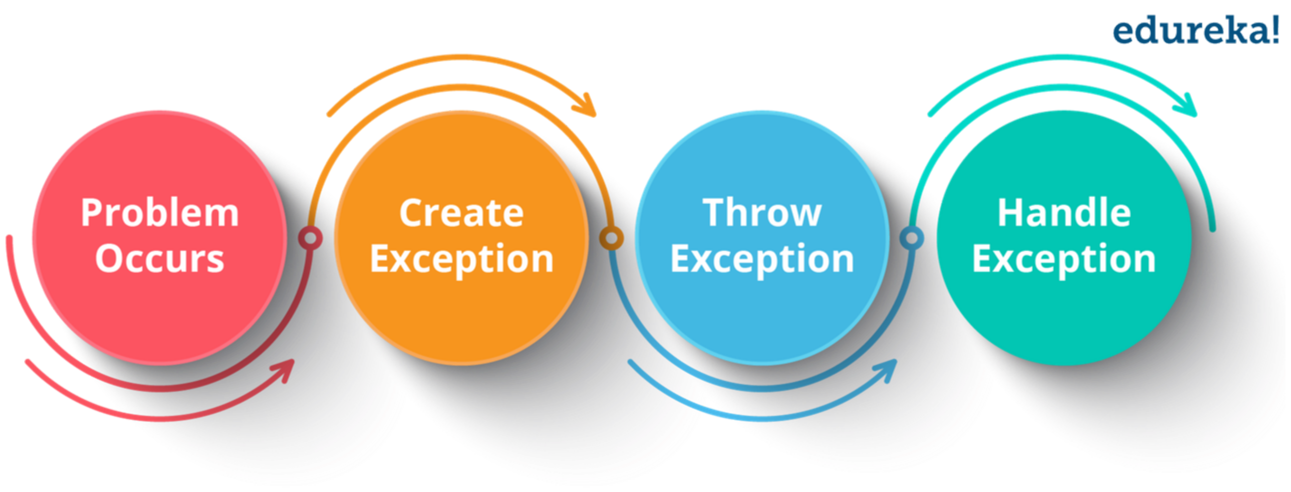
Week 11 - Exception Handling

Louis Botha, `louis.botha@tuni.fi`



Exception Handling in Java | A Beginners Guide to Java Exceptions | Edureka

What is an Exception?

An exception is generally an unwanted event that interrupts the flow of the program, most of the time it means your program crash.

Checked Exceptions
Checked Exceptions are the exceptions that we can typically foresee and plan ahead in our application. These are also exceptions that the Java Compiler requires us to either **handle-or-declare** when writing code.

The handle-or-declare rule refers to our responsibility to either declare that a method throws an exception up the call stack - without doing much to prevent it or handle the exception with our own code, which typically leads to the recovery of the program from the exceptional condition.

This is the reason why they're called *checked exceptions*. The compiler can detect them before runtime, and you're aware of their potential existence while writing code.

Unchecked Exceptions
Unchecked Exceptions are the exceptions that typically occur due to human, rather than an environmental error. These exceptions are not checked during compile-time, but at runtime, which is the reason they're also called *Runtime Exceptions*.

They can often be countered by implementing simple checks before a segment of code that could potentially be used in a way that forms a runtime exception, but more on that later on.

Errors
Errors are the most serious exceptional conditions that you can run into. They are often irrecoverable from and there's no real way to handle them. The only thing we, as developers, can do is optimize the code in hopes that the errors never occur.

Errors can occur due to human and environmental errors. Creating an infinitely recurring method can lead to a `StackOverflowError`, or a memory leak can lead to an `OutOfMemoryError`.

An example

The following code is something you might have used before. `Interger.parseInt()` takes a number in string format and and turns it into a integer.

```
class ExceptionHandling {
    public static void main(String[] args) {
        int i = Integer.parseInt("1");
        System.out.println(i);
        System.out.println("End of app, all good!");
    }
}

// Output
1
```

```
End of app, all good!
```

What will happens if it isn’t a number?

```
class ExceptionHandling {
    public static void main(String[] args) {
        int i = Integer.parseInt("one");
        System.out.println("End of app, all good!");
    }
}


// Output
❯ java ExceptionHandling
Exception in thread "main" java.lang.NumberFormatException: Fo
r input string: "one"
        at java.base/java.lang.NumberFormatException.forInputS
tring(NumberFormatException.java:67)
        at java.base/java.lang.Integer.parseInt(Integer.java:6
68)
        at java.base/java.lang.Integer.parseInt(Integer.java:7
86)
        at ExceptionHandling.main(Example.java:3)
```

The program crash and we are shown that a `NumberFormatException` occurred. Even if the text is `one`, it can’t be changed into a 1.

This is where **Exception Handling** comes in.

Exception Handling

Instead of the program just crashing, you take precautions that something can go wrong and handle the situation if it does.

throw and throws

The easiest way to take care of a compiler error when dealing with a checked exception is to simply throw it.

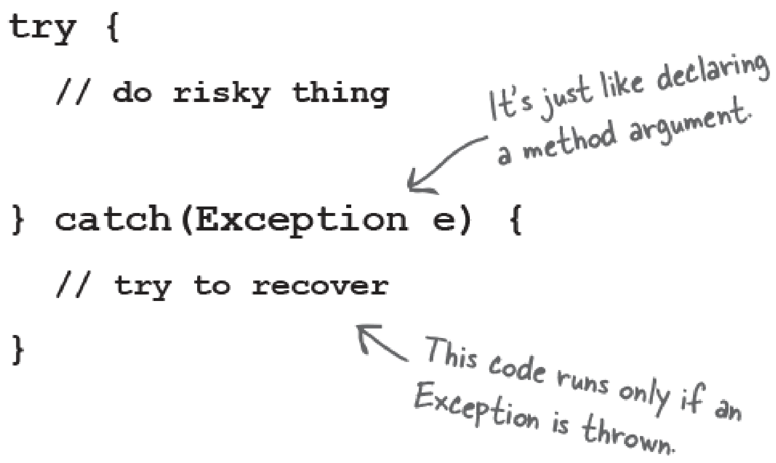
```
public File getFile(String url) throws FileNotFoundException {
    // some code
    throw new FileNotFoundException();
}
```

We are required to mark our method signature with a `throws` clause. A method can add as many exceptions as needed in its `throws` clause, and can throw them later on in the code, but doesn't have to. This method doesn't require a `return` statement, even though it defines a return type. This is because it throws an exception by default, which ends the flow of the method abruptly. The `return` statement, therefore, would be unreachable and cause a compilation error.

> *:attention: Keep in mind that anyone who calls this method also needs to follow the handle-or-declare rule.*

When throwing an exception, we can either throw a new exception, like in the preceding example, or a *caught* exception.

try/catch


```
try {
  // do risky thing

} catch(Exception e) {
  // try to recover
}
```
It's just like declaring a method argument.
This code runs only if an Exception is thrown.

Head First Java

We do this by surrounding the code that can throw an exception with a `try {}` code block and catch the exception in a `catch{}` code block

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt("one");
        } catch (NumberFormatException nfe) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        }
        System.out.println("End of app, all good!");
    }
}

// Output
❯ java ExceptionHandling
Come on, it has to at least look like a number!
```

> *:attention: Remember that the code after the place where the exception occurred will not be executed.*

The catch block is **only** executed when something goes wrong. If the code in the try code block execute and succeeds then the code in the catch block will never run.

```
 class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt("1");
        } catch (NumberFormatException nfe) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        }
        System.out.println("End of app, all good!");
    }
}

// Output
❯ java ExceptionHandling
End of app, all good!
```

> *:attention: **THROW** Exception → **CATCH** Exception*

finally

Your `try/catch` code can include a `finally` block that will execute no matter what happened.

Let’s see how finally works. Here the code doesn’t throw an exception, finally is called.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt("1");
        } catch (NumberFormatException e) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        } finally {
            System.out.println("Finally called!");
        }
        System.out.println("End of app, all good!");
    }
}

// Output
❯ java ExceptionHandling
Finally called!
End of app, all good!
```

Here the code throw an exception, catch and finally is called.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt("one");
        } catch (NumberFormatException e) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        } finally {
            System.out.println("Finally called!");
        }
        System.out.println("End of app, all good!");
    }
}

// Output
```

```
❯ java ExceptionHandling
Come on, it has to at least look like a number!
Finally called!
End of app, all good!
```

Normally it is when we have code that needs to execute regardless of whether an exception occurs, often to close the resources that were opened in the `try` block since an arising exception would skip the code closing them.

```
public String readFirstLine(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if(br != null) br.close();
    }
}
```

try-with-resources

Since Java 7, to accomplish the same as `try/finally`, you can and should use try-with-resources.

The previously complex and verbose `try/finally` block can be substituted with:

```
static String readFirstLineFromFile(String path) throws IOException {
    try(BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

It's much cleaner and it's obviously simplified by including the declaration within the parentheses of the `try` block.

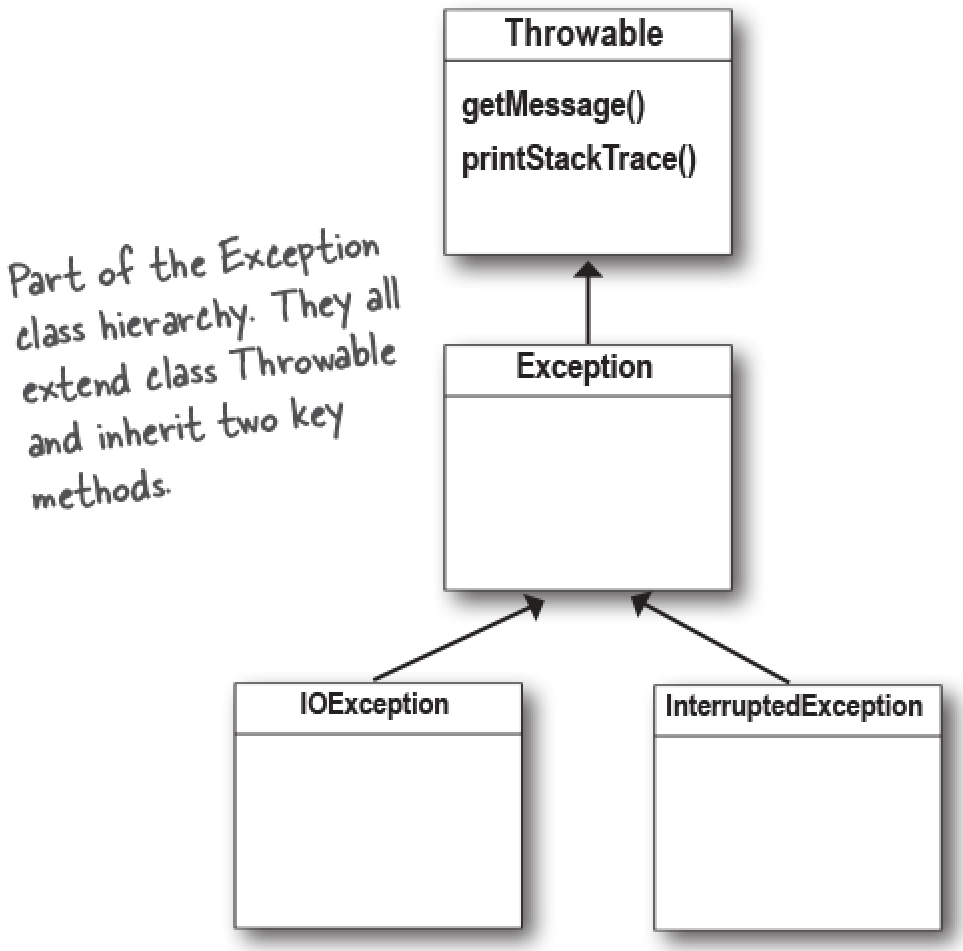
Additionally, you can include multiple resources in this block, one after another:

```
static String multipleResources(String path) throws IOExceptio
n {
    try(BufferedReader br = new BufferedReader(new FileReader
(path));
        BufferedWriter writer = new BufferedWriter(path, chars
et)) {
        // some code
    }
}
```

This way, you don't have to concern yourself with closing the resources yourself, as the *try-with-resources* block ensures that the resources will be closed upon the end of the statement.

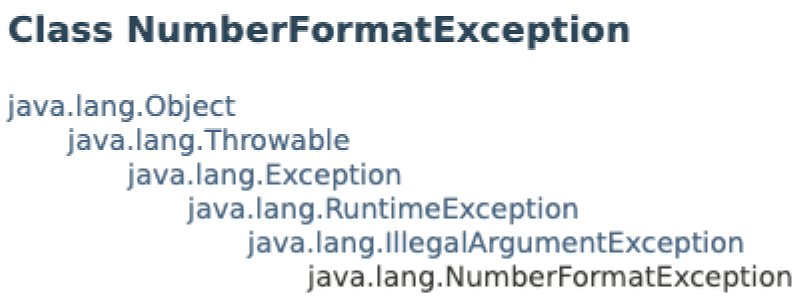
Exception Hierarchy

Exceptions belong to the exception hierarchy. All Java exception classes are a direct or indirect subclass of *Exception*, which is the top class in the exception hierarchy. We can extend this hierarchy to create our own custom exception classes.


Throwable
getMessage()
printStackTrace()
Exception
IOException
InterruptedException
Part of the Exception class hierarchy. They all extend class Throwable and inherit two key methods.

Head First Java

This means that if we catch the `Exception` which is higher up in the hierarchy that the NumberFormatException inherits from, we will also include the NumberFormatException we used earlier and many others.



Class NumberFormatException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
```

Now we will catch any type of exception.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt("one");
        } catch (Exception e) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        }
        System.out.println("End of app, all good!");
    }
}

// Output
❯ java ExceptionHandling
Come on, it has to at least look like a number!
End of app, all good!
```

Although this can be beneficial, it is not really a good idea. It is better to catch specific exceptions and add specific handling. The handling for a exception related to networking will be handled differently then say that of an exception related to writing to a file.

You don't have to catch the exceptions at the exact same line that it happens. When a exception is thrown and it is not caught immediately then it will be thrown up to the method that called it, the **CALL STACK.** The exception goes up the call stack until it is caught by a catch statement.

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
            getInt();
        } catch (Exception e) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        }
        System.out.println("End of app, all good!");
    }

    public static int getInt() {
        int i = Integer.parseInt("one");
        return i;
    }
}

// Output
❯ java ExceptionHandling
Come on, it has to at least look like a number!
End of app, all good!
```

You can have multiple catch blocks for different types of exceptions

```
class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int i = Integer.parseInt("one");
        } catch (NumberFormatException e) {
            System.out.println("Come on, it has to at least lo
ok like a number!");
        } catch (NullPointerException e) {
            System.out.println("Come on, a null?");
        }

        System.out.println("End of app, all good!");
    }
}
```

Throw Exceptions

Sometimes, we don't want to handle exceptions. In such cases, we should only concern ourselves with generating them when needed and allowing someone else, **calling our method**, to handle them appropriately.

Throwing a Checked Exception

When something goes wrong, like the number of users currently connecting to our service exceeding the maximum amount for the server to handle seamlessly, we want to `throw` an exception to indicate an exceptional situation:

```
    public void countUsers() throws TooManyUsersException {
       int numberOfUsers = 0;
           while(numberOfUsers < 500) {
               // some code
               numberOfUsers++;
        }
        throw new TooManyUsersException("The number of users e
xceeds our maximum
            recommended amount.");
    }
}
```

This code will increase `numberOfUsers` until it exceeds the maximum recommended amount, after which it will throw an exception. Since this is a checked exception, we have to add the `throws` clause in the method signature.

Throwing an Unchecked Exception

Throwing runtime exceptions usually boils down to validation of input, since they most often occur due to faulty input - either in the form of an `IllegalArgumentException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException`, or a `NullPointerException`.

```
public void setMake(String make) throws IllegalArgumentExcepti
on {
    if (make != null && !make.isEmpty()) {
        this.make = make;
    } else {
        throw new IllegalArgumentException("Invalid make valu
e.");
    }
}
```

Since we're throwing a runtime exception, there's no need to include it in the method signature, like in the example above, but it's often considered good practice to do so, at least for the sake of documentation.

Rethrowing

Rethrowing refers to the process of throwing an already caught exception, rather than throwing a new one.

```
public String readFirstLine(String url) throws FileNotFoundExc
eption {
    try {
        Scanner scanner = new Scanner(new File(url));
        return scanner.nextLine();
    } catch(FileNotFoundException ex) {
        throw ex;
    }
}
```

Custom Exceptions

We can create our own custom exceptions by declaring our own custom exception class and inheriting from the exception base class.

Here is for example a custom exception for when the amount of users exceeds out system limit.

```
public class TooManyUsersException extends Exception {
    public TooManyUsersException(String message) {
        super(message);
    }
}
```

Here is for example a custom runtime exception for when a user is not authenticated.

```
public class UserNotAuthenticatedException extends RuntimeExce
ption {
    public UserNotAuthenticatedException(String message) {
        super(message);
    }
}
```

Best and Worst Practices

Best Exception Handling Practices

- **Avoid Exceptional Conditions,** some exceptions are preventable by adding good checks and preventative measures to the code.

- **Use *try-with-resources,*** use this more concise and cleaner approach when working with resources.

- **Close resources in *try-catch-finally, i***f you're not utilising the previous advice for any reason, at least make sure to close the resources manually in the finally block.

Worst Exception Handling Practices

- **Swallowing Exceptions,** refers to the act of catching an exception and not fixing the issue.

```
public void parseFile(String filePath) {
    try {
        // some code that forms an exception
    } catch (Exception ex) {}
}
```

  This way, the compiler is satisfied since the exception is caught, but all the relevant useful information that we could extract from the exception for debugging is lost, and we didn't do anything to recover from this exceptional condition.

- **Print Stack Trace**, while it is better than simply ignoring the exception, by printing out the relevant information, this doesn't handle the exceptional condition any more than ignoring it does.

```
public void parseFile(String filePath) {
    try {
        // some code that forms an exception
    } catch(Exception ex) {
        ex.printStackTrace();
    }
}
```

- **Return in a *finally* Block,** by abruptly returning from a `finally` block, the JVM will drop the exception from the `try` block and all valuable data from it will be lost.

- **Throwing in a *finally* Block,** using `throw` in a `finally` block will drop the exception from the *try-catch* block.

- **Logging and Throwing,** don't both log and throw the exception, doing this is redundant and will simply result in a bunch of log messages which aren't really needed.

- **Catching Exception or Throwable,** unless there's a good, specific reason to catch any of these two, it's generally not advised to do so.

  Catching `Exception` will catch both checked and runtime exceptions. Runtime exceptions represent problems that are a direct result of a programming problem, and as such shouldn't be caught since it can't be reasonably expected to recover from them or handle them.

  Catching `Throwable` will catch **everything**. This includes all errors, which aren't actually meant to be caught in any way.

List of RuntimeException examples

The 10 most common examples of RuntimeExceptions in Java are:

1. ArithmeticException
2. NullPointerException
3. ClassCastException
4. DateTimeException
5. ArrayIndexOutOfBoundsException
6. NegativeArraySizeException
7. ArrayStoreException
8. UnsupportedOperationException
9. NoSuchElementException
10. ConcurrentModificationException

Some more information on these common RuntimeExceptions.

Common Exceptions

- *IOException* – This exception is typically a way to say that something on the network, filesystem, or database failed.
- *ArrayIndexOutOfBoundsException* – this exception means that we tried to access a non-existent array index, like when trying to get index 5 from an array of length 3.
- *ClassCastException* – this exception means that we tried to perform an illegal cast, like trying to convert a *String* into a *List*. We can usually avoid it by

performing defensive *instanceof* checks before casting.

- *IllegalArgumentException* – this exception is a generic way for us to say that one of the provided method or constructor parameters is invalid.
- *IllegalStateException* – This exception is a generic way for us to say that our internal state, like the state of our object, is invalid.
- *NullPointerException* – This exception means we tried to reference a *null* object. We can usually avoid it by either performing defensive *null* checks or by using *Optional.*
- *NumberFormatException* – This exception means that we tried to convert a *String* into a number, but the string contained illegal characters, like trying to convert “5f3” into a number.