

PROGRAMMING

Lecture 18

Sushil Paudel

PREVIOUS TOPIC

- Abstraction
- Abstract Class
- Interface

TODAY'S TOPIC

- Exception Handling
- Try Catch

EXCEPTION HANDLING

- Bad things happens.
- You have to pretend that everything is going to fail.

EXCEPTION HANDLING

- An exception (or exceptional event) is a problem that arises during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

EXCEPTION HANDLING

- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
 - A user has entered an invalid data.
 - A file that needs to be opened cannot be found.
 - A network connection has been lost in the middle of communications, etc.

EXCPETION HANDLING

- The core advantage of exception handling is **to maintain the normal flow of the application.**
- An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

SCENARIO

- statement 1;
- statement 2;
- statement 3;
- statement 4;
- statement 5;//exception occurs
- statement 6;
- statement 7;
- statement 8;
- statement 9;
- statement 10;

SCENARIO

- Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.
- If we perform exception handling, the rest of the statement will be executed.
- That is why we use exception handling in Java.

TYPES OF EXCEPTION

There are two types of exceptions as below:

- Checked Exception
- Unchecked Exception

CHECKED EXCEPTION

- Checked exceptions are checked at compile time to ensure you are handling them, either by catching them or declaring the containing method throws the exception.
- If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. Else it will throw compile time error.

Eg. FileNotFoundException

```
File file = new File("E://file.txt");  
FileReader fr = new FileReader(file);
```

UNCHECKED EXCEPTION

- An unchecked exception is an exception that occurs at the time of execution.
- These are also called as **Runtime Exceptions**.
- These include programming bugs, such as logic errors or improper use of an API.
- Runtime exceptions are ignored at the time of compilation.
- For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array the an *ArrayIndexOutOfBoundsException* occurs.

UNCHECKED EXCEPTION

- **Example**

```
int num[] = { 1, 2, 3, 4 };  
System.out.println(num[5]);
```

- **Output**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5  
    at com.exception.ExceptionTest.exceptionUnchecked(ExceptionTest.java:19)  
    at com.exception.ExceptionTest.main(ExceptionTest.java:9)
```

EXCEPTION METHOD

Sr.No.	Method & Description
1	<code>public String getMessage()</code> Returns a detailed message about the exception that has occurred.
2	<code>public Throwable getCause()</code> Returns the cause of the exception as represented by a Throwable object.
3	<code>public String toString()</code> Returns the name of the class concatenated with the result of <code>getMessage()</code> .
4	<code>public void printStackTrace()</code> Prints the result of <code>toString()</code> along with the stack trace to <code>System.err</code> , the error output stream.

COMMON SCENARIO OF EXCEPTION

Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

COMMON SCENARIO OF EXCEPTION

Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;  
System.out.println(s.length()); //NullPointerException
```


SCENARIO 3

Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";  
int i=Integer.parseInt(s);//NumberFormatException
```

SCENARIO 4

Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];  
a[10]=50; //ArrayIndexOutOfBoundsException
```

EXCEPTION HANDLING METHODS

- Try-catch
- Throws
- Throw

JAVA EXCEPTION KEYWORDS

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

TRY-CATCH

- Java try block is used to enclose the code that might throw an exception. It must be used within the method. Java try block must be followed by either catch or finally block.

```
try {  
    //code that may throw exception  
} catch (Exception ex) {  
  
}
```

EXAMPLE

```
try {  
    int data = 50 / 0;  
} catch (ArithmeticException e) {  
    e.printStackTrace();  
}  
System.out.println("Hello after try-catch...");
```

TRY CATCH FINALLY

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc. Java finally block is always executed whether exception is handled or not. Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

```
try{  
  
}catch(Exception ex){  
  
}finally {  
  
}
```

EXAMPLE

```
try{
    System.out.println("Try 1");
}catch(Exception ex){
    System.out.println("Catch 1");
}finally {
    System.out.println("Finally 1");
}
```

```
System.out.println("\n");
```

```
try{
    System.out.println("Try 2");
    String str = null;
    int length = str.length();
}catch(Exception ex){
    System.out.println("Catch 2");
}finally {
    System.out.println("Finally 2");
}
```


OUTPUT

```
Try 1  
Finally 1
```

```
Try 2  
Catch 2  
Finally 2
```

MULTIPLE CATCH

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

EXAMPLE

```
public class MultipleCatchBlock1 {  
    public static void main(String[] args) {  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(ArithmeticException e) {  
            System.out.println("Arithmetic Exception occurs");  
        }  
        catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("ArrayIndexOutOfBoundsException occurs");  
        }  
        catch(Exception e) {  
            System.out.println("Parent Exception occurs");  
        }  
        System.out.println("rest of the code");  
    }  
}
```

OUTPUT

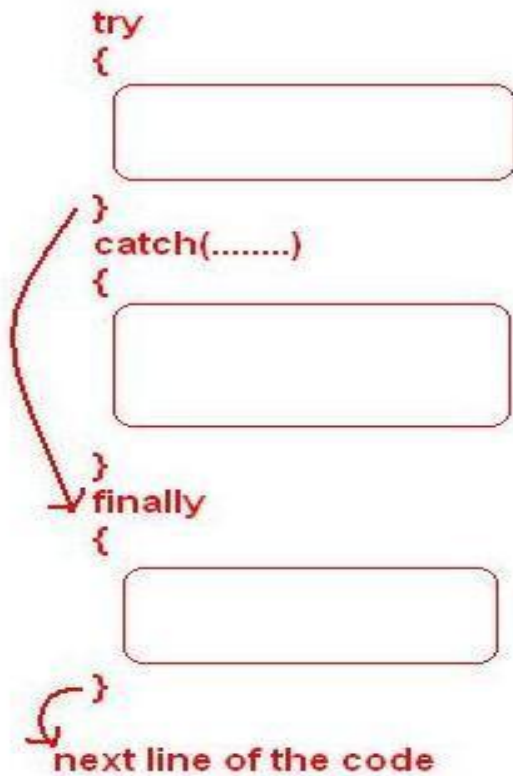
- Arithmetic Exception occurs
- rest of the code

NESTED TRY CATCH

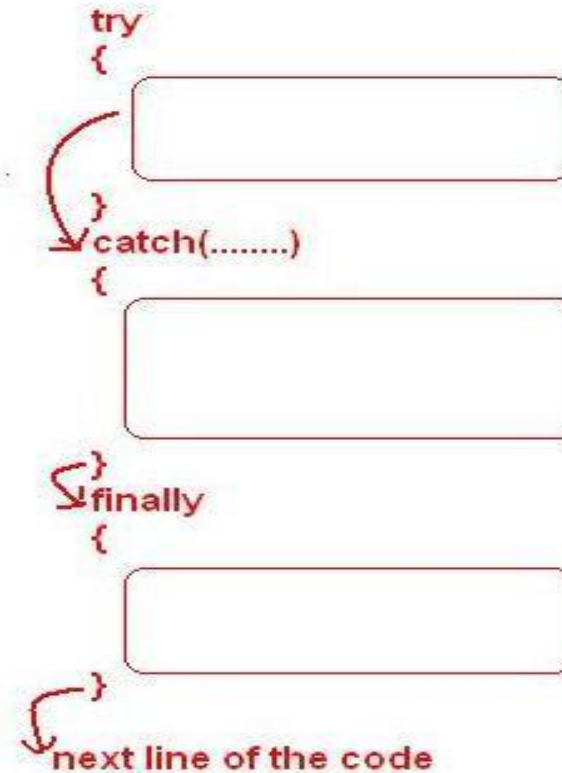
- **try** statement can be **nested** inside another block of **try**. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner **try** block does not have a **catch** handler for a particular exception then the outer **try catch block** is checked for match.

TRY CATCH FINALLY

No exceptions thrown:



An exception arises :

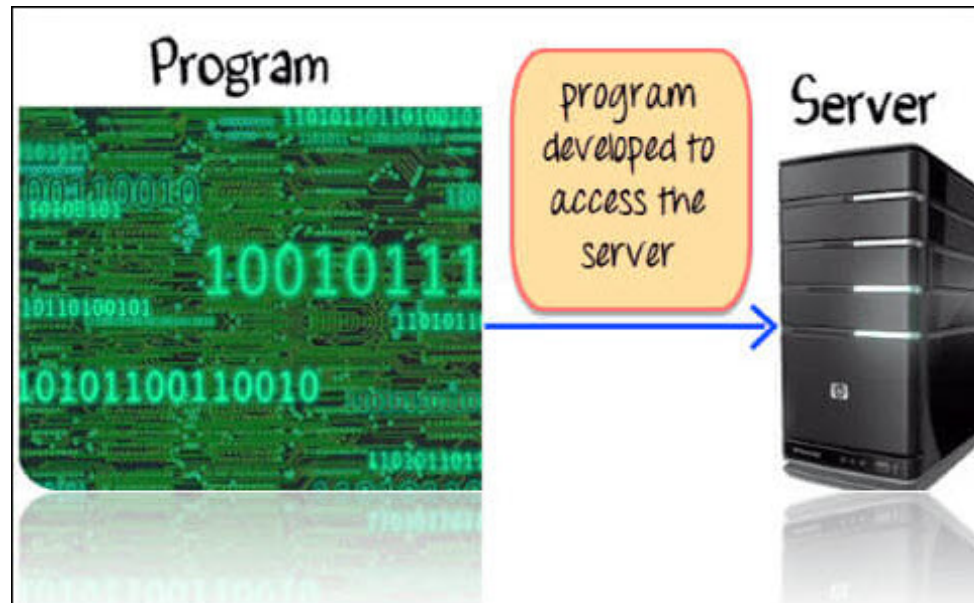


EXAMPLE :D



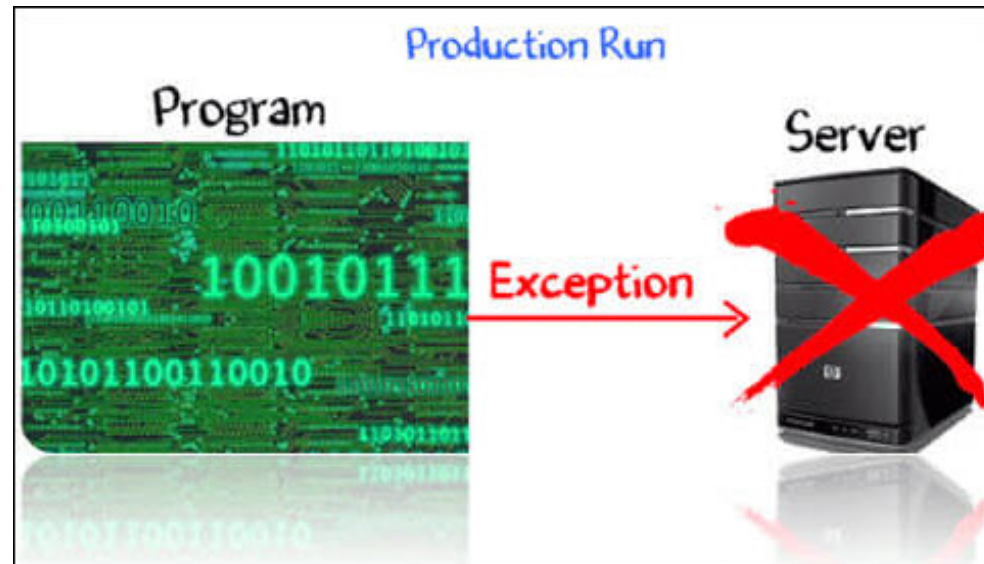
EXAMPLE

- Suppose you have coded a program to access the server. Things worked fine while you were developing the code.



EXAMPLE

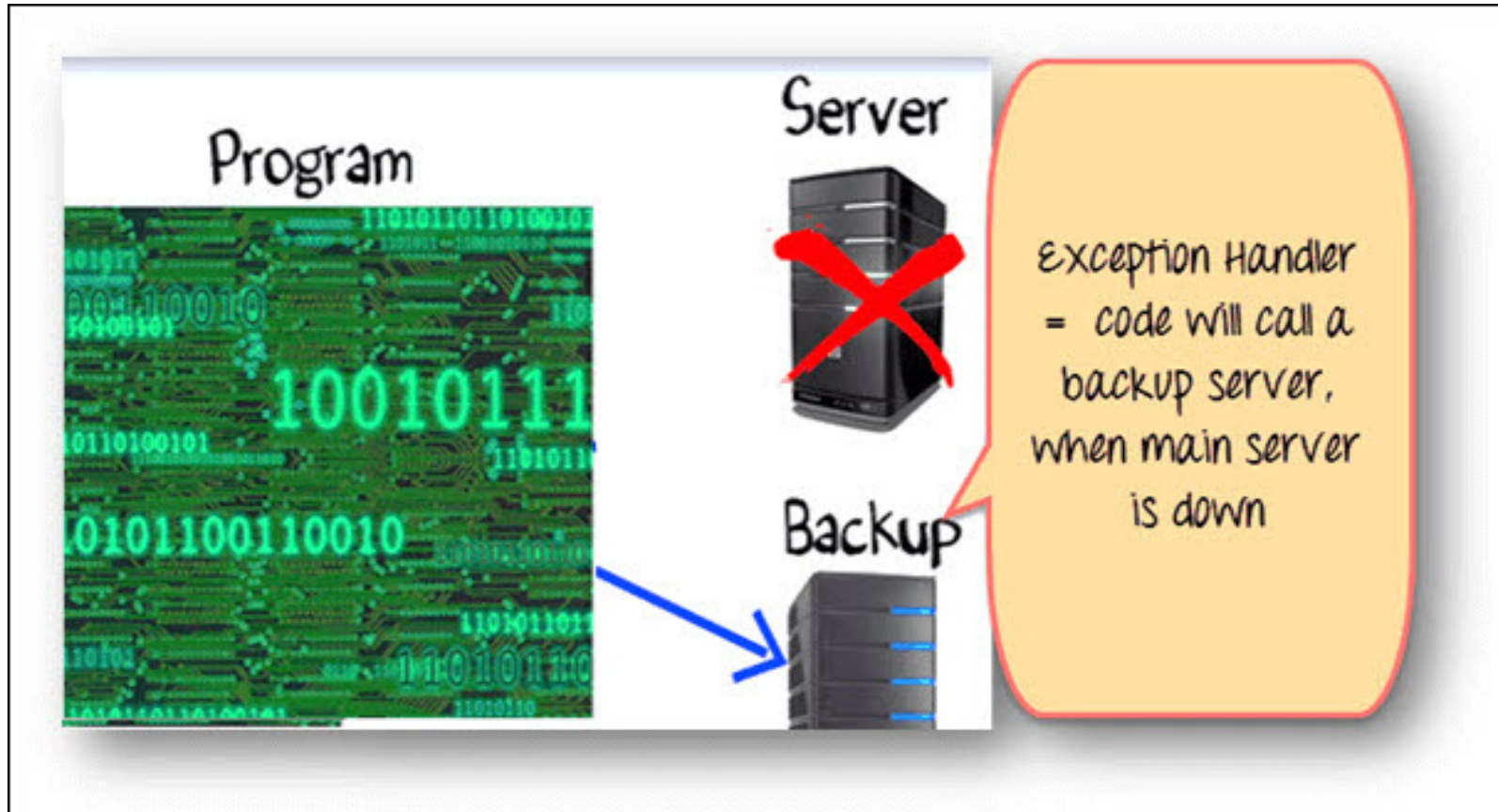
- During the actual production run, the server is down. When your program tried to access it, an exception is raised.



HOW TO HANDLE EXCPETION

- So far, we have seen, exception is beyond developer's control.
- But blaming your code failure on environmental issues is not a solution.
- You need a Robust Programming, which takes care of exceptional situations.
- Such code is known as **Exception Handler**.
- In our example, good exception handling would be, when the server is down, connect to the backup server.

SOLUTION



EXAMPLE

```
public class MyClass {  
    public static void main(String[ ] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        }  
    }  
}
```

OUTPUT

Something went wrong.

EXAMPLE

```
public class TryCatchExample1 {  
  
    public static void main(String[] args) {  
  
        int data= 1;  
        System.out.println("Rest of the code");  
    }  
}
```

OUTPUT

Rest of the code

EXAMPLE

```
public class TryCatchExample2 {  
    public static void main(String[] args) {  
        try {  
            int data="f";  
        } catch(ArithmeticException e) {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
}
```


OUTPUT

Compilation Error

ANY QUESTIONS?

Thank you!