EXCEPTIONS 10

This chapter discusses an aspect of Java that you will think has been lifted right off the System i. You are, no doubt, plenty familiar with the concept of exceptions on the System i. They were part of the original architecture of the System i (and System/38) that was a harbinger of things to come. Well, now their time has arrived! Let's begin by briefly reviewing the i5/OS exception architecture.

SYSTEM I EXCEPTION MODEL OVERVIEW

On the System i, the idea of sending messages from one program to another is a long-established part of the programming model. All operating system APIs and functions send messages when something unexpected happens—something "exceptional." These are sent both explicitly when you code a call to these APIs, and implicitly when they are invoked by a language runtime (such as an RPG database input/output operation). Language runtimes themselves also send messages when a programming error such as "divide by zero" or "array index out of bounds" happens.

Messages on the System i embed two important pieces of information:

- Error-message text, often with runtime substitution variables to pinpoint the problem (such as a source sequence number or error code).
- The severity, which for program-to-program messages is either `*ESCAPE`, `*STATUS`, or `*NOTIFY.`

All error messages have a unique seven-character *message identifier* that can be explicitly monitored for.

The System i message-exception model is most obvious when you are writing CL (Control Language) programs and you code explicit `MONMSG` (*Monitor Message*) statements for each command call. It is possible to monitor for explicit messages (such as `MCH0601`), a range of messages (such as by using 0000 for the numeric part of the message ID), or *function checks* (`CPF9999`). The function check monitors typically give sweeping "if anything at all happens, tell me about it" messages. Notice also that CL programs often send their own messages for diagnostic, status, or exceptional situations by using the `SNDPGMMSG` (*Send Program Message*) command. Programmers have learned that messages, when used properly, can be an effective way out of a troublesome situation such as receiving unexpected input.

The OPM exception model

In the *Original Programming Model* (*OPM*, meaning pre-ILE) days, exception messages were handled like this:

1. Does the program call-stack entry that received the message handle it (monitor for it or have code waiting to receive it)?
2. If yes, done.
3. If no, send a function check (message `CPF9999`) to that same program call-stack entry.
4. Does the program call-stack entry that received the message handle `CPF9999`?
5. If yes, done.

6. If no, blow away that program and send (*percolate*) that `CPF9999` to the previous entry in the call stack.

7. Repeat the previous step until the `CPF9999` is handled. (Ultimately, the job ends or the interactive command line returns control.)

The ILE exception model

When writing new ILE programs, the exception model is changed in the following ways:

- The original exception message is passed all the way up the call stack until a handler is found for it (that is, code that is willing to receive it). It is not converted to a function check right away.

- If nobody on the call stack (to the *control boundary*, which is an activation group, an OPM program, or the job boundary) handles this message, it is converted to a function check (`CPF9999`) and the process is repeated for the function check.

- If the original message is handled by somebody on the call stack, the entries above it are terminated.

- If nobody handles the original message, each is then given a chance to handle the function check, starting at the original call-stack entry that received the message.

Each entry in the call stack that does not handle the function check is typically removed from the call stack (depending on the user's answer to an inquiry message), and the next entry is given a chance to handle it. Further, the call stack itself is different in ILE. Not only does it contain programs, but it also contains procedures, which can have their own unique exception-handling support.

RPG III exception-handling

Now that you have seen the generic system support for exceptions, let's look closer at what is involved in RPG itself. As you recall, RPG III divides exceptions into two camps:

- *File errors*. These can occur when processing files, such as "record not found."

- *Program errors*. These are programming errors, such as "divide by zero."

RPG III offers three ways to handle exceptions:

- Error indicators (*Resulting Error Indicator*) on many op-codes. These are set by the language at runtime if the op-code fails.
- The `INFSR` error subroutine for file errors.
- The `*PSSR` error subroutine for program errors.

You can also code special data structures (`INFDS` and `PSDS`) that the language will update at runtime to indicate the error that occurred (in the `*STATUS` subfield). When returning from an error subroutine, the value of factor two on the `ENDSR` (*End Subroutine*) op-code can be used to determine where control returns. We will not bore you with further details because we assume that you are already intimately familiar with this process and architecture.

ILE RPG (RPG IV) exception-handling

How have things changed for RPG IV? That is a good question, but the answer could easily fill an entire chapter on its own. For more detailed information, consult the *ILE RPG/400 Programmer's Guide* (SC09-2074). However, in a nutshell, the basics follow:

- You still have error indicators, `INFSR` and `*PSSR` subroutines, and `INFDS` and `PSDS` data structures.
- The `INFSR` subroutine and `INFDS` data structures are identified on the F-spec with the new `INFSR(xxx)` and `INFDS(xxx)` keywords.
- The `INFSR` subroutines apply only to the mainline code, *not* to procedures. You will have to rely on error indicators for file-processing in a procedure.
- The `*PSSR` subroutines are local to the procedure where they are declared. (Yes, you can define subroutines inside procedures.) This means you need one for every procedure (although they could all call a common subprocedure) *and* one for the mainline code.

- The `*PSSR` subroutines inside procedures must have a blank factor two on the `ENDSR` statement—and if control reaches that far, the procedure will end there. Unfortunately, you have to rely on `GOTO` prior to `ENDSR` to continue processing.

- The `INFDS` and `PSDS` data structures are global in scope. That means that they are accessible by all procedures.

- There is an entirely new option: an ILE exception-handling bindable API, `CEEHDLR`. This registers an ILE exception handler for this procedure, and its undo cousin (`CEEHDLU`) "unregisters" an ILE exception handler. Using these APIs gives you a language-neutral way of dealing with exceptions in ILE. Typically, then, you code a call to `CEEHDLR` at the beginning of your procedure and a call to `CEEHDLU` at the end.

- The `%ERROR` built-in function can replace error indicators. If you specify an error extender on your op-code (by adding `E` in parentheses after the op-code), you can test if `%ERROR` returns one after the operation to establish whether an error occurred. You can also use the new `%STATUS` built-in-function to return the status code of that error. Other related built-in-functions are `%OPEN` to test if the given file is open, and `%EOF`, `%EQUAL`, and `%FOUND` to test if the previous operation resulted in an end of file, an exact record match, or a record match, respectively.

New MONITOR operation in RPG V5R1

Worth noting is an awesome capability added in V5R1 of RPG that makes exception-handling in much easier, and offers support very much similar to the Java exception support you will see shortly. As of V5R1, you can place one or more operation statements that may result in errors between a `MONITOR` and `ENDMON` set of op-code statements. The idea is that if any of the statements inside the monitor group results in an error, control will go to a particular `ON-ERROR` group.

You code one or more `ON-ERROR` operations within the monitor group, but after the statement you are monitoring. Each `ON-ERROR` op-code specifies a colon-separated list of status codes in free-form factor two, for which it is responsible. If an error happens during execution of any of the monitored statements, control will flow to the `ON-ERROR` statement that matches the status code of the error. You place your statements for handling the error after the `ON-ERROR` statement. All statements up to the next `ON-ERROR` or `ENDMON` statement are executed in this case. To handle the "otherwise" cases, you can specify special values in factor two instead of status codes. These are `*FILE`, `*PROGRAM`, and `*ALL`, which match on any file error, program error, or any error at all, respectively.

The following example is from the RPG IV reference manual:

```
  * The MONITOR block consists of the READ statement and the IF group.
  * - The first ON-ERROR block handles status 1211 which
  *   is issued for the READ operation if the file is not open.
  * - The second ON-ERROR block handles all other file errors.
  * - The third ON-ERROR block handles the string-operation status
  *   code 00100 and array index status code 00121.
  * - The fourth ON-ERROR block (which could have had a factor 2
  *   of *ALL) handles errors not handled by the specific ON-ERROR
  *   operations.
  * If no error occurs in the MONITOR block, control passes from the
  * ENDIF to the ENDMON.
 C                   MONITOR
 C                   READ      FILE1
 C                   IF        NOT %EOF
 C                   EVAL      Line = %SUBST(Line(i) :
 C                                    %SCAN('***': Line(i)) + 1)
 C                   ENDIF
 C                   ON-ERROR  1211
 C                    ... handle file-not-open
 C                   ON-ERROR  *FILE
 C                    ... handle other file errors
 C                   ON-ERROR  00100 : 00121
 C                    ... handle string error and array-index error
 C                   ON-ERROR
 C                    ... handle all other errors
 C                   ENDMON
```

If you have not discovered `MONITOR` and `ON-ERROR` yet, check them out!

EXCEPTIONS IN JAVA

The System i and RPG exception model has taught discipline when it comes to proactively designing support for error situations. If you don't follow this practice, you risk exposing those ugly function checks to your users. So, doing more work up-front prevents problems in the long run. You produce more robust, fault-tolerant code that is cheaper to maintain. (So, it is safe to say that RPG programmers are *exceptional*!) The Java designers have taken these noble goals to heart. (Actually, they are a reasonably standard OO thing.)

Java also provides the feature of exceptions for unexpected situations, and it has language constructs for sending and monitoring them. The consequences of ignoring them are even more frightening than on the System i. In fact, Java goes a step further than simply ending your program at runtime if you fail to monitor for an exception that happens. It actually tries to catch, at compile time, where you missed coding in monitors for potential exceptions. To accomplish this, it has further language syntax for defining, for each method, the exceptions that callers of this method need to monitor for.

The Java exception model at a glance

In a nutshell, Java's exception support includes the following:

- Exceptions in Java are simply objects of classes that extend the `Throwable` class.
- Java comes with many predefined exception classes, which can be found in the JDK documentation.
- System errors extend the `Error` class, while your own exceptions extend the `Exception` class. Both of these classes extend the `Throwable` class.
- Exception objects include message text retrievable via the `getMessage` method.
- Any code can throw an exception when it detects an error, by using the `throw` operator and passing it an exception class object.
- Any method that throws an exception must identify all exceptions it throws on the `throws` clause of the method definition.
- Many methods in Java-supplied classes throw exceptions.
- To call any method that throws exceptions, you must put the method call inside a `try` block, followed by one or more `catch` blocks.
- The `catch` block defines a parameter that is an object of an exception class. If an exception of that class or a child of that class is thrown, this `catch` block gets control.

- The `catch` block for a given `try` block can optionally be followed by a `finally` block, which is always executed whether an exception is thrown or not.

- As an alternative to putting an exception-throwing method call inside a `try` block, you can percolate the exception up the call stack by just defining the `throws` clause for those potential exceptions in your own method.

- Constructors can throw exceptions, too. If they do, the `new` operation is cancelled, and no object is created.

The following "exceptional" sections expand on these concepts.

Exception objects in Java

In contrast to RPG, Java does not have error indicators. It provides only the concept of exception messages, such as the System i exception model. What are these messages? They are Java objects, of course! There is a Java-defined class named `Throwable`, which all Java exceptions inherit from. This class is in the Java-supplied package `java.lang`, which all Java code implicitly imports. Any Java class that directly or indirectly extends `Throwable` is an "exception" in Java, whether that class is Java-supplied or written by you. You use unique language syntax to send these exceptions back up the method call-stack, and to monitor for them in code you call.

Objects of the `Throwable` class contain a string describing the exception, which is retrievable with the `getMessage` method. Another useful method in this class is `printStackTrace`, which prints out a method call-stack trace from the point where this exception was sent. Java programmers are particularly fond of this method because it is very useful in debugging. Here is an example of `printStackTrace`:

```
java.lang.NumberFormatException: abc
  at java.lang.Integer.parseInt(Integer.java:409)
  at java.lang.Integer.parseInt(Integer.java:458)
  at
ShowPrintStackTrace.convertStringToInt(ShowPrintStackTrace.java:20)
  at ShowPrintStackTrace.main(ShowPrintStackTrace.java:8)
```

This is from the exception `NumberFormatException` that the method `parseInt` in class `Integer` throws when it is given a string to convert to integer and that string contains non-numeric characters. You see that the stack trace starts with the name of the exception class, followed on the same line by the message text from the exception object (in this case abc, which is the invalid input used). Following that is the method-call stack, starting with the method that threw the exception (parseInt, in this case). Note the same method is listed twice in the example, indicating it calls itself recursively. The stack trace ends with the method that called printStackTrace. In this case, this is the main method in class ShowPrintStackTrace, shown in Listing 10.1. (The syntax of the try/catch blocks shown in Listing 10.1 is discussed later in this chapter.)

Listing 10.1: The Class ShowPrintStackTrace, which Generates an Exception and Stack Trace

```
public class ShowPrintStackTrace
{

    public static void main(String args[])
    {
        try
        {
           convertStringToInt("abc");
        }
        catch(NumberFormatException exc)
        {
           System.out.println(exc.getMessage());
           exc.printStackTrace();
        }
    }

    public static int convertStringToInt(String stringToConvert)
                     throws NumberFormatException
    {
        return (Integer.parseInt(stringToConvert));
    }
} // end ShowPrintStackTrace class
```

System i exceptions have a severity associated with them, as well as a unique message ID. Java exceptions (or Throwable objects) have this information, too. The severity and unique ID is implicit with the particular class of the exception object. In other words, there are many exception classes (that extend the Throwable class), so the exact error can be determined by the exact exception object used. This means the class, itself, is equivalent to a message ID because it uniquely identifies the error.

There really is no explicit severity associated with an exception in Java, but you can think of the child classes of Error as being critical, of RuntimeException as being severe, and all others as being normal. (These child classes are described shortly.) There are no informational or warning exceptions, since all Java exceptions can cause a program abend if not prevented or handled. Regular return codes are used instead of exceptions for informational and warning situations.

In addition to the implicit ID and severity that the class type implies for Java exceptions, message text is associated with each exception object, retrieved using `getMessage`, as mentioned earlier. This is also true of System i exceptions, of course.

The primary subclasses of `Throwable` are `Error` and `Exception`. The `Error` exceptions are typically non-recoverable system errors, such as "out of memory." The `Exception` errors are further subclassed by `RunTimeException` and other classes, as shown in Figure 10.1. The `RunTimeException` errors are programming errors you make, such as an array index out of bounds. (Hey, it happens.) The other subclasses of `Exception` are, typically, related to preventable user-input errors.

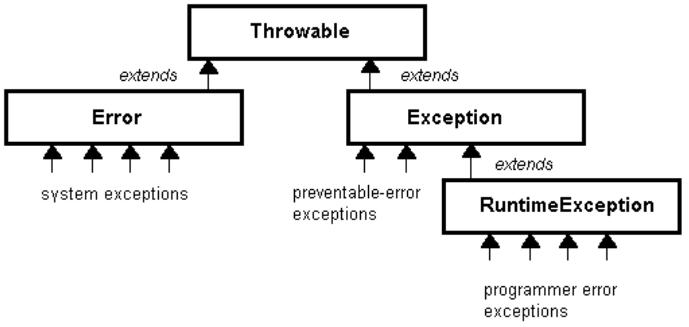

Figure 10.1: The major child classes of the Java `Throwable` class

You typically do not monitor for `Error` exceptions or subclasses in Java. For one thing, you usually can't do much about them. What's more, you will never send one of these exceptions yourself. These exceptions are sent only by the system. What you need to be concerned with are `Exception` exceptions (good name, is it not?) and their subclasses—both for sending and for handling. You have our permission to also ignore `RunTimeException` and its subclasses. These are used by Java to tell *you* that you made a programming error, not for you to tell *others* that you made a programming error. So, your code will probably send and handle only subclasses of `Exception`, except those that subclass `RuntimeException`.

There is little point in listing all of the subclasses here because, as you will see, every class you use clearly documents the `Exception` subclass objects it might send. You will learn them as you need them. (After all, it's probably safe to say that you don't know all the System i system and language runtime exception message IDs by heart.) We have no doubt that you will need them!

The last point to make about `Throwable` objects (it is only `Exception` objects that you really care about) is that you can define your own. You will probably need to do this in Java if you are writing robust code or, more precisely, *when* you are writing robust code. If you discover an unexpected error situation in your error-checking code, such as bad input or an expected resource not found, you should send an exception, not a return code. Return codes, such as an integer value, should be used to identify valid possible results, not to identify exceptional situations. For example, "end of file" is a valid possible result, while "file not found" is an exceptional situation. The first will almost always happen; with good input, the latter should almost never happen. It, in other words, is a frequency call. Having decided that you should send an exception, your next step is to peruse the Java documentation for an existing `Exception` subclass that applies to your situation.

Searching Java documentation for classes

It is time to learn how to find the JDK documentation for a particular class. In this case, you are interested in seeing a list of the classes that extend the `Throwable` class in the `java.lang` package, since that will show all the available predefined exceptions in Java. This will be important when you write your Java code, so you can look for existing exceptions for your own code to throw. Hopefully, the name of the exception class will give a clue to its intended use; from there, you can drill down to the detailed documentation about that class.

First, you must have downloaded and unzipped the JDK documentation file. (When using WinZip to do this, just specify the `c:\` drive or the drive you installed the JDK itself on, and select the "use folder names" checkbox from the WinZip Classic interface.)

Once your JDK documentation is properly expanded, navigate to the `docs\api` subdirectory of the `jdk` folder. This is where all your JDK documentation searches start. Typically, when looking for documentation for a particular package, class, or method, you simply double-click or open the `index.html` file in this `docs\api` directory. However, in this case, you are looking for something special: a list of all the JDK classes that extend `Exception`. To find this, go into subdirectory `java\lang` and open file e package-tree.html. Scroll down until the java.lang.Exception class, and you'll see something like this:

```
class java.lang.Exception
  class java.lang.ClassNotFoundException
  class java.lang.CloneNotSupportedException
  class java.lang.IllegalAccessException
  class java.lang.InstantiationException
  class java.lang.InterruptedException
  class java.lang.NoSuchFieldException
  class java.lang.NoSuchMethodException
```

This shows you a nice tree view of how classes extend each other. Remember, every class that extends Exception (but not RuntimeException) is an exception you might potentially use. To get more detail on any one class, just click its name. Mind you, this is only of limited value, as it does not show you all the child exception classes in other packages. We suggest you also open this file in the java\util and java\io directories, as they contain most of the useful and reusable exception classes.

If you find an existing exception that meets your needs, such as IOException, use it by throwing an object of that class. If you cannot find one that will work in your situation, or you prefer your own exceptions, create a new class that extends Exception (or one of its children), and design the constructor to call the parent's constructor with a string to be used as the error text. You can either hard-code this string or accept it as a parameter to your own constructor. An example of such a custom exception is shown in Listing 10.2, designed to report a string that is not a valid United States zip code (postal code).

Listing 10.2: An Extension of the Exception Class

```
public class BadZipCode extends Exception
{
    public BadZipCode(String errorText) // constructor
    {
        super(errorText);
    }
} // end class BadZipCode
```

This is the simplest possible exception class. The constructor simply takes a string and passes it to its parent (Exception), which will store it so code can later retrieve it via the getMessage method. You could elaborate on it though, and store additional information accessible with new getXXX methods, if you wanted. For example, you could ask the code that throws this exception to pass to the constructor a string telling the name of the method and class it is being thrown from, and log this information in a file. Since it is your class, you can do whatever you desire. However, at a minimum, you need to call the parent's constructor and pass a string.

Remember, all your new exceptions will automatically inherit the `getMessage` and `printStackTrace` methods of the root exception class `Throwable`.

Sending Java exceptions with the throw operator

Having decided that you will send an exception in your error-checking code, how do you do it? First, you have to instantiate an instance of the particular `Exception` child class, which almost always requires a string parameter that is the text extractable with a `getMessage` call later. Then, you use the Java `throw` operator:

```
BadZipCode excObj = new BadZipCode("Zip code is not all numeric");
throw (excObj);
```

These steps can be combined into one statement, of course:

```
throw (new BadZipCode("Zip code is not all numeric"));
```

The `throw` operator is similar to CL's `SNDPGMMSG` command on the System i. You can either send an instance of one of Java's predefined exception classes (such as `IOException`), or you can send an instance of your own exception class (such as `BadZipCode`). This choice is comparable to deciding whether to use a supplied `CPF` or `MCH` message on the System i, or to create your own new message in your own message file. By the way, the generic message `CPF9898 ("&1")` that many of us use on the System i is similar to the generic `Exception` class in Java. On the System i, you substitute your own message text in `CPF9898`. You can do the same in the constructor of `Exception`, as shown below:

```
throw (new Exception("You made a big mistake there pal!"));
```

It is important to remember that you never throw exceptions that are of type `Error`. You use `Exception` because `Error` exceptions are for dramatic system errors and are thrown only by the system.

What does using `throw` do? It ends your method! Any code following the `throw` statement is not executed. You have done the equivalent of sending an escape message in a CL program. The current method is removed from the stack, and the exception is sent back to the line of code that called this method. If that code does not monitor for this exception, the method it is in is also terminated. The exception is sent back to the caller of the method, just as in RPG function-check percolation. It continues until it finds an entry in the call stack that monitors for this exact exception (or one of the parents of this particular exception class, as you will see).

Who throws what

In Java, callers of your method must monitor for any exceptions that you throw. They do this using a `try/catch` block, identifying the exception to monitor for in the `catch` block parameter. This is not unlike System i programming, where calls to CL programs must take care to monitor for any messages sent by the called CL program.

If you have done any CL programming, you know how painful it can be to get those `MONMSG` statements just right. You typically have to examine the CL reference manual for each CL command you call, to see what messages that command might send. And you can only hope that the list includes any messages that its nested command or program calls might send.

How many times have you wished for an automated way to determine this list? For example, it would be nice to have a tool that, given a CL command as input, returns a list of all the possible messages that particular CL command might send. OO language programmers face a similar problem when trying to determine the exceptions any particular method call might result in. Java designers thought about this problem. They knew that if they didn't come up with a solution, the exception architecture in Java would suffer two real-use problems:

- Programmers would not use it enough, which would lead to too much error-prone code (human nature being what it is). This would lead to a bad image of Java.

- Programmers who did decide to place sensitive method calls inside `try/catch` blocks would find it painful to determine what exceptions each method could possibly throw. Programmers would be dependent on all methods having proper and up-to-date documentation about what exceptions they throw (much as you are dependent on this for System i commands).

The Java designers decided to force method designers to specify up-front, in the method signature, what exceptions are thrown by that method. This is done by specifying a `throws` clause on your method signature, like this:

```
public void myMethod(ZipCode zipcode) throws BadZipCode
```

You *must* specify this if you explicitly throw an exception or your compile will fail. If you throw multiple exceptions, they must all be specified, comma-separated, as in:

```
public void openFile(String filename) throws FileNotFound,
                                             FileNotAvailable
```

By putting this information into the method declaration, it automatically ends up in the JavaDoc documentation, which solves the documentation problem. Further, it explicitly tells the compiler what exceptions your method throws, so the compiler can subsequently force all code that calls this method to monitor for those exceptions. If any calling code does not have a `catch` block for each of the exceptions listed, then that calling code will not compile. This solves the lazy-programmer problem.

It may be that the calling code does not know how to handle the error indicated by the exception. If so, there is a way out. As an alternative to monitoring for exceptions with a `try/catch` block, the method containing the called code can simply repeat the `throws` clause on its own signature. If this is the case, if the called method throws an exception, the calling code simply percolates it back up to its own caller. The stack is peeled at the line of code that called the exception-throwing method. This can continue all the way up the stack to the root method—`main` in the first class. If it does not specify a `try/catch` block, you have a problem, as `main` is the root of the call stack, so it has no calling-method to percolate to. In this case, if `main` does not specify a `try/catch` for the offending exception, that code will simply not compile. If there were a way to compile it, though, the program would die at runtime in much the same way an System i program dies with an "unmonitored exception." Isn't it nice that the compiler works so hard to eliminate all these errors up-front, before your users get the chance to?

Let's bring this all together with an example. Listing 10.3 shows a class designed to encapsulate a United States zip code. It takes the string with the zip code as input in the constructor and stores it away. As a convenience, it also offers a static `verify` method to ensure that a given string is a valid United States zip code. (Note that it handles both versions, with and without a box office code.)

Listing 10.3: The ZipCode Class to Encapsulate a Zip Code

```
public class ZipCode
{
    protected String code;
    protected static final String DIGITS = "0123456789";

    public ZipCode(String zipcode) throws BadZipCode
    {
        if (verify(zipcode))
          code = zipcode;
    }

    public static boolean verify(String code) throws BadZipCode
    {
        code = zipcode.trim();
        int codeLen = code.length();
        StringBuffer codeBuffer = new StringBuffer(code);
        BadZipCode excObj = null;
        switch (codeLen)
        {
           case 10: // must be nnnnn-nnnn
               if (code.charAt(5) != '-')
                 {
                   excObj = new BadZipCode("Dash missing in
                           6th position for '" + code + "'");
                   break;
                 }
               else
                 codeBuffer.setCharAt(5, '0');
               // deliberately fall through remaining case
           case 5: // must be nnnnn
               if (RPGString.check(DIGITS, codeBuffer.toString())
                   != -1)
                 excObj = new BadZipCode("Non-numeric zip code '"
                                        + code + "'");
               break;
           default:
               excObj = new BadZipCode("Zip code '" + code +
                         "' not of form nnnnn or nnnnn-nnnn");
        } // end switch
        if (excObj != null)
          throw excObj;
        return (excObj != null);
    } // end verify method

    public String toString()
    {
        return code;
    }
} // end ZipCode class
```

In the interest of reuse, this class uses the static `check` method from Chapter 7 to verify that the string has only digits. Alternatively, you could have walked the string, calling the static method `isDigit` (from the `Character` class) on each character.

The `verify` method throws the `BadZipCode` exception from Listing 10.2 if it detects an error. It simply places different text in the exception for each error situation. The constructor calls the `verify` method to ensure it has been given a valid string. Because this call to `verify` is not encased in a `try/catch` block, the `throws` clause must be re-specified on the constructor itself. This means all code that tries to instantiate this class must put the call to `new` inside a `try` block, followed by a `catch` block for `BadZipCode`. If the constructor does throw the exception, the `new` operation will be aborted. (You will see an example of this after the `try/catch` syntax is discussed in the next section.)

Sometimes, your calling code might decide to handle a thrown exception, but then throw it again anyway. This is legal, and can be done with a simple `throw exc;` statement in your `catch` block. In this case, because you are throwing this exception (albeit, again), you must define it in your method's `throw` clause.

In summary, if your method code does not monitor for an exception it might receive, you must specify that exception in your method's `throws` clause in addition to any exceptions your code explicitly throws, or re-throws.

Monitoring for Java exceptions with try/catch blocks

Now that you know how to send or throw an exception to the callers of your code when you have detected an error, let's discuss what those callers do to monitor for or process it. To monitor for an exception, there is additional Java language syntax. The Java syntax for monitoring for messages builds on this, allowing you to specify a `try/catch` combination, as follows:

```
try
{
  // try-block: one or more statements of code
}
catch (Exception exc)
{
  // catch-block: code to handle the exception
}
```

The idea is to place any method call statement that might throw exceptions inside a `try` block. Because it is a block, you can actually place one or more statements inside it. If any of the statements inside the `try` block do throw an exception, the `catch` block will get control, and any code after the exception-throwing call will not be executed. The control flows immediately to the `catch` block upon receipt of a thrown exception.

The `catch` block defines a parameter, which is the exception it will handle. Java passes that exception object at runtime if an exception is thrown. Your `catch` block code can use methods on the object to display information to the end-user, if desired. For example, you may do something like the following inside your `catch`-block:

```
System.out.println(exc.getMessage());
```

Recalling the zip code example, Listing 10.4 is a method included in the `ZipCode` class for testing purposes. Given a string, it will try to instantiate and return a `ZipCode` object. Also included is a `main` method that uses this method.

Listing 10.4: Testing the ZipCode Class

```
public static ZipCode testZipCode(String code)
{
    System.out.println("Testing '" + code + "'...");
    ZipCode testCode = null;
    try
    {
      testCode = new ZipCode(code);
    }
    catch (BadZipCode exc)
    {
      System.out.println(" ERROR: " + exc.getMessage());
    }
    return testCode;
}
// For testing from the command line.
public static void main(String args[])
{
    // test two valid zip codes, 3 invalid zip codes...
    testZipCode("12345");
    testZipCode("12345-6789");
    testZipCode("1234567890");
    testZipCode("abc");
    testZipCode("123");
}
```

If you want to see this in action, here is the output of running this class:

```
Testing '12345'...
Testing '12345-6789'...
Testing '1234567890'...
 ERROR: Dash missing in 6th position for '1234567890'
Testing 'abc'...
 ERROR: Zip code 'abc' not of form nnnnn or nnnnn-nnnn
Testing '123'...
 ERROR: Zip code '123' not of form nnnnn or nnnnn-nnnn
```

As another example of the same problem (verifying an input string and, if valid, creating an object to wrapper it), Listing 10.5 is a `PhoneNumber` class that throws a `PhoneNumberException` exception.

Listing 10.5: The PhoneNumber Class that Throws PhoneNumberException

```
public class PhoneNumber
{
    protected String number;
    protected static final String PHONEDIGITS = "0123456789.- ";

    protected PhoneNumber(String number)
    {
       this.number = number;
    }
    public String toString()
    {
        return number;
    }

    public static PhoneNumber createPhoneNumber(String number)
                              throws PhoneNumberException
    {
        PhoneNumber numberObject = null;
        if ((RPGString.check(PHONEDIGITS, number) != -1) ||
            (number.length() < 10) ||
            (number.length() > 12))
          throw new PhoneNumberException("Phone number '" +
                    number + "' does not appear to be valid");
        else
          numberObject = new PhoneNumber(number);
        return numberObject;
    }

    public static void main(String args[])
    {
```

Listing 10.5: The PhoneNumber Class that Throws PhoneNumberException (continued)

```
        System.out.println("Testing...");
        testPhoneNumber("5551112222");
        testPhoneNumber("555-111-2222");
        testPhoneNumber("555.111.2222");
        testPhoneNumber("555 111 2222");
        testPhoneNumber("a");
        testPhoneNumber("1");
        testPhoneNumber("555/666/7777");
        testPhoneNumber("123456789012345");
    }
    private static void testPhoneNumber(String number)
    {
        PhoneNumber nbr = null;
        try {
           nbr = createPhoneNumber(number);
           System.out.println(nbr + " is valid");
        } catch (PhoneNumberException exc) {
           System.out.println("Error: " + exc.getMessage());
        }
    }
}
```

Notice that Listing 10.5 takes a slightly different approach with the constructor. Rather than making the constructor public and defining it to throw exceptions, this code makes it protected so that only family members can instantiate it, and designs it not to throw exceptions. You don't want the public using new to instantiate `PhoneNumber` objects; rather, you want them to call your `createPhoneNumber` factory method, which will return a new `PhoneNumber` object. However, it won't do that unless the input string is a valid phone number, so you can guarantee the input to the constructor is always valid. If it is not, then the `PhoneNumberException` exception object is thrown. (This is not shown, but it is very similar to `BadZipCode` in Listing 10.2.) There is some code in `main` to test this, and running it gives this:

```
5551112222 is valid
555-111-2222 is valid
555.111.2222 is valid
555 111 2222 is valid
Error: Phone number 'a' does not appear to be valid
Error: Phone number '1' does not appear to be valid
Error: Phone number '555/666/7777' does not appear to be valid
Error: Phone number '123456789012345' does not appear to be valid
```

Note that the class doesn't pretend to know a universal syntax for phone numbers. We're sure you can improve on the validation routine.

Monitoring for multiple Java exceptions

The `catch` statement, not the `try` statement, is actually equivalent to CL's `MONMSG`. Although both `try` and `catch` are necessary syntactically, it is the `catch` statement that tells Java which exception type you are monitoring for. If your `try` block gets an exception that the `catch` statement did not identify in its parameter, it is as though you never had the `try/catch` block. Your method is ended, and either the exception is sent back to the previous call-stack entry, or compile will fail if you don't identify the missed exception on your `throws` statement on your method signature.

What if you call a method that throws more than one possible exception? How do you define the `catch` statement when you need to monitor for multiple possible exceptions? Two options exist:

- Suppose it does not matter to you which exception happened; it only matters that some exception happened. You can define a parent exception class type on the `catch`. The `catch` will actually get control of any exception that is of the defined type or lower on the hierarchy chain. This is similar to specifying `MCH0000` on the CL `MONMSG` command. Alternatively, to catch all exceptions, specify the root parent of all catchable exceptions: `Exception`. (Some people use `Throwable`.) This is equivalent to specifying `CPF9999` on the CL `MONMSG` command.

- Define multiple `catch` blocks after the `try` block. This is perfectly legal. The exception object received will be compared to each `catch` statement's parameter, in turn, until a match on type is found (or the `catch` defines a child of the thrown exception class type). Use this technique when it is important to your error-recovery code to know exactly what exception was thrown. The need for unique error-handling code is also a good criterion to use when deciding whether you need to define your own exception classes.

Here is an example:

```
try
{
   someObject.callSomeMethod();
}
catch (FileNotFound exc)
{
  . . .
}
catch (FileNotAvailable exc)
{
  . . .
}
```

Finally, there is `finally`. This is an optional block you can define at the end of all your `catch` statements:

```
try
  block
catch (exception-type-1 identifier)
  block
catch (exception-type-2 identifier)
  block
finally
  block
```

You might think this is what will get control in an exception situation if none of the `catch` statements handled a particular exception type. This is only partly correct, however. The `finally` block, if present, is *always* executed. That is, it is executed whether or not an exception was received in the `try` block, and whether or not a `catch` block processed it. For example, if a `BadZipCode` exception is thrown by code in the `try` block, the code inside the `BadZipCode` `catch` block will be executed as well as the code inside the `finally` block.

The `finally` statement is typically used to do code that has to be done no matter what, such as closing any open files. No statement inside a `try` block, not even a `return` statement, can circumvent the `finally` block, if it is present. If the `try` block does have a `return` statement, then the `finally` block will be run and the `try` block's `return` statement will be honored. (It is, however, possible to override the `try` block's `return` in the `finally` statement by coding another `return` statement.)

SUMMARY

This chapter covered the following:

- A review of the System i and RPG exception model
- An introduction to the Java exception model
- The Java `Exception` class hierarchy
- The Java `throw` operator, which is like CL's `SNDPGMMSG`
- The Java `try/catch/finally` statement, which is like CL's `MONMSG`
- The `catch` block, which catches the defined exception or any exception that is a child of it
- The `finally` block, which if present is always executed
- The Java `throws` clause for method signatures
- The fact that throwing an exception in a constructor is legal, and cancels the instantiation
- The two popular methods all exceptions have: `printStackTrace` and `getMessage`
- How to write your own exception classes by extending the `Exception` class or one of its children.
- The two hierarchies into which Java-supplied exceptions are divided: those that extend `Error` and do not need to be monitored, and those that extend `Exception` and do need to be monitored