4
Exception Handling

In *Chapter 1*, *Getting Started with Java 17*, we briefly introduced exceptions. In this chapter, we will treat this topic more systematically. There are two kinds of exceptions in Java: checked and unchecked. We'll demonstrate each of them, and the differences between the two will be explained. Additionally, you will learn about the syntax of the Java constructs related to exception handling and the best practices to address (that is, handle) those exceptions. The chapter will end on the related topic of assertion statements, which can be used to debug the code in production.

In this chapter, we will cover the following topics:

- The Java exceptions framework
- Checked and unchecked (runtime) exceptions
- The `try`, `catch`, and `finally` blocks
- The `throws` statement
- The `throw` statement
- The `assert` statement
- Best practices of exception handling

So, let's begin!

Technical requirements

To be able to execute the code examples that are provided in this chapter, you will need the following:

- A computer with a Microsoft Windows, Apple macOS, or Linux operating system
- Java SE version 17, or later
- An IDE or code editor that you prefer

The instructions for how to set up a Java SE and IntelliJ IDEA editor were provided in *Chapter 1*, *Getting Started with Java 17*. The files with the code examples for this chapter are available on GitHub in the `https://github.com/PacktPublishing/Learn-Java-17-Programming.git` repository. Please search in the `examples/src/main/java/com/packt/learnjava/ch04_exceptions` folder.

The Java exceptions framework

As described in *Chapter 1*, *Getting Started with Java 17*, an unexpected condition can cause the **Java Virtual Machine** (**JVM**) or the application code to create and throw an exception object. As soon as that happens, the control flow is transferred to the `catch` clause, that is, if the exception was thrown inside a `try` block. Let's look at an example. Consider the following method:

```
void method(String s){
    if(s.equals("abc")){
        System.out.println("Equals abc");
    } else {
        System.out.println("Not equal");
    }
}
```

If the input parameter value is `null`, you could expect to see the output as `Not equal`. Unfortunately, that is not the case. The `s.equals("abc")` expression calls the `equals()` method on an object referred to by the `s` variable; however, if the `s` variable is `null`, it does not refer to any object. Let's see what happens next.

Let's run the following code (that is, the `catchException1()` method in the `Framework` class):

```
try {
    method(null);
```

```
} catch (Exception ex){
    System.out.println("catchException1():");
    System.out.println(ex.getClass().getCanonicalName());
                        //prints: java.lang.NullPointerException
    waitForStackTrace();
    ex.printStackTrace();  //prints: see the screenshot
    if(ex instanceof NullPointerException){
        //do something
    } else {
        //do something else
    }
}
```

The preceding code includes the `waitForStackTrace()` method, allowing you to wait a bit until the stack trace has been generated. Otherwise, the output would be out of sequence. The output of this code appears as follows:

```
catchException1():
java.lang.NullPointerException
java.lang.NullPointerException: Cannot invoke "String.
equals(Object)" because "s" is null
     at com.packt.learnjava.ch04_exceptions.Framework.
method(Framework.java:14)
     at com.packt.learnjava.ch04_exceptions.Framework.
catchException1(Framework.java:24)
     at com.packt.learnjava.ch04_exceptions.Framework.
main(Framework.java:8)
```

As you can see, the method prints the name of the exception class, followed by a **stack trace**. The name of **stack trace** comes from the way the method calls are stored (as a stack) in JVM memory: one method calls another, which, in turn, calls another, and so on. After the most inner method returns, the stack is walked back, and the returned method (**stack frame**) is removed from the stack. We will talk about the JVM memory structure, in more detail, in *Chapter 9*, *JVM Structure and Garbage Collection*. When an exception happens, all the stack content (such as the stack frames) is returned as the stack trace. This allows us to track down the line of code that caused the problem.

In the preceding code example, different blocks of code were executed depending on the type of the exception. In our case, it was `java.lang.NullPointerException`. If the application code does not catch it, this exception would propagate through the stack of the called methods into the JVM, which will then stop executing the application. To avoid this happening, the exception can be caught and code can be executed to recover from the exceptional condition.

The purpose of the exception handling framework in Java is to protect the application code from an unexpected condition and, if possible, recover from it. In the following sections, we will dissect this concept in more detail and rewrite the given example using the framework capability.

Checked and unchecked exceptions

If you look up the documentation of the `java.lang` package API, you will discover that the package contains almost three dozen exception classes and a couple of dozen error classes. Both groups extend the `java.lang.Throwable` class, inherit all the methods from it, and do not add other methods. The methods that are most often used in the `java.lang.Throwable` class include the following:

- `void printStackTrace()`: This outputs the stack trace (stack frames) of the method calls.
- `StackTraceElement[] getStackTrace()`: This returns the same information as `printStackTrace()` but allows programmatic access of any frame of the stack trace.
- `String getMessage()`: This retrieves the message that often contains a user-friendly explanation of the reason for the exception or error.
- `Throwable getCause()`: This retrieves an optional object of `java.lang.Throwable` that was the original reason for the exception (but the author of the code decided to wrap it in another exception or error).

All errors extend the `java.lang.Error` class, which, in turn, extends the `java.lang.Throwable` class. Typically, an error is thrown by JVM and, according to the official documentation, *indicates serious problems that a reasonable application should not try to catch*. Here are a few examples:

- `OutOfMemoryError`: This is thrown when the JVM runs out of memory and cannot clean the memory using garbage collection.
- `StackOverflowError`: This is thrown when the memory allocated for the stack of the method calls is not enough to store another stack frame.

- `NoClassDefFoundError`: This is thrown when the JVM cannot find the definition of the class requested by the currently loaded class.

The authors of the framework assumed that an application cannot recover from these errors automatically, which proved to be a largely correct assumption. That is why programmers, typically, do not catch errors, but that is beyond the scope of this book.

On the other hand, exceptions are typically related to application-specific problems and often do not require us to shut down the application and allow recovery. Usually, that is why programmers catch them and implement an alternative (to the main flow) path of the application logic, or at least report the problem without shutting down the application. Here are a few examples:

- `ArrayIndexOutOfBoundsException`: This is thrown when the code tries to access the element by the index that is equal to, or bigger than, the array length (remember that the first element of an array has an index of `0`, so the index is equal to the array length points outside of the array).
- `ClassCastException`: This is thrown when the code casts a reference to a class or an interface not associated with the object referred to by the variable.
- `NumberFormatException`: This is thrown when the code tries to convert a string into a numeric type, but the string does not contain the necessary number format.

All exceptions extend the `java.lang.Exception` class, which, in turn, extends the `java.lang.Throwable` class. That is why, by catching an object of the `java.lang.Exception` class, the code catches an object of any exception type. In the *The Java exceptions framework* section, we demonstrated this by catching `java.lang.NullPointerException` in the same way.

One of the exceptions is `java.lang.RuntimeException`. The exceptions that extend it are called **runtime exceptions** or **unchecked exceptions**. We have already mentioned some of them: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException`, and `NumberFormatException`. The reason they are called runtime exceptions is clear; the reason they are called unchecked exceptions will become clear next.

Those exceptions that do not have `java.lang.RuntimeException` among their ancestors are called **checked exceptions**. The reason for such a name is that the compiler makes sure (checks) that these exceptions have either been caught or listed in the `throws` clause of the method (please refer to the *The throws statement* section). This design forces the programmer to make a conscious decision, either to catch the checked exception or inform the client of the method that this exception might be thrown by the method and has to be processed (handled) by the client. Here are a few examples of checked exceptions:

- `ClassNotFoundException`: This is thrown when an attempt to load a class using its string name with the `forName()` method of the `Class` class has failed.
- `CloneNotSupportedException`: This is thrown when the code tries to clone an object that does not implement the `Cloneable` interface.
- `NoSuchMethodException`: This is thrown when there is no method called by the code.

Not all exceptions reside in the `java.lang` package. Many other packages contain exceptions related to the functionality that is supported by the package. For example, there is a `java.util.MissingResourceException` runtime exception and a `java.io.IOException` checked exception.

Despite not being forced, programmers often catch runtime (unchecked) exceptions to have better control of the program flow, making the behavior of an application more stable and predictable. By the way, all errors are also runtime (unchecked) exceptions. However, as we mentioned already, typically, it is not possible to handle them programmatically, so there is no point in catching descendants of the `java.lang.Error` class.

The try, catch, and finally blocks

When an exception is thrown inside a `try` block, it redirects the control flow to the first `catch` clause. If there is no `catch` block that can capture the exception (but the `finally` block has to be in place), the exception propagates up and out of the method. If there is more than one `catch` clause, the compiler forces you to arrange them so that the child exception is listed before the parent exception. Let's look at the following example:

```
void someMethod(String s){
    try {
       method(s);
    } catch (NullPointerException ex){
       //do something
```

```
    } catch (Exception ex){
       //do something else
    }
}
```

In the preceding example, a `catch` block with `NullPointerException` is placed before the block with `Exception` because `NullPointerException` extends `RuntimeException`, which, in turn, extends `Exception`. We could even implement this example, as follows:

```
void someMethod(String s){
    try {
        method(s);
    } catch (NullPointerException ex){
        //do something
    } catch (RuntimeException ex){
        //do something else
    } catch (Exception ex){
        //do something different
    }
}
```

Note that the first `catch` clause only catches `NullPointerException`. Other exceptions that extend `RuntimeException` will be caught by the second `catch` clause. The rest of the exception types (that is, all of the checked exceptions) will be caught by the last `catch` block. Note that errors will not be caught by any of these `catch` clauses. To catch them, you should add a `catch` clause for `Error` (in any position) or `Throwable` (after the last `catch` clause in the previous example). However, usually, programmers do not do it and allow errors to propagate into the JVM.

Having a `catch` block for each exception type allows us to provide specific exception type processing. However, if there is no difference in the exception processing, you can simply have one `catch` block with the `Exception` base class to catch all types of exceptions:

```
void someMethod(String s){
    try {
        method(s);
    } catch (Exception ex){
        //do something
```

```
    }
}
```

If none of the clauses catch the exception, it is thrown further up until it is either handled by a `try...catch` statement in one of the method callers or propagates all the way out of the application code. In such a case, the JVM terminates the application and exits.

Adding a `finally` block does not change the described behavior. If present, it is always executed, whether an exception has been generated or not. Usually, a `finally` block is used to release the resources, to close a database connection, a file, or similar. However, if the resource implements the `Closeable` interface, it is better to use the `try-with-resources` statement, which allows you to release the resources automatically. The following demonstrates how it can be done with Java 7:

```
try (Connection conn = DriverManager
                .getConnection("dburl", "username", "password");
     ResultSet rs = conn.createStatement()
                .executeQuery("select * from some_table")) {
    while (rs.next()) {
        //process the retrieved data
    }
} catch (SQLException ex) {
    //Do something
    //The exception was probably caused
    //by incorrect SQL statement
}
```

The preceding example creates the database connection, retrieves data and processes it, and then closes (calls the `close()` method) the `conn` and `rs` objects.

Java 9 enhances the `try-with-resources` statement's capabilities by allowing the creation of objects that represent resources outside the `try` block, along with the use of them in a `try-with-resources` statement, as follows:

```
void method(Connection conn, ResultSet rs) {
    try (conn; rs) {
        while (rs.next()) {
            //process the retrieved data
        }
    } catch (SQLException ex) {
```

```
        //Do something
        //The exception was probably caused
        //by incorrect SQL statement
    }
}
```

The preceding code looks much cleaner, although, in practice, programmers prefer to create and release (close) resources in the same context. If that is your preference too, consider using the `throws` statement in conjunction with the `try-with-resources` statement.

The throws statement

We have to deal with `SQLException` because it is a checked exception, and the `getConnection()`, `createStatement()`, `executeQuery()`, and `next()` methods declare it in their `throws` clause. Here is an example:

```
Statement createStatement() throws SQLException;
```

This means that the method's author warns the method's users that it might throw such an exception, forcing them to either catch the exception or to declare it in the `throws` clause of their methods. In our preceding example, we have chosen to catch it using two `try...catch` statements. Alternatively, we can list the exception in the `throws` clause and, thus, remove the clutter by effectively pushing the burden of the exception handling onto the users of our method:

```
void throwsDemo() throws SQLException {
    Connection conn =
      DriverManager.getConnection("url","user","pass");
    ResultSet rs = conn.createStatement().executeQuery(
      "select * ...");
    try (conn; rs) {
        while (rs.next()) {
            //process the retrieved data
        }
    } finally {
        try {
           if(conn != null) {
              conn.close();
```

```
            }
        } finally {
            if(rs != null) {
                rs.close();
            }
        }
    }
}
```

We got rid of the catch clause, but we need the finally block to close the created conn and rs objects.

Please, notice how we included code that closes the conn object in a try block and we included the code that closes the rs object in the finally block. This way we make sure that an exception during closing of the conn object will not prevent us from closing the rs object.

This code looks less clear than the try-with-resources statement we demonstrated in the previous section. We show it just to demonstrate all the possibilities and how to avoid possible danger (of not closing the resources) if you decide to do it yourself, not letting the try-with-resources statement do it for you automatically.

But let us get back to the discussion of the throws statement.

The `throws` clause allows but does not require us to list unchecked exceptions. Adding unchecked exceptions does not force the method's users to handle them.

Finally, if the method throws several different exceptions, it is possible to list the base `Exception` exception class instead of listing all of them. That will make the compiler happy; however, this is not considered a good practice because it hides the details of particular exceptions that a method's user might expect.

Please note that the compiler does not check what kind of exception the code in the method's body can throw. So, it is possible to list any exception in the `throws` clause, which might lead to unnecessary overhead. If, by mistake, a programmer includes a checked exception in the `throws` clause that is never actually thrown by the method, the method's user could write a `catch` block for it that is never executed.

The throw statement

The `throw` statement allows the throwing of any exception that a programmer deems necessary. You can even create your own exception. To create a checked exception, extend the `java.lang.Exception` class as follows:

```
class MyCheckedException extends Exception{
    public MyCheckedException(String message){
        super(message);
    }
    //add code you need to have here
}
```

Also, to create an unchecked exception, extend the `java.lang.RunitmeException` class, as follows:

```
class MyUncheckedException extends RuntimeException{
    public MyUncheckedException(String message){
        super(message);
    }
    //add code you need to have here
}
```

Notice the *add code you need to have here* comment. You can add methods and properties to the custom exception as with any other regular class, but programmers rarely do it. In fact, the best practices explicitly recommend avoiding the use of exceptions for driving business logic. Exceptions should be what the name implies, covering only exceptional or very rare situations.

However, if you need to announce an exceptional condition, use the `throw` keyword and the `new` operator to create and trigger the propagation of an exception object. Here are a few examples:

```
throw new Exception("Something happened");
throw new RuntimeException("Something happened");
throw new MyCheckedException("Something happened");
throw new MyUncheckedException("Something happened");
```

It is even possible to throw `null`, as follows:

```
throw null;
```

The result of the preceding statement is the same as the result of this one:

```
throw new NullPointerException;
```

In both cases, an object of an unchecked `NullPointerException` exception begins to propagate through the system, until it is caught either by the application or the JVM.

The assert statement

Once in a while, a programmer needs to know whether a particular condition has happened in the code, even after the application has already been deployed to production. At the same time, there is no need to run this check all the time. That is where the `assert` branching statement comes in handy. Here is an example:

```
public someMethod(String s){
    //any code goes here
    assert(assertSomething(x, y, z));
    //any code goes here
}
boolean assertSomething(int x, String y, double z){
    //do something and return boolean
}
```

In the preceding code, the `assert()` method takes input from the `assertSomething()` method. If the `assertSomething()` method returns `false`, the program stops executing.

The `assert()` method is executed only when the JVM is run using the `-ea` option. The `-ea` flag should not be used in production, except maybe temporarily for testing purposes. This is because it creates an overhead that affects the application performance.

Best practices of exception handling

Checked exceptions were designed to be used for recoverable conditions when an application can do something automatically to amend or work around the problem. In practice, this doesn't happen very often. Typically, when an exception is caught, the application logs the stack trace and aborts the current action. Based on the logged information, the application support team modifies the code to address the condition that is unaccounted for or to prevent it from occurring in the future.

Each application is different, so best practices depend on the particular application requirements, design, and context. In general, it seems that there is an agreement in the development community to avoid using checked exceptions and to minimize their propagation in the application code. The following is a list of a few other recommendations that have proved to be useful:

- Always catch all checked exceptions that are close to the source.
- If in doubt, catch unchecked exceptions that are also close to the source.
- Handle the exception as close to the source as possible because that is where the context is the most specific and where the root cause resides.
- Do not throw checked exceptions unless you have to because you force the building of extra code for a case that might never happen.
- Convert third-party checked exceptions into unchecked ones by re-throwing them as `RuntimeException` with the corresponding message if you have to.
- Do not create custom exceptions unless you have to.
- Do not drive business logic by using the exception handling mechanism unless you have to.
- Customize generic `RuntimeException` exceptions by using the system of messages and, optionally, the `enum` type instead of using the exception type to communicate the cause of the error.

There are many other possible tips and recommendations; however, if you follow these, you are probably going to be fine in the vast majority of situations. With that, we conclude this chapter.

Summary

In this chapter, you were introduced to the Java exception handling framework, and you learned about two kinds of exceptions—checked and unchecked (runtime)—and how to handle them using `try-catch-finally` and `throws` statements. Also, you learned how to generate (throw) exceptions and how to create your own (custom) exceptions. The chapter concluded with the best practices of exception handling which, if followed consistently, will help you to write clean and clear code, which is pleasant to write and easy to understand and maintain.

In the next chapter, we will talk about strings and their processing in detail, along with input/output streams and file reading and writing techniques.

Quiz

1. What is a stack trace? Select all that apply:

   A. A list of classes currently loaded

   B. A list of methods currently executing

   C. A list of code lines currently executing

   D. A list of variables currently used

2. What kinds of exceptions are there? Select all that apply:

   A. Compilation exceptions

   B. Runtime exceptions

   C. Read exceptions

   D. Write exceptions

3. What is the output of the following code?

```
try {
    throw null;
} catch (RuntimeException ex) {
    System.out.print("RuntimeException ");
} catch (Exception ex) {
    System.out.print("Exception ");
} catch (Error ex) {
    System.out.print("Error ");
} catch (Throwable ex) {
    System.out.print("Throwable ");
} finally {
    System.out.println("Finally ");
}
```

   A. A `RuntimeException` error

   B. `Exception Error Finally`

   C. `RuntimeException Finally`

   D. `Throwable Finally`

4. Which of the following methods will compile without an error?

```
void method1() throws Exception { throw null; }
void method2() throws RuntimeException { throw null; }
void method3() throws Throwable { throw null; }
void method4() throws Error { throw null; }
```

   A. `method1()`
   B. `method2()`
   C. `method3()`
   D. `method4()`

5. Which of the following statements will compile without an error?

```
throw new NullPointerException("Hi there!"); //1
throws new Exception("Hi there!");           //2
throw RuntimeException("Hi there!");        //3
throws RuntimeException("Hi there!");      //4
```

   A. 1
   B. 2
   C. 3
   D. 4

6. Assuming that `int x = 4`, which of the following statements will compile without an error?

```
assert (x > 3); //1
assert (x = 3); //2
assert (x < 4); //3
assert (x = 4); //4
```

   A. 1
   B. 2
   C. 3
   D. 4

7. Which are the best practices from the following list?

   A. Always catch all exceptions and errors.
   B. Always catch all exceptions.
   C. Never throw unchecked exceptions.
   D. Try not to throw checked exceptions unless you have to.