Checked and Unchecked Exceptions in Java

Introduction

In this article from my free Java 8 course, I will introduce you to Checked and Unchecked Exceptions in Java.

Handling exceptions is the process by which you handle an “exceptional condition”. These situations happen rarely, and in very specific instances. You could think of these as a specific type of bug, that you expect not to happen in normal programming, but you still want to protect against.

History of Exception Handling

The mechanism of exceptions does not exist in all languages. It was introduced in the Lisp programming language in 1972, which is about 30 years after the first programing language was invented. Initially there were only unchecked exceptions, but in 1995 the Java programming language introduced the concept of “Checked Exceptions”, which forces you to handle an exception before being allowed to run the program.

Java has also been **the first and last programming language to add checked exceptions**, which *may* imply that there are some disadvantages to this mechanism that I will discuss later in the article.

Unchecked Exceptions

Unchecked exceptions are exceptions subclassed from the class `RuntimeException` and are not enforced at compile time, but rather are simply thrown at runtime. An example of such an exception is the `NullPointerException` which is thrown when you access a member of a `null` reference. This type of exception will stop your code at runtime, so typically programmers use validation in their methods to prevent these exceptions from being thrown.

Exceptional cases are generally programming errors or (other) system failures. For example, if your code contains an enum that has states like `GREEN YELLOW RED` and a switch iterating over it, the current version of the code probably deals with the enum thinking it has only those three states. If at a later time the code was changed, and now the enum had more states, this would be a situation that you could not expect. You never know what will happen to your code in the future. If you had thrown a `RuntimeException` in the case of an unknown enum value, you’d be following a “fail early” approach. Throwing the exception allows you to quickly find and eliminate what is causing the exception. This is much better than ‘hiding’ the error until it becomes a bigger problem for the system.

Checked Exceptions

As briefly mentioned before, the Java designers devised checked exceptions as a mechanism to enforce the handling of exceptions. Checked exceptions are exceptions subclassed from class `Exception` which are checked at compile time. If you do not **handle** these exceptions in your code, it will not run. The only time you do not have to handle a method that throws an exception is if the method that you are writing declares that it too throws the exception, essentially passing the handling of the exception up the hierarchy.

In my opinion, the use of checked exceptions is contradictory to the idea of exceptions. As the name implies, **an *exception* is supposed to be *exceptional*. In other words, *unexpected***. Since you are *expected* to handle a Checked Exception, this implies that you know it can happen, so a Checked Exception **cannot**, by definition, be *exceptional*. If you know that under certain conditions something can happen, then you can plan for it and directly handle it. There will be no need for exception handling in this case. For example, it is common for users to send invalid input. You can validate the user's input and show an error to the user if the input was invalid.

Validating Input

So how do we validate our arguments before actually using them? I'll show you below how to write a method that does this. I'm going to call this method `isValid()`. It accepts one argument, and returns a `boolean` that tells us whether or not the argument is valid.

```
public class CarSelector {

     public static void main(String[] arguments) throws Exception {

          CarService carService = new CarService();
          for (String argument : arguments) {
               if(isValid(argument)){
                    carService.process(argument);
               } else {
                    System.out.println("Invalid argument:" + argument);
               }
          }
     }
}
```

Example 1

Now let's write the `isValid()` method. Since our `main()` method is `static`, this method needs to be `static`. We want the method to return `true` if the argument is valid and `false` if it isn't:

```
private static boolean isValid(String argument) {
```

```
    try {
        CarState.valueOf(argument);
    } catch (IllegalStateException e) {
        return false;
    }
    return true;
}
```

Example 2

In the isValid method, we are going to have a `try`/`catch` block. First, we are going to `try` using the `valueOf()` method of the `enum` class. This method is part of the Java `enum`. It directly converts a `String` to the corresponding `enum` value. This method will throw an `IllegalStateException` in the case of an illegal value and if it does, we can `catch` that and return `false`. Otherwise our argument was valid. If you're interested in learning more about enums you can read my article about enums.

The power of Exceptions

Exception handling is actually a very powerful mechanism. When an exception is thrown, the normal program flow is interrupted. But this power comes at a price: performance cost. However, this is not an issue for *exceptional cases* because they are only expected to happen occasionally and in the case of an exception, performance is not your biggest problem.

When you are creating exceptions never use Checked Exceptions. For unexpected exceptions, use a Runtime Exception and log the error. If you have something that is expected to be problematic, such as user input, instead of throwing an exception for invalid input, you should validate and directly handle invalid input.

Throwing an Exception

Let's dive into a code example. In Example 3, we have an `enum CarState` with different state values, and a method `public static from()` that converts a `String` with a state name to an `enum` value (Since Java 7 it's been possible to use a `String` inside a switch-statement, but this is not generally recommended. However, in this case we'll use it, since the user input is in the form of a `String`).

We will add exception handling, in case the state name is not valid. We'll do that by throwing an exception from the default case of the switch-statement. To throw an exception, you write the word `throw`, followed by creating a `new Exception()` object.

```
public enum CarState {
      DRIVING, WAITING, PARKING;
```

```
	public static CarState from(String state) {
		switch (state) {
		case "DRIVING":
			return DRIVING;
		case "WAITING":
			return WAITING;
		case "PARKING":
			return PARKING;
		default:
			throw new Exception();
		}
	}
}
```

Example 3

In this case, the exception we are throwing occurs whenever there is an invalid state being sent to the `from()` method.

The exception that we threw in Example 1 is a **Checked Exception**, so we have to handle it whenever we call the `from()` method. Typically, you don't want to handle an exception in the method where it is created, as this isn't the point of an exception. Our goal is to have this exception handled when another class uses this method, so that it is protected in case the exception is thrown. To pass along this exception we add `throws Exception` to the method declaration:

```
public enum CarState {
	DRIVING, WAITING, PARKING;

	public static CarState from(String state) throws Exception {
		[...]
	}
}
```

Example 4

Now, let's write a class that calls the `from()` method. We'll call this class `CarService`. Again, I'm just gonna pass this exception up the chain, so to speak.

```
public class CarService {
     private final Logger log =
LoggerFactory.getLogger(CarService.class);

     public void process(String input) throws Exception {
          log.debug("processing car:" + input);
          CarState carState = CarState.from(input);
```

```
    }
}
```

Example 5

Ok, now we get the next method in our code, but again, I'm just gonna write `throws Exception`.

```
public class CarSelector {

    public static void main(String[] arguments) throws Exception {

        CarService carService = new CarService();
        for (String argument : arguments) {
            carService.process(argument);
        }
    }
}
```

Example 6

As you see, we ended up adding the exception in every calling method, all the way up to our `main()` method. The exception is not handled, and if it occurs, the program will stop with an error. This is a typical situation, in which you end up adding the `throws` statement to each and every function in your code. This is why a checked exception is frankly not very helpful; we never even bother to handle it and we end up getting the same stack trace printed as our output.

Instead of doing this, we could write an unchecked exception. If we write an unchecked exception, we can remove every single `throws Exception` that we have just written, and replace our `Exception()` throw with `throw new RuntimeException();` With this exception, we can also provide some information on why our exception was thrown. In this case, I'm going to tell the user that they sent us an unknown state, and then tell them which state it was that caused this exception to be thrown.

```
public enum CarState {
    DRIVING, WAITING, PARKING;

    public static CarState from(String state) {
        switch (state) {
        case "DRIVING":
            return DRIVING;
        case "WAITING":
            return WAITING;
        case "PARKING":
```

```
                return PARKING;
            default:
                throw new RuntimeException("Unknown State: " + state);
            }
        }
}
```

Example 7

Handling an Exception

Let's say we don't want our code to stop running when the user sends us an unknown state. We could handle the exception by surrounding our `from()` method call with a `try/catch` block. Usually, you want to have this try/catch block **as early on as possible** in the code, so wherever the problem will potentially be caused, you should surround that call in the try/catch. In this case, that is in our `main()` method.

First, we wrap our method in a try block. After the try block ends, we add a catch block that accepts the Exception as its argument. All exceptions have a method called `printStackTrace()` that prints a list of all the methods called in reverse order. This is great for debugging because we can see exactly where in each class the exception happened.

In Example 8's try block, if an exception occurs, none of the lines after the line that caused the exception will be executed. If there isn't an exception, then you can safely assume that the call succeeded in the following lines. Because of this, we can write our try block of the code as if everything works, while handling the exceptions separately.

```
public class CarSelector {

    public static void main(String[] arguments){

        CarService carService = new CarService();
        for (String argument : arguments) {
            try{
                carService.process(argument);
            }catch (RuntimeException e){
                e.printStackTrace();
            }
        }
    }
}
```

Example 8

The finally-Block

Exceptions, as I've noted, interrupt the normal program flow, but often we have code that we want to run whether or not our try block works. For example, when handling resources such as IO or database connections, we want to properly free them in the case of an exception.

In fact, it is a classic bug for programs to leak resources or improperly close them when exceptions occur, because the code that handles freeing them is interrupted, or isn't executed.

For this reason Java defines the `finally` keyword, to add an optional `finally` block to the `try/catch` construct.

```
try{
     carService.process(argument);
} catch (RuntimeException e){
     LOG.error(e.getMessage(), e);
} finally {
     System.out.println("I print no matter what");
}
```

Example 9

With a valid argument, the code in the `try` block will be executed. With an invalid argument the normal program flow will be interrupted, and the code in the `try` block will not be executed, but we will instead log our error. No matter what, the line *"I print no matter what*" will still print.

Ways to Handle Exceptions

Many Java books and tutorials never teach how to properly handle an exception and append a comment along the lines of "do proper exception handling here", without explaining what to "properly" write. This might be why many programmers often print a stack trace instead of choosing a more beneficial way to handle their errors.

`e.printStackTrace` prints the exception's stack trace to the standard error output, which usually is the raw console. In simple terms, the stack trace shows the order of nested method calls that lead to the exception, which can be helpful to find the cause of the exception. In other words, this is a basic attempt to generically signal that an error occured, along with the potential source of the error.

This can be useful in very specific situations, like for small developer tools run from the console. Usually however, this approach is not optimal. If possible, you should always handle an exception. For example, if reading a value from a database fails, it might be possible to return a cached value instead. If you can't handle the exception, the minimum you can do is to log the exception as well as keep hold of the complete state the system was in when the exception happened - as close as possible.

Generally, you want to have information such as the time when the error occurred, the name of the system, the full name of class and the method, and the exact line number where the error occurred in the code. Additionally, you should log the status of relevant objects, such as the parameters used by the faulty method. With the help of a logging framework like Logback, this can be achieved without much effort. It can be configured to log to a file or a database, which will permanently persist the error. To learn more about logging, read my article on Logging with SLF4J and LOGBack.

Using a logging framework for error notification is generally preferable to directly printing out the error to the console, as it is a more flexible and powerful way of getting hold of the cause of an exception and to persist it to a storage medium such as a file or a database.

Thanks for reading!