Chapter 6

Loops and Strings

6.1 The `do-while` Loop

The following section is ©Alan Downey and Chris Mayfield, and is material that was originally in *Think Java*, but later removed. I think it's an important topic, so I am including it, slightly modified, here.

The `while` and `for` statements are **pretest loops**; that is, they test the condition first and at the beginning of each pass through the loop.

Java also provides a **posttest loop**: the `do-while` statement. This type of loop is useful when you need to run the body of the loop at least once.

For example, you can use a `do-while` loop to keep reading input until it's valid:

```
Scanner in = new Scanner(System.in);
boolean okay;
do {
    System.out.print("Enter a number: ");
    if (in.hasNextDouble()) {
        okay = true;
    } else {
        okay = false;
        String word = in.next();
        System.err.println(word + " is not a number");
    }
} while (!okay);
double x = in.nextDouble();
```

Although this code looks complicated, it is essentially only three steps:

1. Display a prompt.
2. Check the input; if invalid, display an error and start over.
3. Read the input.

The code uses a flag variable, `okay`, to indicate whether we need to repeat the loop body. If `hasNextDouble()` returns `false`, we consume the invalid input by calling `next()`. We then display an error message via `System.err`. The loop terminates when `hasNextDouble()` return `true`.

Again, the important difference between `while` and `do-while` is this: In a `while` loop, the test happens *before* the loop body is executed. If the condition evaluates to `false`, the loop body doesn't get executed.

In a `do-while` loop, the loop body is always executed at least once, and only then is the condition evaluated. This turns out to be a good choice for validating input, because you must get the user's input before you can determine whether it's valid or not.

6.2 Pre vs. Post Increment and Decrement

The book discusses the increment `++` and decrement `--` operators, which increase or decrease a variable's value by one.

You can put these operators before or after a variable name. For example, in this code fragment:

```
int n = 47;
n++; // n post-increment to 48
++n; // n pre-increment to 49
n--; // n post-decrement to 48
--n; // n pre-decrement to 47
```

When a variable is all by itself, there is no difference between the operator before or after the variable. However, you may see people using these operators inside an expression, and then there is a very big difference.

Consider this code fragment:

```
int n = 47;
int result = n++;
System.out.println("result is now " + result);
```

Because this is a post-increment, Java access the value of `n` and assigns it to `result`, and *then* adds one to `n`. It's as if we wrote it this way:

```
int n = 47;
int result = n;
n = n + 1; // increment after assignment
System.out.println("result is now " + result);
```

If we place the `++` *before* the variable (a pre-increment):

```
int n = 47;
int result = ++n;
System.out.println("result is now " + result);
```

Java will increment `n` *before* it assigns the value to `result`; it's as if we had written:

```
int n = 47;
n = n + 1; // increment before assignment
int result = n;
System.out.println("result is now " + result);
```

While the following is not something you should do in your programs, you may see other people coding this way, and you'll need to understand it:

```
int a = 7;
int b = 11;
int c = a++ * --b;
```

Here's the trick. First, find all the pre-increments and pre-decrements and evaluate them before the expression:

```
int a = 7;
int b = 11;
b = b - 1; // do the pre-decrement
int c = a++ * b;
```

Then, find all the post-increments and post-decrements and evaluate them after the expression:

```
int a = 7;
int b = 11;
b = b - 1;
int c = a * b;
a = a + 1;
```

Now there are no more increment and decrement operators, and you have just "plain Java expressions," with the result being `c` being assigned `70`.

6.3 Compound Assignment

The book mentions that you can increment a variable by two in this way:

```
int n = 12;
n += 2; // same as n = n + 2;
```

I read this expression as "`n` plus and becomes two."

In fact, in any expression where the variable on the left side of the assignment operator is the same as the first variable on the right hand side, you can use this shortcut:

This:	is the same as:
`n += 2;`	`n = n + 2;`
`n -= 3;`	`n = n - 3;`
`n *= 4;`	`n = n * 4;`
`n /= 5;`	`n = n / 5;`
`n %= 6;`	`n = n % 6;`

You can use compound assignment in more complex expressions: `n += m * 5;` is equivalent to `n = n + (m * 5);` However, you can *not* use it if the left-hand variable is part of a parenthesized expression. `n = (n + 7) * 3;` cannot be simplifed using compound assignment.

6.4 The break statement

Some of this material is adapted from material that was in an earlier edition of Think Java.

Sometimes neither a pretest nor a posttest loop will provide exactly what you need. Consider this code, which finds the smallest factor of a number n greater than two:

```
int n = input.nextInt();
int factor = 2;
boolean found = false;

while (factor <= n && !found) {
    if (n % factor == 0) {
        found = true;
    } else {
        factor++;
    }
}
```

In the preceding code, the "test" happens in the middle of the loop. As a result, we used a flag variable and an `if-else` statement.

A simpler way to solve this problem is to use a `break` statement. When a program reaches a `break` statement, it exits the current loop.

```
int n = input.nextInt();

int factor = 2;
while (factor <= n) {
    if (n % factor == 0) {
        break;
    }
    factor++;
}
```

Though `break` is useful, it should be used sparingly. If you think through the condition for the loop carefully, you can often avoid `break` entirely, as in the following code:

```
int n = input.nextInt();

int factor = 2;
while (factor <= n && (n % factor != 0)) {
    factor++;
}
```

Whatever you do, avoid this sort of code:

```
while (true) {
    // process data;
    if (conditon) {
        break;
    }
    // more processing of data
}
```

This "infinite loop with a break" shouts out "I did not bother to carefully think about the exit condition for my loop!"

6.5 Exercises

Exercise 6.1 Write a program that asks the user for a starting amount of money, an annual interest rate as a percent, and a number of years. Use this information to print a table that shows the balance with accumulated compound interest. Use a `for` loop. Here is what your output might look like:

```
Enter starting amount: $100
Enter annual percent interest: 5.3
Enter number of years: 4
Year  Balance
0     $100.00
1     $105.30
2     $110.88
3     $116.76
4     $122.95
```

Exercise 6.2 When a business buys an asset such as a computer or cell phone, it *depreciates*, or loses its vaue over time. In this program, you will calculate the depreciation of an asset. Note: accountants have many different ways to calculate depreciation. This is *not* one of them! Your program will ask for:

- The original purchase price of the asset.
- The *salvage price*—how much you will get when you sell the asset after it is of no further use.
- The *depreciation rate*—the percentage by which the asset's value goes down every year.

The program will then print a table of the asset's current value until its value drops to or below the salvage price. It will then show the final value of the asset.

For example, let's say you purchase a computer for $1000 and will sell it for salvage for $100, and the depreciation rate is 20%. At the end of the first year,it will have lost $200 of value: 1000×0.20, so its value is now $800. At the end of the second year, it will lose 800×0.20, or $160 dollars of value, and the asset will now be worth $640, and so on.

Here is what the output for a purchase price of $1500, salvage price of $500, and depreciation rate of 20.5% might look like:

```
Enter asset price: $1500
Enter salvage price: $500
Enter depreciation rate as a percent: 20.5
Year   Value
   0  $1500.00
   1  $1192.50
   2  $948.04
   3  $753.69
   4  $599.18
Final value: $476.35
```

Exercise 6.3 When you go to the store, the clerk doesn't know in advance how many items you want to buy. Instead, they keep totaling the items until they see one of those plastic dividers to indicate that your order is complete.

Write a program that repeatedly asks the user for the price of an item until they enter a zero for the price. This is the digital equivalent of the plastic divider. Its technical name is a *sentintel value*. As the user enters prices, keep track of the total number of items and sum of the prices. After encountering the sentinel value, your program will print the number of items purchased, the subtotal, the tax (at a rate of 6.5%), and the grand total. Here is what the program might look like. Note that it does not allow (or count) negative prices.

Hint: Use a `while` loop. The exercise says "until they enter a zero..." The word "until" is the opposite of "while", so your loop needs to keep going `while` the entry is *not* zero.

```
Enter price, or 0 when finished: $3.50
Enter price, or 0 when finished: $-2
Prices can not be negative.
Enter price, or 0 when finished: $5.99
Enter price, or 0 when finished: $4.83
Enter price, or 0 when finished: $0

Number of items: 3
Subtotal:  $   14.32
Tax:       $    0.93
Total:     $   15.25
```

Exercise 6.4 This exercise will give you practice with a more complicated condition for a `while` loop. Write a program named *ComplexWhile.java* that lets people enter integers until they've entered five numbers or the sum of the entries is greater than 100, whichever comes first. Here's an example of the output from running the program twice:

```
Enter integer #1: 30
Enter integer #2: 49
Enter integer #3: 24
The sum of your entries is 103.

Enter integer #1: 13
Enter integer #2: 17
Enter integer #3: 20
Enter integer #4: 14
Enter integer #5: 18
You hit the five-entry limit.
The sum of your entries is 82.
```

Hint: You loop until there are five entries or the sum is greater than 100. That means the `while` condition must be "not (five entries or sum greater than 100)." Use De Morgan's laws to simplify this condition.

Exercise 6.5 Write a program that repeatedly asks the user to enter a sentence until they press only ENTER. For each sentence, tell how many vowels, consonants, digits, and "other" characters (anything that is not a letter or digit) are in the sentence. For this exercise, presume that the letter "y" is a

vowel. Continue asking for sentences until the user presses only the ENTER key for input. When that happens, the `String` that you read will equal the empty string `""`. You can test this condition to end the loop.

After exiting the loop, give grand totals for each category.

Sample output:

```
Input a sentence (just ENTER to quit): 4 score & 7 years ago.
Vowels:   7  Consonants:  6
Digits:   2  Others:      7

Input a sentence (just ENTER to quit): 2*x=17
Vowels:   0  Consonants:  1
Digits:   3  Others:      2

Input a sentence (just ENTER to quit): The Quick Brown Fox
Vowels:   5  Consonants: 11
Digits:   0  Others:      3

Input a sentence (just ENTER to quit):
----------
Totals
Vowels:  12  Consonants: 18
Digits:   5  Others:     12
```

Hint: Make `String` variables with contents such as `"aeiouy"`, `"bcdfghjklmnpqrstvwxz"`, etc. and use `indexOf` to determine whether a character belongs to that group of characters.

Exercise 6.6 Write a method called `switchOrder`, which takes a `String` parameter with a person's name in the form `"first middle last"` and returns the name in the form `"last, first middle"`. Your code should work for people with only a single name, people with no middle name, and people with several middle names. Use a `while` loop with the `indexOf` and `substring` methods to do this exercise. (You could do it more easily with the `lastIndexOf` method, but we want you to get practice using loops.)

Write a `main` method that uses a `while` or `do-while` loop to repeatedly ask the user for a name and then calls the `switchOrder` method and prints the result. Repeat until the user presses just ENTER. Here is some sample output:

```
Input a name (just ENTER to quit): Grace Murray Hopper
Hopper, Grace Murray

Input a name (just ENTER to quit): Donald Knuth
Knuth, Donald

Input a name (just ENTER to quit): Prince
Prince

Input a name (just ENTER to quit): C. Anthony Richard Hoare
Hoare, C. Anthony Richard

Input a name (just ENTER to quit):
```

Exercise 6.7 Write a program named *PhoneWord.java* that prompts the user for a "phone word," an alphabetic mnemonic for a phone number. Then, print out the phone number corresponding to that sequence.

Here is how your program must translate letters to numbers:

ABC	2
DEF	3
GHI	4
JKL	5
MNO	6
PQRS	7
TUV	8
WXYZ	9

Keep digits as digits (see the second example output). Don't forget about zero!

You must accept letters in either upper or lower case. If the phone word translates to more than seven digits, keep only the first seven. If the phone word translates to fewer than seven digits, print an error message. Ignore any characters other than a letter or digit.

Here is an example of several runs of the program:

```
Enter a phone word: warbler
The number is 9272537.

Enter a phone word: GOOD4U2
The number is 4663482.

Enter a phone word: OMG
Your phone word is not long enough for a phone number.

Enter a phone word: GREAT DEALS
The number is 4732833.

Enter a phone word: got-food?
The number is 4683663.
```

Hint: Do *not* check the input string to see if its length is greater than or equal to seven. The string `"C-A-T-S!"` is eight characters long, but there are only four letters, so it will not translate to a valid phone word.

Instead, convert all the letters in the string to digits, no matter how many or how few, and *then* check the length of the result to see if it is seven characters or more.

Extra challenge: Print the phone number with a hyphen; for example: 473-2833.

Exercise 6.8 Given an integer from 1 to 10 (inclusive), print a pattern of stars based on that number. Example:

```
Enter number of rows: 4
*
**
***
****
```

Exercise 6.9 Given an integer from 1 to 10 (inclusive), print an inverted triangle of plus signs based on that number. Example:

```
Enter number of rows: 6
++++++++++
 +++++++++
  +++++++
   +++++
    +++
     +
```

Hint: Before you start coding this program, plan it by writing down a table with the line number (starting at zero or one, but zero is probably better), the number of leading spaces you will need on the line, and the number of plus signs. Then figure out the mathematical relationships between the line number and the other two values. They will be linear relationships.