Chapter 8 Iterative Statements and `Strings`

Introduction

Repetition is part of life. Everyone is familiar with it. The sun comes up, the sun goes down. The sun comes up, the sun goes down. Spring turns to summer; fall arrives and students gather in classrooms -- over and over. Babies grow to children, mature to adults, have more babies, and die; for the last million years or so. Civilizations rise and fall; again and again. The galaxy turns, but very slowly to our eyes.

On Being Conscious

If parents are unthinking, not conscious of what they are doing, they tend to raise their children as they were raised. They do the things that were done to them as children to their own children. This is not always optimal. Similarly, teachers, if they have not reflected on the process of teaching and learning, may have their students do the same tedious, unproductive exercises they were forced to do. If the world, or the field has changed, this may not be the best plan. Mindless repetition is what computers do very well; it is better for people to choose their actions.

Imitation and Culture, or Monkey See, Monkey Do?

Researchers have shown that humans are much better at imitation than other great apes, even when it makes no sense. Here is a revealing experiment. The subjects (human and chimpanzee) watch a person trying to reach a banana from behind bars. It is out of reach, but there is a rake-like tool lying at hand. On one side, it has four tines, on the other two. The demonstrator picks it up and drags over the banana with the two tined side. Then the subject is put behind the bars, and a new banana is placed out of reach. Both people and chimpanzees use the rake to get the banana, but the people use the two tined side far more often than the chimps, even though it is much more difficult than the other side. The question is, why?

One hypothesis is the people are biased, unconsciously, towards doing things the way they have seen them done. Why might this be a species trait? One explanation is that this might lead to the emergence of language and culture. If one group of proto-humans tends to do things and express things the same way automatically, and another doesn't, the former tends to develop coherent language, which allows them to communicate better and gives them tremendous adaptive advantage. Same with culture in general. Whether or not this is true, it is an interesting speculation.

This Chapter

This chapter introduces Java statements that repeat. Together with conditional statements, these comprise the bulk of control structure in Java. Good control structure, coupled with well-designed class structure, can yield powerful and useful software.

Like the previous chapter, this one will be almost devoid of classes. It focuses on the mechanics of loops and `Strings`. Perhaps you can learn this material simply by reading the text, but it may help these constructs stick in memory if you type them into a program and run them (and they

are all essential to programming). This would also give you a chance to experiment with small changes to them; to play around with them. That's one of the best ways for most people to learn.

Iteration: Repeated Action

Iterate is another word for repeat. To make a program do something a number of times use an iterative statement. There are three iterative statements in Java; see BNF 8.1. This chapter will cover `while` and `for`.

BNF 8.1 Iterative Statement

<iterative stmt> ::= <while stmt> | <for stmt> | <do-while stmt>

The `while` Loop

Syntax and Semantics

The `while` loop is the simplest looping construct; its syntax and semantics are in BNF 8.2.

BNF 8.2 The `while` Statement

<while stmt> ::= while (<boolean expression>) <stmt>
Semantics: 1. Evaluate the <boolean expression> 2. If the value of the expression is true, a) execute the <stmt> after <expression> b) Go back to step 1.

Notice that the syntax is very much like an `if` statement (see BNF 7.1 "The `if` statement"). The difference in semantics is that after the <stmt> is executed, the <boolean expression> is evaluated again, and if it is still `true`, that process repeats, possibly forever. The programmer must remember to make sure that `while` loops are not infinite!

Example 1: Counting to Ten

Listing 8.1 shows a `while` loop that prints the numbers from one to ten.

Listing 8.1 Counting to Ten

```
1    int count=1;
2
3    while (count<=10) {
4        System.out.println("count=" + count);
5        count++;
6    } // while
```

Line 1: Declares an `int` variable named `count` and sets it to the value 1.

Lines 3-6: The `while` loop, which will run while `count` <= 10.

Line 4: Prints the value of `count` (along with a descriptive label).

Line 5: Adds one to `count`.

This loop runs over and over until the `boolean` expression `count<=10` evaluates to `false` at the top of the loop (it is only rechecked before the entire loop body executes). So, the execution of

the loop is controlled by the value of `count`. Thus, `count` is the *control variable* for this loop. In this case the control variable is also being used to display the count; sometimes the control variable only controls the number of times the loop executes. If you omit line 5, this will print `count=1`, forever! This is called an *infinite loop*.

One way to understand what a section of code does is analysis; simply look at it and see what it does. If that works, fine. If that doesn't work, one way to discover what it does is called *hand simulation*. To hand simulate code, you need paper and pencil. Write down the variables involved and step through the code, carrying out the semantics of each statement one by one, recording the values of the variables as you go. Table 8.1 shows the hand trace for Listing 8.1; make sure you understand it; then do it yourself. It is slow work, but sometimes it provides insight into the mysterious inner workings of code.

Table 8.1 Hand Simulation for Listing 8.1

line	**action**	**count**
1	declare `count`	1
3	`count`**`<=10`**? yes, do body	1
4	output `count`=1	1
5	`count`++	2
3	`count`**`<=`**`10`? yes, do body	2
4	output `count`=2	2
5	`count`**`++`**	3
3	`count<=10`? yes, do body	3
...	...	...
5	`count++`	10
3	`count<=10`? yes, do body	10
4	output `count`=10	10
5	`count++`	11
3	`count`<=10? no!, done!	11

Example 2: Doing Something Ten Times

Listing 8.2 is a generic loop to do something ten times. Because it is in a loop that goes around ten times, `something()` will be executed ten times. What that something is, is up to the programmer.

Listing 8.2 Doing Something Ten Times

```
1    int count=1;
2
3    while (count<=10) {
4        something();
5        count++;
6    } // while
```

For example, this loop can be used to output the squares and cubes of the numbers from one to ten, as shown in Listing 8.3, which uses Listing 8.2 as the body of `main()`.

Listing 8.3 Using That `while` Loop to Print a List of Squares and Cubes

```
1    public class TestIteration {
2
3        private static void sqAndCube(int x) {
4            System.out.println("x=" + x + "x^2=" + x*x + "x^3=" + x*x*x);
5        }
6
7        public static void main(String[] args) {
8            int count=1;
9            while (count <= 10) {
10               sqAndCube(count);
11               count++;
12           }
13       }
14   }
```

Line 3: The heading for `sqAndCube()`. Notice that it is a class method.

The control variable is passed as the value to square and cube each time. Notice that there are some formatting issues.

This code does what it is supposed to, but the output doesn't look very nice. Figure 8.1 has the output for this loop from my machine.

Figure 8.1

```
x=1x^2=1x^3=1
x=2x^2=4x^3=8
x=3x^2=9x^3=27
x=4x^2=16x^3=64
x=5x^2=25x^3=125
x=6x^2=36x^3=216
x=7x^2=49x^3=343
x=8x^2=64x^3=512
x=9x^2=81x^3=729
x=10x^2=100x^3=1000
```

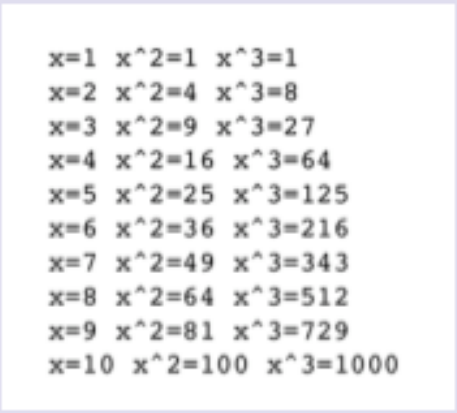
Output from Code Example 8.3 -- obviously in need of spaces.

Although there are spaces in the `println()`, there are none inside the ""s. That `println()` would create better looking output if it had a few spaces; like this:

```
System.out.println("x="+ x + " x^2=" + x*x + " x^3=" + x*x*x);
```

Figure 8.2 has the output with that change.

Figure 8.2



```
x=1 x^2=1 x^3=1
x=2 x^2=4 x^3=8
x=3 x^2=9 x^3=27
x=4 x^2=16 x^3=64
x=5 x^2=25 x^3=125
x=6 x^2=36 x^3=216
x=7 x^2=49 x^3=343
x=8 x^2=64 x^3=512
x=9 x^2=81 x^3=729
x=10 x^2=100 x^3=1000
```

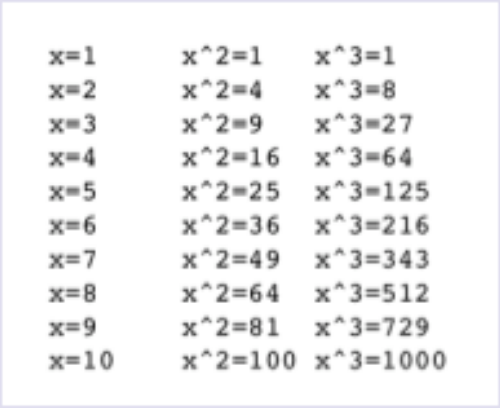
Slightly better output from Code Example 8.3 -- still in need of tabs.

This is better, but the different widths of the larger numbers rather wreck the readability. This is a good place to use the tab character (`\t`):

```
System.out.println("x="+x+ "\tx^2=" + x*x + "\tx^3=" + x*x*x);
```

Output from this appears in Figure 8.3 and looks rather better. The general problem of formatting text from Java will be postponed indefinitely.

Figure 8.3



```
x=1     x^2=1   x^3=1
x=2     x^2=4   x^3=8
x=3     x^2=9   x^3=27
x=4     x^2=16  x^3=64
x=5     x^2=25  x^3=125
x=6     x^2=36  x^3=216
x=7     x^2=49  x^3=343
x=8     x^2=64  x^3=512
x=9     x^2=81  x^3=729
x=10    x^2=100 x^3=1000
```

Output from Code Example 8.3 -- with tabs.

Example 3: Doing Something *n* Times

In the more generic loop in Listing 8.4, the control variable is compared to another variable, *N*, instead of 10.

Listing 8.4 Code A Generic `while` Loop -- Does `something()` *N* Times, Suitable for Memorization

```
1    static final int N=10;
2    int count=1;
3
4    while (count<=N) {
5        something();
6        count++;
7    } // while
```

This can make a big difference in a more complex situation. Changing a 10 to 1000 is easy, changing eight 10s to 1000s, less so; and what if you forget one of them?

Example 4: Counting to Ten -- Take Two

Listing 8.1 started the value of `count` at 1, like a person would. But, it is common `for` loops in programs to start from 0 instead, for reasons that will be seen below. Listing 8.5 shows the same loop counting from 0. There are three changes: `count` is initialized to 0, the expression is `(count<10)` instead of `(count<=10)` and lines 4 and 5 are exchanged. Convince yourself that this loop does the same thing as the other.

Listing 8.5 Counting to Ten -- Take Two

```
1    int count=0;
2
3    while (count<10) {
4        count++;
5        System.out.println("count=" + count);
6    } // while
```

This also prints the numbers 1-10; but using slightly different logic.

Infinite Loops

The `while` loop gives the programmer tremendous power. But with increased power comes increased capacity for error. If the control variable is never updated inside the loop, and the loop starts (i.e. the condition is true the first time), then it is always an infinite loop. If the control variable is not correctly updated, it may be an infinite loop.

The `for` Loop

Listing 8.6 does exactly what Listing 8.4 did, but uses a `for` loop instead of a `while` loop.

Listing 8.6 Doing Something Ten times with a Generic `for` Loop

```
1    static final int N=10
2
3    for (int count=1; count<=N; count++) {
4        something();
5    } // for
```

Notice that initialization, checking and incrementing the control variable, all happen in the first line of the `for`. This makes them much more difficult to forget.

It is a bit more compact than the equivalent `while` loop, and until you are used to it, more confusing. One young programmer who was familiar with `while` loops but not `for` loops refused to learn `for` loops because, “They are too complicated!”. This was a mistake.

Sometimes learning one thing makes another more difficult to learn. This is called proactive interference. The effect is particularly strong when you know one way to do something and someone wants you to learn another. The way you know (in that young programmer’s case, the `while` loop) seems simple and obvious; the new way difficult and confusing. When you try to solve a problem with a new technique, you immediately know how to solve it with the old technique (which interferes with applying the new technique). So you resist the new way. This is true at all scales; from iterative statements to programming languages and paradigms, operating systems, and even life-styles. But, change is good; if you stop learning, your life is essentially over. Oops, back to the `for` loop.

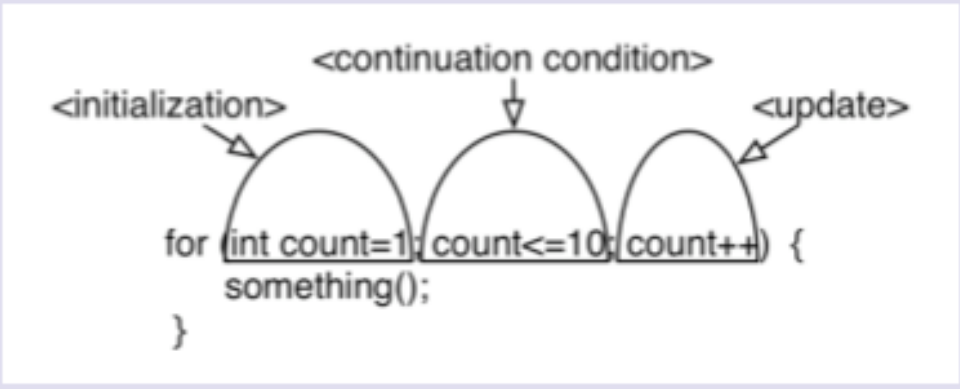
The syntax and semantics of a `for` loop is very similar to that of a `while` loop. As BNF 8.3 shows, it starts with the word for, then some things in ()s, then a single statement.

BNF 8.3 The `for` Statement

<for stmt> ::= for (<initialization>;<continuation condition>; <update>) <stmt>
Semantics: 1. Execute <initialization>. 2. If the value of the <continuation condition>. is true, a) execute the <stmt>. b) Execute <update>. c) Go back to step 2 (that's right, step **2**!).

There are three things in the ()s; the one in the middle, the continuation condition, is a `boolean` expression (see BNF 8.3), exactly like the `while` loop. The first thing, <initialization>, is done once, before anything (like initialization always is!). The third, <update>, is done after the body of the loop is executed (each time it is executed). See Figure 8.4 for an illustration.

Figure 8.4



For loop illustration.

BNF 8.4 `for` Loop Details

<initialization> ::= <stmt> <continuation condition> ::= <boolean expression> <update> ::= <stmt>
Both <initialization> and <update> are simply statements, the <continuation condition> is simply a `boolean` expression

The big advantage of a `for` loop is that the initialization and update are included in the statement. That way they are easy to find and hard to omit accidently.

Programming Example

Here is a slightly more complicated example.

Task:
Write a program that will count to 1000 by twos, displaying the count to `System.out`.

Several methods might be employed to accomplish this task. First, since we already have a loop that counts to 10 by ones, we could simply change the 10 to 1000 and only print the even numbers, as in Listing 8.7.

Listing 8.7 Counting to 1000 by Twos -- Method One

```
static final int N=1000;
 ...
static void countToTen() {
    for (int count=1; count<=N; count++) {
        if (count is even)
            System.out.println("" + count);
    } // for
 }
```

The `for` loop iterates 1000 times, with `count` going from 1 to 1000. Only even numbers print.

Notice here that the first part of the `String` parameter is "". That way it doesn't say "count=" every time. With some compilers, if you omit that and just write `System.out.println(count);` it will generate a compiler error complaining that an `int` is not a `String` (recall "`int` to `String`" in Chapter 5, "Towards Consistent Classes.")

How can we determine if `count` is even? Here's a hint: even numbers are evenly divisible by 2. Need another hint? The `%` operator yields the remainder after `int` division. One more hint? If a number divides another evenly, the remainder is zero. Okay?

A second approach to this problem is to change the update so that each time around the loop it adds 2 to `count` instead of 1. I.e. change the `count++` to `count=count+2`; -- see Listing 8.8.

Listing 8.8 Counting to 1000 by Twos -- Method Two

```
static final int N=1000;
 ...
static void countToTen() {
    for (int count=2; count<=N; count=count+2) {
        System.out.println("" + count);
    } // for
 }
```

By changing both the initialization and the update, we get clean simple code.

Third, we could just count from 1 to 500 and print twice the count each time -- see Listing 8.9, where `count` goes from 1-500, and `count*2` is printed each time around the loop.

Listing 8.9 Counting to 1000 by Twos -- Method Three

```
static final int N=500;
 ...
static void countToTen() {
    for (int count=1; count<=N; count++) {
        System.out.println("" + count*2);
    } // for
 }
```

Which of these methods would be better? It's a matter of style.

Cleaning Up the Output

If you run that code, it produces 500 lines of output. That's excessive. it would be nicer if more than one number were printed on each line. The `println()` method ends the line after it outputs its parameter. There is another method, `print()`, that does not end the line. Thus, the simple solution would seem to be to send `print()` instead of `println()` to `System.out`; do you know what would go wrong in that case?

Using `print()`, all the numbers are concatenated, without any spaces, on one line. The beginning of the output is shown in Figure 8.5

Figure 8.5

```
246810121416182022242628303234363840424446485052545658606264666870727476788
0828486889092949698100102104
```

Using print() instead of println() produces more compact output. Possibly too compact!

That's two problems. The first is very easy to solve; see Listing 8.10.

Listing 8.10 Counting to 1000 by Twos -- On One Line

```
1    for (int count=2; count<=N; count=count+2) {
2        System.out.print(" " + count);
3    } // for
```

Notice the `print()` instead of `println()`. The space in the ""s puts spaces between the numbers.

The second problem is more complicated.

Printing Ten Numbers Per Line

One idea would be to print a fixed number of numbers on each line; say ten. Pseudo-code for this is shown in Listing 8.11.

Listing 8.11 Pseudocode for Printing Ten numbers on Each Line

```
1    for (int count=2; count<=N; count=count+2) {
2        System.out.print(" " + count);
3        if (have printed 10 on this line)
4            System.out.println();  // go to next line
5    } // for
```

How can we tell (in Line 3) if ten numbers have been printed on this line? Here's two possibilities: 1) add a counter; initialize it to 0, and each time we print a number, add one; when we do the `println()`, reset it to 0. Or, 2) Figure out some clever way to discern when ten numbers have been printed. The former is rather more code, but more general; the latter is, well, clever, but likely to be hard to generalize (i.e. to use in another context).

Adding Another Counter

With a counter, the pseudocode appears in Listing 8.12.

Listing 8.12 Pseudocode for Printing Ten Numbers on Each Line Using a Counter

```
1    for (int count=2; count<=N; count=count+2) {
2        System.out.print(" " + count);
3        wordCount++;     // increment wordCount
4        if (wordCount == 10) {
5            System.out.println();  // go to next line
6            reset wordCount to 0;
7        } // if
8    } // for
```

Of course, the variable `wordCount` must be declared and initialized to 0 before the loop. This produces the output shown in Figure 8.6.

Figure 8.6

```
2 4 6 8 10 12 14 16 18 20
22 24 26 28 30 32 34 36 38 40
42 44 46 48 50 52 54 56 58 60
62 64 66 68 70 72 74 76 78 80
82 84 86 88 90 92 94 96 98 100
102 104 106 108 110 112 114 116 118 120
122 124 126 128 130 132 134 136 138 140
```

Ten even numbers on each line; too bad the alignment is so bad.

A Clever Trick

If there were some property shared by all the final numbers on a line, the code could check that property instead of keeping a counter to tell when it is time to end the line. Looking at that output, can you see anything unique about the last number on each line? Right, they are all

multiples of 20. So, to tell if it is time to end the line we could just check if `count%20==0`, as in Listing 8.13.

Listing 8.13 Pseudocode for Printing Ten Numbers on Each Line Using a Trick

```
1    for (int count=2; count<=N; count=count+2) {
2        System.out.print(" " + count);
3        if (count % 20 == 0)
4            System.out.println();  // go to next line
5    } // for
```

This kind of coding trick is fun, but can lead to disaster if later the code must be modified. Nonetheless, it is useful to remember.

The Empty Statement

Here's an unusual statement, the empty statement. BNF 8.5 shows its syntax and semantics. It consists of nothing and does nothing.

BNF 8.5 The Empty Statement

<empty stmt> ::=
Semantics: Does nothing.

What good is it? It allows the programmer to omit a statement in a place where syntactically a statement must appear and still match the grammar. In other words it allows the programmer to get around the rigidity of the compiler, which can be a very good thing.

An Infinite `For` Loop

The format of a `for` loop makes it difficult to forget to update the control variable and so, one is less likely to write an infinite `for` loop. But, sometimes you want to write an infinite `for` loop (as you will see in the next chapter). Listing 8.14 shows how to send the `something()` message forever. Note that the `{}`'s are not necessary.

Listing 8.14 Doing `something()` Forever

```
1    for (;;) {
2        something();
3    } // for
```

The line `something();` is a single message statement, so Listing 8.15 is legal as well.

Listing 8.15 Listing 8.14 Without the `{}`s

```
1    for (;;)      // forever!
2        something();
```

The empty statement sometimes causes nasty bugs (yet another example of the fact that most things have two sides). Can you see what is wrong with the code in Listing 8.16?

Listing 8.16 An Infinite Loop that Doesn't do Anything -- Forever!

```
1    for (;;);    // forever!
```

```
2         anything();
```

If you don't see it immediately, try matching the symbols one by one with the BNF 8.3. It starts like this: for matches for, (matches (, absolutely nothing matches the empty statement, which is a <statement> which is an <initialization>...

If you ran this code it would never do anything, but unlike the empty statement, which does nothing immediately, it would do nothing, forever!

`Strings`: A Very Brief Introduction

The Java `String` class is used to store literal sequences of characters. There are many methods that operate on `Strings`; you can read about them in the documentation; it's about time you got used to reading Sun's API documentation. This section will introduce a bare minimum of `String` methods with the signatures: `int length()`, `char charAt(int)`, `boolean equals(String)`, and `String toUpperCase()`. Notice that these four methods return `int`, `char`, `boolean` and `String` values, respectively. Then it will present a better way of breaking lines of even numbers.

A Few `String` Methods

Here are several `String` methods that will get you started using `Strings`.

The `int length()` Method

As you might guess, if you send `length()` to a `String`, it will return its length. So the code:

```
String s = "hello";
System.out.println("s=" + s + " s.length()=" + s.length());
```

will print `s=hello s.length()=5`. No surprises.

`char charAt(int)`

A `String` is a series of characters. The Java type for characters is `char`; each `char` holds a single character. Character constants have single quotes around them, like `'a'` or `'$'`. The first character in a `String` is at location 0. That's zero, not one, but *zero*. Forgetting this will cause little annoying bugs. Rather like mosquitoes. Not really harmful, but pesky. The reason the first `char` is at zero is that `Strings` are implemented as arrays, which are coming up in a few chapters. Arrays in Java, like in C++ and C, start at zero. No way around it. Deal with it. Accept it. Remember it!

With `String s="hello";` the following are true:

`s.charAt(0)` returns `'h'`
`s.charAt(1)` returns `'e'`
`s.charAt(2)` returns `'l'`
`s.charAt(3)` returns `'l'`
`s.charAt(4)` returns `'o'`

You could write a loop to produce those five lines. Pseudocode for it is shown in Listing 8.17.

Listing 8.17 Pseudocode to Print All the `chars` in a `String`

```
1    String s="Howdy!";
2    for (each char in s) {
3        output the next char with suitable description
4    } // for
```

This is a pattern for many methods that process `Strings`. If you wanted to become expert at Java programming it would be worth committing to memory; both the pseudocode and the code. Your choice.

When you wish to access each of the `char`s in a `String` from first to last in order, you use the idiom shown in Listing 8.18.

Listing 8.18 Printing the `chars` in a `String` One Per Line

```
1    String s="Howdy!";
2    for (int i=0; i<s.length(); i++) {
3        System.out.println("charAt(" + i + ") is: '" + s.charAt(i) + "'");
4    } // for
```

Notice the various parts of the `String` parameter sent to `System.out.println()` that produce the readable output.

Line 2: The generic `for` loop you write to access the `char`s in the `String`, `s`, sequentially. In the body of the loop, `s.charAt(i)` is each `char`, one after another each time around the loop. In pseudocode `s.charAt(i)` is "the next `char`." The output of Listing 8.18 is shown in Figure 8.7.

Figure 8.7

```
charAt(0) is: 'H'
charAt(1) is: 'o'
charAt(2) is: 'w'
charAt(3) is: 'd'
charAt(4) is: 'y'
charAt(5) is: '!'
```

Output from code in Code Example 8.18

Notice how the `String` "`charAt(0) is: `" is constructed in the `println()` on line 3 in Listing 8.18. It is inconvenient, but highly informative to output such descriptive text along with the data one is outputting. If you are printing `chars`, the delimiters can be surprisingly important. The difference between printing nothing and printing an invisible character is important, but invisible. The single quotes bracketing the `char` let the reader see this difference.

The `boolean equals(String)` Method

You cannot compare `Strings` with ==. Instead you must use `equals()`. Worse, if you forget, and use ==, no error message will appear, because Java will compare the memory locations of the two `Strings`. Not what you expected, or wanted. So, if you wanted to know if the same text is in two `TextFields`, you must write code like:

```
String s1 = aTF.getText();
String s2 = bTF.getText();
if (s1.equals(s2))
    "yes! they are the same";
else "no. not equal"
```

`String toUpperCase()`

This method does what you might expect, it makes an uppercase copy of the `String` it is sent to and returns that. So:

```
String s = "abcDEFghi";
System.out.println(s.toUpperCase());
```

would output: `ABCDEFGHI`.

There are many useful methods that you should learn if you were going to do any extensive programming with `Strings`, including: `substring()`, `replaceAll()`, and `subSequence()`.

Breaking Lines Using `Strings`

The technique to print ten even numbers per line in Listing 8.12 worked, but it was rigid and produced somewhat ugly output. This example will do the same job more generally and elegantly. It will count to 1000 by twos and print as many numbers on each line as will fit.

Decoupling the Output

An important step to writing elegant, general code is to decouple the logic and the output. This is true of all code. Input and output are inherently messy and idiosyncratic. A standard technique is to consolidate output in a method call `emit()`, as shown in Listing 8.19.

Listing 8.19 Using `emit()` to Decouple the Output

```
1    static void countTo_N_Improved() {
2        for (int count=2; count<=N; count=count+2) {
3            emit(" " + count);
4        } // for
5    }
```

Each `String` to be output is sent to `emit()` which handles the formatting and output. This way the logic of the loop is kept simple and whatever needs to be done with output `Strings` happens in one place, `emit()`.

Buffers

An area of working storage is called a *buffer*. There is a built-in Java class called `StringBuffer`, that some people would argue should be used here. But, that would be one more class to learn,

and your task now is to understand the basics of `Strings`. If you were going to be doing extensive work with buffering `Strings`, definitely look up `StringBuffer`!

The job of `emit()` here is to put as many output `Strings` on each line as will fit. It will accomplish this with an output buffer. The buffer will start empty. Whenever a `String` is sent to `emit()`, if it fits in the buffer, it will be concatenated onto the end. If it overfills the buffer two things will happen: first the buffer will be output (or flushed, as is sometimes said); second the `String` will be added to the now empty buffer as the first thing on the next line. Pseudocode for `emit()` appears in Listing 8.20.

Listing 8.20 Pseudocode for `emit()` -- First Try

```
1    static void emit(String nextChunk) {
2        if (nextChunk would overfill the buffer) {
3            flush buffer
4            add nextChunk to the buffer
5        }
6        else add nextChunk to the buffer
7    }
```

Notice that lines 4 and 6 are identical. When the same thing is done at the end of both the `if` and else parts of an `if-else` you can do what is called *bottom factoring*.

Bottom Factoring
Since either the `if` part or the `else` part of an `if-else` is bound to execute, when the last thing in both is the same, it is always the last thing done before the next statement. Thus, it can be moved after the `if-else`. This is illustrated in Listing 8.21.

Listing 8.21 Pseudocode for `emit()` -- After Bottom Factoring

```
1    static void emit(String nextChunk) {
2        if (nextChunk would overfill the buffer)
3            flush buffer
4        add nextChunk to the buffer
5    }
```

Since the next chunk is always added to the buffer, this code does the same thing as Listing 8.20. It is simpler and easier to read.

Having thought through the logic and written detailed pseudocode, it is fairly easy to write the actual code; see Listing 8.22.

Listing 8.22 Breaking Lines at 70 `chars` -- A Complete Program

```
1    public class TestIteration {
2
3        public static void main(String[] args) {
4            countTo_N_Improved();
5        }
6
7         private final static int MAX_LINE_LENGTH=70;
8         private static String buffer = "";
9         private static void emit(String nextChunk) {
```

```
10              if (buffer.length() + nextChunk.length() > MAX_LINE_LENGTH) {
11                  System.out.println(buffer);
12                  buffer = "";  // empty the buffer
13              }
14              buffer += nextChunk;
15          }
16
17          private static final int N=1000;
18          private static void countTo_N_Improved() {
19              for (int count=2; count<=N; count=count+2) {
20                  emit(" " + count);
21              } // for
22          }
23      }
```

Line 8: Declares and initializes the buffer.

Line 9: Method heading for `emit()`. Notice that this method is `static`, like `countTo_N_Improved()`. These methods are not instance methods, but class methods. Thus they cannot access instance variable (and there are none in this class. They also cannot be invoked unless messages are sent to the `TestIteration` class.

Line 10: Checks if adding this next chunk would overfill it. If so; it must be flushed.

Line 11: Outputs it.

line 12: Empties the buffer.

Line 14: Always adds the new output to the end of the buffer.

Make sure you understand how this listing implements the pseudocode; these are standard techniques you will need to know.

The reason static methods are appropriate there is that there will never be multiple instances of a `TestIteration`; thus it is simpler to make them class methods.

The output using this line breaking scheme is shown in Figure 8.8. It is much better balanced and adaptable than the original. This technique could be easily adapted to a word processing application.

Figure 8.8

```
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94
96 98 100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130
132 134 136 138 140 142 144 146 148 150 152 154 156 158 160 162 164
166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198
```

Output from code in Code Example 8.18

Palindromes

A palindrome is a word that is the same backwards and forwards; like “noon” or “racecar”. This section will develop a program that inputs a `String` and reports whether it is a palindromic. Palindromic, here, means a `String` would be a palindrome if we neglected case, spaces and punctuation. A famous palindromic sentence is: “A man, a plan, a canal -- Panama!”. This and many others are at various websites, including:

http://www.derf.net/palindromes/old.palindrome.html

Design: Two Approaches

There is usually more than one way to approach a problem. Sometimes the first approach you think of is the best way to handle a problem, but other times a fresh perspective will yield better results. If you conceptualize two different approaches before beginning to implement either, you reduce the likelihood of frustrating missteps; and if you do start off on a path that turns out to be rocky, you have an alternative in the wings. To illustrate this, two approaches to this problem will be presented.

Approach Number One: Converging Pointers

Formulating an algorithm to decide if a `String` is palindromic is not entirely trivial. There may be spaces, punctuation and lower or uppercase letters at any position in it. So, it makes sense to solve a subset of the problem.

A simpler version of this problem is deciding if a single word with all lowercase letters is a palindrome. This is a sensible place to start thinking about this problem. Could you solve this simpler problem? If so, you could likely skip ahead to the next section. If not, here's a way to get started.

One technique for designing an algorithm to convert into a program to solve some problem is to solve a simple example of that problem yourself. If you keep track of what you did to solve it, then that technique can be cast as pseudocode and then converted to actual code.

Problem Solving Technique: *How would you do it without a computer?*
If you can't think of an algorithm for a problem, start solving it yourself and then convert the technique you would use into an algorithm.

One way a human might check if a word were a palindrome is to point one finger at the first letter and another at the last letter. If those two letters are the same, move the fingers towards each other; if they are different, the word is not a palindrome and you stop. Otherwise, continue this procedure until the fingers cross.

Most people point with their first finger; that's why it is called the index finger. Figure 8.9 shows the initial state of the algorithm for the two strings “racecar” and “dogfood”.

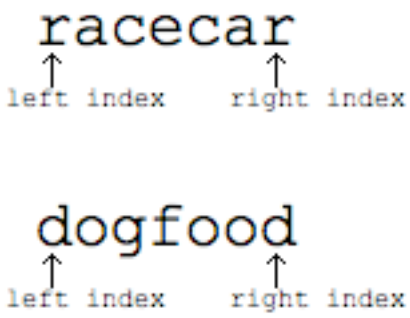


Figure 8.9 Initially the pointers are at the first and last letters.

Since `'r'=='r'` and `'d'=='d'`, both sets of pointers will next move towards each other. At that next step, shown in Figure 8.10, both `'a'=='a'`, in “racecar”, `and 'o'=='o'` in “dogfood”.

Figure 8.10 Since the first and last letters of each word are the same, the pointers move towards each other by one position.

So, again, both words may still be palindromes, and both sets of pointers move towards each other. Then, as shown in Figure 8.11, `'c'=='c'`, but `'g' != 'o'`, so we know “dogfood” is not a palindrome.

Figure 8.11 In “racecar” the pointers still point at identical letters, so it may still be a palindrome, but 'g' is not 'o', so “dogfood” cannot be a palindrome.

Continuing, when the indices are advanced once more in “racecar”, see Figure 8.12, they are both pointed at the `'e'` in the middle. Since the fingers have run into each other, “racecar” is judged to be a palindrome.

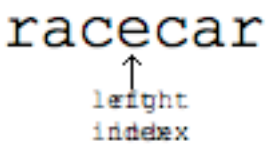


Figure 8.12 The two pointers meet without ever pointing to different letters, so, the word must be a palindrome.

Pseudocode for `isPalindrome(String)`

Having thought carefully about the algorithm to decide if a `String` is a palindrome, we are ready to write pseudocode for it.

```
set leftIndex to the start of the String
set rightIndex to the end of the String

while (pointers not crossed and letters at pointers are ==) {
    move left pointer right 1 letter
    move right pointer left 1 letter
}

if (pointers are crossed)
    the String is a palindrome
else it is not
```

Java Code for `isPalindrome(String)`

With the pseudocode in hand, the Java code is straightforward to write, assuming you recall several details.

1. You can access a particular character in a `String` using `charAt(int)`
2. The first character is `charAt(0)`

3. You can learn the length of a `String` by `length()`
4. Thus, the last char is `charAt(length()-1),` since the first index in a `String` is 0.
5. You can increment an `int` with `++` and decrement it with --
6. Methods with non-`void` return types must return values
7. When you exit a `while` loop, you know the condition is `false`; therefore the `return` statement returns `true` (meaning it is a palindrome) just in case the pointers have crossed -- if they have not, it must be the case that two letters the same distance from the ends of the `String` were `!=`, since otherwise you would still be in the `while` loop.

The code for the `isPalindrome()` method appears in Listing 8.23.

Listing 8.23 The `isPalindrome()` Method

```
 1    boolean isPalindrome(String s) {
 2        int leftIndex = 0;
 3        int rightIndex = s.length()-1;
 4
 5        while (leftIndex < rightIndex
 6                  && s.charAt(leftIndex) == s.charAt(rightIndex)) {
 7            leftIndex++;
 8            rightIndex--;
 9        }
10
11        return leftIndex >= rightIndex;
12    }
```

Adjourn to the screen now; type in this method and test it. The simplest way is to use a `JFrame` with two `TextFields`; one for input, one for output. Remember that if you are trying to learn to program, instead of just transcribing the code, a good technique is to recreate it from memory; this will force you to internalize the various language and logic features. So, study the code in Listing 8.23, and try to type it in without looking. If you have to peek, that's okay, but don't just copy it!

Converting `isPalindrome(String)` to `isPalindromic(String)`

Having solved the simpler problem, we are ready to adapt that solution to the more difficult problem. One's first idea of how to convert `isPalindrome(String)` to `isPalindromic(String)` might be to modify it to handle the more complex case. I.e., the basic structure of the algorithm is the same, but each time around the loop, instead of just incrementing one pointer and decrementing the other, we must move them until they are both pointing to a letter (as opposed to a space, or punctuation).

There is a method in the `Character` class, `isLetter(char),` which tells if a `char` is a letter. `Character.isLetter('a')` returns `true`, whereas `Character.isLetter('8')` returns `false`.

Thus, the code might be rewritten as in Listing 8.24.

Listing 8.24 The `isPalindromic(String)` Method: Take One

```
 1    boolean isPalindromic(String s) {
 2        int leftIndex = 0;
 3        int rightIndex = s.length()-1;
 4
 5        while (leftIndex < rightIndex
 6                  && s.charAt(leftIndex) == s.charAt(rightIndex)) {
 7            leftIndex++;
 8            rightIndex--;
 9            while (!Character.isLetter(s.charAt(leftIndex)))
10                leftIndex++;
11            while (!Character.isLetter(s.charAt(rightIndex)))
12                rightIndex--;
13        }
14
15    return leftIndex >= rightIndex;
16  }
```

Lines 9-10: Move the `leftIndex` right one position at a time until it is pointed at a letter.

Lines 11-12: Move the `righttIndex` left one position at a time until it is pointed at a letter.

This would work, so long as all the letters were either upper or lowercase, but if you compare `'a'` with `'A'` they are not equal. The other problem is that there are two `while` loops inside a third. This is not necessarily wrong, but it is rather complicated, and complicated code leads to mysterious bugs.

There is a method to convert all the letters in a `String` to lowercase; it is called `toLowerCase()`. So, if you convert the `String` to lowercase before starting the loop, that problem is avoided. I.e., insert the line:

```
s = s.toLowerCase();
```

as the first line in the method.

Approach Number Two: Filters

There are two ways to reuse classes, inheritance and composition (recall Chapter 6, "Software Reuse"). Similarly, there are two ways to reuse a method. The first way is to modify it (as shown above); the second is to invoke it in the context of another method. That conversion of the `String` to lowercase is the first step in this second approach to the palindromic algorithm. We already have a simple method that tells if a `String` is a palindrome, assuming it only contains lowercase letters. If we could transform the `String` containing letters of both cases and non-letters (spaces, and punctuation), into one with only lowercase letters, we could then use that method to test the result. The pseudocode looks like:

```
convert to all lowercase
remove all non-letters
use palindrome() to determine if the remaining letters are a palindrome
```

This transduction is shown in Figure 8.13.

A man, a plan a canal -- Panama!

...becomes...

a man, a plan, a canal -- panama!

...becomes...

amanaplanacanalpanama

Figure 8.13 The filtering method for deciding palindromic.

The only part of that pseudocode that needs explication is `remove all non-letters`. What is required is to look at each `char` in the `String` and filter out everything except the letters. Thus, logically, we must:

```
create a String to hold the filtered input
for each char in s
    if (it is a letter)
        add it to the filtered String
```

As you may recall (see Listing 8.18), there is an idiom for accessing each `char` in a `String`, one at a time from first to last. Using that idiom we obtain Listing 8.25.

Listing 8.25 The `isPalindromic()` Method as a Filter

```
 1    boolean isPalindromic(String s) {
 2        s = s.toLowerCase();
 3        String filteredS = "";
 4
 5        for (int i=0; i<s.length(); i++)
 6            if (Character.isLetter(s.charAt(i)))
 7                filteredS += s.charAt(i);
 8
 9        return isPalindrome(filteredS);
10    }
```

Line 2: Convert to lowercase
Line 3: Create the filtered `String`
Lines 5-7: Filter out all non-letters
Line 5: For each `char` in `s`
Line 6: If it is a letter
Line 7: Add it to the end of the filtered `String`
Line 9: Return whatever `isPalindrome()` returns for the filtered `String`.

Which of these two techniques is better? It is difficult to say, but the latter method is certainly simpler; and simplicity is a virtue.

Conclusion

This chapter introduced loops and `Strings`. Once you are familiar with classes, methods, instances, `Strings`, `chars`, conditional and iterative statements, you have the skills to solve a

great number of problems. The next three chapters will introduce `StringTokenizer`, file I/O, and lists (both arrays and `Vectors`). Those are the last new topics in this text.

Review Questions

1. What is the difference between a `while` loop and a `for` loop syntactically?
2. What can you do with a `while` loop that you can't do with a `for` loop?
3. If `String s="stuff";` what is `s.charAt(1)`?
4. What is a buffer?
5. What is bottom factoring?
6. Hand simulate Listing 8.5.
7. Why is it a good idea to come up with two ways to approach a problem before beginning implementation?

Programming Exercises

1. Write and run the three methods for counting to 1000 by twos.
2. Write a method, `String reverse(String)`, that returns its parameter backwards.
3. Write a method `int countEs(String)`, that returns the number of `'e'`s in the parameter.
4. Write a method `int countVowels(String)`, that returns the number of vowels in the parameter.
5. Use `reverse()` to write a very simple method, `boolean palindrome(String)`, that returns `true` just if its parameter is the same backwards as forwards. E.g. "noon," "madam" and "aha" are palindromes.
6. Extend the previous method to handle spaces, punctuation and capital letters. E.g. "A man, a plan, a canal. Panama!" is a palindrome. Hint: use a filtering scheme. First make everything uppercase (`toUpperCase()`), then remove punctuation and spaces (`Character.isLetter(char)`), then use the method from above.
7. Write a method `boolean anagram(String, String)`, that returns `true` just if its parameters are anagrams. Hint: this is not simple! It will take careful design. You might want to ask you instructor how to go about it.