Previous in the Series	Current Tutorial	Next in the Series
What Is an Exception?	**Catching and Handling Exceptions**	Throwing Exceptions

Catching and Handling Exceptions

Catching and Handling Exceptions

This section describes how to use the three exception handler components — the `try`, `catch`, and `finally` blocks — to write an exception handler. Then, the try-with-resources statement, introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use `Closeable` resources, such as streams.

The last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named `ListOfNumbers`. When constructed, `ListOfNumbers` creates an `ArrayList` that contains 10 `Integer` elements with sequential values 0 through 9. The `ListOfNumbers` class also defines a method named `writeList()`, which writes the list of numbers into a text file called `OutFile.txt`. This example uses output classes defined in `java.io`, which are covered in the Basic I/O section.

```
// Note: This class will not comp
import java.io.*;
import java.util.List;
import java.util.ArrayList;
```

In this tutorial

```
6
7   public class ListOfNumbers {
8
9       private List<Integer> list;
10      private static final int SIZE
11
12      public ListOfNumbers () {
13          list = new ArrayList<>(SI
14          for (int i = 0; i < SIZE;
15              list.add(i);
16          }
17      }
18
19      public void writeList() {
20      // The FileWriter constructor
21          PrintWriter out = new Pri
22
23          for (int i = 0; i < SIZE;
24              // The get(int) metho
25              out.println("Value at
26          }
27          out.close();
28      }
    }
```

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be

`IOException`. The second boldface line is a call to the `ArrayList` class's get method, which throws an `IndexOutOfBoundsException` if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the `ArrayList`.

If you try to compile the `ListOfNumbers` class, the compiler prints an error message about the exception thrown by the `FileWriter` constructor. However, it does not display an error message about the exception thrown by `get()`. The reason is that the exception thrown by the constructor, `IOException`, is a checked exception, and the one thrown by the `get()` method, `IndexOutOfBoundsException`, is an unchecked exception.

Now that you're familiar with the `ListOfNumbers` class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

The Try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a `try` block. In general, a `try` block looks like the following:

```
try {
    code
}
catch and finally blocks . . .
```

The segment in the example labeled code contains one or more legal lines of code that could throw an exception. (The `catch` and `finally` blocks are explained in the next two subsections.)

To construct an exception handler for the `writeList()` method from the `ListOfNumbers` class, enclose the exception-throwing statements of the `writeList()` method within a `try` block. There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the `writeList()` code within a single `try` block and associate multiple handlers with it. The following listing uses one `try` block for the entire method because the code in question is very short.

```
private List<Integer> list;
private static final int SIZE = 10

public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entere
        out = new PrintWriter(new
        for (int i = 0; i < SIZE;
            out.println("Value at
        }
    }
    catch and finally blocks  . .
}
```

If an exception occurs within the `try` block, that exception is handled by an exception handler associated with it. To associate an exception handler with a `try` block, you must put a `catch` block after it; the next section, The catch Blocks, shows you how.

The Catch Blocks

You associate exception handlers with a `try` block by providing one or more catch blocks directly after the `try` block. No code can be between the end of the `try` block and the beginning of the first `catch` block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Each `catch` block is an exception handler that handles the type of exception indicated by its argument. The argument type, `ExceptionType`, declares the type of exception that the handler can handle and must be the name of a class that inherits from the `Throwable` class. The handler can refer to the exception with name.

The `catch` block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose `ExceptionType` matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the `writeList()` method:

```
try {

} catch (IndexOutOfBoundsExceptior
    System.err.println("IndexOutO
} catch (IOException e) {
    System.err.println("Caught IOE
}
```

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using

chained exceptions, as described in the Chained Exceptions section.

Multi-Catching Exceptions

You can catch more than one type of exception with one exception handler, with the multi-catch pattern.

In Java SE 7 and later, a single `catch` block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the `catch` clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (`|`):

```
1 catch (IOException|SQLException ex
2     logger.log(ex);
3     throw ex;
4 }
```

Note: If a catch block handles more than one exception type, then the `catch` parameter is implicitly `final`. In this example, the `catch` parameter `ex` is `final` and therefore you cannot assign any values to it within the `catch` block.

The Finally Block

The finally block always executes when the `try` block exits. This ensures that the `finally` block is executed even if an unexpected exception occurs. But `finally` is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`. Putting cleanup code in a `finally` block is always a good practice, even when no exceptions are anticipated.

> *Note: If the JVM exits while the `try` or `catch` code is being executed, then the*

> *`finally` block may not execute.*

The `try` block of the `writeList()` method that you've been working with here opens a `PrintWriter`. The program should close that stream before exiting the `writeList()` method. This poses a somewhat complicated problem because `writeList()`'s `try` block can exit in one of three ways.

1. The new `FileWriter` statement fails and throws an `IOException`.
2. The `list.get(i)` statement fails and throws an `IndexOutOfBoundsException`.
3. Everything succeeds and the `try` block exits normally.

The runtime system always executes the statements within the `finally` block regardless of what happens within the `try` block. So it's the perfect place to perform cleanup.

The following `finally` block for the `writeList()` method cleans up and then closes the `PrintWriter`.

```
finally {
    if (out != null) {
        System.out.println("Closi
        out.close();
    } else {
        System.out.println("Printl
    }
}
```

> *Important: The `finally` block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a `finally` block to ensure that resource is always recovered.*
>
> *Consider using the try-with-resources statement in these situations, which automatically releases system resources when no longer needed. The try-with-resources Statement section has more information.*

The Try-with-resources Statement

The try-with-resources statement is a `try` statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

The following example reads the first line from a file. It uses an instance of `BufferedReader` to read data from the file. `BufferedReader` is a resource that must be closed after the program is finished with it:

```
1 static String readFirstLineFromFi
2     try (BufferedReader br =
3                     new BufferedRe
4         return br.readLine();
5     }
6 }
```

In this example, the resource declared in the try-with-resources statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the `try` keyword. The class `BufferedReader`, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the `BufferedReader` instance is declared in a try-with-resource statement, it will be closed regardless of whether the `try` statement completes normally or abruptly (as a result of the method `BufferedReader.readLine()` throwing an `IOException`.

Prior to Java SE 7, you can use a `finally` block to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly. The following example uses a `finally` block instead of a try-with-resources statement:

```
static String readFirstLineFromFi

    BufferedReader br = new Buffe
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

However, in this example, if the methods `readLine()` and close both throw exceptions, then the method `readFirstLineFromFileWithFinallyBloc k()` throws the exception thrown from the `finally` block; the exception thrown from the `try` block is suppressed. In contrast, in the example `readFirstLineFromFile()`, if exceptions are thrown from both the `try` block and the try-with-resources statement, then the method `readFirstLineFromFile()` throws the exception thrown from the `try` block; the exception thrown from the try-with-resources block is suppressed. In Java SE 7 and later, you can retrieve suppressed exceptions; see the section Suppressed Exceptions for more information.

You may declare one or more resources in a try-with-resources statement. The following example retrieves the names of the files packaged in the zip file `zipFileName` and creates a text file that contains the names of these files:

```
public static void writeToFileZipF



    java.nio.charset.Charset chars
         java.nio.charset.Standard
    java.nio.file.Path outputFileP
         java.nio.file.Paths.get(o

    // Open zip file and create ou
    // try-with-resources statemen

    try (
        java.util.zip.ZipFile zf =
             new java.util.zip.Zip
        java.io.BufferedWriter wri
            java.nio.file.Files.ne
    ) {
        // Enumerate each entry
        for (java.util.Enumeratio
```

```
                                                zi
                // Get the entry name
                String newLine = Syste
                String zipEntryName =
                     ((java.util.zip.Z
                     newLine;
                writer.write(zipEntryN
            }
        }
    }
```

In this example, the try-with-resources statement contains two declarations that are separated by a semicolon: `ZipFile` and `BufferedWriter`. When the block of code that directly follows it terminates, either normally or because of an exception, the `close()` methods of the `BufferedWriter` and `ZipFile` objects are automatically called in this order. Note that the close methods of resources are called in the opposite order of their creation.

The following example uses a try-with-resources statement to automatically close a `java.sql.Statement` object:

```
public static void viewTable(Conne

    String query = "select COF_NAM

    try (Statement stmt = con.creat
        ResultSet rs = stmt.execut

        while (rs.next()) {
            String coffeeName = rs
            int supplierID = rs.ge
            float price = rs.getFl
            int sales = rs.getInt(
            int total = rs.getInt(

            System.out.println(cof
                                pri
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.prin
    }
}
```

The resource `java.sql.Statement` used in this example is part of the JDBC 4.1 and later API.

Note: A try-with-resources statement can have `catch` and `finally` blocks just like an ordinary `try` statement. In a try-with-resources statement, any `catch` or `finally`

block is run after the resources declared have been closed.

Suppressed Exceptions

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example `writeToFileZipFileContents()`, an exception can be thrown from the `try` block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the `ZipFile` and `BufferedWriter` objects. If an exception is thrown from the `try` block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the `writeToFileZipFileContents()` method. You can retrieve these suppressed exceptions by calling the `Throwable.getSuppressed()` method from the exception thrown by the `try` block.

Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the `AutoCloseable` and `Closeable` interfaces for a list of classes that implement either of these interfaces. The `Closeable` interface extends the `AutoCloseable` interface. The `close()` method of the `Closeable` interface throws exceptions of type `IOException` while the `close()` method of the `AutoCloseable` interface throws exceptions of type `Exception`. Consequently, subclasses of the `AutoCloseable` interface can override this behavior of the `close()` method to throw specialized exceptions, such as `IOException`, or no exception at all.

Putting It All Together

The previous sections described how to construct the `try`, `catch`, and `finally` code blocks for the `writeList()` method in the `ListOfNumbers` class. Now, let's walk through the code and investigate what can happen.

When all the components are put together, the `writeList()` method looks like the following.

```
1  public void writeList() {
2      PrintWriter out = null;
3
4      try {
5          System.out.println("Enter
6
7          out = new PrintWriter(new
8          for (int i = 0; i < SIZE;
9              out.println("Value at
10         }
11     } catch (IndexOutOfBoundsExcep
12         System.err.println("Caught
13                             +  e.ge
14
15     } catch (IOException e) {
16         System.err.println("Caught
17
18     } finally {
19         if (out != null) {
20             System.out.println("Cl
21             out.close();
22         }
23         else {
24             System.out.println("Pr
25         }
26     }
27 }
```

As mentioned previously, this method's `try` block has three different exit possibilities; here are two of them.

1. Code in the `try` statement fails and throws an exception. This could be an `IOException` caused by the new `FileWriter` statement or an `IndexOutOfBoundsException` caused by a wrong index value in the `for` loop.
2. Everything succeeds and the `try` statement exits normally.

Let's look at what happens in the `writeList()` method during these two exit possibilities.

Scenario 1: An Exception Occurs

The statement that creates a `FileWriter` can fail for a number of reasons. For example, the constructor for the `FileWriter` throws an `IOException` if the program cannot create or write to the file indicated.

When `FileWriter` throws an `IOException`, the runtime system immediately stops executing the `try` block; method calls being executed are not completed. The runtime system then starts searching at the top of the method call stack for an appropriate exception handler. In this example, when the `IOException` occurs, the `FileWriter` constructor is at the top of the call stack. However, the `FileWriter` constructor doesn't have an appropriate exception handler, so the runtime system checks the next method — the `writeList()` method — in the method call stack. The `writeList()` method has two exception handlers: one for `IOException` and one for `IndexOutOfBoundsException`.

The runtime system checks `writeList()`'s handlers in the order in which they appear after the `try` statement. The argument to the first exception handler is `IndexOutOfBoundsException`. This does not match the type of exception thrown, so the runtime system checks the next exception handler — `IOException`. This matches the type of exception that was thrown, so the runtime system ends its search for an appropriate exception handler. Now that the runtime has found an appropriate handler, the code in that `catch` block is executed.

After the exception handler executes, the runtime system passes control to the `finally` block. Code in the `finally` block executes regardless of the exception caught above it. In this scenario, the `FileWriter` was never opened and doesn't need to be closed. After the `finally` block

finishes executing, the program continues with the first statement after the `finally` block.

Here's the complete output from the `ListOfNumbers` program that appears when an `IOException` is thrown.

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

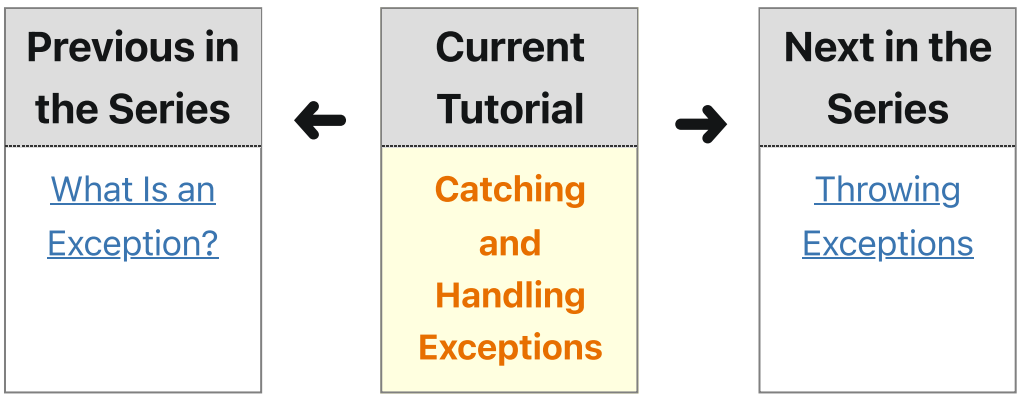
Scenario 2: The try Block Exits Normally

In this scenario, all the statements within the scope of the `try` block execute successfully and throw no exceptions. Execution falls off the end of the `try` block, and the runtime system passes control to the `finally` block. Because everything was successful, the `PrintWriter` is open when control reaches the `finally` block, which closes the `PrintWriter`. Again, after the `finally` block finishes executing, the program continues with the first statement after the `finally` block.

Here is the output from the `ListOfNumbers` program when no exceptions are thrown.

```
Entering try statement
Closing PrintWriter
```

Last update: *September 14, 2021*



Home > Tutorials > Exceptions > Catching and Handling Exceptions

About

- About Java
- About OpenJDK
- Getting Started
- Oracle Java SE Subscription

Downloads

- All Releases
- Source Code

Learning Java

- Documentation
- Java 21 API Docs
- Tutorials
- FAQ
- Java YouTube

Community

- Java User Groups
- Java Conferences
- Contributing

Stay Informed

- Inside.java
- Newscast
- Podcast
- Java Magazine
- Java YouTube
- @java on Twitter