Java SE & Spring

Module 1: Java SE

12.Exception Handling

What is an Exception?

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons. Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file

Further reading: https://howtodoinjava.com/java/exception-handling/

Exception Types

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

- **Checked exceptions**: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions**: A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Errors**: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Types

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

Exception Types

A runtime exception happens due to a programming error. They are also known as unchecked exceptions.

These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:

- Improper use of an API - IllegalArgumentException
- Null pointer access (missing the initialization of a variable) - NullPointerException
- Out-of-bounds array access - ArrayIndexOutOfBoundsException
- Dividing a number by 0 - ArithmeticException
- You can think about it in this way. “If it is a runtime exception, it is your fault”.

The NullPointerException would not have occurred if you had checked whether the variable was initialized or not before using it. An ArrayIndexOutOfBoundsException would not have occurred if you tested the array index against the array bounds.

Exception Handling

When an exception occurs within a method, it creates an object. This object is called the exception object.

It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred. Here's a list of different approaches to handle exceptions in Java.

- try-catch-finally block
- throw and throws keyword

try-catch-finally

try block:

The try block contains the application code which is expected to work in normal conditions. For example, reading a file, writing to databases or performing complex business operations. A try block is written with try keyword followed with the curly braces.

Syntax:

```
1  try {
2      //application code
3  }
```

try-catch-finally

try-catch block:

The optional catch block(s) follows the try block and MUST handle the checked exceptions thrown by try block as well as any possible unchecked exceptions. An application can go wrong in N different ways. That's why we can associate multiple catch blocks with a single try block. In each catch block, we can handle one or more specific exceptions in a unique way.

When one catch block handles the exception, the next catch blocks are not executed. Control shifts directly from the executed catch block to execute the remaining part of the program, including finally block.

Syntax:

```
1  try {
2      //code
3  }
4  catch(Exception e) {
5      //handle exception
6  }
```

try-catch-finally

finally block:

An optional finally block gives us a chance to run the code which we want to execute EVERYTIME a try-catch block is completed – either with errors or without any error. The finally block statements are guaranteed of execution even if the we fail to handle the exception successfully in catch block.

Syntax:

```
1  try {
2      //open file
3      //read file
4  }
5  catch(Exception e) {
6      //handle exception while reading the file
7  }
8  finally {
9      //close the file
10 }
```

try-catch-finally

Example:

```
1  try {
2      int a = 5/0;
3      System.out.println(a);
4
5  }
6  catch(ArithmeticException e) {
7      System.out.println(e.getMessage());
8  }
9  finally {
10     System.out.println("This is the finally block of the try-catch statement");
11 }
```

try with Resources

The try-with-resources statement automatically closes all the resources at the end of the statement. A resource is an object to be closed at the end of the program.

Syntax:

```
try (resource declaration) {
  // use of the resource
} catch (ExceptionType e1) {
  // catch block
}
```

As seen from the above syntax, we declare the try-with-resources statement by,

- declaring and instantiating the resource within the try clause.
- specifying and handling all exceptions that might be thrown while closing the resource.

Note: The try-with-resources statement closes all the resources that implement the AutoCloseable interface.

throw and throws

As we said in introduction section of the chapter, exceptions can be categorized in two section:,

- **Unchecked Exceptions**: They are not checked at compile-time but at run-time.For example: **ArithmeticException**, **NullPointerException**, **ArrayIndexOutOfBoundsException**, exceptions under Error class, etc.
- **Checked Exceptions**: They are checked at compile-time. For example, **IOException**, **InterruptedException**, etc.

In this section, we will now focus on how to handle **checked exceptions** using throw and throws

throw

To throw an exception from a method or constructor, use **throw** keyword along with an instance of exception class.

Syntax:

```
1  public void method()
2  {
3     //throwing an exception
4     throw new SomeException("message");
5  }
```

If we throw an unchecked exception from a method, it is not mandatory to handle the exception or declare in throws clause. For example, **NullPointerException** or **ArithmeticException** is an unchecked exception.

throw

Example:

```
1  public double divide(double dividend, double divisor){
2    if (divisor == 0){
3       throw new ArithmeticException("Divide by Zero!");
4    }
5    return dividend / divisor;
6  }
```

The throw keyword is used to explicitly throw a single exception. When an exception is thrown, the flow of program execution transfers from the try block to the catch block. We use the throw keyword within a method.

Note: ArithmeticException is an unchecked exception. It's usually not necessary to handle unchecked exceptions.

throws

We use the throws keyword in the method declaration to declare the type of exceptions that might occur within it.

Syntax:

```
accessModifier returnType methodName()  throws ExceptionType1, ExceptionType2 … {
  // code
}
```

We can declare both types of exceptions using throws clause i.e. checked and unchecked exceptions. **But the method calling the given method must handle only checked exceptions**. Handling of unchecked exceptions is optional.

Practice Time: Let’s Code Together!

Let’s code together!🚀

Questions ?

Next:Java IO API