Exceptions handling (try\_catch\_finally\_ throw\_ throws)

**What Is an Exception?**

For example, an application prompts a user for an input file name, then opens the file by passing the name to the constructor for java.io.FileReader. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

**Definition:** An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

**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 . . .*

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**

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 and 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

try {

} catch (FileNotFoundException e) {

System.err.println("FileNotFoundException: " + e.getMessage());

throw new SampleException(e);

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

}

**Catching More Than One Type of Exception with One Exception Handler**

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 (|):

catch (**IOException|SQLException** ex) {

logger.log(ex);

throw ex;

}

**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.

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. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

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.

**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.

**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.

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

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:

**try (**

**java.util.zip.ZipFile zf = new java.util.zip.ZipFile(zipFileName);**

**java.io.BufferedWriter writer = java.nio.file.Files.newBufferedWriter(outputFilePath, charset)**

**)** {

// Enumerate each entry

for (java.util.Enumeration entries =

zf.entries(); entries.hasMoreElements();) {

// Get the entry name and write it to the output file

String newLine = System.getProperty("line.separator");

String zipEntryName =

((java.util.zip.ZipEntry)entries.nextElement()).getName() +

newLine;

writer.write(zipEntryName, 0, zipEntryName.length());

}

}

}

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.

**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.

public void writeList() {

PrintWriter out = null;

try {

System.out.println("Entering" + " try statement");

out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++)

out.println("Value at: " + i + " = " + vector.elementAt(i));

} catch (ArrayIndexOutOfBoundsException e) {

System.err.println("Caught ArrayIndexOutOfBoundsException: "

+ e.getMessage());

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

} finally {

if (out != null) {

System.out.println("Closing PrintWriter");

out.close();

}

else {

System.out.println("PrintWriter not open");

}

}

}

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 ArrayIndexOutOfBoundsException 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 ArrayIndexOutOfBoundsException.

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 ArrayIndexOutOfBoundsException. 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

The boldface code in the following listing shows the statements that get executed during this scenario:

public void writeList() {

**PrintWriter out = null;**

**try {**

**System.out.println("Entering try statement");**

**out = new PrintWriter(new FileWriter("OutFile.txt"));**

for (int i = 0; i < SIZE; i++)

out.println("Value at: " + i + " = " + vector.elementAt(i));

} catch (ArrayIndexOutOfBoundsException e) {

System.err.println("Caught ArrayIndexOutOfBoundsException: "

+ e.getMessage());

} **catch (IOException e) {**

**System.err.println("Caught IOException: " + e.getMessage());**

**} finally {**

**if (out != null) {**

System.out.println("Closing PrintWriter");

out.close();

}

**else {**

**System.out.println("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

The boldface code in the following sample shows the statements that get executed during this scenario.

public void writeList() {

**PrintWriter out = null;**

**try {**

**System.out.println("Entering try statement");**

**out = new PrintWriter(new FileWriter("OutFile.txt"));**

**for (int i = 0; i < SIZE; i++)**

**out.println("Value at: " + i + " = " + vector.elementAt(i));**

**}** catch (ArrayIndexOutOfBoundsException e) {

System.err.println("Caught ArrayIndexOutOfBoundsException: "

+ e.getMessage());

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

} **finally {**

**if (out != null) {**

**System.out.println("Closing PrintWriter");**

**out.close();**

**}**

else {

System.out.println("PrintWriter not open");

}

}

}

# Specifying the Exceptions Thrown by a Method( throws ) :

If the method doesn't catch the checked exceptions that can occur within it, the method must specify that it can throw these exceptions. Let's modify the original writeList method to specify the exceptions it can throw instead of catching them.

// Note: This method won't compile by design!

public void writeList() {

PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++) {

out.println("Value at: " + i + " = " + vector.elementAt(i));

}

out.close();

}

To specify that writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method. The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.

public void writeList() **throws IOException, ArrayIndexOutOfBoundsException** {

Remember that ArrayIndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following.

public void writeList() **throws IOException** {

# How to Throw Exceptions (The throw Statement):

Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

Checked vs Unchecked Exceptions in Java

In Java, there two types of exceptions:

**1) Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword

**2) Unchecked** are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.  
In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

+-----------+

| Throwable |

+-----------+

/ \

/ \

+-------+ +-----------+

| Error | | Exception |

+-------+ +-----------+

/ | \ / | \

\\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_/ \

+------------------+

unchecked checked | RuntimeException |

+------------------+

/ | | \

\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

unchecked

Here are the few other Checked Exceptions –

* SQLException
* IOException
* DataAccessException
* ClassNotFoundException
* InvocationTargetException
* FileNotFoundException  
  ParseException  
  ClassNotFoundException  
  CloneNotSupportedException  
  InstantiationException  
  InterruptedException  
  NoSuchMethodException  
  NoSuchFieldException

Here are the few most frequently seen un-checked exceptions –

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException
* ArrayStoreException  
  BufferOverflowException  
  BufferUnderflowException  
  CannotRedoException  
  CannotUndoException  
  ClassCastException  
  CMMException  
  ConcurrentModificationException  
  DOMException  
  EmptyStackException  
  IllegalArgumentException  
  IllegalMonitorStateException  
  IllegalPathStateException  
  IllegalStateException  
  ImagingOpException  
  IndexOutOfBoundsException  
  MissingResourceException  
  NegativeArraySizeException  
  NoSuchElementException  
  NullPointerException  
  ProfileDataException  
  ProviderException  
  RasterFormatException  
  SecurityException  
  SystemException  
  UndeclaredThrowableException  
  UnmodifiableSetException  
  UnsupportedOperationException