Herbert Schildt's *Java: A Beginner's Guide*, 6th ed.

Chapter 3 Self Test Answers

1. **Write a program that reads characters from the keyboard until a period is recieved. Have the program count the number of spaces. Report the total at the end of the program.**

   The following program satisfies the criteria outlined above:

```
public class SpacesCounter {
  public static void main (String args[]) throws java.io.IOException{

    char input = 0;
    int counter = 0;

    System.out.println("Enter a statment.  End your statement with a period:");

    while(input != '.'){
      input = (char) System.in.read();
      if (input == ' '){
        counter++;
      }
    } // end while

    System.out.println("There were " + counter + " spaces in your statement.");

  } // end main
} // end SpacesCounter
```

2. **Show the general form of a *if-else-if* ladder.**

   The general form of an *if-else-if* ladder is:

```
if (condition){
  // perform some action
} else if (condition){
  // perform some (other) action
} else {
  // some (usually default) action
}
```

3. **Given the following code, to what *if* does the last *else* associate?**

```
if (x < 10)
  if(y > 100){
  if (!done) x = z;
  else y = z;
  }
else System.out.println("error"); // what if?
```

   The second else in this statement refers the *if (y >100)* statement. This is due to the nested nature of the code. However, this code can be criticized for unclearness in its layout. It would be much clearer if formatted out as the following:

```
if (x < 10){
  if(y > 100){
    if (!done) {
      x = z;
    } else {
      y = z;
    }
  } else {
    System.out.println("error");
  }
}
```

   The code blocks above will produce the exact same result, but the second is much clearer as the curly braces align for ease of reading. Some programmers would find the second formatting as adding too many spaces, but remember Java does not consider white spaces in its code. So spacing code out to make it easy to read is better than collapsing it into small blocks that are less readable.

4. **Show the *for* statement for a loop that counts from 1000 to 0 by -2.**

   I created a small program that answers this question:

```
public class CountDownForLoop {

  public static void main(String args[]){

    int maxValue = 1000;
    int minValue = 0;
    int decrementBy = 2;

    for (; maxValue >= minValue; maxValue -= decrementBy){
      System.out.println(maxValue);
    }

  } // end main
} // end CountDownForLoop
```

5. **Is the following fragment valid?**

```
for (int i = 0; i < num; i++)
  sum += i;

count = i;
```

   The short answer to this question is **no**. There are several variables in this fragment which have not been declared or initialized, so this fragment definitely would not compile for this reason. Further, the code is attempting to assign the value of the *i* variable to the *count* variable outside the scope of the for loop, meaning that *i* is not defined in this scope and cannot be assigned here. Clearly this fragment of code is not valid.

6. **Explain what *break* does. Be sure to explain both of its forms.**

   According to Schildt (page 88): "When a **break** statement is encountered inside a loop, the loop is terminated and the program control resumes at the next statement following the loop." An example of this in code is (from page 88 of Schildt):

```
// Using break to exit a loop
class BreakDemo {
  public static void main(String args[]){

    int num = 100;

    // loop while i-squared is less than num
    for (int i = 0; i < num; i++){
      if (i*i >= num) break; // terminate the loop if i*i >= 100
      System.out.println(i + " ");
    }
    System.out.println("Loop complete");
  } // end main
} // end BreakDemo
```

   In the above code, once *i* reaches 10, the loop exits. There is another, similar use for *break*, but instead of simply breaking out of the control loop, the *break* statement can be accompanied by a label, and now the *break* statement defines where the code should recommence.

   A label is defined as:

```
one: {
  // insert code here
}
```

7. **In the following fragment, after the *break* statement executes, what is displayed?**

```
for (i = 0; i < 10; i++){
  while(running){
  if (x<y) break;
  //...
  }
  System.out.println("after while");
}
System.out.println("After for");
```

   In this code fragment, after *break* executes, the statement "after while" will be displayed in the console. As there is no indication of the values of *x* or *y*, there is no way to know how many times this will be executed. However, the maximum number of times this could possibly displayed would be 10, as the for loop it is (grand)nested will only run 10 times total.

8. **What does the following fragment print?**

```
for (int i = 0; i<10; i++){
  System.out.print(i + " ");
  if ((i%2) == 0) continue;
  System.out.println();
}
```

The output of the above fragment would be:

0 1

2 3

4 5

6 7

8 9

9. **The iteration expression in a *for* loop need not always alter the loop control variable by a fixed amount. Instead, the loop control variable can change in any arbitrary way. Using this concept, write a program that uses a *for* loop to generate and display the progression 1, 2, 4, 8, 16, 32 and so on.**

The following program satisfies the criteria outlined above:

```
/*
* This program answers Question 9 of Chapter 3 Self-Test
* from Herbert Schidt's Java: A Beginner's Guide
* It iterates over a for loop and multiplies its
* value by 2 up to 64.
*/

public class MultiplyBy2 {
  public static void main(String args[]){

    for (int i = 2; i <= 64; i *= 2){
      System.out.print(i + ", ");
    }

    System.out.println();

  } // end main
} // end MultiplyBy2
```

10. **The ASCII lowercase letters are separated from the uppercase letters by 32. Thus, to convert a lowercase letter to uppercase, subtract 32 from it. Use this information to write a program that reads characters from the keyboard. Have it convert all lowercase letters to uppercase, and all uppercase letters to lowercase, displaying the result. Make no changes to any other character. Have the program stop when the user enters a period. At the end, have the program display the number of case changes that have taken place.**

    The following program satisfies the criteria outlined above:

```
public class CaseChanger {
  public static void main (String args[]) throws java.io.IOException{

    char input = 0;

    System.out.println("Enter a statement that ends in a period.
            The program will switch the case each of the letters:");

    while (input != '.'){
      input = (char) System.in.read();

      if (input == ' '){
        System.out.print(' ');
      }

      if (input >= 65 && input <= 90){
        input += 32;
        System.out.print(input);
      } else if (input >= 97 && input <= 122){
        input -= 32;
        System.out.print(input);
      }
    }// end while

    System.out.println("\n");
  } // end main
} // end CaseChanger
```

11. **What is an infinite loop?**

    An infinite loop is a loop which does not have a circumstance in which it will terminate. It is particularly easy to create an infinite loop using the *for* loop, for example if there is no possible way the condition of the *for* loop could be false, then the loop will continue forever (or until it is manually escaped by the user).

12. **When using *break* with a label, must the label be on a block that contains the *break*?**

    No, it is not necessary that the *break* be contained within the code block that bears the label. A labelled break simply assigns a location for the flow of code to jump to.