Chapter 3: Exception Handling

Exception Handling

An exception is an event that interrupts the normal flow of a program's logic. In this chapter, we'll discuss a bit about the different types of exceptions you might encounter. Then we'll look at how to *trap* them so (we hope) they won't blow up your programs. Then you'll find out how to generate exceptions of your own if you need them. (Yes, you really might want to test your code by putting errors in it!)

Perhaps the most common problem that programs encounter in programs is *unexpected input*. We mentioned that briefly in Lesson 7. For instance, if a user entered letters when we expected a number before we checked the input, we got an InputMismatchException that caused our program to crash.

Java's *application program interface* (API) defines several exceptions, and every method's documentation includes (or should include) a list of the exceptions it might generate, or *throw*.

The terminology in Java for generating an exception is throwing it. A thrown exception will cause the program to blow up . . . unless the program handles it by trapping it. When we trap an exception and process it, programmers call that *catching* it.

Let's look at some of the basic exception types.

All Java exceptions are subclasses (more on that shortly) of the Exception classes, and they come in two basic "varieties": *runtime exceptions* and *checked exceptions*.

Checked exceptions are ones that the compiler can check and catch; the compiler will make sure there's code to process them, either by handling the problem or by "passing the buck" to another level.

Runtime exceptions typically come from programming problems that the compiler can't really anticipate. These include dividing by zero, using a null reference, or making invalid input that leads to an InputMismatchException.

Even though it's a bit more work to set up checked exceptions than it is to set up runtime exceptions, we should use checked exceptions whenever possible for a couple of reasons.

- Checked exceptions allow the compiler to make sure that programmers consider what should happen when the program encounters exceptions—and that reduces the likelihood of program crashes.
- The exceptions themselves are their own documentation of what might happen, and they help a programmer to plan for problems. So they reduce system problems in the long run.

I've probably thoroughly confused the issue by discussing a lot about concepts we haven't dealt with yet! Let's get into how to use exceptions.

Catching Exceptions

Let's suppose someone wants to use our Temperature class, but the person just does what the first version of our driver program did: try to read a double value for a temperature and not check it first. The user could end up with the same InputMismatchException I showed you in the last lesson.

One way to deal with this problem is checking the input data first (which is a better, more efficient solution, by the way). But another way is to catch the exception. I'm going to show you how to do that, even though it's not a good practice to catch exceptions when we could have avoided them in the first place.

To catch an exception, we need to put the code that can cause the exception into what's called a *try block*. Then the code that deals with exceptions goes into a *catch block*. Here's how our do-while loop could look for our InputMismatchException:

```
do {
    System.out.print("Enter a temperature: ");
    try {
        inputTemperature = keyInput.nextDouble();
        if (Temperature.isTemperatureValid(temperatureType, inputTemperature)) {
            goodTemperature = true;
        }
        else {
            System.out.println("You entered an invalid temperature!");
            System.out.println("It must be greater than absolute zero.");
            System.out.println("Try again.");
            keyInput.next();
        }
    }
    catch (InputMismatchException e) {
        System.out.println("You entered an invalid temperature!");
        System.out.println("It must be a numeric value.");
        System.out.println("Try again.");
        keyInput.next();
    }
} while ( ! goodTemperature);
```

What will happen when this code runs? The first line of the do-while loop asks the user for a temperature value. After the user enters a value, the next line will try to read a double value, and if it's valid, it will proceed to the code that follows in the if statement. The program will ignore the catch block.

If, however, the user enters anything besides a valid number, the program will generate an InputMismatchException, and that in turn will cause the code in the catch block to execute. The program will ignore the remainder of the try block.

So using that try-catch technique, we can achieve the same results as we did before. As I said, though, this code is considerably less efficient than calling the hasNextDouble() method to check for numeric input, so this isn't the preferred way to code.

Let's look at the general format of the try-catch mechanism, since this technique is important in trapping errors. Here's the general format:

```
try {
    // code that might cause an exception
}
catch (<ExceptionType1> name) {
    // code for the first exception
}
catch (<ExceptionType2> name) {
    // code for the second exception
}
 . . .
finally {
    // code to execute whether there's an exception or not
}
```

You can see from the format that it's possible to have more than one catch block for a try block. You can have as many as you need to trap all the exceptions the code in the try block might generate. And after all of them, a *finally block* is optional, but you can use it for code that the program should execute after the try block whether or not there's an exception.

For example, a file-reading process might cause several exceptions. (I know we haven't looked at processing files yet, but I'm just using this as an example.) Among the exceptions a file process could encounter are FileNotFoundException, InputMismatchException, and IOException.

Here's a try-catch-finally sequence to handle all those. Since the program should close a file regardless of whether the program encountered an error, I'll include a finally block with logic to close the file.

```
try {
    // code to process file(s)
}
catch (FileNotFoundException e) {
    // code to handle a missing file condition
}
catch (InputMismatchException e) {
    // code to handle invalid data conditions
}
catch {IOException e) {
    // code to handle I/O exceptions
}
finally {
    // code to close the file(s)
}
```

I mentioned that the compiler checks to make sure some exceptions get handled in a program. How would the compiler know which checked exceptions a method might throw? For that matter, now that you know how to handle exceptions, how do you know which exceptions to code for? Read on!