3 MAY 2018 / #JAVA

# Why you should ignore exceptions in Java and how to do it correctly

by Rainer Hahnekamp

In this article, I will show how to ignore checked exceptions in Java. I pattern to resolve this issue. Then I will present some libraries for that purpose.

## Checked and Unchecked Exceptions

In Java, a method can force its caller to deal with the occurrence of potential exceptions. The caller can use the try/catch clause, where the try contains the actual code and catch contains the code to execute when the exception occurs.

Alternatively, the caller can pass on that burden to its **parent caller**. This can go upwards until the main method is reached. If the main method also passes on the exception, the application will crash when an exception happens.

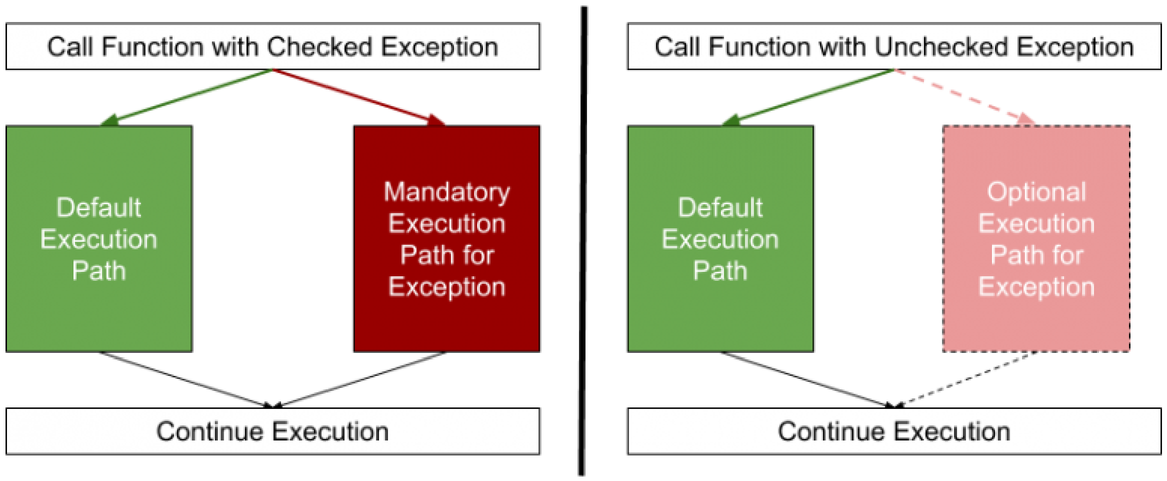
In the case of an exception, there are many scenarios where the application cannot continue to run and needs to stop. There are no alternative paths. Unfortunately, that means Java forces us to write code for a situation where the application shouldn't run anymore. Quite useless!

An option is to minimise that boilerplate code. We can wrap the exception into a **RuntimeException**, which is an unchecked exception. This has the effect that, even though the application still crashes, we don't have to provide any handling code.

By no means do we log the exception and let the application continue like nothing has happened. It is possible, but is similar to opening Pandora's Box.

We call these exceptions, for which we have to write extra code, **checked exceptions.** The others of the **RuntimeException** type we call

Donate


Call Function with Checked Exception
Default Execution Path
Mandatory Execution Path for Exception
Continue Execution
Call Function with Unchecked Exception
Default Execution Path
Optional Execution Path for Exception
Continue Execution

Checked and Unchecked Exceptions

## Why Checked Exceptions at all?

We can find lots of checked exceptions in third-party libraries, and even in the Java Class Library itself. The reason is pretty straightforward. A library vendor cannot predict in which context the developer will use their code.

Logically, they don’t know if our application has alternative paths. So they leave the decision to us. Their responsibility is to “label” methods that can potentially throw exceptions. Those labels give us the chance to implement counter-actions.

A good example is the connection to a database. The library vendor marks the connection retrieval method with an exception. If we use the database as a cache, we can send our queries directly to our primary database. This is the alternative path.

If our database is not the cache, there is no way the application can continue to run. And it’s OK if the application crashes:

Donate

https://imgflip.com/i/26h3xi

## A lost database

Let’s put our theoretical example to real code:

```
public DbConnection getDbConnection(String username, String passwoı
```

The database is not used as a cache. In the event of a lost connection, we need to stop the application at once.

As described above, we wrap the **DbConnectionException** into a **RuntimeException**.

The required code is relatively verbose and always the same. This creates lots of duplication and decreases the readability.

## The RuntimeException Wrapper

We can write a function to simplify this. It should wrap a **RuntimeException** over some code and return the value. We cannot simply pass code in Java. The function must be part of a class or interface. Something like this:

```
public interface RuntimeExceptionWrappable<T> {  T execute() throws
```

The **RuntimeException** wrapping has been extracted into its own class. In terms of software design, this might be the more elegant solution. Still, given the amount of code, we can hardly say the situation got better.

With Java 8 lambdas, things got easier. If we have an interface with one method only, then we just write the specific code of that method. The compiler does the rest for us. The unnecessary or "syntactic sugar code" to create a specific or anonymous class is not required any more. That's the basic use case for Lambdas.

Donate

```
@FunctionalInterfacepublic interface RuntimeExceptionWrappable<T> 
```

The difference is quite clear: the code is more concise.

## Exceptions in Streams & Co.

`RuntimeExceptionWrappable` is a very generic interface. It is just a function that returns a value. Use cases for that function, or its variations, appear all over. For our convenience, Java's Class Library has a set of such common Interfaces built-in. They are in the package `java.util.function` and are better known as the "Functional Interfaces." Our `RuntimeExceptionWrappable` is similar to `java.util.function.Supplier<` ;T>.

These interfaces form the prerequisite of the powerful Stream, Optional, and other features which are also part of Java 8. In particular, Stream comes with a lot of different methods for processing collections. Many of these methods have a "Functional Interface" as parameter.

Let's quickly switch the use case. We have a list of URL strings that we want to map into a list of objects of type java.net.URL.

The following code **does not compile**:

```
public List<URL> getURLs() {  return Stream    .of("https://www.hal
```

There is a big problem when it comes to exceptions. The Interfaces defined in `java.util.function` don't throw exceptions. That's why our method **createURL** doesn't have the same signature as `java.util.function.Function`, which is the parameter of the map method.

What we can do is to write the try/catch block inside the lambda:

```
public List<URL> getURLs() {  return Stream    .of(“https://www.hal
```

This compiles, but doesn't look nice either. We can now take a step further and write a wrapper function along a new interface similar to `RuntimeExceptionWrappable`` :

```
@FunctionalInterfacepublic interface RuntimeWrappableFunction<T, R:
```

And apply it to our Stream example:

```
public List<URL> getURLs() {  return Stream    .of(“https://www.hal
```

Great! Now we have a solution, where we can:

- run code without catching checked exceptions, and on.

## SneakyThrow to the rescue

The SneakyThrow library lets you skip copying and pasting the code snippets from above. Full disclosure: I am the author.

SneakyThrow comes with two static methods. One runs code without catching checked exceptions. The other method wraps an exception-throwing lambda into one of the Functional Interfaces:

```
//SneakyThrow returning a resultpublic DbConnection getDbConnectio
```

## Alternative Libraries

### ThrowingFunction

```
//ThrowingFunction returning a resultpublic DbConnection getDbConne
```

In contrast to SneakyThrow, ThrowingFunction can't execute code directly. Instead, we have to wrap it into a Supplier and call the Supplier afterwards. This approach can be more verbose than SneakyThrow.

If you have multiple unchecked Functional Interfaces in one class, then you have to write the full class name with each static method. This is because unchecked does not work with method overloading.

On the other hand, ThrowingFunction provides you with more features than SneakyThrow. You can define a specific exception you want to wrap. It is also possible that your function returns an Optional, otherwise known as "lifting."

I designed SneakyThrow as an opinionated wrapper of ThrowingFunction.

## Vavr

Vavr, or "JavaSlang," is another alternative. In contrast to SneakyThrow and ThrowingFunction, it provides a complete battery of useful features that enhance Java's functionality.

For example, it comes with pattern matching, tuples, its own Stream and much more. If you haven't heard of it, it is definitely worth a look. Prepare to invest some time in order to understand its full potential.

```
//Vavr returning a resultpublic DbConnection getDbConnection(Strin
```

## Project Lombok

Such a list of libraries is not complete without mentioning Lombok. Like Vavr, it offers much more functionality than just wrapping checked exceptions. It is a code generator for boilerplate code and creates full Java Beans, Builder objects, logger instances, and much more.

Lombok achieves its goals by bytecode manipulation. Therefore, we

Donate

@SneakyThrows is Lombok's annotation for manipulating a function with a checked exception into one that doesn't. This approach doesn't depend on the usage of lambdas, so you can use it for all cases. It is the least verbose library.

Please keep in mind that Lombok manipulates the bytecode which can cause problems with other tooling you might use.

```
//Lombok returning a result@SneakyThrowspublic DbConnection getDbCo
```

## Further Reading

The code is available on GitHub

- https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html
- https://www.artima.com/intv/handcuffs.html
- http://www.informit.com/articles/article.aspx?p=2171751&seqNum=3
- https://github.com/rainerhahnekamp/sneakythrow
- https://projectlombok.org/features/SneakyThrows
- https://github.com/pivovarit/ThrowingFunction
- https://github.com/vavr-io/vavr
- http://www.baeldung.com/java-lambda-exceptions

*Originally published at www.rainerhahnekamp.com on March 17, 2018.*

Donate

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

| Our Nonprofit | Trending Guides |
|---|---|
| About | 2019 Web Developer Roadmap |
| Alumni Network | Python Tutorial |
| Open Source | CSS Flexbox Guide |
| Shop | JavaScript Tutorial |
| Support | Python Example |
| Sponsors | HTML Tutorial |
| Academic Honesty | Linux Command Line Guide |