Throwing Exceptions

Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the `writeList()` method in the `ListOfNumbers` class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing the `ListOfNumbers` class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to not catch the exception and to allow a method further up the call stack to handle it.

If the `writeList()` method doesn't catch the checked exceptions that can occur within it, the `writeList()` 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. To remind you, here's the original version of the `writeList()` method that won't compile.

```
public void writeList() {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(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. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

```
1 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.

```
1 public void writeList() throws IOException {
```

How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. 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.

As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the `Throwable` class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

You can also create chained exceptions. For more information, see the Chained Exceptions section.

The Throw Statement

All methods use the `throw` statement to throw an exception. The `throw` statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the `Throwable` class. Here's an example of a `throw` statement.

```
throw someThrowableObject;
```

Let's look at the `throw` statement in context. The following `pop()` method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

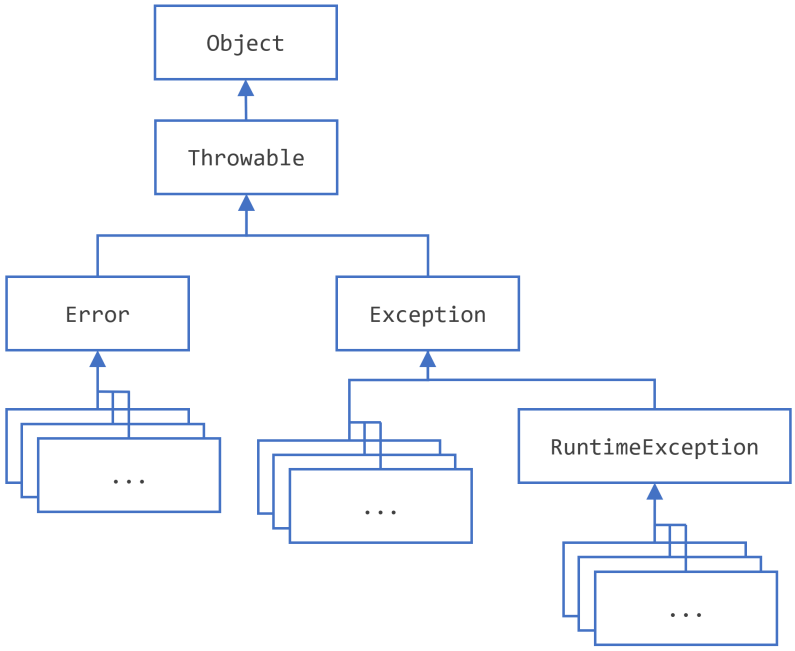
```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The `pop()` method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to 0), pop instantiates a new `EmptyStackException` object, a member of `java.util` and throws it. The Creating Exception Classes section in this chapter explains how to create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the `java.lang.Throwable` class.

Note that the declaration of the `pop()` method does not contain a `throws` clause. `EmptyStackException` is not a checked exception, so pop is not required to state that it might occur.

Throwable Class and Its Subclasses

The objects that inherit from the `Throwable` class include direct descendants (objects that inherit directly from the `Throwable` class) and indirect descendants (objects that inherit from children or grandchildren of the `Throwable` class). The figure below illustrates the class hierarchy of the `Throwable` class and its most significant subclasses. As you can see, `Throwable` has two direct descendants: `Error` and `Exception`.



The Throwable hierarchy

Error Class

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an `Error`. Simple programs typically do not catch or throw instances of `Error`.

Exception Class

Most programs throw and catch objects that derive from the `Exception` class. An `Exception` indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch instances of `Exception` as opposed to `Error`.

The Java platform defines the many descendants of the `Exception` class. These descendants indicate various types of exceptions that can occur. For example, `IllegalAccessException` signals that a particular method could not be found, and `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.

One `Exception` subclass, `RuntimeException`, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is `NullPointerException`, which occurs when a method tries to access a member of an object through a null reference. The section Unchecked Exceptions — The Controversy discusses why most applications shouldn't throw runtime exceptions or subclass `RuntimeException`.

Chained Exceptions

An application often responds to an exception by throwing another exception. In effect, the first exception causes the second exception. It can be very helpful to know when one exception causes another. Chained Exceptions help the programmer do this.

The following are the methods and constructors in `Throwable` that support chained exceptions.

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

The `Throwable` argument to `initCause()` and the `Throwable` constructors is the exception that caused the current exception. `getCause()` returns the exception that caused the current exception, and `initCause()` sets the current exception's cause.

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() + "()");
    }

```

```
    }
```

Logging API

The next code snippet logs where an exception occurred from within the `catch` block. However, rather than manually parsing the stack trace and sending the output to `java.util.logging`, it sends the output to a file using the logging facility in the `java.util.logging` package.

```
try {
    Handler handler = new FileHandler("OutFile.log");
    Logger.getLogger("").addHandler(handler);

} catch (IOException e) {
    Logger logger = Logger.getLogger("package.name");
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        logger.log(Level.WARNING, elements[i].getMethodName());
    }
}
```

Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

- Do you need an exception type that isn't represented by those in the Java platform?
- Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?

- If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?

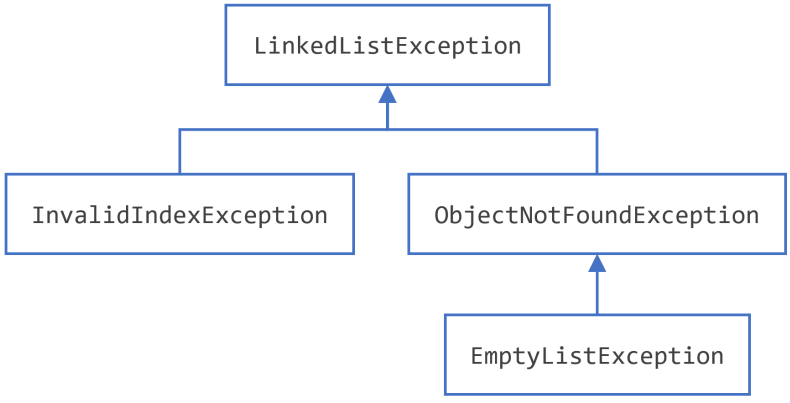
An Example

Suppose you are writing a linked list class. The class supports the following methods, among others:

- `objectAt(int n)` — Returns the object in the nth position in the list. Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.
- `firstObject()` — Returns the first object in the list. Throws an exception if the list contains no objects.
- `indexOf(Object o)` — Searches the list for the specified Object and returns its position in the list. Throws an exception if the object passed into the method is not in the list.

The linked list class can throw multiple exceptions, and it would be convenient to be able to catch all exceptions thrown by the linked list with one exception handler. Also, if you plan to distribute your linked list in a package, all related code should be packaged together. Thus, the linked list should provide its own set of exception classes.

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



Example exception class hierarchy

Choosing a Superclass

Any `Exception` subclass can be used as the parent class of `LinkedListException`. However, a quick perusal of those subclasses shows that they are inappropriate because they are either too specialized or completely unrelated to `LinkedListException`. Therefore, the parent class of `LinkedListException` should be Exception.

Most applications you write will throw objects that are instances of `Exception`. Instances of `Error` are normally used for serious, hard errors in the system, such as those that prevent the JVM from running.

Note: For readable code, it's good practice to append the string `Exception` to the names of all classes that inherit (directly or indirectly) from the `Exception` class.

Last update: *September 14, 2021*

Catching and Handling Exceptions ← **Throwing Exceptions** → Unchecked Exceptions — The Controversy

Home > Tutorials > Exceptions > Throwing Exceptions