Exception Handling in Java

Last updated: May 11, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

> Reviewed by: Josh Cummings (https://www.baeldung.com/editor/josh-cummings)

Core Java (https://www.baeldung.com/category/java/core-java)

Definition (https://www.baeldung.com/tag/definition)

Exception (https://www.baeldung.com/tag/exception)

ung.com/scala/exception-	Kotlin (https://www.baeldung.com/kotlin/exception-handling)

1. Overview (/)

In this tutorial, we'll go through the basics of exception handling in Java as well as some of its gotchas.

2. First Principles

2.1. What Is It?

To better understand exceptions and exception handling, let's make a real-life comparison.

Imagine that we order a product online, but while en-route, there's a failure in delivery. A good company can handle this problem and gracefully re-route our package so that it still arrives on time.

Likewise, in Java, the code can experience errors while executing our instructions. Good *exception handling* can handle errors and gracefully re-route the program to give the user still a positive experience.

2.2. Why Use It?

We usually write code in an idealized environment: the filesystem always contains our files, the network is healthy, and the JVM always has enough memory. Sometimes we call this the "happy path".

In production, though, filesystems can corrupt, networks break down, and JVMs run out of memory. The wellbeing of our code depends on how it deals with "unhappy paths".

We must handle these conditions because they affect the flow of the application negatively and form *exceptions*:

```
public static List<Player> getPlayers() throws IOException {
    Path path = Paths.get("players.dat");
    List<String> players = Files.readAllLines(path);

    return players.stream()
      .map(Player::new)
      .collect(Collectors.toList());
}
```

This code chooses not to handle the *IOException*, passing it up the call stack instead. In an idealized environment, the code works fine.

But what might happen in production if *players.dat* is missing?

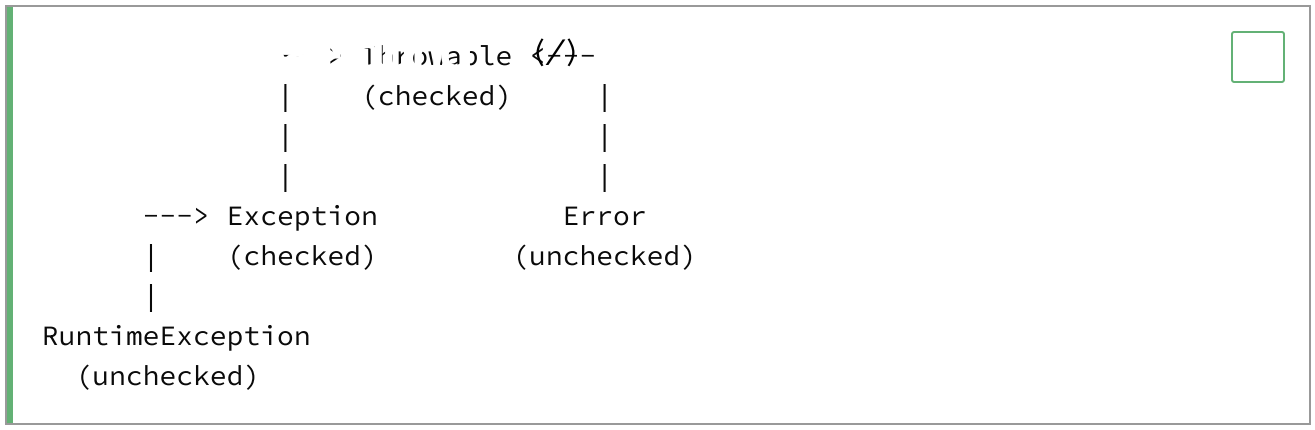
```
Exception in thread "main" java.nio.file.NoSuchFileException: players.dat
<-- players.dat file doesn't exist
    at sun.nio.fs.WindowsException.translateToIOException(Unknown Source)
    at sun.nio.fs.WindowsException.rethrowAsIOException(Unknown Source)
    // ... more stack trace
    at java.nio.file.Files.readAllLines(Unknown Source)
    at java.nio.file.Files.readAllLines(Unknown Source)
    at Exceptions.getPlayers(Exceptions.java:12) <-- Exception arises in
getPlayers() method, on line 12
    at Exceptions.main(Exceptions.java:19) <-- getPlayers() is called by
main(), on line 19
```

Without handling this exception, an otherwise healthy program may stop running altogether! We need to make sure that our code has a plan for when things go wrong.

Also note one more benefit here to exceptions, and that is the stack trace itself. Because of this stack trace, we can often pinpoint offending code without needing to attach a debugger.

3. Exception Hierarchy

Ultimately, *exceptions* are just Java objects with all of them extending from *Throwable*:



There are three main categories of exceptional conditions:

- Checked exceptions
- Unchecked exceptions / Runtime exceptions
- Errors

Runtime and unchecked exceptions refer to the same thing. We can often use them interchangeably.

3.1. Checked Exceptions

Checked exceptions are exceptions that the Java compiler requires us to handle. We have to either declaratively throw the exception up the call stack, or we have to handle it ourselves. More on both of these in a moment.

Oracle's documentation (https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) tells us to use checked exceptions when we can reasonably expect the caller of our method to be able to recover.

A couple of examples of checked exceptions are *IOException* and *ServletException.*

3.2. Unchecked Exceptions

Unchecked exceptions are exceptions that the Java compiler does *not* require us to handle.

Simply put, if we create an exception that extends *RuntimeException*, it will be unchecked; otherwise, it will be checked.

And while this sounds convenient, Oracle's documentation (https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html) tells us that there are good reasons for both concepts, like differentiating between a situational error (checked) and a usage error (unchecked).

Some examples of unchecked exceptions are *NullPointerException*, *IllegalArgumentException*, and *SecurityException*.

3.3. Errors

Errors represent serious and usually irrecoverable conditions like a library incompatibility, infinite recursion, or memory leaks.

And even though they don't extend *RuntimeException*, they are also unchecked.

In most cases, it'd be weird for us to handle, instantiate or extend *Errors*. Usually, we want these to propagate all the way up.

A couple of examples of errors are a *StackOverflowError* and *OutOfMemoryError*.

4. Handling Exceptions

In the Java API, there are plenty of places where things can go wrong, and some of these places are marked with exceptions, either in the signature or the Javadoc:

```
/**
 * @exception FileNotFoundException ...
 */
public Scanner(String fileName) throws FileNotFoundException {
   // ...
}
```

As stated a little bit earlier, when we call these "risky" methods, we *must* handle the checked exceptions, and we *may* handle the unchecked ones. Java gives us several ways to do this:

4.1. *throws* (/)

The simplest way to "handle" an exception is to rethrow it:

```
public int getPlayerScore(String playerFile)
  throws FileNotFoundException {

    Scanner contents = new Scanner(new File(playerFile));
    return Integer.parseInt(contents.nextLine());
}
```

Because *FileNotFoundException* is a checked exception, this is the simplest way to satisfy the compiler, but **it does mean that anyone that calls our method now needs to handle it too!**

parseInt can throw a *NumberFormatException*, but because it is unchecked, we aren't required to handle it.

4.2. *try-catch*

If we want to try and handle the exception ourselves, we can use a *try-catch* block. We can handle it by rethrowing our exception:

```
public int getPlayerScore(String playerFile) {
    try {
        Scanner contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException noFile) {
        throw new IllegalArgumentException("File not found");
    }
}
```

Or by performing recovery steps:

```
public in  ge  ayerS  e(Str ng playerFile) {
    try {
        Scanner contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } catch ( FileNotFoundException noFile ) {
        logger.warn("File not found, resetting score.");
        return 0;
    }
}
```

4.3. *finally*

Now, there are times when we have code that needs to execute regardless of whether an exception occurs, and this is where the *finally* keyword comes in.

In our examples so far, there 's been a nasty bug lurking in the shadows, which is that Java by default won't return file handles to the operating system.

Certainly, whether we can read the file or not, we want to make sure that we do the appropriate cleanup!

Let's try this the "lazy" way first:

```
public int getPlayerScore(String playerFile)
  throws FileNotFoundException {
    Scanner contents = null;
    try {
        contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } finally {
        if (contents != null) {
            contents.close();
        }
    }
}
```

Here, the *finally* block indicates what code we want Java to run regardless of what happens with trying to read the file.

Even if a *FileNotFoundException* is thrown up the call stack, Java will call the contents of *finally* before doing that.

We can also both handle the exception *and* make sure that our resources get closed:

```
public int getPlayerScore(String playerFile) {
    Scanner contents;
    try {
        contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException noFile ) {
        logger.warn("File not found, resetting score.");
        return 0;
    } finally {
        try {
            if (contents != null) {
                contents.close();
            }
        } catch (IOException io) {
            logger.error("Couldn't close the reader!", io);
        }
    }
}
```

Because *close* is also a "risky" method, we also need to catch its exception!

This may look pretty complicated, but we need each piece to handle each potential problem that can arise correctly.

4.4. *try*-with-resources

Fortunately, as of Java 7, we can simplify the above syntax when working with things that extend *AutoCloseable*:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile))) {
      return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException e ) {
      logger.warn("File not found, resetting score.");
      return 0;
    }
}
```

When we place references that are *AutoClosable* in the *try* declaration, then we don't need to close the resource ourselves.

We can still use a *finally* block, though, to do any other kind of cleanup we want.

Check out our article dedicated to *try*-with-resources (/java-try-with-resources) to learn more.

4.5. Multiple *catch* Blocks

Sometimes, the code can throw more than one exception, and we can have more than one *catch* block handle each individually:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile))) {
        return Integer.parseInt(contents.nextLine());
    } catch (IOException e) {
        logger.warn("Player file wouldn't load!", e);
        return 0;
    } catch (NumberFormatException e) {
        logger.warn("Player file was corrupted!", e);
        return 0;
    }
}
```

Multiple catches give us the chance to handle each exception differently, should the need arise.

Also note here that we didn't catch *FileNotFoundException*, and that is because it *extends IOException*. Because we're catching *IOException*, Java will consider any of its subclasses also handled.

Let's say, though, that we need to treat *FileNotFoundException* differently from the more general *IOException*:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile)) ) {
        return Integer.parseInt(contents.nextLine());
    } catch (FileNotFoundException e) {
        logger.warn("Player file not found!", e);
        return 0;
    } catch (IOException e) {
        logger.warn("Player file wouldn't load!", e);
        return 0;
    } catch (NumberFormatException e) {
        logger.warn("Player file was corrupted!", e);
        return 0;
    }
}
```

Java lets us handle subclass exceptions separately, **remember to place them higher in the list of catches.**

4.6. Union *catch* Blocks

When we know that the way we handle errors is going to be the same, though, Java 7 introduced the ability to catch multiple exceptions in the same block:

```
public int getPlayerScore(String playerFile) {
    try (Scanner contents = new Scanner(new File(playerFile))) {
        return Integer.parseInt(contents.nextLine());
    } catch (IOException | NumberFormatException e) {
        logger.warn("Failed to load score!", e);
        return 0;
    }
}
```

5. Throwing Exceptions

If we don't want to handle the exception ourselves or we want to generate our exceptions for others to handle, then we need to get familiar with the *throw* keyword.

Let's say that we have the following checked exception we've created ourselves:

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

and we have a method that could potentially take a long time to complete:

```
public List<Player> loadAllPlayers(String playersFile) {
    // ... potentially long operation
}
```

5.1. Throwing a Checked Exception

Like returning from a method, we can *throw* at any point.

Of course, we should throw when we are trying to indicate that something has gone wrong:

```
public List<Player> loadAllPlayers(String playersFile) throws
TimeoutException {
    while ( !tooLong ) {
        // ... potentially long operation
    }
    throw new TimeoutException("This operation took too long");
}
```

Because *TimeoutException* is checked, we also must use the *throws* keyword in the signature so that callers of our method will know to handle it.

5.2. *Throw*ing an Unchecked Exception

If we want to do something like, say, validate input, we can use an unchecked exception instead:

```
public List<Player> loadAllPlayers(String playersFile) throws
TimeoutException {
    if(!isFilenameValid(playersFile)) {
        throw new IllegalArgumentException("Filename isn't valid!");
    }

    // ...
}
```

Because *IllegalArgumentException* is unchecked, we don't have to mark the method, though we are welcome to.

Some mark the method anyway as a form of documentation.

5.3. Wrapping and Rethrowing

We can also choose to rethrow an exception we've caught:

```
public List<Player> loadAllPlayers(String playersFile)
  throws IOException {
    try {
        // ...
    } catch (IOException io) {
        throw io;
    }
}
```

Or do a wrap and rethrow:

```
public List<Player> loadAllPlayers(String playersFile)
  throws PlayerLoadException {
    try {
        // ...
    } catch (IOException io) {
        throw new PlayerLoadException(io);
    }
}
```

This can be nice for consolidating many different exceptions into one.

5.4. Rethrowing *Throwable* or *Exception*

Now for a special case.

If the only possible exceptions that a given block of code could raise are *unchecked* exceptions, then we can catch and rethrow *Throwable* or *Exception* without adding them to our method signature:

```
public List<Player> loadAllPlayers(String playersFile) {
    try {
        throw new NullPointerException();
    } catch (Throwable t) {
        throw t;
    }
}
```

While simple, the above code can't throw a checked exception and because of that, even though we are rethrowing a checked exception, we don't have to mark the signature with a *throws* clause.

This is handy with proxy classes and methods. More about this can be found here (http://4comprehension.com/sneakily-throwing-exceptions-in-lambda-expressions-in-java/).

5.5. Inheritance

When we mark methods with a *throws* keyword, it impacts how subclasses can override our method.

In the circumstance where our method throws a checked exception:

```
public class Exceptions {
    public List<Player> loadAllPlayers(String playersFile)
      throws TimeoutException {
        // ...
    }
}
```

A subclass can have a "less risky" signature:

```
public class FewerExceptions extends Exceptions {
    @Override
    public List<Player> loadAllPlayers(String playersFile) {
        // overridden
    }
}
```

But not a "*more* riskier" signature:

```
public class MoreExceptions extends Exceptions {
    @Override
    public List<Player> loadAllPlayers(String playersFile) throws
MyCheckedException {
        // overridden
    }
}
```

This is because contracts are determined at compile time by the reference type. If I create an instance of *MoreExceptions* and save it to *Exceptions*:

```
Exceptions exceptions = new MoreExceptions();
exceptions.loadAllPlayers("file");
```

Then the JVM will only tell me to *catch* the *TimeoutException*, which is wrong since I've said that *MoreExceptions#loadAllPlayers* throws a different exception.

Simply put, subclasses can throw *fewer* checked exceptions than their superclass, but not *more*.

6. Anti-Patterns

6.1. Swallowing Exceptions

Now, there's one other way that we could have satisfied the compiler:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch (Exception e) {} // <== catch and swallow
    return 0;
}
```

The above is called swallowing an exception. Most of the time, it would be a little mean for us to do this because it doesn't address the issue *and* it keeps other code from being able to address the issue, too.

There are times when there's a checked exception that we are confident will just never happen. **In those cases, we should still at least add a comment stating that we intentionally ate the exception**:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch (IOException e) {
        // this will never happen
    }
}
```

Another way we can "swallow" an exception is to print out the exception to the error stream simply:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch (Exception e) {
        e.printStackTrace();
    }
    return 0;
}
```

We've improved our situation a bit by a least writing the error out somewhere for later diagnosis.

It'd be better, though, for us to use a logger:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch (IOException e) {
        logger.error("Couldn't load the score", e);
        return 0;
    }
}
```

While it's very convenient for us to handle exceptions in this way, we need to make sure that we aren't swallowing important information that callers of our code could use to remedy the problem.

Finally, we can inadvertently swallow an exception by not including it as a cause when we are throwing a new exception:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch (IOException e) {
        throw new PlayerScoreException();
    }
}
```

Here, we pat ourselves on the back for alerting our caller to an error, but **we fail to include the *IOException* as the cause.** Because of this, we've lost important information that callers or operators could use to diagnose the problem.

We'd be better off doing:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch (IOException e) {
        throw new PlayerScoreException(e);
    }
}
```

Notice the subtle difference of including *IOException* as the *cause* of *PlayerScoreException*.

6.2. Using *return* in a *finally* Block

Another way to swallow exceptions is to *return* from the *finally* block. This is bad because, by returning abruptly, the JVM will drop the exception, even if it was thrown from by our code:

```
public int getPlayerScore(String playerFile) {
    int score = 0;
    try {
        throw new IOException();
    } finally {
        return score; // <== the IOException is dropped
    }
}
```

According to the Java Language Specification (https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20.2):

> If execution of the try block completes abruptly for any other reason *R*, then the finally block is executed, and then there is a choice.
>
> If the *finally* block completes normally, then the try statement completes abruptly for reason R.
>
> If the *finally* block completes abruptly for reason S, then the try statement completes abruptly for reason S (and reason R is discarded).

6.3. Using *throw* in a *finally* Block

Similar to using *return* in a *finally* block, the exception thrown in a *finally* block will take precedence over the exception that arises in the catch block.

This will "erase" the original exception from the *try* block, and we lose all of that valuable information:

```
public int getPlayerScore(String playerFile) {
    try {
        // ...
    } catch ( IOException io ) {
        throw new IllegalStateException(io); // <== eaten by the finally
    } finally {
        throw new OtherException();
    }
}
```

6.4. Using *throw* as a *goto*

Some people also gave into the temptation of using *throw* as a *goto* statement:

```
public void doSomething() {
    try {
        // bunch of code
        throw new MyException();
        // second bunch of code
    } catch (MyException e) {
        // third bunch of code
    }
}
```

This is odd because the code is attempting to use exceptions for flow control as opposed to error handling.

7. Common Exceptions and Errors

Here are some common exceptions and errors that we all run into from time to time:

7.1. Checked Exceptions

- *IOException* – This exception is typically a way to say that something on the network, filesystem, or database failed.

7.2. RuntimeExceptions

- *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.

7.3. Errors

- *StackOverflowError* – this exception means that the stack trace is too big. This can sometimes happen in massive applications; however, it usually means that we have some infinite recursion happening in our code.
- *NoClassDefFoundError* – this exception means that a class failed to load either due to not being on the classpath or due to failure in static initialization.
- *OutOfMemoryError* – this exception means that the JVM doesn't have any more memory available to allocate for more objects. Sometimes, this is due to a memory leak.

8. Conclusion

In this article, we've gone through the basics of exception handling as well as some good and poor practice examples.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

(/)

OUR PARTNERS (/PARTNERS/)

PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)

EBOOKS (/LIBRARY/)

FAQ (/LIBRARY/FAQ)

BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)

PRIVACY MANAGER