



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

If there's a golden rule of programming it's this: Errors occur in software programs. This we know. But what really matters is what happens *after* the error occurs. How is the error handled? Who handles it? Can the program recover, or should it just die?

What's an Exception and Why Do I Care?

The Java language uses exceptions to provide error-handling capabilities for its programs. An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

Your First Encounter with Java Exceptions

If you have done any amount of Java programming at all, you have undoubtedly already encountered exceptions. Your first encounter with Java exceptions was probably in the form of an error message from the compiler like this one:

```
InputFile.java:11: Exception java.io.FileNotFoundException
must be caught, or it must be declared in the throws clause
of this method.
    in = new FileReader(filename);
           ^
```

This message indicates that the compiler found an exception that is not being handled. The Java language requires that a method either catch all "checked" exceptions (those that are checked by the runtime system) or specify that it can throw that type of exception.

Java's Catch or Specify Requirement

This section discusses the reasoning behind this requirement and what it means to you and your Java programs.

Dealing with Exceptions

This section features an example program that can throw two different kinds of exceptions. Using this program, you can learn how to catch an exception and handle it and, alternatively, how to specify that your method can throw it.

Throwing Exceptions

The Java runtime system and many classes from Java packages throw exceptions under some circumstances by using the throw statement. You can use the same mechanism to throw exceptions in your Java programs. This section shows you how to throw exceptions from your Java code.

Runtime Exceptions--The Controversy

Although Java requires that methods catch or specify checked exceptions, they do not have to catch or specify runtime exceptions, that is, exceptions that occur within the Java runtime system. Because catching or specifying an exception is extra work, programmers may be tempted to write code that throws only runtime exceptions and therefore doesn't have to catch or specify them. This is "exception abuse" and is not recommended. The last section in this lesson, explains why.

Note to C++ Programmers: Java exception handlers can have a finally block, which allows programs to clean up after the try block. See [The finally Block](#) for information about how to use finally.



[Start of Tutorial](#) > [Start of Trail](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

What's an Exception and Why Do I Care?

The term *exception* is shorthand for the phrase "exceptional event." It can be defined as follows:

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

Many kinds of errors can cause exceptions--problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element. When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. The exception object contains information about the exception, including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error. In Java terminology, creating an exception object and handing it to the runtime system is called *throwing an exception*.

After a method throws an exception, the runtime system leaps into action to find someone to handle the exception. The set of possible "someones" to handle the exception is the set of methods in the call stack of the method where the error occurred. The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate *exception handler*. An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler. Thus the exception bubbles up through the call stack until an appropriate handler is found and one of the calling methods handles the exception. The exception handler chosen is said to *catch the exception*.

If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

By using exceptions to manage errors, Java programs have the following advantages over traditional error management techniques:

- [Advantage 1: Separating Error Handling Code from "Regular" Code](#)
- [Advantage 2: Propagating Errors Up the Call Stack](#)
- [Advantage 3: Grouping Error Types and Error Differentiation](#)

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

In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, suppose that you have a function that reads an entire file into memory. In pseudo-code, your function might look something like this:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

At first glance this function seems simple enough, but it ignores all of these potential errors:

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To answer these questions within your `read_file` function, you'd have to add a lot of code to do error detection, reporting and handling. Your function would end up looking something like this:

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            }
        }
    }
}
```

```

        }
    } else {
        errorCode = -2;
    }
} else {
    errorCode = -3;
}
close the file;
if (theFileDidntClose && errorCode == 0) {
    errorCode = -4;
} else {
    errorCode = errorCode and -4;
}
} else {
    errorCode = -5;
}
return errorCode;
}

```

With error detection built in, your original 7 lines (in bold) have been inflated to 29 lines of code--a bloat factor of almost 400 percent. Worse, there's so much error detection, reporting, and returning that the original 7 lines of code are lost in the clutter. And worse yet, the logical flow of the code has also been lost in the clutter, making it difficult to tell if the code is doing the right thing: Is the file *really* being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing after you modify the function three months after writing it. Many programmers "solve" this problem by simply ignoring it--errors are "reported" when their programs crash.

Java provides an elegant solution to the problem of error management: exceptions. Exceptions enable you to write the main flow of your code and deal with the, well, exceptional cases elsewhere. If your `read_file` function used exceptions instead of traditional error management techniques, it would look something like this:

```

readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    }
}

```

```

        } catch (sizeDeterminationFailed) {
            doSomething;
        } catch (memoryAllocationFailed) {
            doSomething;
        } catch (readFailed) {
            doSomething;
        } catch (fileCloseFailed) {
            doSomething;
        }
    }
}

```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors. What exceptions do provide for you is the means to separate all the grungy details of what to do when something out-of-the-ordinary happens from the main logic of your program.

In addition, the bloat factor for error management code in this program is about 250 percent--compared to 400 percent in the previous example.

Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile` method is the fourth method in a series of nested method calls made by your main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`.

```

method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call readFile;
}

```

Suppose also that `method1` is the only method interested in the errors that occur within `readFile`. Traditional error notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`--the only method that is interested in them.

```

method1 {

```

```

        errorCodeType error;
        error = call method2;
        if (error)
            doErrorProcessing;
        else
            proceed;
    }
    errorCodeType method2 {
        errorCodeType error;
        error = call method3;
        if (error)
            return error;
        else
            proceed;
    }
    errorCodeType method3 {
        errorCodeType error;
        error = call readfile;
        if (error)
            return error;
        else
            proceed;
    }
}

```

As you learned earlier, the Java runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception. A Java method can "duck" any exceptions thrown within it, thereby allowing a method further up the call stack to catch it. Thus only the methods that care about errors have to worry about detecting errors.

```

method1 {
    try {
        call method2;
    } catch (exception) {
        doErrorProcessing;
    }
}
method2 throws exception {
    call method3;
}
method3 throws exception {
    call readfile;
}

```

}

However, as you can see from the pseudo-code, ducking an exception does require some effort on the part of the "middleman" methods. Any checked exceptions that can be thrown within a method are part of that method's public programming interface and must be specified in the throws clause of the method. Thus a method informs its callers about the exceptions that it can throw, so that the callers can intelligently and consciously decide what to do about those exceptions.

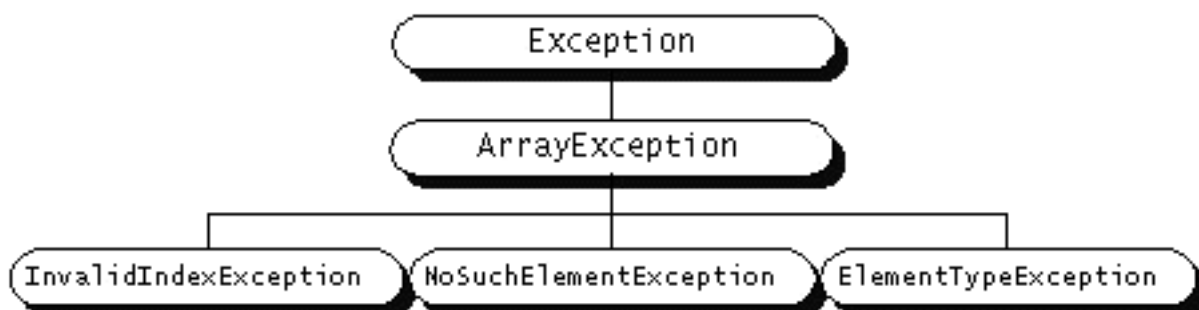
Note again the difference in the bloat factor and code obfuscation factor of these two error management techniques. The code that uses exceptions is more compact and easier to understand.

Advantage 3: Grouping Error Types and Error Differentiation

Often exceptions fall into categories or groups. For example, you could imagine a group of exceptions, each of which represents a specific type of error that can occur when manipulating an array: the index is out of range for the size of the array, the element being inserted into the array is of the wrong type, or the element being searched for is not in the array. Furthermore, you can imagine that some methods would like to handle all exceptions that fall within a category (all array exceptions), and other methods would like to handle specific exceptions (just the invalid index exceptions, please).

Because all exceptions that are thrown within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy. Java exceptions must be instances of Throwable or any Throwable descendant. As for other Java classes, you can create subclasses of the Throwable class and subclasses of your subclasses. Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions.

For example, in the following diagram, ArrayException is a subclass of Exception (a subclass of Throwable) and has three subclasses.



InvalidIndexException, ElementTypeException, and NoSuchElementException are all leaf

classes. Each one represents a specific type of error that can occur when manipulating an array. One way a method can catch exceptions is to catch only those that are instances of a leaf class. For example, an exception handler that handles only invalid index exceptions has a catch statement like this:

```
catch (InvalidIndexException e) {
    . . .
}
```

`ArrayException` is a node class and represents any error that can occur when manipulating an array object, including those errors specifically represented by one of its subclasses. A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all array exceptions regardless of their specific type, an exception handler would specify an `ArrayException` argument:

```
catch (ArrayException e) {
    . . .
}
```

This handler would catch all array exceptions including `InvalidIndexException`, `ElementTypeException`, and `NoSuchElementException`. You can find out precisely which type of exception occurred by querying the exception handler parameter `e`. You could even set up an exception handler that handles any `Exception` with this handler:

```
catch (Exception e) {
    . . .
}
```

Exception handlers that are too general, such as the one shown here, can make your code more error prone by catching and handling exceptions that you didn't anticipate and therefore are not correctly handled within the handler. We don't recommend writing general exception handlers as a rule.

As you've seen, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

What's Next?

Now that you understand what exceptions are and the advantages of using exceptions in

**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

Your First Encounter with Java Exceptions

The following error message is one of two similar error messages you will see if you try to compile the class `InputFile`, because the `InputFile` class contains calls to methods that throw exceptions when an error occurs:

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException
must be caught, or it must be declared in throws clause of this method.
    in = new FileReader(filename);
        ^
```

The Java language requires that methods either *catch* or *specify* all checked exceptions that can be thrown within the scope of that method. (Details about what this actually means are covered in the next section, [Java's Catch or Specify Requirement](#).) If the compiler detects a method, such as those in `InputFile`, that doesn't meet this requirement, it issues an error message like the one shown above and refuses to compile the program.

Let's look at `InputFile` in more detail and see what's going on.

The `InputFile` class wraps a `FileReader` and provides a method, `getWord`, for reading a word from the current position in the reader.

```
// Note: This class won't compile by design!
import java.io.*;

public class InputFile {

    private FileReader in;

    public InputFile(String filename) {
        in = new FileReader(filename);
    }

    public String getWord() {
        int c;
        StringBuffer buf = new StringBuffer();

        do {
            c = in.read();
            if (Character.isWhitespace((char)c))
```

```

        return buf.toString();
    else
        buf.append((char)c);
    } while (c != -1);

    return buf.toString();
}
}

```

The compiler prints the first error message because of the bold line in the above code listing. The bold line creates a new `FileReader` object and uses it to open a file whose name is passed into the `FileReader` constructor.

So what should the `FileReader` do if the named file does not exist on the file system? Well, that depends on what the program using the `FileReader` wants to do. The implementers of `FileReader` have no idea what the `InputFile` class wants to do if the file does not exist. Should the `FileReader` kill the program? Should it try an alternate filename? Should it just create a file of the indicated name? There's no possible way the `FileReader` implementers could choose a solution that would suit every user of `FileReader`. So, they punted, or rather, threw, an exception. If the file named in the argument to the `FileReader` constructor does not exist on the file system, the constructor throws a `java.io.FileNotFoundException`. By throwing an exception, `FileReader` allows the calling method to handle the error in whatever way is most appropriate for it.

As you can see from the code, the `InputFile` class completely ignores the fact that the `FileReader` constructor can throw an exception. However, as stated previously, the Java language requires that a method either catch or specify all checked exceptions that can be thrown within the scope of that method. Because the `InputFile` class does neither, the compiler refuses to compile the program and prints an error message.


In addition to the first error message shown above, you also see the following similar error message when you compile the `InputFile` class:

```

InputFile.java:15: Warning: Exception java.io.IOException
must be caught, or it must be declared in throws clause
of this method.
    while ((c = in.read()) != -1) {
                ^

```

The `InputFile` class's `getWord` method reads from the `FileReader` that was opened in `InputFile`'s constructor. The `FileReader` `read` method throws a `java.io.IOException` if for some reason it can't read from the file. Again, the `InputFile` class makes no attempt to catch or specify this exception. Thus you see the second error message.

At this point, you have two options. You can either arrange to catch the exceptions within the appropriate methods in the `InputFile` class, or the `InputFile` methods can "duck" and allow other methods further up the call stack to catch them. Either way, the `InputFile` methods must do something, either catch or specify the exceptions, before the `InputFile` class can be compiled. For the diligent, there's a class, [InputFileDeclared](#) , that fixes the bugs in `InputFile` by specifying the exceptions.



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

Java's Catch or Specify Requirement

As stated previously, Java requires that a method either catch or specify all checked exceptions that can be thrown within the scope of the method. This requirement has several components that need further description: "catch", "specify," "checked exceptions," and "exceptions that can be thrown within the scope of the method."

Catch

A method can catch an exception by providing an exception handler for that type of exception. The next page, [Dealing with Exceptions](#), introduces an example program, talks about catching exceptions, and shows you how to write an exception handler for the example program.

Specify

If a method chooses not to catch an exception, the method must specify that it can throw that exception. Why did the Java designers make this requirement? Because any exception that can be thrown by a method is really part of the method's public programming interface: callers of a method must know about the exceptions that a method can throw in order to intelligently and consciously decide what to do about those exceptions. Thus, in the method signature you specify the exceptions that the method can throw.

The next page, [Dealing with Exceptions](#), talks about specifying exceptions that a method throws and shows you how to do it.

Checked Exceptions

Java has different types of exceptions, including I/O Exceptions, runtime exceptions, and exceptions of your own creation, to name a few. Of interest to us in this discussion are runtime exceptions. Runtime exceptions are those exceptions that occur within the Java runtime system. This includes arithmetic exceptions (such as when dividing by zero), pointer exceptions (such as trying to access an object through a null reference), and indexing exceptions (such as attempting to access an array element through an index that is too large or too small).

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. The cost of checking for runtime exceptions often exceeds the benefit of catching or specifying them. Thus the compiler does not require that you catch or specify runtime exceptions, although you can. *Checked exceptions* are exceptions that are not runtime exceptions and are checked by the compiler; the compiler checks that these exceptions are caught or specified.

Some consider this a loophole in Java's exception handling mechanism, and programmers are tempted to make all exceptions runtime exceptions. In general, this is not recommended. [Runtime Exceptions--The Controversy](#) contains a thorough discussion about when and how to use runtime exceptions.

Exceptions that can be thrown within the scope of the method

The statement "exceptions that can be thrown within the scope of the method" may seem obvious at first: just look for the throw statement. However, this statement includes more than just the exceptions that can be thrown directly by the method: the key is in the phrase *within the scope of*. This phrase includes any exception that can be thrown while the flow of control remains within the method. This statement includes both

- Exceptions that are thrown directly by the method with Java's throw statement.
- Exceptions that are thrown indirectly by the method through calls to other methods



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

Dealing with Exceptions

[Your First Encounter with Java Exceptions](#) briefly described how you were (probably) first introduced to Java exceptions: a compiler error complaining that you must either catch or specify exceptions. Then, [Java's Catch or Specify Requirement](#) discussed what exactly the error message means and why the Java language designers decided to make this requirement. We're now going to show you both how to catch an exception and how to specify one.

The ListOfNumbers Example

The sections below, both the one on catching an exception and the one on specifying an exception, use the same example. This example defines and implements a class named [ListOfNumbers](#)♦. The ListOfNumbers class calls two methods from classes in the Java packages that can throw exceptions. [Catching and Handling Exceptions](#) will show you how to write exception handlers for both exceptions, and [Specifying the Exceptions Thrown by a Method](#) will show you how to specify those exceptions instead of catching them.

Catching and Handling Exceptions

Once you've familiarized yourself with the ListOfNumbers class and where the exceptions can be thrown, you can learn how to write exception handlers to catch and handle those exceptions.

This section covers the three components of an exception handler--the try, catch, and finally blocks--by showing you how to use them to write an exception handler for the ListOfNumbers class's writeList method. In addition, this section contains a page that walks through the writeList method and analyzes what occurs within the method during various scenarios.

Specifying the Exceptions Thrown by a Method

If it is not appropriate for your method to catch and handle an exception thrown by a method that it calls, or if your method itself throws its own exception, you must specify in the

**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

The ListOfNumbers Example

The following example defines and implements a class named [ListOfNumbers](#). The ListOfNumbers class calls two methods from classes in the Java packages that can throw exceptions.

```
// Note: This class won't compile by design!
// See ListOfNumbersDeclared.java or ListOfNumbers.java
// for a version of this class that will compile.
import java.io.*;
import java.util.Vector;

public class ListOfNumbers {
    private Vector vector;
    private static final int size = 10;

    public ListOfNumbers () {
        vector = new Vector(size);
        for (int i = 0; i < size; i++)
            vector.addElement(new Integer(i));
    }
    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();
    }
}
```

Upon construction, ListOfNumbers creates a Vector that contains ten Integer elements with sequential values 0 through 9. The ListOfNumbers class also defines a method named writeList that writes the list of numbers into a text file called OutFile.txt.

The writeList method calls two methods that can throw exceptions. First, the following line invokes the constructor for FileWriter, which throws an IOException if the file cannot be opened for any reason:

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

Second, the Vector class's elementAt method throws an ArrayIndexOutOfBoundsException if you pass in an index whose value is too small (a negative number) or too large (larger than the number of elements currently contained by the Vector). Here's how ListOfNumbers invokes elementAt:

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

If you try to compile the ListOfNumbers class, the compiler prints an error message about the exception thrown by the FileWriter constructor, but does **not** display an error message about the exception thrown by elementAt. This is because the exception thrown by the FileWriter constructor, IOException, is a checked exception and the exception thrown by the elementAt method, ArrayIndexOutOfBoundsException, is a runtime exception. Java requires that you catch or specify only checked exceptions. For more information, refer to [Java's Catch or Specify Requirement](#).

The next section, [Catching and Handling Exceptions](#), will show you how to write an exception handler for the ListOfNumbers' writeList method.

Following that, a section named [Specifying the Exceptions Thrown By a Method](#) will show you how to specify that the ListOfNumbers' writeList method throws the exceptions instead of catching them.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

Catching and Handling Exceptions

Now that you've familiarized yourself with the `ListOfNumbers` class and where the exceptions can be thrown within it, you can learn how to write exception handlers to catch and handle those exceptions.

The three sections that follow cover the three components of an exception handler -- the try, catch, and finally blocks. They show you how to write an exception handler for the `ListOfNumbers` class's `writeList` method, described in [The ListOfNumbers Example](#). Next, you'll walk through the resulting `writeList` method and see what occurs within the example code during various scenarios.

The try Block

The first step in writing an exception handler is to enclose the statements that might throw an exception within a try block. The try block is said to *govern* the statements enclosed within it and defines the scope of any exception handlers (established by subsequent catch blocks) associated with it.

The catch Block(s)

Next, you associate exception handlers with a try block by providing one or more catch blocks directly after the try block.

The finally Block

Java's finally block provides a mechanism that allows your method to clean up after itself regardless of what happens within the try block. Use the finally block to close files or release other system resources.

Putting It All Together

The previous sections describe how to construct the try, catch, and finally code blocks for the `writeList` example. Now, let's walk through the code and investigate what happens

**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

The try Block

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

```
try {  
    Java statements  
}
```

The segment of code labelled *Java statements* is composed of one or more legal Java statements that could throw an exception.

To construct an exception handler for the `writeList` method from the `ListOfNumbers` class, you need to enclose the exception-throwing statements of the `writeList` method within a try block. There is more than one way to accomplish this task. You could put each statement that might potentially throw an exception within its own try statement, and provide separate exception handlers for each try. Or you could put all of the `writeList` statements within a single try statement and associate multiple handlers with it. The following listing uses one try statement for the entire method because the code tends to be easier to read.

```
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 + " = " + victor.elementAt(i));  
}
```

The try statement governs the statements enclosed within it and defines the scope of any exception handlers associated with it. In other words, if an exception occurs within the try statement, that exception is handled by the appropriate exception handler associated with this try statement.

A try statement *must* be accompanied by at least one catch block or one finally block.


Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

The catch Block(s)

As you learned on the [previous page](#), the try statement defines the scope of its associated exception handlers. You associate exception handlers with a try statement by providing one or more catch blocks directly after the try block:

```
try {
    . . .
} catch ( . . . ) {
    . . .
} catch ( . . . ) {
    . . .
} . . .
```

There can be no intervening code between the end of the try statement and the beginning of the first catch statement. The general form of Java's catch statement is:

```
catch (SomeThrowableObject variableName) {
    Java statements
}
```

As you can see, the catch statement requires a single formal argument. The argument to the catch statement looks like an argument declaration for a method. The argument type, *SomeThrowableObject*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the [Throwable](#) class defined in the [java.lang](#) package. When Java programs throw an exception they are really just throwing an object, and only objects that derive from Throwable can be thrown. You'll learn more about throwing exceptions in [How to Throw Exceptions](#).

variableName is the name by which the handler can refer to the exception caught by the handler. For example, the exception handlers for the writeList method (shown later) each call the exception's getMessage method using the exception's declared name e:

```
e.getMessage( )
```

You access the instance variables and methods of exceptions in the same manner that you access

the instance variables and methods of other objects. `getMessage` is a method provided by the `Throwable` class that prints additional information about the error that occurred. The `Throwable` class also implements two methods for filling in and printing the contents of the execution stack when the exception occurred. Subclasses of `Throwable` can add other methods or instance variables. To find out what methods an exception implements, check its class definition and definitions for any of its ancestor classes.

The catch block contains a series of legal Java statements. These statements are 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 type matches that of the exception thrown.

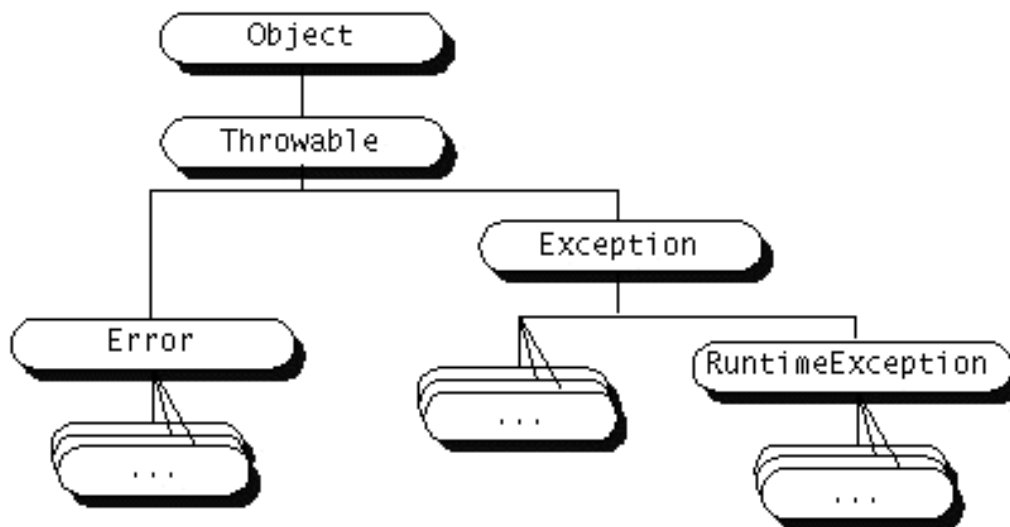
The `writeList` method from the [ListOfNumbers](#) class uses two exception handlers for its try statement, with one handler for each of the two types of exceptions that can be thrown within the try block -- `ArrayIndexOutOfBoundsException` and `IOException`.

```
try {
    . . .
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Caught ArrayIndexOutOfBoundsException: " +
        e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " +
        e.getMessage());
}
```

Catching Multiple Exception Types with One Handler

The two exception handlers used by the `writeList` method are very specialized. Each handles only one type of exception. The Java language allows you to write general exception handlers that handle multiple types of exceptions.

As you know, Java exceptions are `Throwable` objects; they are instances of `Throwable` or a subclass of `Throwable`. The Java packages contain numerous classes that derive from `Throwable` and thus, build a hierarchy of `Throwable` classes.



Your exception handler can be written to handle any class that inherits from Throwable. If you write a handler for a "leaf" class (a class with no subclasses), you've written a specialized handler: it will only handle exceptions of that specific type. If you write a handler for a "node" class (a class with subclasses), you've written a general handler: it will handle any exception whose type is the node class or any of its subclasses.

Let's modify the writeList method once again. Only this time, let's write it so that it handles both IOExceptions and ArrayIndexOutOfBoundsExceptions. The closest common ancestor of IOException and ArrayIndexOutOfBoundsException is the Exception class. An exception handler that handles both types of exceptions looks like this:

```

try {
    . . .
} catch (Exception e) {
    System.err.println("Exception caught: " + e.getMessage());
}

```

The Exception class is pretty high in the Throwable class hierarchy. So in addition to the IOException and ArrayIndexOutOfBoundsException types that this exception handler is intended to catch, it will catch numerous other types. Generally speaking, your exception handlers should be more specialized. Handlers that can catch most or all exceptions are typically useless for error recovery because the handler has to determine what type of exception occurred anyway to determine the best recovery strategy. Also, exception handlers that are too general can make code *more* error prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

The finally Block

The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program. You do this by enclosing the cleanup code within a finally block.

The try block of the writeList method that you've been working with opens a PrintWriter. The program should close that stream before allowing control to pass out of the writeList method. This poses a somewhat complicated problem because writeList's try block has three different exit possibilities:

1. The new FileWriter statement failed and threw an IOException.
2. The victor.elementAt(i) statement failed and threw an ArrayIndexOutOfBoundsException.
3. Everything succeeded and the try block exited normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. Regardless of whether control exits the writeList method's try block due to one of the three scenarios listed previously, the code within the finally block will be executed.

This is the finally block for the writeList method. It cleans up and closes the PrintWriter.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

Is the finally Statement Really Necessary?

At first the need for a finally statement may not be immediately apparent. Programmers often ask "Is the finally statement really necessary or is it just sugar for my Java?" In particular, C++

programmers doubt the need for a finally statement because C++ doesn't have one.

The need for a finally statement is not apparent until you consider the following: how does the `PrintWriter` in the `writeList` method get closed if you don't provide an exception handler for the `ArrayIndexOutOfBoundsException` and an `ArrayIndexOutOfBoundsException` occurs? (It's easy and legal to omit an exception handler for `ArrayIndexOutOfBoundsException` because it's a runtime exception and the compiler won't alert you that the `writeList` contains a method call that might throw one.) The answer is that the `PrintWriter` does not get closed if an `ArrayIndexOutOfBoundsException` occurs and `writeList` does not provide a handler for it-- unless the `writeList` provides a finally statement.

There are other benefits to using the finally statement. In the `writeList` example it is possible to provide for cleanup without the intervention of a finally statement. For example, you could put the code to close the `PrintWriter` at the end of the try block and again within the exception handler for `ArrayIndexOutOfBoundsException`, as shown here:

```
try {
    . . .
    out.close();          // don't do this; it duplicates code
} catch (ArrayIndexOutOfBoundsException e) {
    out.close();          // don't do this; it duplicates code
    System.err.println("Caught ArrayIndexOutOfBoundsException: "
        + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

However, this duplicates code, making the code hard to read and prone to errors if you modify the code later. For example, if you add code to the try block that may throw a new type of exception, you will have to remember to close the `PrintWriter` within the new exception handler (which if you're anything like me, you are bound to forget).



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

Putting It All Together

The previous sections describe how to construct the try, catch, and finally code blocks for the writeList example. Now, let's walk through the code and investigate what happens during three scenarios.

When all of the components are put together, the writeList method looks like this:

```
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 + " = " + victor.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");
        }
    }
}
```

This try block in this method has three different exit possibilities:

1. The new FileWriter statement fails and throws an IOException.
2. The victor.elementAt(i) statement fails and throws an ArrayIndexOutOfBoundsException.
3. Everything succeeds and the try statement exits normally.

This section investigates in detail what happens in the writeList method during each of those exit possibilities.

Scenario 1: An IOException Occurs

The new `FileWriter("OutFile.txt")` statement can fail for any number of reasons: the user doesn't have write permission on the file or directory, the file system is full, or the directory for the file doesn't exist. If any of these situations is true, then the constructor for `FileWriter` throws an `IOException`.

When the `IOException` is thrown, the runtime system immediately stops execution of the try block. Then the runtime system attempts to locate an exception handler appropriate for handling an `IOException`.

The runtime system begins its search at the top of the method call stack. When the exception occurred, the `FileWriter` constructor was 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 in the method call stack--the `writeList` method. The `writeList` method has two exception handlers: one for `ArrayIndexOutOfBoundsException` and one for `IOException`.

The runtime system checks `writeList`'s handlers in the order that they appear following the try statement. The argument to the first exception handler is `ArrayIndexOutOfBoundsException`, but the exception that was thrown is an `IOException`. An `IOException` cannot legally be assigned to an `ArrayIndexOutOfBoundsException`, so the runtime system continues its search for an appropriate exception handler.

The argument to `writeList`'s second exception handler is an `IOException`. The exception thrown by the `FileWriter` constructor is also an `IOException` and can be legally assigned to the handler's `IOException` argument. Thus, this handler is deemed appropriate and the runtime system executes this handler, which prints this statement:

```
Caught IOException: OutFile.txt
```

After the exception handler has run, the runtime system passes control to the finally block. In this particular scenario, the `PrintWriter` never was opened, and thus `out` is null and won't get closed. After the finally block has completed executing, the program continues with the first statement after the finally block.

The complete output that you see from the `ListOfNumbers` program when an `IOException` is thrown is this:

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

Scenario 2: An ArrayIndexOutOfBoundsException Occurs

This scenario is the same as the first except that a different error occurs during the try block. In this scenario, the argument passed to the `Vector`'s `elementAt` method is out of bounds. That is, the argument is either less than 0 or is larger than the size of the array. (The way the code is written, this is actually impossible, but let's suppose a bug is introduced into the code when someone modifies it.)

As in to scenario 1, when the exception occurs the runtime system stops execution of the try block and attempts to locate an exception handler suitable for an `ArrayIndexOutOfBoundsException`. The runtime system searches for an appropriate exception handler as it did before. It comes upon the catch statement in

the `writeList` method that handles exceptions of the type `ArrayIndexOutOfBoundsException`. Since the type of the thrown exception matches the type of the exception handler, the runtime system executes this exception handler.

After the exception handler has run, the runtime system passes control to the `finally` block. In this particular scenario, the `PrintWriter` did get opened, thus the `finally` statement closes it. After the `finally` block has completed executing, the program continues with the first statement after the `finally` block.

The complete output that you see from the `ListOfNumbers` program when an `ArrayIndexOutOfBoundsException` is thrown is something like this:

```
Entering try statement
Caught ArrayIndexOutOfBoundsException: 10 >= 10
Closing PrintWriter
```

Scenario 3: 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 then the runtime system passes control to the `finally` block. Since everything was successful, the `PrintWriter` is open when control reaches the `finally` block, which closes the `PrintWriter`. Again, after the `finally` block has completed executing, the program continues with the first statement after the `finally` block.

Thus, the output that you see from the `ListOfNumbers` program when no exceptions are thrown is:

```
Entering try statement
Closing PrintWriter
```



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

How to Throw Exceptions

The pages listed below explain how to throw exceptions in a Java program.

[The throw Statement](#)

Before you can catch an exception, some Java code somewhere must throw one. Any Java code can throw an exception: your code, code from a package written by someone else (such as the packages that come with the Java development environment), or the Java runtime system. Regardless of who (or what) throws the exception, it's always thrown with the Java throw statement.

[The Throwable Class and Its Subclasses](#)

The Java language requires that exceptions derive from the Throwable class or one of its subclasses.

[Creating Your Own Exception Classes](#)

As you have undoubtedly noticed in your travels (travails?) through the Java language, the packages that ship with the Java development environment provide numerous exception classes. All of these classes are descendants of the Throwable class and allow programs to differentiate between the various types of exceptions that can occur during the execution of a Java program.

You can create your own exception classes, as well, to represent problems that can occur within the classes that you write. Indeed, if you are a package developer you will find that you must create your own set of exception classes to allow your users to differentiate an error that can occur in your package versus those errors that occur in the Java development environment or other packages.



**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

The throw Statement

All Java methods use the throw statement to throw an exception. The throw statement requires a single argument: a *throwable* object. In the Java system, throwable objects are instances of any subclass of the [Throwable](#) class. Here's an example of a throw statement:

```
throw someThrowableObject;
```

If you attempt to throw an object that is not throwable, the compiler refuses to compile your program and displays an error message similar to the following:

```
testing.java:10: Cannot throw class java.lang.Integer;
it must be a subclass of class java.lang.Throwable.
    throw new Integer(4);
           ^
```

The next page, [The Throwable Class and Its Subclasses](#), talks more about the Throwable class.

[PENDING: EmptyStackException is not a checked exception, so technically pop doesn't have to use the throws clause]

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

```
public Object pop() throws EmptyStackException {
    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 if there are any elements on the stack. If the stack is empty (its size is equal to 0), then pop instantiates a new `EmptyStackException` object and throws it. The `EmptyStackException` class is defined in the `java.util` package. Later pages in this lesson describe how you can create your own exception classes. For now, all you really need to remember is that you can throw only objects that inherit from the `java.lang.Throwable` class.

The throws Clause

You'll notice that the declaration of the pop method contains this clause:

```
throws EmptyStackException
```

The throws clause specifies that the method can throw an `EmptyStackException`. As you know, the Java language requires that methods either catch or specify all checked exceptions that can be thrown within the scope of that method. You do this with the throws clause of the method declaration. For more information about this requirement see [Java's Catch or Specify Requirement](#). Also, [Specifying the Exceptions Thrown by a Method](#) shows you in more detail how a method can specify the exceptions it can throw.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



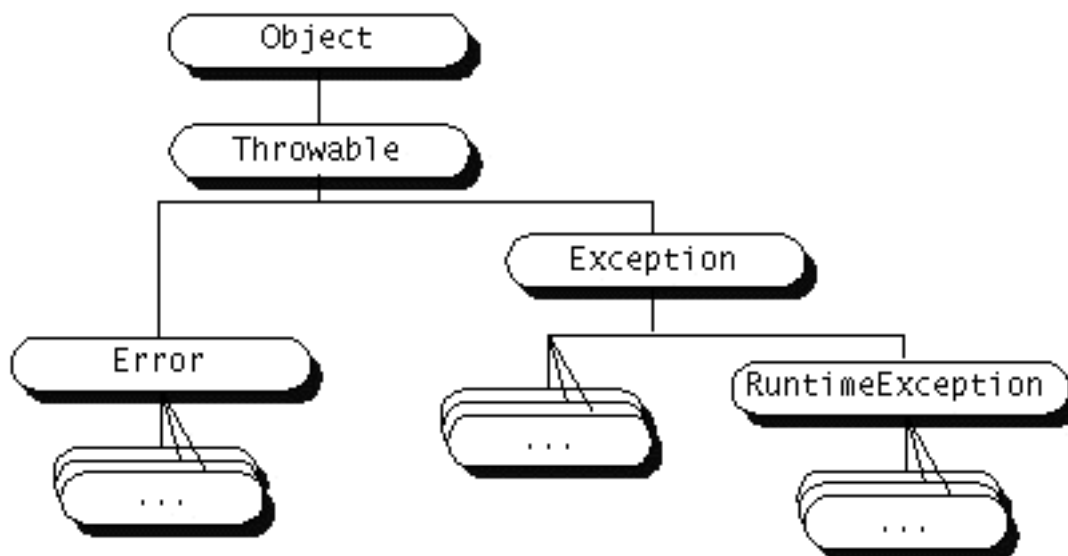
Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

The Throwable Class and Its Subclasses

As you learned on the previous page, you can throw only objects that derive from the Throwable class. This includes direct descendants (that is, objects that derive directly from the Throwable class) as well as indirect descendants (objects that derive from children or grandchildren of the Throwable class).

This diagram illustrates the class hierarchy of the Throwable class and its most significant subclasses.



As you can see from the diagram, Throwable has two direct descendants: Error and Exception.

Errors

When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error. Typical Java programs should not catch Errors. In addition, it's unlikely that typical Java programs will ever throw Errors either.

Exceptions

Most programs throw and catch objects that derive from the Exception class. Exceptions

indicate that a problem occurred but that the problem is not a serious systemic problem. Most programs you write will throw and catch Exceptions.

The Exception class has many descendants defined in the Java packages. 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 has special meaning in the Java language: `RuntimeException`.

Runtime Exceptions

The `RuntimeException` class represents exceptions that occur within the Java virtual machine (during runtime). 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. A `NullPointerException` can occur anywhere a program tries to dereference a reference to an object. The cost of checking for the exception often outweighs the benefit of catching it.

Because runtime exceptions are so ubiquitous and attempting to catch or specify all of them all the time would be a fruitless exercise (and a fruitful source of unreadable and unmaintainable code), the compiler allows runtime exceptions to go uncaught and unspecified.

The Java packages define several `RuntimeException` classes. You can catch these exceptions just like other exceptions. However, a method is not required to specify that it throws `RuntimeExceptions`. In addition, you can create your own `RuntimeException` subclasses. [Runtime Exceptions--The Controversy](#) contains a thorough discussion of when and how to use runtime exceptions.



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

Creating Your Own Exception Classes

When you design a package of Java classes that collaborate to provide some useful function to your users, you work hard to ensure that your classes interact well together and that their interfaces are easy to understand and use. You should spend just as much time thinking about and designing the exceptions that your classes throw.

Suppose you are writing a linked list class that you're planning to distribute as freeware. Among other methods, your linked list class supports these methods:

objectAt(int *n*)

Returns the object in the *n*th position in the list.

firstObject

Returns the first object in the list.

indexOf(Object *n*)

Searches the list for the specified Object and returns its position in the list.

What Can Go Wrong?

Because many programmers will be using your linked list class, you can be assured that many will misuse or abuse your class and its methods. Also, some legitimate calls to your linked list's methods may result in an undefined result. Regardless, in the face of errors, you want your linked list class to be as robust as possible, to do something reasonable about errors, and to communicate errors back to the calling program. However, you can't anticipate how each user of your linked list class will want the object to behave under adversity. So, often the best thing to do when an error occurs is to throw an exception.

Each of the methods supported by your linked list might throw an exception under certain conditions, and each method might throw a different type of exception than the others. For example,

objectAt

Throws an exception if the integer passed into the method is less than 0 or larger than the number of objects currently in the list.

firstObject

Throws an exception if the list contains no objects.

indexOf

Throws an exception if the object passed into the method is not in the list.

But what type of exception should each method throw? Should it be an exception provided with the Java development environment? Or should you create your own?

Choosing the Exception Type to Throw

When faced with choosing the type of exception to throw, you have two choices:

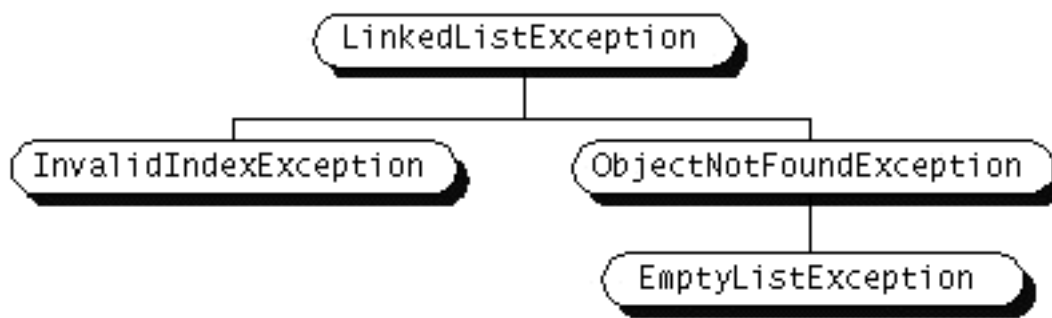
1. Use one written by someone else. The Java development environment provides a lot of exception classes that you could use.
2. Write one of your own.

You should go to the trouble of writing your own exception classes if you answer "yes" to any of the following questions. Otherwise, you can probably get away with using someone else's:

- Do you need an exception type that isn't represented by those in the Java development environment?
- Would it help your 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 your users have access to those exceptions? A similar question is: Should your package be independent and self-contained?

Your 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 for the linked list, you should create your own exception class hierarchy.

The following diagram illustrates one possible exception class hierarchy for your linked list:



LinkedListException is the parent class of all the possible exceptions that can be thrown by the linked list class. Users of your linked list class can write a single exception handler to

handle all linked list exceptions with a catch statement like this:

```
catch (LinkedListException) {
    . . .
}
```

Or, users could write more specialized handlers for each subclass of `LinkedListException`.

Choosing a Superclass

The diagram above does not indicate the superclass of the `LinkedListException` class. As you know, Java exceptions must be `Throwable` objects (they must be instances of `Throwable` or a subclass of `Throwable`). So, your temptation might be to make `LinkedListException` a subclass of `Throwable`. However, the `java.lang` package provides two `Throwable` subclasses that further divide the type of problems that can occur within a Java program: `Errors` and `Exceptions`. Most of the applets and applications that you write will throw objects that are `Exceptions`. (`Errors` are reserved for serious hard errors that occur deep in the system.)

Theoretically, any `Exception` subclass could be used as the parent class of `LinkedListException`. However, a quick perusal of those classes show that they are either too specialized or completely unrelated to `LinkedListException` to be appropriate. Thus, the parent class of `LinkedListException` should be `Exception`.

Because runtime exceptions don't have to be specified in the `throws` clause of a method, many packages developers ask: "Isn't it just easier if I make all of my exception inherit from `RuntimeException`?" The answer to this question is covered in detail on [Runtime Exceptions - The Controversy](#). The bottom line is that you shouldn't subclass `RuntimeException` unless your class really is a runtime exception! For most of you, this means "No, your exceptions shouldn't inherit from `RuntimeException`."

Naming Conventions

It's good practice to append the word "Exception" to the end of all classes that inherit (directly or indirectly) from the `Exception` class. Similarly, classes that inherit from the `Error` class should end with the string "Error".



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.

**Trail:** Essential Java Classes**Lesson:** Handling Errors with Exceptions

Runtime Exceptions--The Controversy

Because the Java language does not require methods to catch or specify runtime exceptions, it's tempting for programmers to write code that throws only runtime exceptions or to make all of their exception subclasses inherit from `RuntimeException`. Both of these programming shortcuts allow programmers to write Java code without bothering with all of the nagging errors from the compiler and without bothering to specify or catch any exceptions. While this may seem convenient to the programmer, it sidesteps the intent of Java's catch or specify requirement and can cause problems for the programmers using your classes.

```
InputFile.java:8: Warning: Exception
java.io.FileNotFoundException must be caught, or it must be
declared in throws clause of this method.
    fis = new FileInputStream(filename);
           ^
```

Why did the Java designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Because any exception that can be thrown by a method is really part of the method's public programming interface: callers of a method must know about the exceptions that a method can throw in order to intelligently and consciously decide what to do about those exceptions. The exceptions that a method can throw are as much a part of that method's programming interface as its parameters and return value.

Your next question might be: "Well then, if it's so good to document a method's API including the exceptions that it can throw, why not specify runtime exceptions, too?"

Runtime exceptions represent problems that are detected by the runtime system. This includes arithmetic exceptions (such as when dividing by zero), pointer exceptions (such as trying to access an object through a null reference), and indexing exceptions (such as attempting to access an array element through an index that is too large or too small).

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. Typically, the cost of checking for runtime exceptions exceeds the benefit of catching or specifying them. Thus the compiler does not require that you catch or specify

runtime exceptions, although you can.

Checked exceptions represent useful information about the operation of a legally specified request that the caller may have had no control over and that the caller needs to be informed about--for example, the file system is now full, or the remote end has closed the connection, or the access privileges don't allow this action.

What does it buy you if you throw a `RuntimeException` or create a subclass of `RuntimeException` just because you don't want to deal with specifying it? Simply, you get the ability to throw an exception without specifying that you do so. In other words, it is a way to avoid documenting the exceptions that a method can throw. When is this good? Well, when is it ever good to avoid documenting a method's behavior? The answer is "hardly ever."

Rules of Thumb:

- A method can detect and throw a `RuntimeException` when it's encountered an error in the virtual machine runtime. However, it's typically easier to just let the virtual machine detect and throw it. Normally, the methods you write should throw `Exceptions`, not `RuntimeException`.
 - Similarly, you create a subclass of `RuntimeException` when you are creating an error in the virtual machine runtime (which you probably aren't). Otherwise you should subclass `Exception`.
 - Do not throw a runtime exception or create a subclass of `RuntimeException` simply because you don't want to be bothered with specifying the exceptions your methods can throw.
-



[Start of Tutorial](#) > [Start of Trail](#) > [Start of Lesson](#)

[Search](#)
[Feedback Form](#)

[Copyright](#) 1995-2003 Sun Microsystems, Inc. All rights reserved.



Trail: Essential Java Classes

Lesson: Handling Errors with Exceptions

Questions and Exercises: Exceptions

Questions

1. Is the following code legal?

```
try {  
    ...  
} finally {  
    ...  
}
```

2. What exception types can be caught by the following handler?

```
catch (Exception e) {  
    ...  
}
```

What is wrong with using this type of exception handler?

3. What exceptions can be caught by the following handler?

```
...  
} catch (Exception e) {  
    ...  
} catch (ArithmeticException a) {  
    ...  
}
```

Is there anything wrong with this exception handler as written? Will this code compile?

4. Match each situation in the first column with an item in the second column.

- | | |
|---|---|
| <ol style="list-style-type: none"> 1. <code>int[] A;</code>
<code>A[0] = 0;</code> 2. The Java VM starts running your program, but the VM can't find the Java platform classes. (The Java platform classes reside in <code>classes.zip</code> or <code>rt.jar</code>.) 3. A program is reading a stream and reaches the end of stream marker. 4. Before closing the stream and after reaching the end of stream marker, a program tries to read the stream again. | <ol style="list-style-type: none"> 1. error 2. checked exception 3. runtime exception 4. no exception |
|---|---|

Exercises

1. Add a `readList` method to [ListOfNumbers.java](#)◆. This method should read in `int` values from a file, print each value, and append them to the end of the vector. You should catch all appropriate errors. You will also need a text file containing numbers to read in.
2. Modify the following `cat` method so that it will compile:

```
public static void cat(File named) {
    RandomAccessFile input = null;
    String line = null;

    try {
        input = new RandomAccessFile(named, "r");
        while ((line = input.readLine()) != null) {
            System.out.println(line);
        }
        return;
    } finally {
        if (input != null) {
            input.close();
        }
    }
}
```

[Check your answers.](#)◆