Theory: Custom exceptions

35 minutes

Skip this topic | Start practicing

Java exceptions cover a lot of standard exceptions we have to deal with in programming. However, sometimes we might need to create them ourselves. One of the main reasons why is to handle logical exceptions which are specific to your program. Imagine you are implementing a client-server application. The server processes user information and validates its correctness, or throws an exception. An essential condition for the server to work properly is that it should validate all the fields with user data and not stop at the first incorrect one. If some fields turn out to be incorrect, we want to receive a detailed report about what went wrong. In this case, a standard exception like `IllegalArgumentException` will not be enough for us, which means it's time to create a custom exception.

§1. How to create and throw a custom exception

To create a custom exception you need to extend the `Exception` (checked) or `RuntimeException` (unchecked) classes.

Here is an example:

```
1  public class MyAppException extends Exception {
2
3      public MyAppException(String msg) {
4          super(msg);
5      }
6
7      public MyAppException(Exception cause) {
8          super(cause);
9      }
10 }
```

In the example above, a new class of exceptions is declared. It is a checked exception because it extends the `Exception` class. The declared class has two constructors for creating instances, and they call the corresponding constructor of the base class.

Now, we can throw an instance of the class:

```
1  public static void someMethod() throws MyAppException {
2      throw new MyAppException("Something bad");
3  }
```

For more information about throwing exceptions, please refer to our topic on the subject.

Now let's learn some rules of creating custom exceptions.

§2. Best practices for custom exceptions

First things first, make sure that your application will benefit from creating a custom exception. Otherwise, use standard Java exceptions.

Secondly, follow the naming convention — end the class name with "Exception", for example `MyAppException`.

Also, provide the constructor that sets a cause in case your program catches a standard exception before throwing a custom one.

For example, let's look at the code snippet below. Here we capture the root cause of the exception with the `Throwable` argument, which is passed to the parent class constructor.

2 required topics

- ✕ Exception handling
- ✕ Throwing exceptions

Table of contents:

↑ Custom exceptions

§1. How to create and throw a custom exception

§2. Best practices for custom exceptions

§3. Conclusion

Discussion

```
public class CustomException extends Exception {

   public CustomException(String message, Throwable cause) {

       super(message, cause);

   }

}
```

Is creating a custom exception always a good idea? Although the custom exception feature greatly enhances the error handling mechanism, its use is not always justified. We advise you to use standard exceptions whenever possible for a number of reasons, such as:

- Standard exceptions are widely known by other programmers. One can understand the type of problem just by looking at the name of the exception.
- By opting for standard exceptions, you follow the reusability principle. It makes your code clearer and more professional.

§3. Conclusion

Custom exceptions are a great tool for handling inconsistencies in your program. In this topic, we learned how to create and throw them, along with some best practices to follow. However, we highly recommend creating a custom exception only when it is justified. Standard Java exceptions are often a safer and no less efficient choice.