Input Validation

When humans interact with technology, a great diversity in experience is brought to bear on the task, and things often go amiss. If the technology is well-designed, mistakes or unusual or unexpected interactions are anticipated and accommodated in a safe and non-destructive manner. Computer software is no exception. When a user enters a syntactically incorrect value or selects a disabled feature, for example, the software must recognize, accommodate, and recover from the anomalous condition with minimal disruption. In other words, our software must expect the unexpected. In this section, we will illustrate how to write simple programs that recognize and recover from invalid user input. We call this input validation.

Many of our programs were designed to receive input from the keyboard using the readLine() method of the BufferedReader class. The readLine() method returns a string holding a full line of input. If the user entered an integer or floating point number, the string was converted to an int using the Integer.parseInt() method or to a double using the Double.parseDouble() method. For example

```java
String s = readLine();
int value = Integer.parseInt(s);
```

But, what if the user entered something like "thirteen", instead of "13"? Here's the reaction from the DemoMethod program presented earlier:

```
PROMPT> java DemoMethod
Enter an integer: thirteen
Exception in thread "main" java.lang.NumberFormatException: thirteen
  at java.lang.Integer.parseInt(Compiled Code)
  at java.lang.Integer.parseInt(Integer.java:458)
  at DemoMethod.main(DemoMethod.java:21)
PROMPT>
```

Obviously, thirteen is not our lucky number. The output is hardly what we'd call a user-friendly error message. The program has crashed, and we're left high and dry. The critical term above is "NumberFormatException". There was an attempt to parse the string "thirteen" into an int and the parseInt() method didn't like what it found. The string "thirteen" has no correspondence to an integer — at least, not in the eyes of the parseInt() method. It reacted by throwing a "number format exception".

In this section, we'll examine a simple way to implement input validation. If the user input is not correctly format, we want to gracefully recover, if possible. And we definitely don't want our program to crash. Input validation is well-suited to implementation in methods; so, this a good place to introduce the topic. We'll re-visit the topic in more detail later when we examine exceptions.

The trick to input validation, as presented in this section, is to inspect the input string before passing it to a parsing method. If an invalid string is parsed, then it's too late for graceful recovery. We want to identify the problem before parsing, and recover in a manner that seems appropriate for the given program. The demo program InputInteger illustrates one approach to the problem (see Figure 1).
```java
import java.io.*;

public class InputInteger {
    public static void main(String[] args) throws IOException {
        // setup 'stdin' as handle for keyboard input
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in), 1);

        // send prompt, get input and check if valid
        String s;
        do {
            System.out.print("Enter an integer: ");
            s = stdin.readLine();
        } while (!isValidInteger(s));

        // convert string to integer (safely!)
        int i = Integer.parseInt(s);

        // done!
        System.out.println("You entered " + i + " (Thanks!)");
    }

    // check if string contains a valid integer
    public static boolean isValidInteger(String str) {
        // return 'false' if empty string
        if (str.length() == 0)
            return false;

        // skip over minus sign, if present
        int i;
        if (str.indexOf(' - ') == 0)
            i = 1;
        else
            i = 0;

        // ensure all characters are digits
        while (i < str.length())
            if (!Character.isDigit(str.charAt(i)))
                break;
        i++;

        // if reached the end of the line, all characters are OK!
        if (i == str.length())
            return true;
        else
            return false;
    }
}
```

Figure 1. InputInteger.java

A sample dialogue with this program follows:
PROMPT> java InputInteger
Enter an integer: (blank line)
Enter an integer: - (only a minus sign entered)
Enter an integer: ninety nine (alpha characters not allowed)
Enter an integer: -ninety nine (sorry! try again)
Enter an integer: 99 (leading spaces not allowed)
Enter an integer: 99 98 97 (no spaces allowed)
Enter an integer: 99 (finally!)
You entered 99 (Thanks!)

A variety of incorrect responses are shown above. In all cases, the program reacted by re-issuing the prompt and inputting another line. As evident in the 55 lines of source code, even this simple implementation of input validation is tricky. The good news is that the input routine is packaged in a method. At the level of "using" the method, the code is very clean (see lines 11-18).

Let's look inside the definition of the isValidInteger() method. For the purpose of this program, an integer string has the following characteristics: (a) it must contain no leading or trailing spaces, (b) it may contain an optional minus sign at the beginning, and (c) it must contain only digit characters until the end of the string.

The first "processing" task is to ensure that the string is not an empty string. If the line length is zero, the method returns false — the string does not contain a valid integer! (See lines 30-32). The next order of business is to check if the first character is a minus sign. An index variable i is initialized with 0 if the first character is not a minus sign, or 1 otherwise to skip over the minus sign. The string is scanned from this point to the end checking that each character is a digit. The check is performed in line 44 as follows:

```java
if (!Character.isDigit(str.charAt(i)))
```

The character at position i in the string is extract using the charAt() method of the String class and presented to the isDigit() method of the Character wrapper class. The return value is true if the character is a digit, false otherwise. If false is returned the relational test is true — because of the NOT operator (!) — and the following statement executes. The following statement is a break, which causes an immediate exit from the while loop. The final value of the index variable i is either str.length() if all the characters checked out as digits or something less than str.length() if the loop terminated early. This condition is check in line 50 and the appropriate boolean result is returned: true if the string is a valid integer, false otherwise.

Despite the best intentions, the approach to input validation just shown is not as robust as we'd like. If the user enters 12345678999, for example, the program still crashes with a number format exception. (The largest integer represented by an int is \(2^{31} - 1 = 2,147,483,647\).) As well, it is not entirely clear that we should reject input with leading or trailing spaces. A far better approach to input validation is to "deal with" the built-in exceptions generated by Java's API classes. We'll learn how to do this later. For the moment, the approach shown in InputInteger will serve us well.

Since isValidInteger() is a static method, it is available to other programs, provided the InputInteger.class file is reachable by the compiler. We could, for example, include the following lines in another Java program:
String s;
s = stdin.readLine();
if (InputInteger.isValidInteger(s))
{
    /** process input */
}

The prefix InputInteger is simply identifies the the class where the static method isValidInteger() is found, much the same as “Math” in Math.sqrt(25) identifies the class where the sqrt() method is found.