Chapter 11
Command-Line Arguments

Command Line Arguments

We now turn our attention to an alternate means of gaining input to our program. Whenever we run a program:

```
demo$ java MyProgram
```

We actually have the chance to pass as many String values in as we want:

```
demo$ java MyProgram str1 str2 str3 …
```

We receive these strings through the String[] args parameter of the main method:

```
public class TestCLArgs {
	public static  void main (String[] args) {
		System.out.println("Received command line args: ");
		for(String arg : args) {
			System.out.println(arg);
		}
	}
}
```

Copy the above code into a new file, and then try running it at the terminal:

```
demo $ javac TestCLArgs.java
demo $ java TestCLArgs hello world 1 2 3
```

The strings are separated by spaces. If you want a space character as part of a string, just surround the entire string value in quotes (single or double will work here):

```
demo $ java TestCLArgs "light blue" red 'hot pink' "1 2 3"
```

No matter what you type after `java TestCLArgs`, you will get them as String values. If you want to read in numbers, you must convert from a String to the appropriate numeric type:

```
public class TestCLArgs {
	public static void main (String[] args) {
		// if there aren't exactly three args, print an error message and quit.
		if (args.length !=3) {
			System.err.println("usage: java TestCLArgs # # #");
			System.exit(1);
		}

		int a = Integer.parseInt(args[0]);
		short b = Short.parseShort(args[1]);
		double c =Double.parseDouble(args[2]);
		boolean result = (double)a/b == c;
		System.out.println("a="+a+"\nb="+b+"\nc="+d+"\nresult="+result);
	}
}
```

We see that we can convert from `String` to any other basic type through the wrapper classes' *parseX* methods.

Digression #1
We see that we can print to the error-reporting stream (`System.err`) as easily as we print to standard output (`System.out`). Although this output is usually just shown on the terminal window intermingled with `System.out`'s output, they are two separate streams of characters that can be dealt with separately. Any message that relates to program malfunction ought to be sent to `System.err`, while expected behaviors should result in output to `System.out`. Even "whoops, try again!" should be sent to `System.out`: we expect users to see this message and then type better input next time; that is a normal behavior of the program.

Digression #2
We also see how to immediately exit a program: `System.exit(someIntValue).` An exit value of zero represents normal termination (success); any other `int` value indicates abnormal quitting, and the chosen number should mean something to the programmer to indicate what sort of abnormal reason for quitting occurred.

Your Turn!

- Write a program that accepts an arbitrary number of integers on the command line, and then prints their sum and average.
- Write a program that accepts exactly 5 `String` arguments (or else it quits with an error message), and then checks against a secret password `String` that is only written in the source file. Either print `"You said the password, among other things!"`, or `"you didn't say the password."`
- Questions:
    - What are some key differences between using command line arguments versus the keyboard? Versus reading from a file?
    - When is each a better choice?
    - Does using the command-line redirection operator (<) still work with command-line arguments, or not? What is an example where you would even want to use both? Construct a sample call using both, or describe how you would work around not being able to use both, based on what you find out. (recall that on the command line, we can direct the contents of a file to be used as the "keyboard input" that System.in reads. This is done such as **`java MyProg < readFromHere.txt`**).
- **Larger Example:** Create an enumeration with three values: sparse, normal, verbose. Allow a flag to be passed in through a command-line argument that is either –s, -n, or –v to represent each value. Write a program that now gives sparse, normal, or verbose messages each printing time. You could even have a method that is always given three messages, and prints the correct one based on the current verbosity.
  → with different ways to represent it, many programs allow for "verbose output." Though this is perhaps not a common way to implement it, the same idea of varied levels of printed chatter definitely shows up in many command-line-based programs.