Chapter 10 - Exceptions

Unchecked Exceptions

An *unchecked exception* is any exception that does not need to be declared or handled by the application code where it is thrown. Unchecked exceptions are often referred to as runtime exceptions, although in Java, **unchecked exceptions include any class that inherits `RuntimeException` or `Error`.**

Throwing an Exception

Additionally, you should know that an `Exception` is an `Object`. This means you can store in a variable, and this is legal:

```
1 Exception e = new RuntimeException();
2 throw e;
```

TABLE 10.1 Types of exceptions and errors

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of `RuntimeException`	Yes	No
Checked exception	Subclass of `Exception` but not subclass of `RuntimeException`	Yes	Yes
Error	Subclass of `Error`	No	No

RuntimeException Classes

`RuntimeException` and its subclasses are unchecked exceptions that don't have to be handled or declared. They can be thrown by the programmer or by the JVM. Common `RuntimeException` classes include the following:

`ArithmeticException` Thrown when code attempts to divide by zero

`ArrayIndexOutOfBoundsException` Thrown when code uses an illegal index to access an array

`ClassCastException` Thrown when an attempt is made to cast an object to a class of which it is not an instance

`NullPointerException` Thrown when there is a `null` reference where an object is required

`IllegalArgumentException` Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument

`NumberFormatException` Subclass of `IllegalArgumentException` thrown when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format

For the exam, you need to know that `NumberFormatException` is a subclass of `IllegalArgumentException`

Checked *Exception* Classes

Checked exceptions have `Exception` in their hierarchy but not `RuntimeException`. They must be handled or declared. Common checked exceptions include the following:

`IOException` Thrown programmatically when there's a problem reading or writing a file

`FileNotFoundException` Subclass of `IOException` thrown programmatically when code tries to reference a file that does not exist

For the exam, you need to know that these are both checked exceptions. You also need to know that `FileNotFoundException` is a subclass of `IOException`. You'll see shortly why that matters.

Error Classes

Errors are unchecked exceptions that extend the `Error` class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see these:

`ExceptionInInitializerError` Thrown when a static initializer throws an exception and doesn't handle it

`StackOverflowError` Thrown when a method calls itself too many times (This is called *infinite recursion* because the method typically calls itself without end.)

`NoClassDefFoundError` Thrown when a class that the code uses is available at compile time but not runtime

Handling Exceptions

Using *try* and *catch* Statements

```
try { // DOES NOT COMPILE
   fall();
}
```

This code doesn't compile because the `try` block doesn't have anything after it. Remember, the point of a `try` statement is for something to happen if an exception is thrown. Without another clause, the `try` statement is lonely. As you will see shortly, there is a special type of `try` statement that includes an implicit `finally` block, although the syntax for this is quite different from this example.

For the exam, you won't be asked to create your own exception, but you may be given exception classes and need to understand how they function. Here's how to tackle them. First, you must be able to recognize if the exception is a checked or an unchecked exception. Second, you need to determine whether any of the exceptions are subclasses of the others.

Also, remember that an exception defined by the `catch` statement is only in scope for that `catch` block. For example, the following causes a compiler error since it tries to use the exception class outside the block for which it was defined:

```
public void visitManatees() {
   try {
   } catch (NumberFormatException e1) {
      System.out.println(e1);
   } catch (IllegalArgumentException e2) {
      System.out.println(e1);  // DOES NOT COMPILE
   }
}
```

Java intends multi-catch to be used for exceptions that aren't related, and it prevents you from specifying redundant types in a multi-catch. Do you see what is wrong here?

```
try {
   throw new IOException();
} catch (FileNotFoundException | IOException p) {} // DOES NOT COMPILE
```

Specifying it in the multi-catch is redundant, and the compiler gives a message such as this:

```
The exception FileNotFoundException is already caught by the alternative IOException
```

There is one additional rule you should know for `finally` blocks. If a `try` statement with a `finally` block is entered, then the `finally` block will always be executed, regardless of whether the code completes successfully. Take a look at the following `goHome()` method. Assuming an exception may or may not be thrown on line 14, what are the possible values that this method could print? Also, what would the return value be in each case?

```
12: int goHome() {
13:    try {
14:       // Optionally throw an exception here
15:       System.out.print("1");
16:       return -1;
17:    } catch (Exception e) {
18:       System.out.print("2");
19:       return -2;
20:    } finally {
21:       System.out.print("3");
22:       return -3;
23:    }
24: }
```

If an exception is not thrown on line 14, then the line 15 will be executed, printing `1`. Before the method returns, though, the `finally` block is executed, printing `3`. If an exception is thrown, then lines 15–16 will be skipped, and lines 17–19 will be executed, printing `2`, followed by `3` from the `finally` block. While the first value printed may differ, the method always prints `3` last since it's in the `finally` block.

SYSTEM.EXIT()

There is one exception to "the `finally` block always be executed" rule: Java defines a method that you call as `System.exit()`. It takes an integer parameter that represents the error code that gets returned.

```
try {
   System.exit(0);
} finally {
   System.out.print("Never going to get here");  // Not printed
}
```

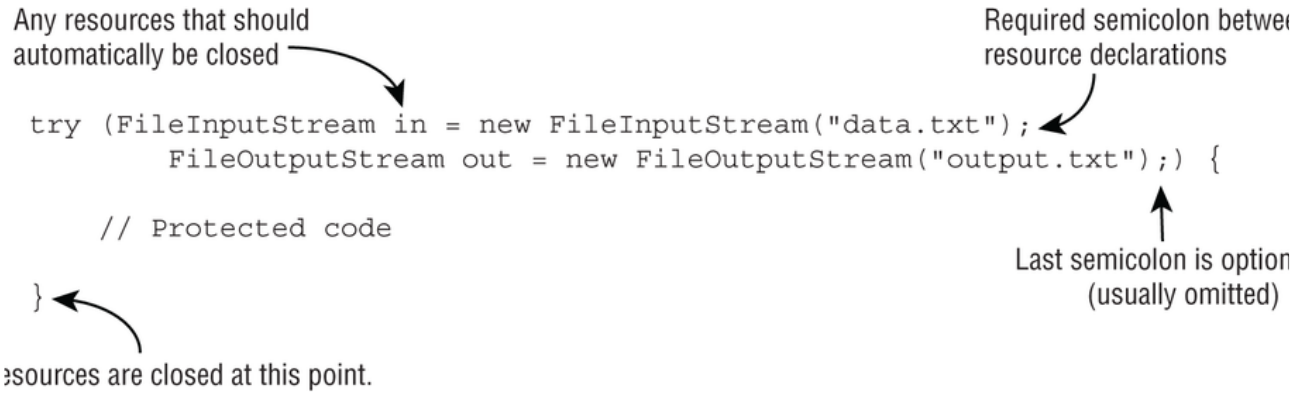
`System.exit()` tells Java, "Stop. End the program right now. Do not pass go. Do not collect $200." When `System.exit()` is called in the `try` or `catch` block, the `finally` block does not run.

Java includes the *try-with-resources* statement to automatically close all resources opened in a `try` clause. This feature is also known as *automatic resource management*, because Java automatically takes care of the closing.

More importantly, though, by using a try-with-resources statement, we guarantee that as soon as a connection passes out of scope, Java will attempt to close it within the same method.

Behind the scenes, the compiler replaces a try-with-resources block with a `try` and `finally` block. We refer to this "hidden" `finally` block as an implicit `finally` block since it is created and used by the compiler automatically. You can still create a programmer-defined `finally` block when using a try-with-resources statement; just be aware that the implicit one will be called first.

Figure 10.5 shows what a try-with-resources statement looks like. Notice that one or more resources can be opened in the `try` clause. When there are multiple resources opened, they are closed in the *reverse* order from which they were created. Also, notice that parentheses are used to list those resources, and semicolons are used to separate the declarations. This works just like declaring multiple indexes in a `for` loop.



What happened to the `catch` block in Figure 10.5? Well, it turns out a `catch` block is optional with a try-with-resources statement. For example, we can rewrite the previous `readFile()` example so that the method rethrows the exception to make it even shorter:

In fact, if the code within the `try` block throws a checked exception not declared by the method in which it is defined or handled by another `try`/`catch` block, then it will need to be handled by the `catch` block. Also, the `catch` and `finally` blocks are run in addition to the implicit one that closes the resources. For the exam, you need to know that the implicit `finally` block runs *before* any programmer-coded ones.

While try-with-resources does support declaring multiple variables, each variable must be declared in a separate statement. For example, the following do not compile:

```
try (MyFileClass is = new MyFileClass(1),  // DOES NOT COMPILE
      os = new MyFileClass(2)) {
}

try (MyFileClass ab = new MyFileClass(1),  // DOES NOT COMPILE
      MyFileClass cd = new MyFileClass(2)) {
}
```

You can declare a resource using `var` as the data type in a try-with-resources statement, since resources are local variables.

```
try (var f = new BufferedInputStream(new FileInputStream("it.txt"))) {
   // Process file
}
```

The resources created in the `try` clause are in scope only within the `try` block.

Following Order of Operation

- Resources are closed after the `try` clause ends and before any `catch`/`finally` clauses.
- Resources are closed in the reverse order from which they were created.

inside a `catch` block on the exam, check and make sure the code in the associated `try` block is capable of throwing the exception or a subclass of the exception. If not, the code is unreachable and does not compile. Remember that this rule does not extend to unchecked exceptions or exceptions declared in a method signature.