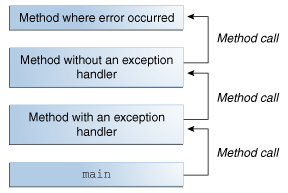
# Lesson: Exceptions

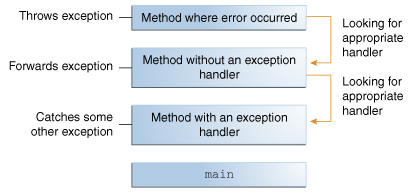
## What Is an Exception?

The term *exception* is shorthand for the phrase "exceptional event."



The call stack.

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*.



Searching the call stack for the exception handler.

## The Catch or Specify Requirement

### The Three Kinds of Exceptions

The first kind of exception is the *checked exception*.

Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

The second kind of exception is the *error*.

Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

The third kind of exception is the *runtime exception*.

Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by RuntimeException and its subclasses.

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

## Catching and Handling Exceptions

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 try Block

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.)

### The catch Blocks

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*.

#### 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;

}

**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.

**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.

**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 [The try-with-resources Statement](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html)section has more information.

### The try-with-resources Statement

The try-with-resources statement is a try statement that declares one or more resources.

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

public static void viewTable(Connection con) throws SQLException {

String query = "select COF\_NAME, SUP\_ID, PRICE, SALES, TOTAL from COFFEES";

**try (Statement stmt = con.createStatement())** {

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + ", " + supplierID + ", " +

price + ", " + sales + ", " + total);

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

}

}

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.

### Putting It All Together

## Specifying the Exceptions Thrown by a Method

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

Remember that IndexOutOfBoundsException 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

if (size == 0) {

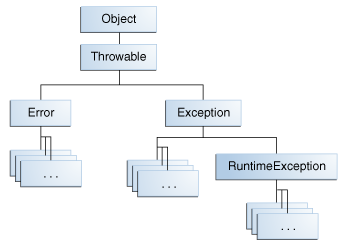
**throw new EmptyStackException();**

}

For now, all you need to remember is that you can throw only objects that inherit from the java.lang.Throwable class.

#### Throwable Class and Its Subclasses

 As you can see, Throwable has two direct descendants: [Error](https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html) and [Exception](https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html).



The Throwable class.

### Chained Exceptions

The following example shows how to use a chained exception.

try {

} catch (IOException e) {

throw new SampleException("Other IOException", e);

}

In this example, when an IOException is caught, a new SampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

#### Accessing Stack Trace Information

Now let's suppose that the higher-level exception handler wants to dump the stack trace in its own format.

**Definition:** A *stack trace* provides information on the execution history of the current thread and lists the names of the classes and methods that were called at the point when the exception occurred. A stack trace is a useful debugging tool that you'll normally take advantage of when an exception has been thrown.

The following code shows how to call the getStackTrace method on the exception object.

catch (Exception cause) {

StackTraceElement elements[] = cause.getStackTrace();

for (int i = 0, n = elements.length; i < n; i++) {

System.err.println(elements[i].getFileName()

+ ":" + elements[i].getLineNumber()

+ ">> "

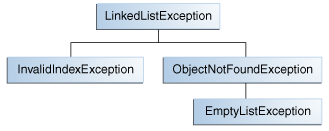
+ elements[i].getMethodName() + "()");

}

}

### Creating Exception Classes

The next figure illustrates one possible class hierarchy for the exceptions thrown by the linked list.



Example exception class hierarchy.

## Unchecked Exceptions — The Controversy（争论）

Generally speaking, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

## Advantages of Exceptions

### Advantage 1: Separating Error-Handling Code from "Regular" Code

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

### Advantage 2: Propagating Errors Up the Call Stack(传播错误到堆栈的最顶层)

### Advantage 3: Grouping and Differentiating Error Types