Handling Errors using Exceptions

Table Of Contents

- Handling Errors using Exceptions
  - What's an Exception and Why Do I Care?
  - Your First Encounter with Java Exceptions
  - Java's Catch or Declare Requirement
  - Dealing with Exceptions
    - The Example
    - Catching and Handling Exceptions
      - The try Block
      - The catch Block(s)
      - The finally Block
      - Putting It All Together
    - Declaring the Exceptions Thrown by a Method
  - How to Throw Exceptions
    - The throw Statement
    - The Throwable Class and Its Subclasses
    - Creating Your Own Exception Classes
  - Runtime Exceptions––The Controversy

Handling Errors using Exceptions

Table of Contents

Handling Errors using Exceptions

If there's one 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?

The Java language uses *exceptions* to provide error handling capabilities for its programs. In this lesson you will learn what an exception is, how to throw and catch exceptions, what to do with an exception once you've caught it, and how to best use the exception class hierarchy provided by the Java development environment.

Note on terminology: Some languages (or their development environments) use the terms *raise* and *handle* the same way that Java uses *throw* and *catch*. Java inherits its terminology and much of its exception syntax from C++.

What's an Exception and Why Do I Care?

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

Now that you understand what exceptions are and the advantages of using exceptions in your Java programs, let's start writing some Java code to put exceptions to use.

If you have done any amount of Java programming at all, you have undoubtedly seen an error message similar to this one:

```
FileWithGetLine.java:8: Warning: Exception java.io.FileNotFoundException must be caught, or it
        fis = new FileInputStream(filename);
              ^
```

This message indicates that the compiler found an exception that is not being dealt with. The Java language requires that all non−runtime exceptions be caught or declared. At first, this may seem overburdensome to you, however, there are advantages to this requirement.

Java's Catch or Declare Requirement

Java requires that your programs either catch or declare all non−runtime exceptions that can be thrown within the scope of that method.

Dealing with Exceptions

This page introduces an example that we will use to show you how to catch an exception and handle it, and alternatively, how to declare an exception that can be thrown within a method.

Throwing Exceptions

So far, you've been catching exceptions thrown by the Java runtime system or classes from the Java packages. This section shows you how you can throw exceptions from your own Java code using the `throw` statement. All exceptions must be *Throwable*, that is, exceptions must inherit (either directly or indirectly) from the Throwable class defined in the java.lang package.

Runtime Exceptions--The Controversy

Because the Java language does not require methods to catch or declare runtime exceptions, it's tempting for programmers to write code that always throw runtime exceptions, or make all of their exception subclasses inherit from RuntimeException. Both of these are "exception abuse" and are not recommended.

Note to C++ Programmers: Java exception handlers can have a finally block which allows programs to cleanup after the `try` block. See The finally Block for more information about how to use the `finally` statement.

Table of Contents

Handling Errors using Exceptions

What's an Exception and Why Do I Care?

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

Note on terminology: Some languages (or their development environments) use the terms *raise* and *handle* the same way that Java uses *throw* and *catch*. Java inherits its terminology and much of its exception syntax from C++.

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: Separate Error Handling Code from "Regular" Code
- Advantage 2: Propagate Errors Up the Call Stack
- Advantage 3: Grouping Error Types and Error Differentiation

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

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

```
read_file {
    openTheFile;
    determine its size;
    allocate that much memory;
    read the file into memory;
    closeTheFile;
}
```

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 read_file {
    initialize errorCode = 0;
    openTheFile;
    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;
        }
        closeTheFile;
        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?) and 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:

```
read_file {
    try {
        openTheFile;
        determine its size;
        allocate that much memory;
        read the file into memory;
        closeTheFile;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Notice 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.

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: Propagate 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 `read_file` method was the fourth method in a series of nested method calls made by your main program: `method1` called `method2` which called `method3` which finally called `read_file`.

```
method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call read_file;
}
```

Suppose also, that `method1` was the only method interested in the errors that occurred within `read_file`. Traditional error notification techniques force `method2` and `method3` to propagate the error codes returned by `read_file` up the call stack until the error codes finally reached `method1`--the only method that was 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 read_file;
}
```

However, as you can see from the psuedo−code, ducking an exception does require some effort on the part of the "middleman" methods. Any non−runtime exceptions that can be thrown within a method is part of that method's public programming interface and must be declared 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 conciously decide what to do about those exceptions.

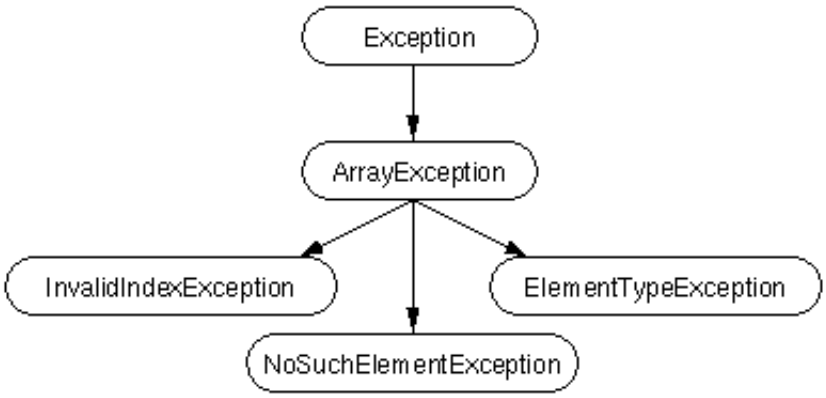
Notice again the difference in the bloat factors of these two error management techniques and the code obfuscation factor. The code that uses exceptions is more compact and easier to understand.

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 represented a specific type of error that can occur when manipulating an array: the index was out of range for the size of the array, the element being inserted into the array was of the wrong type, or the element being searched for was 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 classes and superclasses. Java exceptions must be Throwable, that is, they must be instances of Throwable or any Throwable subclass. Like 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 and each one represents a very specific type of error that can occur when manipulating an array. A method can catch an exception based on its specific type (its immediate class or interface). For example, if your wanted to write an exception handler that handled only invalid index exceptions, your `catch` statement would look like this:

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

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

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

This handler would catch all array exceptions including InvalidIndexException, ElementTypeException and NoSuchElementException. You could even set up an exception handler that handled any Exception with this handler

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

but it is not recommended.

So 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 your Java programs, it's time to learn how.

Handling Errors using Exceptions

Your First Encounter with Java Exceptions

This error message

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException must be caught, or it must be
        fis = new FileInputStream(filename);
              ^
```

is one of two similar error messages you will see if you try to compile this class, InputFile, because the InputFile class contains calls to methods that will throw exceptions when an error occurs. The Java language requires that methods either *catch* or *declare* all non–runtime exceptions that can be thrown within the scope of that method (details about what this actually means are covered on the next page, Java's Catch or Declare Requirement). If the compiler detects a method, such as those in InputFile, that don'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 FileInputStream and provides a method, `getLine()`, for reading a line from the current position in the input stream.

```
import java.io.*;

class InputFile {

    FileInputStream fis;

    InputFile(String filename) {
        fis = new FileInputStream(filename);
    }

    String getLine() {
        int c;
        StringBuffer buf = new StringBuffer();

        while ((c = fis.read()) != -1) {
            if (c == '\n') {
                return buf.toString();
            } else
                buf.append((char)c);
        }
        return null;
    }
}
```

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

So, what should the FileInputStream do if the named file does not exist on the filesystem? Well, that depends on what the program using the FileInputStream wants to do. The implementers of FileInputStream have no idea what the InputFile class wants to do if the file does not exist. Should the FileInputStream 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 FileInputStream implementers could choose a solution that would suit every user of FileInputStream. So, they punted, or rather, threw, an exception. That is, if the file named in the argument to the FileInputStream constructor does not exist on the filesystem, the constructor throws a java.io.FileNotFoundException. By throwing an exception, FileInputStream 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 FileInputStream constructor can throw an exception. However, as stated previously, the Java language requires that methods either catch or declare all non−runtime exceptions that can be thrown within the scope of that method. Because the InputFile class does neither, the compiler prints an error message to that effect when it encounters the call to the FileInputStream constructor and refuses to compile the class.

In addition to the first error message shown above, you will also see this 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
        while ((c = fis.read()) != -1) {
                            ^
```

The InputFile class's `getLine()` method reads from the FileInputStream that was opened in InputFile's constructor. The FileInputStream `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 declare 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 another method further up the call stack to catch them. Either way, the InputFile methods must do something, either catch or declare the exceptions, before the InputFile class can be compiled.

The next page describes in further detail Java's Catch or Declare Requirement. The subsequent pages show you how to comply to the requirement.

InputFile.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
import java.io.*;

class InputFile {

    FileInputStream fis;

    InputFile(String filename) {
        fis = new FileInputStream(filename);
    }

    String getLine() {
        int c;
        StringBuffer buf = new StringBuffer();

        while ((c = fis.read()) != -1) {
            if (c == '\n') {
                return buf.toString();
            } else
                buf.append((char)c);
        }
        return null;
    }
}
```

Java's Catch or Declare Requirement

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

Declare

If a method chooses not to catch an exception, then the method must declare that it can throw it. 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 conciously decide what to do about those exceptions. Thus, you declare the exceptions that can be thrown within the scope of a Java methods in the method signature.

The next page, Dealing with Exceptions, talks about declaring exceptions and shows you how to declare exceptions for Java methods that require it.

Non−runtime Exceptions

In Java, there are 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. Often the cost of checking for runtime exceptions exceeds the benefit of catching or declaring them. Thus the compiler does not require that you catch or declare runtime exceptions, though you can.

Often, this is considered a loophole in Java's exception handling mechanism and programmer's are

tempted to declare all exceptions to be runtime exceptions. In general, this is not recommended. Runtime Exceptions––The Controversy contains a thorough discussion about when and how to use runtime exceptions.

The 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. Thus, this statement includes both

- exceptions that are thrown directly by the method with Java's `throw` statement, and
- exceptions that are thrown indirectly by the method through calls to other methods

---

Handling Errors using Exceptions

Dealing with Exceptions

Your First Encounter with Java Exceptions briefly described what your first encounter with Java exceptions was probably like: it was likely in the form of an error message from the compiler complaining that you must either catch or declare exceptions. Then, Java's Catch or Declare Requirement discussed what exactly was meant by the error message and why the Java language designers decided to make this requirement. Now, we're going to show you both how to catch an exception and how to declare one.

The Example

Both sections below, the one on catching an exception and the one on declaring 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 Declaring the Exceptions Thrown by a Method will show you how to declare those exceptions instead of catching them.

Catching and Handling Exceptions

Now that you've familiarized yourself with the ListOfNumbers class and where the exceptions can be thrown, let's 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.

Declaring 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 declare in the method signature that the method throws the exception. Using the ListOfNumbers class this section shows you how to declare exceptions.

The Example

The two sections that cover catching an exception and declaring an exception, use this class as an example.

```
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }
    public void writeList() {
        PrintStream pStr = null;

        System.err.println("Entering try statement");
        int i;
        pStr = new PrintStream(
                  new BufferedOutputStream(
                     new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));

        pStr.close();
    }
}
```

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

The `writeList()` method calls two other methods that can throw exceptions. First, this line:

```
pStr = new PrintStream(new BufferedOutputStream(new FileOutputStream("OutFile.txt")));
```

invokes the constructor for FileOutputStream which will thrown an IOException if the file cannot be opened for any reason.

Second, the Vector class's `elementAt()` method

```
pStr.println("Value at: " + i + " = " + victor.elementAt(i));
```

will throw 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).

If you try to compile the ListOfNumbers class, the compiler will print an error message about the exception thrown by the FileOutputStream constructor, but will **not** display an error message about the exception thrown by `elementAt()`. This is because the exception thrown by the FileOutputStream constructor, IOException, is a non–runtime exception and the exception thrown by the `elementAt()` method, ArrayIndexOutOfBoundsException, is a runtime exception. Java only requires that you catch or declare non–runtime exceptions. For more information, refer to Java's Catch or Declare Requirement.

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

Following that is a section named Declaring the Exceptions Thrown By a Method will show you how to declare that the ListOfNumbers's `writeList()` method throws the exceptions instead of catching them.

Handling Errors using Exceptions

ListOfNumbersWOHandler.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }
    public void writeList() {
        PrintStream pStr = null;

        System.err.println("Entering try statement");
        int i;
        pStr = new PrintStream(
                  new BufferedOutputStream(
                     new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.println("Value at: " + i + " = " + victor.elementAt(i));

        pStr.close();
    }
}
```

Catching and Handling Exceptions

The next several pages show you how to build an exception handler for the `writeList()` method in the example program described in The Example. The first three pages listed below describe three different components of an exception handler and show how those components can be used in the `writeList()` method. The fourth page walks through the resulting `writeList()` method and analyzes what occurs within the example code during various scenarios.

The try Block

The first step in writing any exception handler is putting the Java statements within which an exception can occur into 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 sequential `catch` blocks directly after the `try` block.

The finally Block

Java's `finally` block provides a mechanism which allows your method to cleanup 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 during three possible scenarios that this code presents.

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 potentially 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.

```
PrintStream pstr;

try {
    int i;

    System.err.println("Entering try statement");
    pStr = new PrintStream(
              new BufferedOutputStream(
                 new FileOutputStream("OutFile.txt")));

    for (i = 0; i < size; i++)
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));
}
```

The `try` statement is said to *govern* the statements enclosed within it and defines the scope of any exceptions handlers (established by subsequent `catch` blocks) 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 either by one or more `catch` blocks, or one `finally` block.

Handling Errors using 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 sequential `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 (SomeThrowableClassName 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, *SomeThrowableClassName*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable api class defined in the java.lang api 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 below) 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 and superclass definitions.

The `catch` statement governs a series of legal Java statements. These statements are executed when and if the exception handler is invoked. The runtime system invokes the exception handler when an exception whose type matches that of the `catch` statement's argument is thrown within the handler's `try` block.

The `writeList()` method from the ListOfNumbers class uses two exception handlers for its `try` statement−−one handler for each of the two different 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());
}
```

An IOException Occurs Let's suppose that the FileOutputStream constructor fails and throws an IOException. The runtime system immediately takes over and tries to locate an appropriate exception handler. The runtime system begins its search at the top of the method call stack. When the exception occurred, the FileOutputStream constructor was at the top of the call stack. However, the FileOutputStream 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. (So, the order that you write your exception handlers matters!) 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 FileOutputStream constructor is also an IOException and so it can legally be assigned to the handler's IOException argument. Thus, this handler is deemed appropriate and the runtime system executes this handler which prints this statement:

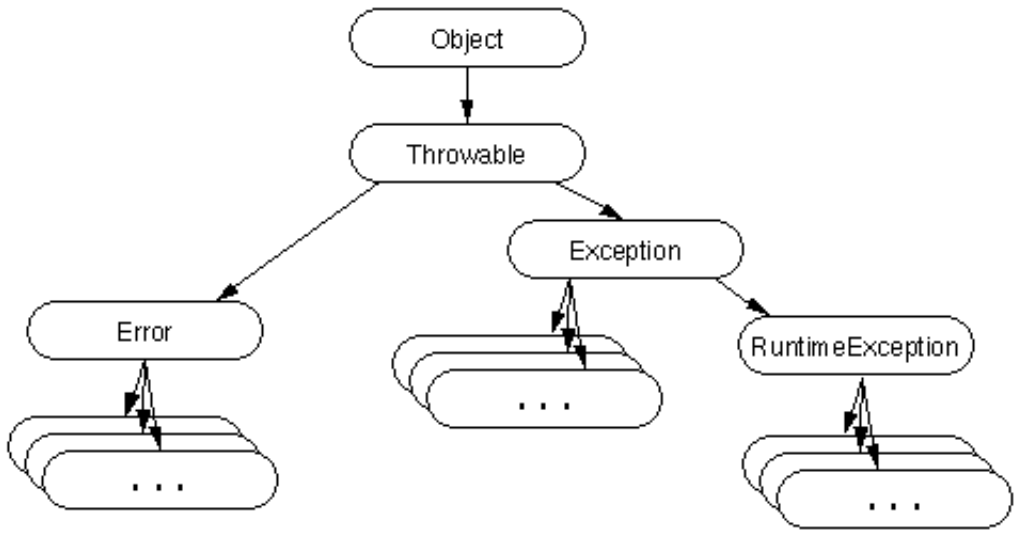
```
Caught IOException: OutFile.txt
```

The runtime system goes through a similar process if an ArrayIndexOutOfBoundsException occurs. For more details, Putting It All Together walks through the `writeList()` method after it's been completed (there's one more step) and investigates what happens during the three possible scenarios that this code presents: the code runs successfully, an IOException occurs, or an ArrayIndexOutOfBoundsException occurs.

Catching Multiple Exception Types with One Handler

The two exception handlers used by the `writeList()` method are very specialized; they handle 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 ancester of IOException and ArrayIndexOutOfBoundsException is the Exception class. Thus an exception handler that would handle both types of exceptions would look 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 also catch numerous other types. Generally speaking, your exception handlers should be more specialized than this one. 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.

Handling Errors using 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 PrintStream. 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 FileOutputStream` 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. That is, regardless of whether control exits the `writeList()` method's `try` block due to scenario 1, 2, or 3 listed above, the code within the `finally` block will be executed.

This is the `finally` block for the `writeList()` method which cleans up and closes the PrintStream.

```
finally {
    if (pStr != null) {
        System.err.println("Closing PrintStream");
        pStr.close();
    } else {
        System.err.println("PrintStream 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 would the PrintStream in the `writeList()` method get closed if you didn't provide an exception handler for the ArrayIndexOutOfBoundsException and an ArrayIndexOutOfBoundsException occurred? (It

would be 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 PrintStream would not get closed if an ArrayIndexOutOfBoundsException occurred and `writeList()` did not provide a handler for it--unless the `writeList()` provided a `finally` statement.

There are other benefits to using the `finally` statement as well. 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 PrintStream at the end of the `try` block and again within exception handler for ArrayIndexOutOfBoundsException as shown here:

```
try {
    . . .
    pStr.close();       // don't do this; it duplicates code
} catch (ArrayIndexOutOfBoundsException e) {
    pStr.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 which makes 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 PrintStream within the new exception handler (which if you're anything like me you are bound to forget).

Handling Errors using Exceptions

Putting It All Together

When all of the components are put togehter, the `writeList()` method will look like this:

```
public void writeList() {
    PrintStream pStr = null;

    try {
        int i;

        System.err.println("Entering try statement");
        pStr = new PrintStream(
              new BufferedOutputStream(
                 new FileOutputStream("OutFile.txt")));

        for (i = 0; i < size; i++)
            pStr.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 (pStr != null) {
            System.err.println("Closing PrintStream");
            pStr.close();
        } else {
            System.err.println("PrintStream not open");
        }
    }
}
```

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

1. The `new FileOutputStream` 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 page investigates in detail what happens in the `writeList()` method during each of those exit possibilities.

Scenario 1: An IOException Occurs

The `new FileOutputStream("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 FileOutputStream

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 immediately takes over and tries to locate an appropriate exception handler. The runtime system begins its search at the top of the method call stack. When the exception occurred, the FileOutputStream constructor was at the top of the call stack. However, the FileOutputStream 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 FileOutputStream 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 PrintStream never got opened, thus `pStr` 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 will see from the ListOfNumbers program when an IOException is thrown is this:

```
Entering try statement
Caught IOException: OutFile.txt
PrintStream 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 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.)

Similar 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 and comes upon the `catch` statement in the `writeList()` method that handles exceptions of the type ArrayIndexOutOfBoundsException and executes it.

After the exception handler has run, the runtime system passes control to the `finally` block. In this particular scenario, the PrintStream 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 will see from the ListOfNumbers program when an ArrayIndexOutOfBoundsException is thrown is this:

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

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, we know that the PrintStream is open when control reaches the `finally` block and must be closed. Again, after the `finally` block has completed executing, the program continues with the first statement after the `finally` block.

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

```
Entering try statement
Closing PrintStream
```

Handling Errors using Exceptions

ListOfNumbers.java

```
/*
 * Copyright (c) 1994 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for NON-COMMERCIAL purposes and without
 * fee is hereby granted provided that this copyright notice
 * appears in all copies. Please refer to the file "copyright.html"
 * for further important copyright and licensing information.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
 * THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
 * TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR
 * ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
import java.io.*;
import java.util.Vector;

class ListOfNumbers {
    private Vector victor;
    final int size = 10;

    public ListOfNumbers () {
        int i;
        victor = new Vector(size);
        for (i = 0; i < size; i++)
            victor.addElement(new Integer(i));
    }
    public void writeList() {
        PrintStream pStr = null;

        try {
            int i;

            System.err.println("Entering try statement");
            pStr = new PrintStream(
                  new BufferedOutputStream(
                     new FileOutputStream("OutFile.txt")));

            for (i = 0; i < size; i++)
                pStr.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 (pStr != null) {
                System.err.println("Closing PrintStream");
                pStr.close();
            } else {
                System.err.println("PrintStream not open");
```

```
                }
            }
        }
}
```

Declaring the Exceptions Thrown by a Method

The previous section showed you how to write an exception handler for the `writeList()` method in the ListOfNumbers class. Sometimes, it's appropriate for your 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 of the users of your package. In this case, it's better to **not** catch the exception and allow someone further up the call stack to handle it.

If the `writeList()` method doesn't catch the exceptions that can occur within it, then the `writeList()` method must declare that it can throw them. Let's modify the `writeList()` method to declare the methods that it can throw. To remind you, here's the `writeList()` method again.

```
public void writeList() {
    System.err.println("Entering try statement");
    int i;
    pStr = new PrintStream(
              new BufferedOutputStream(
                 new FileOutputStream("OutFile.txt")));

    for (i = 0; i < size; i++)
        pStr.println("Value at: " + i + " = " + victor.elementAt(i));
}
```

As you recall, the `new FileOutputStream("OutFile.txt")` statement may throw an IOException (which is not a runtime exception). The `victor.elementAt(i)` statement can throw an ArrayIndexOutOfBoundsException (which is a subclass of RuntimeException).

To declare that `writeList()` throws these two exceptions, you would add a `throws` clause to the method signature for the `writeList()` method. The `throws` clause is composed of the `throws` keyword followed by a comma–separated list of all the exceptions thrown by that method. The `throws` clause goes after the method name and argument list and before the curly bracket that defines the scope of the method. Like this:

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

Remember that ArrayIndexOutOfBoundsException is a runtime exception, so you don't have to declare it in the `throws` clause, although you can.

Handling Errors using 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 all objects you throw with the `throw` statement 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 error and exception classes. All of these classes are descendents of the Throwable class and allow programs to differentiate between the various types of errors or exceptions that can occur during the execution of a Java program.

You, as a Java programmer, can create your own error and exception classes as well to create a set of classes that represents problems that can occur within the classes that you write. Indeed, if you are a package developer you will find that you must create their own set of exceptions 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.

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 api class defined in java.lang api .

```
throw someThrowableObject;
```

If you attempt to throw an object that is not throwable, the compiler will display 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.Throw
            throw new Integer(4);
            ^
```

and refuse to compile your program. The next page, The Throwable Class and Its Subclasses, talks more about the Throwable class.

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. If the stack is empty this method throws an exception:

```
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 only throw 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
```

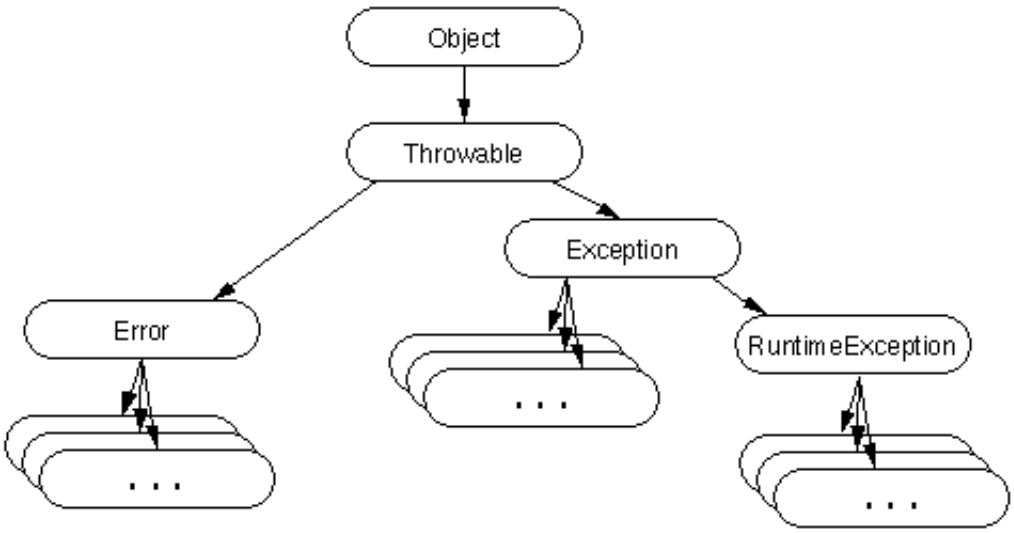
which declares that the method can throw an EmptyStackException. As you know, the Java language requires that methods either catch or declare all non–runtime exceptions that can be thrown within the scope of that method. You do this with the `throws` clause of the method declaration. For more information about this requirement see Java's Catch or Declare Requirement. Also, Declaring the Exceptions Thrown by a Method shows you in more detail how a method can declare the exceptions it can throw.

Handling Errors using Exceptions

The Throwable Class and Its Subclasses

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

This diagram illustrates the class hierarchy of the Throwable class and its 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 problem is not a serious systemic problem. Most programs you write will throw and catch Exceptions.

The Exception class has many descendents defined in the Java packages which indicate various types of exceptions that can occur. For example, IllegalAccessException signals that a particular method

coult not be found, and InstantiationException indicates that a program attempted to instantiate an abstract class or an interface.

One Exception subclass has special meaning in the Java language and deserves mention here: RuntimeException.

RuntimeExceptions

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, and 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 declare 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 undeclared.

The Java packages define several RuntimeException classes. You can catch these exceptions just like other exceptions. However, a method is not required to declare that it throws RuntimeExceptions. In addition, you can create your own RuntimeException subclasses. Runtime Exceptions––The Controversy contains a thorough discussion about when and how to use runtime exceptions.

Handling Errors using 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 program to. You should spend as much time thinking about and designing the exceptions that your classes throw as you do thinking about and designing the classes themselves.

Suppose you were writing a linked list class that you were planning to distribute as freeware software. 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 o)`
: searches the list for the Object specified and returns its position in the list

What Can Go Wrong?

Because many programmers would 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 under the circumstances, do something reasonable about the error, and 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 is to throw an exception.

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

`objectAt()`
: will throw a exception if the integer passed into the method is less than 0 or larger than the number of object currently contained in the list

`firstObject()`
: will throw a exception if the list contains no objects

`indexOf()`
: will throw a exception if the object passed into the method is not contained 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 roll your own?

Choosing the Exception Type to Throw

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

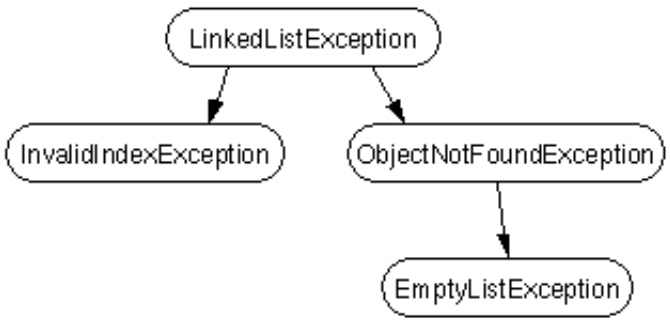
1. use one written by someone else. For example, the Java development enviroment provides a lot of exception classes that you could use.
2. write one of your own

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 handler. Also, if you plan on distributing your linked list in a package, all related code should be packaged together. Thus for the linked list, you should roll your own exception class hierarchy.

The following diagram illustrates one possible exception class hierarchy designed to support the linked list as described above:



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 declared 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 derive from 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".

Handling Errors using Exceptions

Runtime Exceptions––The Controversy

Because the Java language does not require methods to catch or declare runtime exceptions, it's tempting for programmers to write code that throws only runtime exceptions or 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

```
InputFile.java:8: Warning: Exception java.io.FileNotFoundException must be caught, or it must be decl
        fis = new FileInputStream(filename);
              ^
```

errors from the compiler and without bothering to declare or catch any exceptions. While this may seem convenient to the programmer, it sidesteps the intent of Java's catch or declare requirement and can cause problems for the programmers using your classes.

Why did the Java designers decide to force a method to declare all uncaught non–runtime 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 conciously 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 declare runtime exceptions, too?"

RuntimeExceptions represent a programming problem that was detected by the runtime system or an inappropriate use of an API. 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 declaring them. Thus the compiler does not require that you catch or declare runtime exceptions, though you can.

Non–runtime 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 filesystem 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 declaring it? Simply, you get the ability to throw an exception without declaring 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 an Exception.
- 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 declaring them.

Handling Errors using Exceptions