Chapter 6
Repetition

Goals

This chapter introduces the third major control structure—repetition (sequential and selection being the first two). Repetition is discussed within the context of two general algorithmic patterns—the determinate loop and the indeterminate loop. Repetitive control allows for execution of some actions either a specified, predetermined number of times or until some event occurs to terminate the repetition. After studying this chapter, you will be able to

- Use the Determinate Loop pattern to execute a set of statements until an event occurs to stop.
- Use the Indeterminate Loop pattern to execute a set of statements a predetermined number of times
- design loops

6.1 Repetition

Repetition refers to the repeated execution of a set of statements. Repetition occurs naturally in non-computer algorithms such as these:

- For every name on the attendance roster, call the name. Write a checkmark if present.
- Practice the fundamentals of a sport
- Add the flour ¼-cup at a time, whipping until smooth.

Repetition is also used to express algorithms intended for computer implementation. If something can be done once, it can be done repeatedly. The following examples have computer-based applications:

- Process any number of customers at an automated teller machine (ATM)
- Continuously accept hotel reservations and cancellations
- While there are more fast-food items, sum the price of each item
- Compute the course grade for every student in a class
- Microwave the food until either the timer reaches 0, the cancel button is pressed, or the door opens

Many jobs once performed by hand are now accomplished by computers at a much faster rate. Think of a payroll department that has the job of producing employee paychecks. With only a few employees, this task could certainly be done by hand. However, with several thousand employees, a very large payroll department would be necessary to compute and generate that many paychecks by hand in a timely fashion. Other situations requiring repetition include, but are certainly not limited to, finding an average, searching through a collection of objects for a particular item, alphabetizing a list of names, and processing all of the data in a file.

The Determinate Loop Pattern

Without the selection control structures of the preceding chapter, computers are little more than nonprogrammable calculators. Selection control makes computers more adaptable to varying situations. However, what makes computers powerful is their ability to repeat the same actions accurately and very quickly. Two algorithmic patterns emerge. The first involves performing some action a specific, predetermined

(known in advance) number of times. For example, to find the average of 142 test grades, you would repeat a set of statements exactly 142 times. To pay 89 employees, you would repeat a set of statements 89 times. To produce grade reports for 32,675 students, you would repeat a set of statements 32,675 times. There is a pattern here.

In each of these examples, a program requires that the exact number of repetitions be determined somehow. The number of times the process should be repeated must be established before the loop begins to execute. You shouldn't be off by one. Predetermining the number of repetitions and then executing some appropriate set of statements precisely a predetermined number of times is referred to here as the **Determinate Loop pattern**.

Algorithmic Pattern: Determinate Loop

Pattern: Determinate Loop

Problem: Do something exactly *n* times, where *n* is known in advance.

Outline: Determine *n* as the number of times to repeat the actions
Set a counter to 1
While counter <= *n*, do the following
  Execute the actions to be repeated

Code Example:

```
// Print the integers from 1 through n inclusive
int counter = 1;
int n = 5;
while (counter <= n) {
  System.out.println(counter);
  counter = counter + 1;
}
```

The Java `while` statement can be used when a determinate loop is needed.

General Form: `while` statement

while (*loop-test***) {**
   repeated-part
}

Example

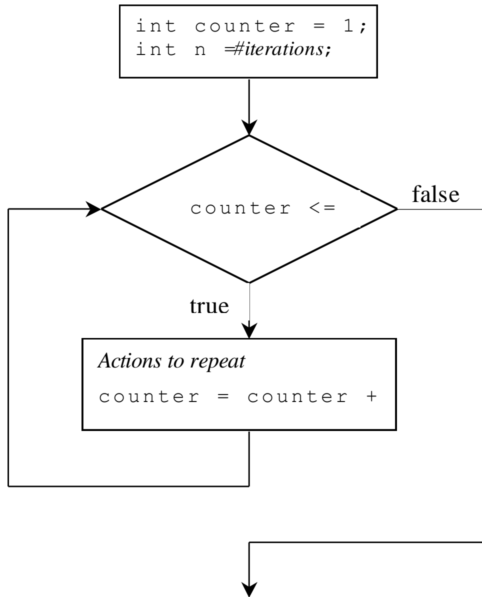
```
int start = 1;
int end = 6;
while (start < end) {
  System.out.println(start + " " + end);
  start = start + 1;
  end = end - 1;
}
```

Output

```
1 6
2 5
3 4
```

The *loop-test* is a `boolean` expression that evaluates to either true or false. The *repeated-part* may be any Java statement, but it is usually a set of statements enclosed in `{` and `}`.
When a `while` loop is encountered, the loop test evaluates to either true or false. If true, the repeated part executes. This process continues while (as long as) the loop test is true.

Flow Chart View of one Indeterminate Loop



To implement the Determinate Loop Pattern you can use some `int` variable—named `n` here—to represent how often the actions must repeat. However, other appropriate variable names are certainly allowed, such as `numberOfEmployees`. The first thing to do is determine the number of repetitions somehow. Let `n` represent the number of repetitions.

n = number of repetitions

The number of repetitions may be input, as in `int n = keyboard.nextInt();` or `n` may be established at compiletime, as in `int n = 124;` or `n` may be passed as an argument to a method as shown in the following method heading.

```
// Return the sum of the first n integers.
// Precondition: n >= 0
public int sumOfNInts(int n)
```

The method call `sumOfNInts(4)` should return the sum of all positive integers from 1 through 4 inclusive or 1 + 2 + 3 +4 = 10. The following test method shows four other expected values with different values for n.

```
@Test public void testSumOfNInts() {
  assertEquals(0, sumOfNInts(0));
  assertEquals(1, sumOfNInts(1));
  assertEquals(3, sumOfNInts(2));
  assertEquals(1 + 2 + 3 + 4 + 5 + 6 + 7, sumOfNInts(7));
}
```

Once `n` is known, another `int` variable, named `counter` in the `sumOfNInts` method below, helps control the number of loop iterations.

```
// Return the sum of the first n integers
public int sumOfNInts(int n) {
  int result = 0;
  int counter = 1;
  // Add counter to result as it changes from 1 through n
  while (counter <= n) {
    result = result + counter;
    counter = counter + 1;
  }
  return result;
}
```

The action to be repeated is incrementing result by the value of counter as it progresses from 1 through n. Incrementing `counter` at each loop iteration gets the loop one step closer to termination.

Determinate Loop with Strings

Sometimes an object carries information to determine the number of iterations to accomplish the task. Such is the case with `String` objects. Consider `numSpaces(String)` that returns the number of spaces in the String argument. The following assertions must pass

```
@Test public void testNumSpaces() {
  assertEquals(0, numSpaces(""));
  assertEquals(2, numSpaces(" a "));
  assertEquals(7, numSpaces("  a  bc   "));
  assertEquals(0, numSpaces("abc"));
}
```

The solution employs the determinate lop pattern to look at each and every character in the String. In this case, `str.length()` represents the number of loop iterations. However, since the characters in a string are indexed from 0 through its length() – 1, index begins at 0.

```
// Return the number of spaces found in str.
public int numSpaces(String str) {
  int result = 0;
  int index = 0;
  while (index < str.length()) {
    if (str.charAt(index) == ' ')
      result = result +1;
    index++;
  }
  return result;
}
```

Infinite Loops

It is possible that a loop may never execute, not even once. It is also possible that a `while` loop never terminates. Consider the following `while` loop that potentially continues to execute until external forces are applied such as terminating the program, turning off the computer or having a power outage. This is an infinite loop, something that is usually undesirable.

```
// Print the integers from 1 through n inclusive
int counter = 1;
int n = 5;
while (counter <= n) {
  System.out.println(counter);
}
```

The loop repeats virtually forever. The termination condition can never be reached. The loop test is always true because there is no statement in the repeated part that brings the loop closer to the termination condition. It should increment counter so it eventually becomes greater than to make the loop test is false. When writing `while` loops, make sure the loop test eventually becomes false.

Self-Check

6-1 Write the output from the following Java program fragments:

```
int n = 3;
int counter = 1;
while (counter <= n) {
  System.out.print(counter + "  ");
  counter = counter + 1;
}
```

```
int low = 1;
int high = 9;
while (low < high) {
  System.out.println(low + " " + high);
  low = low + 1;
  high = high - 1;
}
```

```java
int last = 10;
int j = 2;
while (j <= last) {
  System.out.print(j + "  ");
  j = j + 2;
}
```

```java
int counter = 10;
// Tricky, but an easy-to-make mistake
while (counter >= 0); {
  System.out.println(counter);
  counter = counter - 1;
}
```

6-2 Write the number of times "Hello" is printed. "Zero" and "Infinite" are valid answers.

```java
int counter = 1;
int n = 20;
while (counter <= n) {
  System.out.print("Hello ");
  counter = counter + 1;
}
```

```java
int j = 1;
int n = 5;
while (j <= n) {
  System.out.print("Hello ");
  n = n + 1;
  j = j + 1;
}
```

```java
int counter = 1;
int n = 5;
while (counter <= n) {
  System.out.print("Hello ");
  counter = counter + 1;
}
```

```java
// Tricky
int n = 5;
int j = 1;
while (j <= n)
  System.out.print("Hello ");
  j = j + 1;
```

6-3 Implement method `factorial` that return n!. `factorial(0)` must return 1, `factorial(1)` must return 1, `factorial(2)` must return 2*1, `factorial(3)` must return 3*2*1, and `factorial(4)` must return is 4*3*2*1. The following assertions must pass.

```java
  @Test
  public void testFactorial() {
    assertEquals(1, factorial(0));
    assertEquals(1, factorial(1));
    assertEquals(2, factorial(2));
```

```
    assertEquals(6, factorial(3));
    assertEquals(7 * 6 * 5 * 4 * 3 * 2 * 1, factorial(7));
  }
```

6-4 Implement method `duplicate` that returns a string where every letter is duplicated. Hint: Create an empty String referenced by `result` and concatenate each character in the argument to result twice. The following assertions must pass.

```
  @Test
  public void testDuplicate() {
    assertEquals("", duplicate(""));
    assertEquals("  ", duplicate(" "));
    assertEquals("zz", duplicate("z"));
    assertEquals("xxYYzz", duplicate("xYz"));
    assertEquals("1122334455", duplicate("12345"));
  }
```

6.2 Indeterminate Loop Pattern

It is often necessary to execute a set of statements an undetermined number of times. For example, to process report cards for *every* student in a school where the number of students changes from semester to semester. Programs cannot always depend on prior knowledge to determine the exact number of repetitions. It is often more convenient to think in terms of “process a report card for all students” rather than “process precisely 310 report cards.” This leads to a recurring pattern in algorithm design that captures the essence of repeating a process an unknown number of times. It is a pattern to help design a process of iterating until something occurs to indicate that the looping is finished. The **Indeterminate Loop pattern** occurs when the number of repetitions is not known in advance.

Algorithmic Pattern

Pattern: Indeterminate Loop

Problem: A process must repeat an unknown number of times.

Outline:
while (the termination condition has not occurred) {
  perform the actions
  do something to bring the loop closer to termination
}

Code Example
```
// Return the greatest common divisor of two positive integers.
public int GCD(int a, int b) {
  while (b != 0) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  return a;
}
```

The code example above is an indeterminate loop because the algorithm cannot determine how many times a must be subtracted from b or b from a. The loop repeats until there is nothing more to subtract. When b becomes 0, the loop terminates. When the following test method executes, the loop iterates a varying number of times:

```
@Test
public void testGCD() {
  assertEquals(2, GCD(6, 4));
  assertEquals(7, GCD(7, 7));
  assertEquals(3, GCD(24, 81));
  assertEquals(5, GCD(15, 25));
}
```

GCD(6, 4) → 2

a	b
6	4
2	4
2	2
2	0

GCD(7, 7) → 7

a	b
7	7
7	0

GCD(24, 81) → 3

a	b
24	81
24	57
24	33
24	9
15	9
6	9
6	3
3	3
3	0

GCD(15, 25) → 5

a	b
15	25
15	10
5	10
5	5
5	0

The number of iterations in the four assertions ranges from 1 to 8. However, GCD(1071, 532492) results in 285 loop iterations to find there is no common divisor other than 1. The following alternate algorithm for GCD(a, b) using modulus arithmetic more quickly finds the GCD in seven iterations because b approaches 0 more quickly with %.

```
// Return the greatest common divisor of two
// positive integers with fewer loop iterations
public int GCD(int a, int b) {
  while (b != 0) {
    int temp = a;
    a = b;
    b = temp % b;
  }
  return a;
}
```

a	b
532492	1071
1071	205
205	46
46	21
21	4
4	1
1	0

Indeterminate Loop with Scanner(String)

Sometimes a stream of input from the keyboard or a file needs to be read until there is no more needed input. The amount of input may not be known until there is no more. A convenient way to expose this processing is to use a Scanner with a String argument to represent input from the keyboard or a file.

```
// Constructs a new Scanner that produces values scanned from the specified
// string. The parameter source is the string to scan
public void Scanner(String source)
```

Scanner has convenient methods to determine if there is any more input of a certain type and to get the next value of that type. For example to read white space separated strings, use these two methods from `java.util.Scanner`.

```
// Returns true if this scanner has another token in its input.
// This method may block while waiting for keyboard input to scan.
public boolean hasNext()

// Return the next complete token as a string.
public String next()
```

The following test methods demonstrates how `hasNext()` will eventually return false after `next()` has been called for every token in scanner's string.

```
@Test
public void showScannerWithAStringOfStringTokens() {
  Scanner scanner = new Scanner("Input with four tokens");
  assertTrue(scanner.hasNext());
  assertEquals("Input", scanner.next());
  assertTrue(scanner.hasNext());
  assertEquals("with", scanner.next());
  assertTrue(scanner.hasNext());
```

```
    assertEquals("four", scanner.next());
    assertTrue(scanner.hasNext());
    assertEquals("tokens", scanner.next());

    // Scanner has scaned all tokens, so hasNext() should now be false.
    assertFalse(scanner.hasNext());
  }
```

You can also have the `String` argument in the `Scanner` constructor contain numeric data. You have used `nextInt()` before in Chapter 2's console based programs.

```
  // Returns true if the next token in this scanner's input
  // can be interpreted as an int value.
  public boolean hasNextInt()

  // Scans the next token of the input as an int.
  public int nextInt()
```

The following test method has an indeterminate loop that repeats as long as there is another valid integer to read.

```
  @Test
  public void showScannerWithAStringOfIntegerTokens() {
    Scanner scanner = new Scanner("80 70 90");
    // Sum all integers found as tokens in scanner
    int sum = 0;
    while (scanner.hasNextInt()) {
      sum = sum + scanner.nextInt();
    }
    assertEquals(240, sum);
  }
```

Scanner also has many such methods whose names indicate what they do: hasNextDouble() with nextDouble(), hasNextLine() with nextLine(), and hasNextBoolean() with nextBoolean().

A Sentinel Loop

A **sentinel** is a specific input value used only to terminate an indeterminate loop. A sentinel value should be the same type of data as the other input. However, this sentinel must not be treated the same as other input. For example, the following set of inputs hints that the input of -1 is the event that terminates the loop and that -1 is not to be counted as a valid test score. If it were counted as a test score, the average would not be 80.

Dialogue

```
Enter test score #1 or -1.0 to quit: 80
Enter test score #2 or -1.0 to quit: 90
Enter test score #3 or -1.0 to quit: 70
Enter test score #4 or -1.0 to quit: -1
Average of 3 tests = 80.0
```

This dialogue asks the user either to enter test scores or to enter -1.0 to signal the end of the data. With **sentinel loops**, a message is displayed to inform the user how to end the input. In the dialogue above, -1 is the sentinel. It could have some other value outside the valid range of inputs, any negative number, for example.

Since the code does not know how many inputs the user will enter, an indeterminate loop should be used. Assuming that the variable to store the user input is named currentInput, the termination condition is currentInput == -1. The loop should terminate when the user enters a value that flags the end of the data. The loop test can be derived by taking the logical negation of the termination condition. The while loop test becomes currentInput **!=** -1.

```
while (currentInput != -1)
```

The value for `currentInput` must be read before the loop. This is called a "priming read," which goes into the first iteration of the loop. Once inside the loop, the first thing that is done is to process the `currentInput` from the priming read (add its value to `sum` and add `1` to `n`). Once that is done, the second `currentInput` is read at the "bottom" of the loop. The loop test evaluates next. If `currentInput != -1`, the second input is processed. This loop continues until the user enters -1. Immediately after the `nextInt` message at the bottom of the loop, `currentValue` is compared to `SENTINEL`. When they are equal, the loop terminates. The `SENTINEL` is not added to the running sum, nor is 1 added to the count. The awkward part of this algorithm is that the loop is processing data read in the *previous* iteration of the loop.

The following method averages any number of inputs. It is an instance of the Indeterminate Loop pattern because the code does not assume how many inputs there will be.

```
import java.util.Scanner;
// Find an average by using a sentinel of -1 to terminate the loop
// that counts the number of inputs and accumulates those inputs.
public class DemonstrateIndeterminateLoop {

  public static void main(String[] args) {
    double accumulator = 0.0; // Maintain running sum of inputs
    int n = 0;                // Maintain total number of inputs
    double currentInput;
    Scanner keyboard = new Scanner(System.in);

    System.out.println("Compute average of numbers read.");
    System.out.println();
    System.out.print("Enter number or -1 to quit: ");
    currentInput = keyboard.nextDouble();

    while (currentInput != -1) {
      accumulator = accumulator + currentInput; // Update accumulator
      n = n + 1; // Update number of inputs so far
      System.out.print("Enter number or -1 to quit: ");
      currentInput = keyboard.nextDouble();
    }
```

```
    if (n == 0)
      System.out.println("Can't average zero numbers");
    else
      System.out.println("Average: " + accumulator / n);
  }
}
```

Dialogue

```
Compute average of numbers read.

Enter number or -1.0 to quit: 70.0
Enter number or -1.0 to quit: 90.0
Enter number or -1.0 to quit: 80.0
Enter number or -1.0 to quit: -1.0
Average: 80.0
```

The following table traces the changing state of the important variables to simulate execution of the previous program. The variable named `accumulator` maintains the running sum of the test scores. The loop also increments `n` by +1 for each valid `currentInput` entered by the user. Notice that `-1` is not treated as a valid `currentInput`.

Iteration Number	**currentInput**	**accumulator**	**n**	**currentInput != SENTINEL**
Before the loop	NA	0.0	0	NA
Loop 1	70.0	70.0	1	True
Loop 2	90.0	160.0	2	True
Loop 3	80.0	240.0	3	True
After the loop	NA	240.0	3	NA

Self-Check

6-5 Determine the value assigned to average for each of the following code fragments by simulating execution when the user inputs 70.0, 60.0, 80.0, and -1.0.

```
Scanner keyboard = new Scanner(System.in);
int n = 0;
double accumulator = 0.0;
double currentInput = keyboard.nextDouble();
while (currentInput != -1.0) {
  currentInput = keyboard.nextDouble();
  accumulator = accumulator  + currentInput; // Update accumulator
  n = n + 1; // Update total # of inputs
}
double average = accumulator / n;
```

6-6 If you answered 70.0 for 6-5, try again until you get an answers for !- 70.

6-7 What is the value of `numberOfWords` after this code executes with the dialogue shown (read carefully).

```
String SENTINEL = "QUIT";
Scanner keyboard = new Scanner(System.in);
String theWord = "";
int numberOfWords = 0;
System.out.println("Enter words or 'QUIT' to quit");
while (!theWord.equals(SENTINEL)) {
  numberOfWords = numberOfWords + 1;
  theWord = keyboard.next();
}
System.out.println("You entered " + numberOfWords + " words.");
```

Output

```
Enter words or 'QUIT' to quit
The quick brown fox quit and then jumped over the lazy dog. QUIT
You entered ___ words.
```

The `for` Statement

Java has several structures for implementing repetition. The `while` statement shown above can be used to implement indeterminate and determinate loop patterns. Java also has added a for loop that combines all looping logic into more compact code. The `for` loop was added to programming languages because the Determinate Loop Pattern arises so often. Here is the general form of the Java `for` loop:

***General Form:* `for` statement**

`for (`*initial-statement* **`;`** *loop-test* **`;`** *update-step*`)` **`{`**

    repeated-part **`;`**

`}`

The following `for` statement shows the three components that maintain the Determinate Loop pattern: the initialization (`n = 5` and `j = 1`), the loop test for determining when to stop (`j <= n`), and the update step (`j = j + 1`) that brings the loop one step closer to terminating.

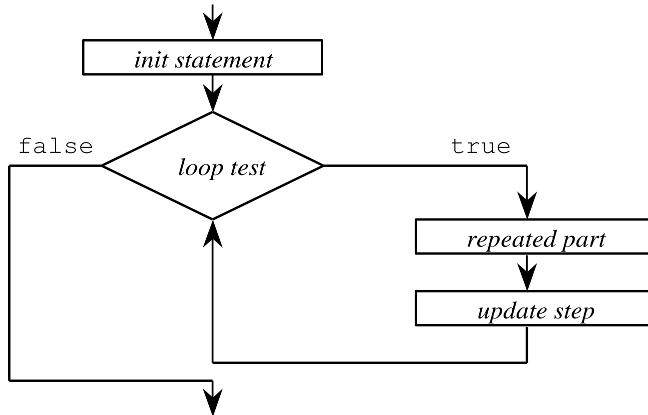
```
// Predetermined number of iterations
int n = 5;
for (int j = 1; j <= n; j = j + 1) {
  // Execute this block n times
}
```

In the preceding `for` loop, `j` is first assigned the value of 1. Next, `j <= n` (`1 <= 5`) evaluates to `true` and the block executes. When the statements inside the block are done, `j` increments by 1 (`j=j+1`). These three components ensure that the block executes precisely `n` times.

```
j = 1       // Initialize counter
j <= n      // Loop test
j = j + 1   // Update counter
```

When a `for` loop is encountered, the *initial-statement* is executed first and only once. The *loop-test* evaluates to either `true` or `false` before each execution of the *repeated-part*. The *update-step* executes after each iteration of the repeated part. This process continues until the loop test evaluates to `false`.

Flowchart view of a for loop



The following `for` statement simply displays the value of the loop counter named `j` as it ranges from 1 through 5 inclusive:

```
int n = 5;
for (int j = 1; j <= n; j = j + 1) {
  System.out.print(j + " ");
}
```

Output

```
1 2 3 4 5
```

Other Increment and Assignment Operators

Assignment operations alter computer memory even when the variable on the left of = is also involved in the expression to the right of =. For example, the variable `int j` is incremented by `1` with this assignment operation:

```
j = j + 1;
```

This type of update—incrementing a variable—is performed so frequently that Java offers operators with the express purpose of incrementing variables. The `++` and `--` operators **increment** and **decrement** a variable by 1, respectively. For example, the expression `j++` adds `1` to the value of `j`, and the expression `x--` reduces `x` by 1. The `++` and `--` unary operators alter the numeric variable that they follow (see the table below).

Statement	State of `j`
`int j = 0;`	0
`j++;`	1
`j++;`	2
`j--;`	1

So, within the context of the determinate loop, the update step can be written as `j++` rather than `j = j + 1`. This `for` loop

```
for (int j = 1; j <= n; j = j + 1) {
  // ...
}
```

may also be written with the `++` operator for equivalent behavior:

```java
for(int j = 1; j <= n; j++) {
  // ...
}
```

These new assignment operators are shown because they provide a convenient way to increment and decrement a counter in `for` loops. Also, most Java programmers use the `++` operator in `for` loops. You will see them often.

Java has several assignment operators in addition to `=`. Two of them, `+=` and `-=`, add and subtract value from the variable to the left, respectively.

Operator	**Equivalent Meaning**
`+=`	Increment variable on left by value on right.
`-=`	Decrement variable on left by value on right.

These two new operators alter the numeric variable that they follow.

Statement	**Value of `j`**
`int j = 0;`	0
`j += 3;`	3
`j += 4;`	7
`j -= 2;`	5

Whereas the operators ++ and -- increment and decrement the variable by one, the operators += and -= increment and decrement the variable by any amount. The += operator is most often used to accumulate values inside a loop.

The following comparisons show the for loop was designed to put the initialization and the update step together with the loop test. The for loops also use the shorter ++ operator. This makes the code a bit more

compact and a bit more difficult to read. However, you will get used to it, especially when the `for` loop will be used extensively in the next chapters.

<table>
<tr><th>While loop</th><th>For loop equivalent</th></tr>
<tr><td><pre><code>public int sumOfNInts(int n) {
  int result = 0;
  int counter = 1;
  while (counter <= n) {
    result = result + counter;
    counter++;
  }
  return result;
}</code></pre></td><td><pre><code>public int sumOfNInts(int n) {
  int result = 0;

  for (int counter = 1; counter <= n; counter++) {
    result = result + counter;
  }

  return result;
}</code></pre></td></tr>
<tr><td><pre><code>public int numSpaces(String str) {
  int result = 0;

  int index = 0;
  while (index < str.length()) {
    if (str.charAt(index) == ' ')
      result++;
    index++;
  }
  return result;
}</code></pre></td><td><pre><code>public int numSpaces(String str) {
  int result = 0;

  for (int index = 0; index < str.length(); index++) {
    if (str.charAt(index) == ' ')
      result++;
  }

  return result;
}</code></pre></td></tr>
</table>

Self-Check

6-8 Does a `for` loop execute the update step at the beginning of each iteration?

6-9 Must an update step increment the loop counter by `+1`?

6-10 Do `for` loops always execute the repeated part at least once?

6-11 Write the output generated by the following `for` loops.

```
for(int j = 0; j < 5; j++) {
  System.out.print(j + " ");
}
```

```
for( int j = 1; j < 10; j += 2) {
  System.out.print(j + " ");
}
```

```
int n = 5;
for( int j = 1; j <= n; j++ ) {
  System.out.print(j + " ");
}
```

```
int n = 0;
System.out.print("before ");
for(int j = 1; j <= n; j++) {
  System.out.print( j + " " );
}
System.out.print(" after");
```

```
int n = 3;
for (int j = -3; j <= n; j += 2) {
  System.out.print(j + " ");
}
```

```
for (int j = 5; j >= 1; j--) {
  System.out.print(j + " ");
}
```

6-12 Write a `for` loop that displays all of the integers from 1 to 100 inclusive on separate lines.

6-13 Write a `for` loop that displays all of the integers from 10 down to 1 inclusive on separate lines.

6.3 Loop Selection and Design

For some people, loops are easy to implement, even at first. For others, infinite loops, being off by one iteration, and intent errors are more common. In either case, the following outline is offered to help you choose and design loops in a variety of situations:

1. Determine which type of loop to use.
2. Determine the loop test.
3. Write the statements to be repeated.
4. Bring the loop one step closer to termination.
5. Initialize variables if necessary.

Determine Which Type of Loop to Use

If the number of repetitions is known in advance or is read as input, it is appropriate to use the Determinate Loop pattern. The `for` statement was specifically designed for this pattern. Although you can use the `while` loop to implement the Determinate Loop pattern, consider using the `for` loop instead. The `while` implementation allows you to omit one of the key parts with no compile time errors thus making any intent errors more difficult to detect and correct. If you leave off one of the parts from a `for` loop, you get an easier-to-detect-and-correct compiletime error.

The Indeterminate Loop pattern is more appropriate when you need to wait until some event occurs during execution of the loop. In this case, use the `while` loop. If you need to process all the data in an input file, consider using a `Scanner` object with one of the `hasNext` methods as the loop test. This is an indeterminate loop.

Determining the Loop Test

If the loop test is not obvious, try writing the conditions that must be true for the loop to terminate. For example, if you want the user to enter `QUIT` to stop entering input, the termination condition is

```
inputName.equals("QUIT")  // Termination condition
```

The logical negation `!inputName.equals("QUIT")` can be used directly as the loop test of a `while` loop.

```
while(! inputName.equals("QUIT")) {
 // . . .
}
```

Write the Statements to Be Repeated

This is why the loop is being written in the first place. Some common tasks include keeping a running sum, keeping track of a high or low value, and counting the number of occurrences of some value. Other tasks that will be seen later include searching for a name in a list and repeatedly comparing all string elements of a list in order to alphabetize it.

Bring the Loop One Step Closer to Termination

To avoid an infinite loop, at least one action in the loop must bring it closer to termination. In a determinate loop this might mean incrementing or decrementing a counter by some specific value. Inputting a value is a way to bring indeterminate loops closer to termination. This happens when a user inputs data until a sentinel is read, for example. In a `for` loop, the repeated statement should be designed to bring the loop closer to termination, usually by incrementing the counter.

Initialize Variables if Necessary

Check to see if any variables used in either the body of the loop or the loop test need to be initialized. Doing this usually ensures that the variables of the loop and the variables used in the iterative part have been initialized. This code attempts to use many variables in expressions before they have been initialized. In certain other languages, these variables are given garbage values and the result is unpredictable. Fortunately, the Java compiler flags these uninitialized variables as errors.

Self-Check

6-14 Which kind of loop best accomplishes these tasks?

- **-a** Sum the first five integers (1 + 2 + 3 + 4 + 5).
- **-b** Find the average for a list of numbers when the size of the list is known.
- **-c** Find the average value for a list of numbers when the size of the list is not known in advance.
- **-d** Obtain a character from the user that must be an uppercase S or Q.

6-15 To design a loop that processes inputs called `value` until `-1` is entered,

- **-a** describe the termination condition.
- **-b** write the Boolean expression that expresses the logical negation of the termination condition. This will be the loop test.

6-16 To design a loop that visits all the characters of `theString`, from the first to the last.

- **-a** describe the termination condition.
- **-b** write the Boolean expression that expresses the logical negation of the termination condition. This will be the loop test.

6-17 Which variables are not initialized but should be?

- **-a** `while(j <= n) {   }`
- **-b** `for(int j = 1; j <= n; j = j + inc) {   }`

Answers to Self-Checks

6-1 1 2 3

1 9
2 8
3 7
4 6

2 4 6 8 10

No output, this is an infinite loop, it does nothing. The code between) and ; (an empty statement) until the program is externally terminated.

6-2 20

Infinite since n grows as fast as j, j will always be less than n

5

Infinite since j++ is not part of the loop. Add { and }

6-3
```
public int factorial(int n) {
  int result = 1;
  int counter = 1;
  while (counter <= n) {
    result = result * counter;
    counter++;
  }
  return result;
}
```

6-4
```
public String duplicate(String str) {
  String result = "";
  int index = 0;
  while (index < str.length()) {
    result = result + str.charAt(index) + str.charAt(index);
```

```
      index++;
    }
    return result;
  }
```

6-5 46.3

6-6 Trace your code again if necessary.

6-7 The answer of 13 includes QUIT. The solution does not include the priming read.

```
You entered 13 words.
```

6-8 No, the update step happens at the end of the loop iteration. The `init` statement happens first, and only once.

6-9 No, you can use increments of any amount, including negative increments (decrements).

6-10 No, consider `for ( int j = 1; j < n; j++ ) { /*do nothing*/ }` when `n == 0`.

6-11

```
 0  1  2  3  4
 1  2  3  4  5
-3 -1  1  3
```

```
1  3  5  7  9
before  after
5  4  3  2  1
```

6-12

```
for(int j = 1; j <= 100; j++) {
   System.out.println( j );
 }
```

6-13

```
for(int k = 10; k >= 1; k--) {
   System.out.println(k);
 }
```

6-14 -a A `for` loop, since number of repetition is known.
-b A `for` loop, since the number of repetitions would be known in advance.
-c An indeterminate loop, perhaps a `while` loop that terminates when the sentinel is read.
-d An indeterminate loop, perhaps a `while` loop that terminates when the sentinel is read.

6-15 -a The `value` just input equals `-1`

-c `value != -1`

6-16 -a An index starting at 0 becomes the length of the string

-c `index < theString.length()`

6-17 -a Both `j` and `n`

-b Both `n` and `inc`