9: Error handling with exceptions

The basic philosophy of Java is that “badly-formed code will not be run.”

As with C++, the ideal time to catch the error is at compile time, before you even try to run the program. However, not all errors can be detected at compile time. The rest of the problems must be handled at run-time through some formality that allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

In C and other earlier languages, there could be several of these formalities, and they were generally established by convention and not as part of the programming language. Typically, you returned a special value or set a flag, and the recipient was supposed to look at the value or the flag and determine that something was amiss. However, as the years passed, it was discovered that programmers who use a library tend to think of themselves as invincible, as in, “Yes, errors might happen to others but not in *my* code.” So, not too surprisingly, they wouldn’t check for the error conditions (and sometimes the error conditions were too silly to check for[38]). If you *were* thorough enough to check for an error every time you called a method, your code could turn into an unreadable nightmare. Because programmers could still coax systems out of these languages they were resistant to admitting the truth: This approach to handling errors was a major limitation to creating large, robust, maintainable programs.

The solution is to take the casual nature out of error handling and to enforce formality. This actually has a long history, since implementations of *exception handling* go back to operating systems in the 1960s and even to BASIC’s **on error goto**. But C++ exception handling was based on Ada, and Java’s is based primarily on C++ (although it looks even more like Object Pascal).

The word “exception” is meant in the sense of “I take exception to that.” At the point where the problem occurs you might not know what to do with it, but you do know that you can’t just continue on merrily; you must stop and somebody, somewhere, must figure out what to do. But you don’t have enough information in the current context to fix the problem. So you hand the problem out to a higher context where someone is qualified to make the proper decision (much like a chain of command).

The other rather significant benefit of exceptions is that they clean up error handling code. Instead of checking for a particular error and dealing with it at multiple places in your program, you no longer need to check at the point of the method call (since the exception will guarantee that someone catches it). And, you need to handle the problem in only one place, the so-called *exception handler*. This saves you code and it separates the code that describes what you want to do from the code that is executed when things

go awry. In general, reading, writing, and debugging code becomes much clearer with exceptions than when using the old way.

Because exception handling is enforced by the Java compiler, there are only so many examples that can be written in this book without learning about exception handling. This chapter introduces you to the code you need to write to properly handle the exceptions, and the way you can generate your own exceptions if one of your methods gets into trouble.

Basic exceptions

An *exceptional condition* is a problem that prevents the continuation of the method or scope that you're in. It's important to distinguish an exceptional condition from a normal problem, in which you have enough information in the current context to somehow cope with the difficulty. With an exceptional condition, you cannot continue processing because you don't have the information necessary to deal with the problem *in the current context*. All you can do is jump out of the current context and relegate that problem to a higher context. This is what happens when you throw an exception.

A simple example is a divide. If you're about to divide by zero, it's worth checking to make sure you don't go ahead and perform the divide. But what does it mean that the denominator is zero? Maybe you know, in the context of the problem you're trying to solve in that particular method, how to deal with a zero denominator. But if it's an unexpected value, you can't deal with it and so must throw an exception rather than continuing along that path.

When you throw an exception, several things happen. First, the exception object is created in the same way that any Java object is created: on the heap, with **new**. Then the current path of execution (the one you couldn't continue, remember) is stopped and the handle for the exception object is ejected from the current context. At this point the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the *exception handler*, whose job is to recover from the problem so the program can either try another tack or simply continue.

As a simple example of throwing an exception, consider an object handle called **t**. It's possible that you might be passed a handle that hasn't been initialized, so you might want to check before trying to call a method using that object handle. You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new NullPointerException();
```

This throws the exception, which allows you – in the current context – to abdicate responsibility for thinking about the issue further. It's just magically handled somewhere else. Precisely *where* will be shown shortly.

Exception arguments

Like any object in Java, you always create exceptions on the heap using **new** and a constructor gets called. There are two constructors in all the standard exceptions; the first is the default constructor, and the second takes a string argument so you can place pertinent information in the exception:

```
if(t == null)
  throw new NullPointerException("t = null");
```

This string can later be extracted using various methods, as will be shown later.

The keyword **throw** causes a number of relatively magical things to happen. First it executes the **new**-expression to create an object that isn't there under normal program execution, and of course, the constructor is called for that object. Then the object is, in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as an alternate return mechanism, although you get into trouble if you take that analogy too far. You can also exit from ordinary scopes by throwing an exception. But a value is returned, and the method or scope exits.

Any similarity to an ordinary return from a method ends here, because *where* you return is someplace completely different from where you return for a normal method call. (You end up in an appropriate exception handler that might be miles away – many levels lower on the call stack – from where the exception was thrown.)

In addition, you can throw any type of **Throwable** object that you want. Typically, you'll throw a different class of exception for each different type of error. The idea is to store the information in the exception object *and* in the type of exception object chosen, so someone in the bigger context can figure out what to do with your exception. (Often, the only information is the type of exception object, and nothing meaningful is stored within the exception object.)

Catching an exception

If a method throws an exception, it must assume that exception is caught and dealt with. One of the advantages of Java exception handling is that it allows you to concentrate on the problem you're trying to solve in one place, and then deal with the errors from that code in another place.

To see how an exception is caught, you must first understand the concept of a *guarded region*, which is a section of code that might produce exceptions, and is followed by the code to handle those exceptions.

The try block

If you're inside a method and you throw an exception (or another method you call within this method throws an exception), that method will exit in the process of throwing. If you don't want a **throw** to leave a method, you can set up a special block within that method to capture the exception. This is called the *try block* because you "try" your various method calls there. The try block is an ordinary scope, preceded by the keyword **try**:

```
try {
  // Code that might generate exceptions
}
```

If you were checking for errors carefully in a programming language that didn't support exception handling, you'd have to surround every method call with setup and error testing code, even if you call the same method several times. With exception handling, you put everything in a try block and capture all the exceptions in one place. This means your code is a lot easier to write and easier to read because the goal of the code is not confused with the error checking.

Exception handlers

Of course, the thrown exception must end up someplace. This "place" is the *exception handler*, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}

// etc...
```

Each catch clause (exception handler) is like a little method that takes one and only one argument of a particular type. The identifier (**id1**, **id2**, and so on) can be used inside the handler, just like a method argument. Sometimes you never use the identifier because the type of the exception gives you enough information to deal with the exception, but the identifier must still be there.

The handlers must appear directly after the try block. If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. (The search for handlers stops once the catch clause is finished.) Only the matching catch clause executes; it's not like a **switch** statement in which you need a **break** after each **case** to prevent the remaining ones from executing.

Note that, within the try block, a number of different method calls might generate the same exception, but you need only one handler.

Termination vs. resumption

There are two basic models in exception-handling theory. In *termination* (which is what Java and C++ support), you assume the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided that there was no way to salvage the situation, and they don't *want* to come back.

The alternative is called *resumption*. It means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled. In this case, your exception is more like a method call – which is how you should set up situations in Java in which you want resumption-like behavior. (That is, don't throw an exception; call a method that fixes the problem.) Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. So although resumption sounds attractive at first, it seems it isn't quite so useful in practice. The dominant reason is probably the *coupling* that results: your handler must often be aware of where the exception is thrown from and contain non-generic code specific to the throwing location. This makes the code difficult to write and maintain, especially for large systems where the exception can be generated from many points.

The exception specification

In Java, you're required to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method. This is civilized because the caller can know exactly what code to write to catch all potential exceptions. Of course, if source code is available, the client programmer could hunt through and look for **throw** statements, but often a library doesn't come with sources. To prevent this from being a problem, Java provides syntax (and *forces* you to use that syntax) to allow you to politely tell the client programmer what exceptions this method throws, so the client programmer can handle them. This is the *exception specification* and it's part of the method declaration, appearing after the argument list.

The exception specification uses an additional keyword, **throws**, followed by a list of all the potential exception types. So your method definition might look like this:

```
void f() throws tooBig, tooSmall, divZero { //...
```

If you say

```
void f() { // ...
```

it means that no exceptions are thrown from the method. (*Except* for the exceptions of type **RuntimeException**, which can reasonably be thrown anywhere – this will be described later.)

You can't lie about an exception specification – if your method causes exceptions and doesn't handle them, the compiler will detect this and tell you that you must either handle the exception or indicate with an exception specification that it may be thrown from your method. By enforcing exception specifications from top to bottom, Java guarantees that exception correctness can be ensured *at compile time*.[39]

There is one place you can lie: you can claim to throw an exception that you don't. The compiler takes your word for it and forces the users of your method to treat it as if it really does throw that exception. This has the beneficial effect of being a placeholder for that exception, so you can actually start throwing the exception later without requiring changes to existing code.

Catching any exception

It is possible to create a handler that catches any type of exception. You do this by catching the base-class exception type **Exception** (there are other types of base exceptions, but **Exception** is the base that's pertinent to virtually all programming activities):

```
catch(Exception e) {
  System.out.println("caught an exception");
}
```

This will catch any exception, so if you use it you'll want to put it at the *end* of your list of handlers to avoid pre-empting any exception handlers that might otherwise follow it.

Since the **Exception** class is the base of all the exception classes that are important to the programmer, you don't get much specific information about the exception, but you can call the methods that come from *its* base type **Throwable**:

String getMessage( )
Gets the detail message.

String toString( )
Returns a short description of the Throwable, including the detail message if there is one.

void printStackTrace( )
void printStackTrace(PrintStream)
Prints the Throwable and the Throwable's call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown.

The first version prints to standard error, the second prints to a stream of your choice. If you're working under Windows, you can't redirect standard error so you might want to use the second version and send the results to **System.out**; that way the output can be redirected any way you want.

In addition, you get some other methods from **Throwable**'s base type **Object** (everybody's base type). The one that might come in handy for exceptions is **getClass( )**, which returns an object representing the class of this object. You can in turn query this **Class** object for its name with **getName( )** or **toString( )**. You can also do more sophisticated things with **Class** objects that aren't necessary in exception handling. **Class** objects will be studied later in the book.

Here's an example that shows the use of the **Exception** methods:

```
//: ExceptionMethods.java
// Demonstrating the Exception Methods
package c09;

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("Here's my Exception");
    } catch(Exception e) {
      System.out.println("Caught Exception");
      System.out.println(
        "e.getMessage(): " + e.getMessage());
      System.out.println(
        "e.toString(): " + e.toString());
      System.out.println("e.printStackTrace():");
      e.printStackTrace();
    }
  }
} ///:~
```

The output for this program is:

```
Caught Exception
e.getMessage(): Here's my Exception
e.toString(): java.lang.Exception: Here's my Exception
e.printStackTrace():
java.lang.Exception: Here's my Exception
```

```
        at ExceptionMethods.main
```

You can see that the methods provide successively more information – each is effectively a superset of the previous one.

Rethrowing an exception

Sometimes you'll want to rethrow the exception that you just caught, particularly when you use **Exception** to catch any exception. Since you already have the handle to the current exception, you can simply re-throw that handle:

```
catch(Exception e) {
  System.out.println("An exception was thrown");
  throw e;
}
```

Rethrowing an exception causes the exception to go to the exception handlers in the next-higher context. Any further **catch** clauses for the same **try** block are still ignored. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

If you simply re-throw the current exception, the information that you print about that exception in **printStackTrace( )** will pertain to the exception's origin, not the place where you re-throw it. If you want to install new stack trace information, you can do so by calling **fillInStackTrace( )**, which returns an exception object that it creates by stuffing the current stack information into the old exception object. Here's what it looks like:

```
//: Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
  public static void f() throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void g() throws Throwable {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Inside g(), e.printStackTrace()");
      e.printStackTrace();
      throw e; // 17
      // throw e.fillInStackTrace(); // 18
    }
  }
  public static void
  main(String[] args) throws Throwable {
```

```
    try {
      g();
    } catch(Exception e) {
      System.out.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
    }
  }
} ///:~
```

The important line numbers are marked inside of comments. With line 17 un-commented (as shown), the output is:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
```

So the exception stack trace always remembers its true point of origin, no matter how many times it gets rethrown.

With line 17 commented and line 18 un-commented, **fillInStackTrace( )** is used instead, and the result is:

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:24)
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.g(Rethrowing.java:18)
        at Rethrowing.main(Rethrowing.java:24)
```

Because of **fillInStackTrace( )**, line 18 becomes the new point of origin of the exception.

The class **Throwable** must appear in the exception specification for **g( )** and **main( )** because **fillInStackTrace( )** produces a handle to a **Throwable** object. Since **Throwable** is a base class of **Exception**, it's possible to get an object that's a **Throwable** but *not* an **Exception**, so the handler for **Exception** in **main( )** might miss it. To make sure everything is in order, the compiler forces an exception specification for **Throwable**. For example, the exception in the following program is *not* caught in **main( )**:

```
//: ThrowOut.java
public class ThrowOut {
  public static void
  main(String[] args) throws Throwable {
    try {
      throw new Throwable();
    } catch(Exception e) {
      System.out.println("Caught in main()");
    }
  }
} ///:~
```

It's also possible to rethrow a different exception from the one you caught. If you do this, you get a similar effect as when you use **fillInStackTrace( )**: the information about the original site of the exception is lost, and what you're left with is the information pertaining to the new **throw**:

```
//: RethrowNew.java
// Rethrow a different object from the one that
// was caught

public class RethrowNew {
  public static void f() throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
      throw new NullPointerException("from main");
    }
  }
} ///:~
```

The output is:

```
originating the exception in f()
Caught in main, e.printStackTrace()
java.lang.Exception: thrown from f()
        at RethrowNew.f(RethrowNew.java:8)
        at RethrowNew.main(RethrowNew.java:13)
java.lang.NullPointerException: from main
        at RethrowNew.main(RethrowNew.java:18)
```

The final exception knows only that it came from **main( )**, and not from **f( )**. Note that **Throwable** isn't necessary in any of the exception specifications.

You never have to worry about cleaning up the previous exception, or any exceptions for that matter. They're all heap-based objects created with **new**, so the garbage collector automatically cleans them all up.

Standard Java exceptions

Java contains a class called **Throwable** that describes anything that can be thrown as an exception. There are two general types of **Throwable** objects ("types of" = "inherited from"). **Error** represents compile-time and system errors that you don't worry about catching (except in special cases). **Exception** is the basic type that can be thrown from any of the standard Java library class methods and from your methods and run-time accidents.

The best way to get an overview of the exceptions is to browse online Java documentation from *http://java.sun.com*. (Of course, it's easier to download it first.) It's worth doing this once just to get a feel for the various exceptions, but you'll soon see that there isn't anything special between one exception and the next except for the name. Also, the number of exceptions in Java keeps expanding; basically it's pointless to print them in a book. Any new library you get from a third-party vendor will probably have its own exceptions as well. The important thing to understand is the concept and what you should do with the exceptions.

`java.lang.Exception`

This is the basic exception class your program can catch. Other exceptions are derived from this. The basic idea is that the name of the exception represents the problem that occurred and the exception name is intended to be relatively self-explanatory. The exceptions are not all defined in **java.lang**; some are created to support other libraries such as **util**, **net,** and **io**, which you can see from their full class names or what they are inherited from. For example, all IO exceptions are inherited from **java.io.IOException**.

The special case of RuntimeException

The first example in this chapter was

```
if(t == null)
  throw new NullPointerException();
```

It can be a bit horrifying to think that you must check for **null** on every handle that is passed into a method (since you can't know if the caller has passed you a valid handle). Fortunately, you don't – this is part of the standard run-time checking that Java performs for you, and if any call is made to a null handle, Java will automatically throw a **NullPointerException**. So the above bit of code is always superfluous.

There's a whole group of exception types that are in this category. They're always thrown automatically by Java and you don't need to include them in your exception specifications. Conveniently enough, they're all grouped together by putting them under a single base class called **RuntimeException**, which is a perfect example of inheritance: it establishes a family of types that have some characteristics and behaviors in common. Also, you never need to write an exception specification saying that a method might throw a **RuntimeException**, since that's just assumed. Because they indicate bugs, you virtually never catch a **RuntimeException** – it's dealt with automatically. If you were forced to check for **RuntimeException**s your code could get messy. Even though you don't typically catch **RuntimeExceptions**, in your own packages you might choose to throw some of the **RuntimeException**s.

What happens when you don't catch such exceptions? Since the compiler doesn't enforce exception specifications for these, it's quite plausible that a **RuntimeException** could percolate all the way out to your **main( )** method without being caught. To see what happens in this case, try the following example:

```
//: NeverCaught.java
// Ignoring RuntimeExceptions

public class NeverCaught {
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
} ///:~
```

You can already see that a **RuntimeException** (or anything inherited from it) is a special case, since the compiler doesn't require an exception specification for these types.

The output is:

```
java.lang.RuntimeException: From f()
        at NeverCaught.f(NeverCaught.java:9)
        at NeverCaught.g(NeverCaught.java:12)
        at NeverCaught.main(NeverCaught.java:15)
```

So the answer is: If a RuntimeException gets all the way out to **main( )** without being caught, **printStackTrace( )** is called for that exception as the program exits.

Keep in mind that it's possible to ignore only **RuntimeException**s in your coding, since all other handling is carefully enforced by the compiler. The reasoning is that a **RuntimeException** represents a programming error:

1. An error you cannot catch (receiving a null handle handed to your method by a client programmer, for example)
2. An error that you, as a programmer, should have checked for in your code (such as **ArrayIndexOutOfBoundsException** where you should have paid attention to the size of the array).

You can see what a tremendous benefit it is to have exceptions in this case, since they help in the debugging process.

It's interesting to notice that you cannot classify Java exception handling as a single-purpose tool. Yes, it is designed to handle those pesky run-time errors that will occur because of forces outside your code's control, but it's also essential for certain types of programming bugs that the compiler cannot detect.

Creating your own exceptions

You're not stuck using the Java exceptions. This is important because you'll often need to create your own exceptions to denote a special error that your library is capable of creating, but which was not foreseen when the Java hierarchy was created.

To create your own exception class, you're forced to inherit from an existing type of exception, preferably one that is close in meaning to your new exception. Inheriting an exception is quite simple:

```
//: Inheriting.java
// Inheriting your own exceptions

class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}

public class Inheriting {
  public static void f() throws MyException {
    System.out.println(
      "Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println(
      "Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace();
    }
```

```
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The inheritance occurs in the creation of the new class:

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String msg) {
    super(msg);
  }
}
```

The key phrase here is **extends Exception**, which says "it's everything an **Exception** is and more." The added code is small – the addition of two constructors that define the way **MyException** is created. Remember that the compiler automatically calls the base-class default constructor if you don't explicitly call a base-class constructor, as in the **MyException( )** default constructor. In the second constructor, the base-class constructor with a **String** argument is explicitly invoked by using the **super** keyword.

The output of the program is:

```
Throwing MyException from f()
MyException
        at Inheriting.f(Inheriting.java:16)
        at Inheriting.main(Inheriting.java:24)
Throwing MyException from g()
MyException: Originated in g()
        at Inheriting.g(Inheriting.java:20)
        at Inheriting.main(Inheriting.java:29)
```

You can see the absence of the detail message in the **MyException** thrown from **f( )**.

The process of creating your own exceptions can be taken further. You can add extra constructors and members:

```
//: Inheriting2.java
// Inheriting your own exceptions

class MyException2 extends Exception {
  public MyException2() {}
  public MyException2(String msg) {
    super(msg);
  }
  public MyException2(String msg, int x) {
    super(msg);
    i = x;
  }
  public int val() { return i; }
```

```
  private int i;
}

public class Inheriting2 {
  public static void f() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from g()");
    throw new MyException2("Originated in g()");
  }
  public static void h() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from h()");
    throw new MyException2(
      "Originated in h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch(MyException2 e) {
      e.printStackTrace();
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace();
      System.out.println("e.val() = " + e.val());
    }
  }
} ///:~
```

A data member **i** has been added, along with a method that reads that value and an additional constructor that sets it. The output is:

```
Throwing MyException2 from f()
MyException2
        at Inheriting2.f(Inheriting2.java:22)
        at Inheriting2.main(Inheriting2.java:34)
Throwing MyException2 from g()
MyException2: Originated in g()
        at Inheriting2.g(Inheriting2.java:26)
        at Inheriting2.main(Inheriting2.java:39)
Throwing MyException2 from h()
MyException2: Originated in h()
        at Inheriting2.h(Inheriting2.java:30)
        at Inheriting2.main(Inheriting2.java:44)
e.val() = 47
```

Since an exception is just another kind of object, you can continue this process of embellishing the power of your exception classes. Keep in mind, however, that all this dressing up might be lost on the client programmers using your packages, since they might simply look for the exception to be thrown and nothing more. (That's the way most of the Java library exceptions are used.) If this is the case, it's possible to create a new exception type with almost no code at all:

```
//: SimpleException.java
class SimpleException extends Exception {
} ///:~
```

This relies on the compiler to create the default constructor (which automatically calls the base-class default constructor). Of course, in this case you don't get a **SimpleException(String)** constructor, but in practice that isn't used much.

Exception restrictions

When you override a method, you can throw only the exceptions that have been specified in the base-class version of the method. This is a useful restriction, since it means that code that works with the base class will automatically work with any object derived from the base class (a fundamental OOP concept, of course), including exceptions.

This example demonstrates the kinds of restrictions imposed (at compile time) for exceptions:

```
//: StormyInning.java
// Overridden methods may throw only the
// exceptions specified in their base-class
// versions, or exceptions derived from the
// base-class exceptions.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  void event () throws BaseballException {
   // Doesn't actually have to throw anything
  }
  abstract void atBat() throws Strike, Foul;
  void walk() {} // Throws nothing
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
```

```
  void rainHard() throws RainedOut;
}

public class StormyInning extends Inning
    implements Storm {
  // OK to add new exceptions for constructors,
  // but you must deal with the base constructor
  // exceptions:
  StormyInning() throws RainedOut,
    BaseballException {}
  StormyInning(String s) throws Foul,
    BaseballException {}
  // Regular methods must conform to base class:
//! void walk() throws PopFoul {} //Compile error
  // Interface CANNOT add exceptions to existing
  // methods from the base class:
//! public void event() throws RainedOut {}
  // If the method doesn't already exist in the
  // base class, the exception is OK:
  public void rainHard() throws RainedOut {}
  // You can choose to not throw any exceptions,
  // even if base version does:
  public void event() {}
  // Overridden methods can throw
  // inherited exceptions:
  void atBat() throws PopFoul {}
  public static void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
    } catch(RainedOut e) {
    } catch(BaseballException e) {}
    // Strike not thrown in derived version.
    try {
      // What happens if you upcast?
      Inning i = new StormyInning();
      i.atBat();
      // You must catch the exceptions from the
      // base-class version of the method:
    } catch(Strike e) {
    } catch(Foul e) {
    } catch(RainedOut e) {
    } catch(BaseballException e) {}
  }
} ///:~
```

In **Inning**, you can see that both the constructor and the **event( )** method say they will throw an exception, but they never do. This is legal because it allows you to force the user to catch any exceptions that you might add in overridden versions of **event( )**. The same idea holds for **abstract** methods, as seen in **atBat( )**.

The **interface Storm** is interesting because it contains one method (**event( )**)that is defined in **Inning**, and one method that isn't. Both methods throw a new type of exception, **RainedOut**. When **StormyInning extends Inning** and **implements**

Storm, you'll see that the **event( )** method in **Storm** *cannot* change the exception interface of **event( )** in **Inning**. Again, this makes sense because otherwise you'd never know if you were catching the correct thing when working with the base class. Of course, if a method described in an **interface** is not in the base class, such as **rainHard( )**, then there's no problem if it throws exceptions.

The restriction on exceptions does not apply to constructors. You can see in **StormyInning** that a constructor can throw anything it wants, regardless of what the base-class constructor throws. However, since a base-class constructor must always be called one way or another (here, the default constructor is called automatically), the derived-class constructor must declare any base-class constructor exceptions in its exception specification.

The reason **StormyInning.walk( )** will not compile is that it throws an exception, while **Inning.walk( )** does not. If this was allowed, then you could write code that called **Inning.walk( )** and that didn't have to handle any exceptions, but then when you substituted an object of a class derived from **Inning**, exceptions would be thrown so your code would break. By forcing the derived-class methods to conform to the exception specifications of the base-class methods, substitutability of objects is maintained.

The overridden **event( )** method shows that a derived-class version of a method may choose to not throw any exceptions, even if the base-class version does. Again, this is fine since it doesn't break any code that is written assuming the base-class version throws exceptions. Similar logic applies to **atBat( )**, which throws **PopFoul**, an exception that is derived from **Foul** thrown by the base-class version of **atBat( )**. This way, if someone writes code that works with **Inning** and calls **atBat( )**, they must catch the **Foul** exception. Since **PopFoul** is derived from **Foul**, the exception handler will also catch **PopFoul**.

The last point of interest is in **main( )**. Here you can see that if you're dealing with exactly a **StormyInning** object, the compiler forces you to catch only the exceptions that are specific to that class, but if you upcast to the base type then the compiler (correctly) forces you to catch the exceptions for the base type. All these constraints produce much more robust exception-handling code.[40]

It's useful to realize that although exception specifications are enforced by the compiler during inheritance, the exception specifications are not part of the type of a method, which is comprised of only the method name and argument types. Therefore, you cannot overload methods based on exception specifications. In addition, because an exception specification exists in a base-class version of a method doesn't mean that it must exist in the derived-class version of the method, and this is quite different from inheriting the methods (that is, a method in the base class must also exist in the derived class). Put another way, the "exception specification interface" for a particular method may narrow during inheritance and overriding, but it may not widen – this is precisely the opposite of the rule for the class interface during inheritance.

Performing cleanup with finally

There's often some piece of code that you want to execute whether or not an exception occurs in a **try** block. This usually pertains to some operation other than memory recovery (since that's taken care of by the garbage collector). To achieve this effect, you use a **finally** clause[41] at the end of all the exception handlers. The full picture of an exception-handling section is thus:

```
try {
  // The guarded region:
  // Dangerous stuff that might throw A, B, or C
} catch (A a1) {
  // Handle A
} catch (B b1) {
  // Handle B
} catch (C c1) {
  // Handle C
} finally {
  // Stuff that happens every time
}
```

To demonstrate that the **finally** clause always runs, try this program:

```
//: FinallyWorks.java
// The finally clause is always executed

public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // post-increment is zero first time:
        if(count++ == 0)
          throw new Exception();
        System.out.println("No exception");
      } catch(Exception e) {
        System.out.println("Exception thrown");
      } finally {
        System.out.println("in finally clause");
        if(count == 2) break; // out of "while"
      }
    }
  }
} ///:~
```

This program also gives a hint for how you can deal with the fact that exceptions in Java (like exceptions in C++) do not allow you to resume back to where the exception was thrown, as discussed earlier. If you place your **try** block in a loop, you can establish a condition that must be met before you continue the program. You can also add a **static** counter or some other device to allow the loop to try several different approaches before giving up. This way you can build a greater level of robustness into your programs.

The output is:

```
Exception thrown
in finally clause
No exception
in finally clause
```

Whether an exception is thrown or not, the **finally** clause is always executed.

What's finally for?

In a language without garbage collection *and* without automatic destructor calls,[42] **finally** is important because it allows the programmer to guarantee the release of memory regardless of what happens in the **try** block. But Java has garbage collection, so releasing memory is virtually never a problem. Also, it has no destructors to call. So when do you need to use **finally** in Java?

finally is necessary when you need to set something *other* than memory back to its original state. This is usually something like an open file or network connection, something you've drawn on the screen or even a switch in the outside world, as modeled in the following example:

```
//: OnOffSwitch.java
// Why use finally?

class Switch {
  boolean state = false;
  boolean read() { return state; }
  void on() { state = true; }
  void off() { state = false; }
}

public class OnOffSwitch {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      sw.off();
    } catch(NullPointerException e) {
      System.out.println("NullPointerException");
      sw.off();
    } catch(IllegalArgumentException e) {
      System.out.println("IOException");
```

```
      sw.off();
    }
  }
} ///:~
```

The goal here is to make sure that the switch is off when **main( )** is completed, so **sw.off( )** is placed at the end of the try block and at the end of each exception handler. But it's possible that an exception could be thrown that isn't caught here, so **sw.off( )** would be missed. However, with **finally** you can place the closure code from a try block in just one place:

```
//: WithFinally.java
// Finally Guarantees cleanup

class Switch2 {
  boolean state = false;
  boolean read() { return state; }
  void on() { state = true; }
  void off() { state = false; }
}

public class WithFinally {
  static Switch2 sw = new Switch2();
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
    } catch(NullPointerException e) {
      System.out.println("NullPointerException");
    } catch(IllegalArgumentException e) {
      System.out.println("IOException");
    } finally {
      sw.off();
    }
  }
} ///:~
```

Here the **sw.off( )** has been moved to just one place, where it's guaranteed to run no matter what happens.

Even in cases in which the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception-handling mechanism continues its search for a handler at the next higher level:

```
//: AlwaysFinally.java
// Finally is always executed

class Ex extends Exception {}

public class AlwaysFinally {
  public static void main(String[] args) {
    System.out.println(
      "Entering first try block");
```

```
    try {
      System.out.println(
        "Entering second try block");
      try {
        throw new Ex();
      } finally {
        System.out.println(
          "finally in 2nd try block");
      }
    } catch(Ex e) {
      System.out.println(
        "Caught Ex in first try block");
    } finally {
      System.out.println(
        "finally in 1st try block");
    }
  }
} ///:~
```

The output for this program shows you what happens:

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught Ex in first try block
finally in 1st try block
```

The **finally** statement will also be executed in situations in which **break** and **continue** statements are involved. Note that, along with the labeled **break** and labeled **continue**, **finally** eliminates the need for a **goto** statement in Java.

Pitfall: the lost exception

In general, Java's exception implementation is quite outstanding, but unfortunately there's a flaw. Although exceptions are an indication of a crisis in your program and should never be ignored, it's possible for an exception to simply be lost. This happens with a particular configuration using a **finally** clause:

```
//: LostMessage.java
// How an exception can be lost

class VeryImportantException extends Exception {
  public String toString() {
    return "A very important exception!";
  }
}

class HoHumException extends Exception {
  public String toString() {
    return "A trivial exception";
  }
}
```

```
public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args)
      throws Exception {
    LostMessage lm = new LostMessage();
    try {
      lm.f();
    } finally {
      lm.dispose();
    }
  }
} ///:~
```

The output is:

```
A trivial exception
        at LostMessage.dispose(LostMessage.java:21)
        at LostMessage.main(LostMessage.java:29)
```

You can see that there's no evidence of the **VeryImportantException**, which is simply replaced by the **HoHumException** in the **finally** clause. This is a rather serious pitfall, since it means that an exception can be completely lost, and in a far more subtle and difficult-to-detect fashion than the example above. In contrast, C++ treats the situation in which a second exception is thrown before the first one is handled as a dire programming error. Perhaps a future version of Java will repair the problem. (The above results were produced with Java 1.1.)

Constructors

When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will this be properly cleaned up?" Most of the time you're fairly safe, but in constructors there's a problem. The constructor puts the object into a safe starting state, but it might perform some operation – such as opening a file – that doesn't get cleaned up until the user is finished with the object and calls a special cleanup method. If you throw an exception from inside a constructor, these cleanup behaviors might not occur properly. This means that you must be especially diligent while you write your constructor.

Since you've just learned about **finally**, you might think that it is the correct solution. But it's not quite that simple, because **finally** performs the cleanup code *every time*, even in the situations in which you don't want the cleanup code executed until the cleanup method runs. Thus, if you do perform cleanup in **finally**, you must set some kind of flag when the constructor finishes normally and don't do anything in the finally block if the flag is set. Because this isn't particularly elegant (you are coupling your code

from one place to another), it's best if you try to avoid performing this kind of cleanup in **finally** unless you are forced to.

In the following example, a class called **InputFile** is created that opens a file and allows you to read it one line (converted into a **String**) at a time. It uses the classes **FileReader** and **BufferedReader** from the Java standard IO library that will be discussed in Chapter 10, but which are simple enough that you probably won't have any trouble understanding their basic use:

```
//: Cleanup.java
// Paying attention to exceptions
// in constructors
import java.io.*;

class InputFile {
  private BufferedReader in;
  InputFile(String fname) throws Exception {
    try {
      in =
        new BufferedReader(
          new FileReader(fname));
      // Other code that might throw exceptions
    } catch(FileNotFoundException e) {
      System.out.println(
        "Could not open " + fname);
      // Wasn't open, so don't close it
      throw e;
    } catch(Exception e) {
      // All other exceptions must close it
      try {
        in.close();
      } catch(IOException e2) {
        System.out.println(
          "in.close() unsuccessful");
      }
      throw e;
    } finally {
      // Don't close it here!!!
    }
  }
  String getLine() {
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      System.out.println(
        "readLine() unsuccessful");
      s = "failed";
    }
    return s;
  }
  void cleanup() {
    try {
      in.close();
    } catch(IOException e2) {
```

```
      System.out.println(
        "in.close() unsuccessful");
    }
  }
}

public class Cleanup {
  public static void main(String[] args) {
    try {
      InputFile in =
        new InputFile("Cleanup.java");
      String s;
      int i = 1;
      while((s = in.getLine()) != null)
        System.out.println(""+ i++ + ": " + s);
      in.cleanup();
    } catch(Exception e) {
      System.out.println(
        "Caught in main, e.printStackTrace()");
      e.printStackTrace();
    }
  }
} ///:~
```

This example uses Java 1.1 IO classes.

The constructor for **InputFile** takes a **String** argument, which is the name of the file you want to open. Inside a **try** block, it creates a **FileReader** using the file name. A **FileReader** isn't particularly useful until you turn around and use it to create a **BufferedReader** that you can actually talk to – notice that one of the benefits of **InputFile** is that it combines these two actions.

If the **FileReader** constructor is unsuccessful, it throws a **FileNotFoundException**, which must be caught separately because that's the one case in which you don't want to close the file since it wasn't successfully opened. Any *other* catch clauses must close the file because it *was* opened by the time those catch clauses are entered. (Of course, this is trickier if more than one method can throw a **FileNotFoundException**. In that case, you might want to break things into several **try** blocks.) The **close( )** method throws an exception that is tried and caught even though it's within the block of another **catch** clause – it's just another pair of curly braces to the Java compiler. After performing local operations, the exception is re-thrown, which is appropriate because this constructor failed, and you wouldn't want the calling method to assume that the object had been properly created and was valid.

In this example, which doesn't use the aforementioned flagging technique, the **finally** clause is definitely *not* the place to **close( )** the file, since that would close it every time the constructor completed. Since we want the file to be open for the useful lifetime of the **InputFile** object this would not be appropriate.

The **getLine( )** method returns a **String** containing the next line in the file. It calls **readLine( ),** which can throw an exception, but that exception is caught so **getLine( )**

doesn't throw any exceptions. One of the design issues with exceptions is whether to handle an exception completely at this level, to handle it partially and pass the same exception (or a different one) on, or whether to simply pass it on. Passing it on, when appropriate, can certainly simplify coding. The **getLine( )** method becomes:

```
String getLine() throws IOException {
  return in.readLine();
}
```

But of course, the caller is now responsible for handling any **IOException** that might arise.

The **cleanup( )** method must be called by the user when they are finished using the **InputFile** object to release the system resources (such as file handles) that are used by the **BufferedReader** and/or **FileReader** objects.[43] You don't want to do this until you're finished with the **InputFile** object, at the point you're going to let it go. You might think of putting such functionality into a **finalize( )** method, but as mentioned in Chapter 4 you can't always be sure that **finalize( )** will be called (even if you *can* be sure that it will be called, you don't know *when*). This is one of the downsides to Java – all cleanup other than memory cleanup doesn't happen automatically, so you must inform the client programmer that they are responsible, and possibly guarantee that cleanup occurs using **finalize( )**.

In **Cleanup.java** an **InputFile** is created to open the same source file that creates the program, and this file is read in a line at a time, and line numbers are added. All exceptions are caught generically in **main( )**, although you could choose greater granularity.

One of the benefits of this example is to show you why exceptions are introduced at this point in the book. Exceptions are so integral to programming in Java, especially because the compiler enforces them, that you can accomplish only so much without knowing how to work with them.

Exception matching

When an exception is thrown, the exception-handling system looks through the "nearest" handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. A derived-class object will match a handler for the base class, as shown in this example:

```
//: Human.java
// Catching Exception Hierarchies

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}
```

```
public class Human {
  public static void main(String[] args) {
    try {
      throw new Sneeze();
    } catch(Sneeze s) {
      System.out.println("Caught Sneeze");
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    }
  }
} ///:~
```

The **Sneeze** exception will be caught by the first **catch** clause that it matches, which is the first one, of course. However, if you remove the first catch clause:

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    }
```

The remaining catch clause will still work because it's catching the base class of **Sneeze**. Put another way, **catch(Annoyance e)** will catch a **Annoyance** *or any class derived from it*. This is useful because if you decide to add more exceptions to a method, if they're all inherited from the same base class then the client programmer's code will not need changing, assuming they catch the base class, at the very least.

If you try to "mask" the derived-class exceptions by putting the base-class catch clause first, like this:

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    } catch(Sneeze s) {
      System.out.println("Caught Sneeze");
    }
```

the compiler will give you an error message, since it sees that the **Sneeze** catch-clause can never be reached.

Exception guidelines

Use exceptions to:

1. Fix the problem and call the method (which caused the exception) again.
2. Patch things up and continue without retrying the method.
3. Calculate some alternative result instead of what the method was supposed to produce.
4. Do whatever you can in the current context and rethrow the *same* exception to a higher context.

5. Do whatever you can in the current context and throw a *different* exception to a higher context.
6. Terminate the program.
7. Simplify. If your exception scheme makes things more complicated, then it is painful and annoying to use.
8. Make your library and program safer. This is a short-term investment (for debugging) and a long-term investment (for application robustness).

Summary

Improved error recovery is one of the most powerful ways that you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, and it's especially important in Java, in which one of the primary goals is to create program components for others to use. To create a robust system, each component must be robust.

The goals for exception handling in Java are to simplify the creation of large, reliable programs using less code than currently possible, with more confidence that your application doesn't have an unhandled error.

Exceptions are not terribly difficult to learn, and are one of those features that provide immediate and significant benefits to your project. Fortunately, Java enforces all aspects of exceptions so it's guaranteed that they will be used consistently by both library designer and client programmer.

Exercises

1. Create a class with a **main( )** that throws an object of class **Exception** inside a **try** block. Give the constructor for **Exception** a string argument. Catch the exception inside a **catch** clause and print out the string argument. Add a **finally** clause and print a message to prove you were there.
2. Create your own exception class using the **extends** keyword. Write a constructor for this class that takes a **String** argument and stores it inside the object with a **String** handle. Write a method that prints out the stored **String**. Create a try-catch clause to exercise your new exception.
3. Write a class with a method that throws an exception of the type created in Exercise 2. Try compiling it without an exception specification to see what the compiler says. Add the appropriate exception specification. Try out your class and its exception inside a try-catch clause.
4. In chapter 5, find the two programs called **Assert.java** and modify these to throw their own type of exception instead of printing to **System.err**. This exception should be an inner class that extends RuntimeException.