am passing here, that means my balance is getting the value 1000 initially, right?

So that is what I am doing here, right? And then withdraw amount. Withdraw amount is 1200, right? Withdraw amount is equal to 1200, that means the user is trying to withdraw 1200, right? So in that case, We are trying here, right?

We are using a try block. So, System.out.println, then attempting to withdraw, right? So, this money you are going to withdraw. So, how much money am I going to withdraw? 1200 will be printed, right?

So, this will be printed. Now, this account object, right? The account object is invoking withdraw method, right? This account object is invoking withdraw method, right? So you are passing this withdraw amount.

```
38. try {
39.     System.out.println("Attempting to withdraw " + withdrawAmount);
40.     account.withdraw(withdrawAmount); // Attempt withdrawal
41. } catch (InsufficientFundsException e) {
42.     System.out.println("Error: " + e.getMessage()); // Handling insufficient
    funds
43. } finally {
44.     // This block will always execute, logging the transaction status
45.     System.out.println("Transaction completed. Current balance: " + account.g
    etBalance());
46. }
47. }
48. }
```




How much is the withdraw amount? 1200. So that means it is invoking withdraw method. So when it is invoking withdraw method, let us say what is happening. Your amount you are trying to withdraw.

This is what you are passing. 1200 you are passing here. And balance we know already what? 1000. So 1200.

Here it is 1200. And the balance is 1000. We know this is true. 1200 is greater than 1000. True.

```
19.
20. public void withdraw(double amount) throws InsufficientFundsException {
21.     if (amount > balance) {
22.         throw new InsufficientFundsException("Insufficient balance. Cannot withdr
    aw " + amount);
23.     }
24.     balance -= amount; ✓
25.     System.out.println("Withdrawal successful. Remaining balance: " + balance); ✓
26. }
27.
28. public double getBalance() {
29.     return balance;
30. }
31. }
32. }
```



So this will throw. All right. So this checked exception is an insufficient funds exception. Right. So that means this will be printed.

Right. Insufficient balance: cannot withdraw will be printed, and amount is 1200 will also be printed. That means the first output I will get is insufficient balance: cannot withdraw. Right. So if at all any other print statement we have.

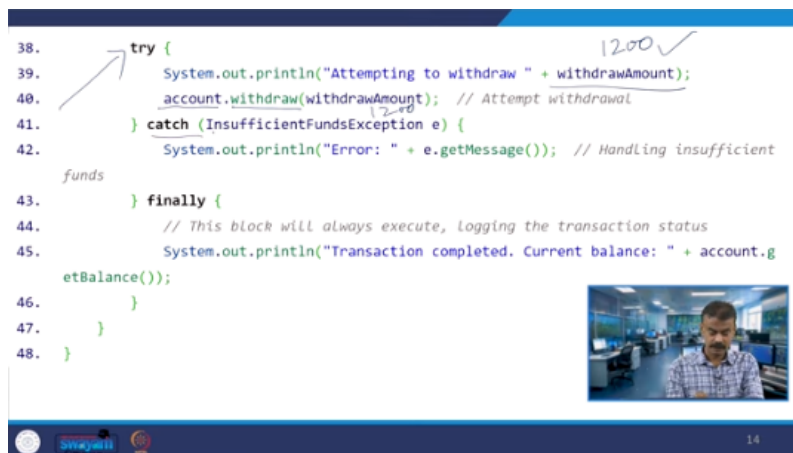
So that will be there. Right. So, what is the print statement? Yes. Yeah.

Attempting to withdraw 1200 will be the first output. Correct. And then. Here you have. This particular statement.

Insufficient balance. Cannot withdraw. The amount is 1200. So that will be printed. So once it throws the exception.

So this is with. Declaration exception, right, that we had seen in the last class, right, the checked exception. So, in that case, this will be caught over here. The name is insufficient funds exception with the message, right. So, what it will print?

```
38. try {
39.     System.out.println("Attempting to withdraw " + withdrawAmount);
40.     account.withdraw(withdrawAmount); // Attempt withdrawal
41. } catch (InsufficientFundsException e) {
42.     System.out.println("Error: " + e.getMessage()); // Handling insufficient
    funds
43. } finally {
44.     // This block will always execute, logging the transaction status
45.     System.out.println("Transaction completed. Current balance: " + account.getBalance());
46. }
47. }
48. }
```



It will print error and this object E, right, this object E will invoke get message. What is the get message? Get balance you have, right? And whatever is available in the superclass exception message, right? So, that will be printed.

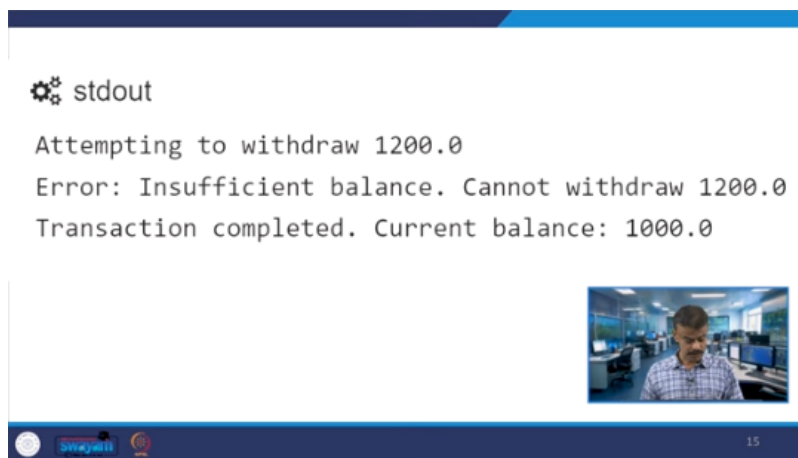
In fact, it is throwing this. The insufficient balance cannot withdraw and with the amount, right? So, this will be thrown. This will be thrown and this message will be printed in get message, right? This will be printed in the get message.

The get message which is available in the superclass exception, right? So, in fact, we are throwing this and that will be printed insufficient funds exception. So, you have this particular message. So, get message which is available in the

superclass. So, therefore, this particular message that means the insufficient balance cannot withdraw 1200 will be printed.

And then it is going to finally like a previous program or the theory so far we had studied. So, this will go to finally and `system.out.println` transaction completed and it in fact will show the current balance or whatever you have current balance. So, that means `account.get balance`. So, that will be displayed 1000. I believe 1000 is balanced.

So, that will be printed. So, you can see what are all the output we are getting. All right. So, in fact, here we are printing error. So, that will be printed.

A screenshot of a terminal window with a dark blue header and footer. The header contains a gear icon and the text 'stdout'. The terminal output consists of three lines: 'Attempting to withdraw 1200.0', 'Error: Insufficient balance. Cannot withdraw 1200.0', and 'Transaction completed. Current balance: 1000.0'. In the bottom right corner of the terminal window, there is a small inset video of a man in a blue shirt. The footer of the terminal window shows a search bar, a 'Run' button, and a page number '15'.

And as I said, the get message is nothing but this one. Insufficient balance cannot withdraw that particular amount. The first one we have already printed attempting to withdraw 1200. Correct. So, this will be printed.

And in the next case error, this message will be printed. Right. Insufficient balance cannot withdraw 1200. And finally, the transaction completed will be printed, right? So, this is the one.

In the finally, transaction completed and the current balance is 1000 will be printed, right? So, this is all about your keywords in Java, all right? So, try, that means in the exception, error handling exceptions, what are the keywords we are using? Try, catch, throw, throws, finally. So, all the examples that we had seen.

And in the case of standard error or standard exception, we have seen try, throw, catch, block in C++, right? Only try, throw, catch. There is no throws and finally in


C++, right? So here I have taken one particular example, overflow error, right? One particular standard exception.

So we will see this. So what will happen? Assume that you are multiplying two numbers. So you are passing those two numbers. When I take the integer, there will be a limit, the maximum.

If it is a 32-bit system or 64-bit system, you have the range, minimum and maximum. So assume that I am handling based on this function, the maximum. maximum number so when you are multiplying A star B right assume that I am taking a maximum number both A and B assume that A and B are both the maximum if I multiply that will definitely go beyond what is the maximum both are maximum and then I am multiplying or even that example that we are going to see let us take 50000 A is 50000 B is 50000 when you multiply it may go beyond this so when I am dividing by B if it goes beyond A right A greater than this correct I mean A is going beyond this maximum by B right. So then you will have the overflow error and this is one of the standard errors we just go through the slides that we had seen right.

std::overflow_error in C++

```
1 #include <iostream>
2 #include <stdexcept>
3 #include <limits>
4
5 int multiply(int a, int b) {
6     if (a > 0 && b > 0 && a > (std::numeric_limits<int>::max() / b)) {
7         throw std::overflow_error("Overflow detected");
8     }
9     return a * b;
10 }
11
```




16

So I took a few examples when we are dealing with C++, and in today's class, I have taken this particular example: overflow error, right. If one of them is true, right? If one of them is false, correct? If one of them is false, the overflow error will occur. So if a is greater than 0 and b is greater than 0, and a is alright—so that means we are taking two, let us say, non-negative numbers, alright? And if it is not satisfying this, There will be an overflow error. So we are printing, 'Overflow detected.'

Otherwise, you simply return A star B—multiply the two numbers. Right. So let us consider trying X = 50000, Y = 50000. I said this example that I am planning to work it out. So you are calling multiply(X, Y). So, 50000 into 50000.

```
--
12 * int main() {
13 *     try {
14 *         int x = 50000;
15 *         int y = 50000;
16 *         int result = multiply(x, y);
17 *         std::cout << "Result: " << result << std::endl;
18 *     } catch (const std::overflow_error& e) {
19 *         std::cout << "Error: " << e.what() << std::endl;
20 *     }
21 *
22 *     return 0;
23 * }
```



This condition will be satisfied. Right. A is positive—true. B is positive—true. This condition will not be satisfied.

False. Right. True, true, and false. So this will be false. Right.

Sorry. In fact, this is true. Yes, it will go beyond the range. A star B will be greater than max. A star B will be greater than max.

So this will also be true. So true, true, and true. So it will go inside. So that means it is throwing an overflow error. All right.

It is throwing overflow error. So overflow detected will be printed. right? So, that will be correct. So, that will be printed, right?

So, what is happening? So, this will catch, right? Because exception will be thrown by this function. So, that will be caught over here, right? And then error and also we had seen many times e dot what, right?

So, that means whatever the message in Java also we had seen similar one. So, what so overflow detected will be printed. That means error overflow detected will be printed. So when I run the code I have to get this output that is what I said error overflow detected will be printed.

So here, all the conditions are true: A is positive 50000, B is 50000, and A star B, when you take it, will definitely go beyond the maximum right beyond the limit,

correct. So when it is going beyond the limit, you are passing this information, which means you are throwing this exception. So, this exception will be caught over here, and E dot—what we had seen several times. So, that means this will print this particular information. So, that means you can see that the error will be printed, and 'overflow detected' will be printed, right.

So, with this, we have seen the complete error handling exception, both in C++ and Java. Thank you.