Chapter 4

Basic Control Structures

Performing Comparisons

- `if` statement can whether a `boolean` expression has the value `true` or `false`.
- Comparisons are performed using the relational operators and the equality operators.
  - operands may have different types, an `int` will be converted to `double` before the comparison
  - The arithmetic operators take precedence over the relational operators

    < > <= >= returning a `boolean` result
  - Equality operators have lower precedence than the relational operators. == !=

    `2 == 2.0` ⇒ *true*

Testing Equality

- Floating-Point Round-off error, `1.2 - 1.1 == 0.1` is false, because the value of `1.2 - 1.1` is 0.09999999999999987, not 0.1.
- Test equality of object x and y, of the same type
  - `x == y` tests whether `x` and `y` refer to the same object (or both `x` and `y` have the value `null`).

`equals` Method in a Class

- Every Java class supports the `equals` method, although the definition of "equals" varies from class to class.
  - For some classes, the value of `x.equals(y)` is the same as `x` == `y`. e.g., String
  - Some classes have the `equals` method, to test whether two objects contain a same data value.

Comparing Strings

- `str1.equals(str2)` to test whether `str1` and `str2` contain the same series of characters.
- The `equalsIgnoreCase` method is similar to `equals` but ignores the case of letters.
- `str1.compareTo(str2)` returns an integer that's less than zero, equal to zero, or greater than zero, depending on whether `str1` is less than `str2`, equal to `str2`, or greater than `str2`, respectively.
  - For example, `"aab"` is less than `"aba"`. `"ab"` is less than `"aba"`.

Comparing Strings

- To determine whether one character is less than another, the `compareTo` method examines the Unicode values of the characters.
- Properties of Unicode characters:
  - Digits are assigned consecutive values; 0 is less than 1, which is less than 2, and so on.
  - Uppercase letters have consecutive values.
  - Lowercase letters have consecutive values.
  - Uppercase letters are less than lowercase letters.
  - The space character is less than any printing character, including letters and digits.

Logical Operators

- ***logical operators*** combine the results of comparisons. `!` `&&` `||`

  `age >= 18 && age <= 65`

- All logical operators expect `boolean` operands and produce `boolean` results.
- `!(9 < 11)` is `false`
- `!` operator is often used to test whether objects (including strings) are not equal:

  `!str1.equals(str2)`

Performing the *And* Operation

- The `&&` operator tests whether two `boolean` expressions are both true.
- Behavior of the `&&` operator:

  Evaluate the left operand. If it's false, return `false`. Otherwise, evaluate the right operand. If it's true, return `true`; if it's false, return `false`.
- The `&&` operator ignores the right operand if the left operand is false. This behavior is often called ***short-circuit evaluation.***

Short-Circuit Evaluation

- Short-circuit evaluation can save time.
- More importantly, short-circuit evaluation can avoid potential errors.
- The following expression tests whether `i` is not 0 before checking whether `j/i` is greater than 0:

```
(i != 0) && (j / i > 0)
```

Performing the *Or* Operation

- The `||` ("or") operator is used to test whether one (or both) of two conditions is true.
- Behavior of the `||` operator:

  Evaluate the left operand. If it's true, return `true`. Otherwise, evaluate the right operand. If it's true, return `true`; if it's false, return `false`.
- The `||` operator also relies on short-circuit evaluation. If the left operand is true, it ignores the right operand.

Precedence and Associativity of *And, Or,* and *Not*

- `!` operator takes precedence over `&&`, which in turn takes precedence over `||`.
- The relational and equality operators take precedence over `&&` and `||`, but have lower precedence than `!`.
- Java would interpret the expression

  ```
  a < b || c >= d && e == f
  ```

  as

  ```
  (a < b) || ((c >= d) && (e == f))
  ```

Precedence and Associativity of *And*, *Or*, and *Not*

- The `!` operator is right associative.
- The `&&` and `||` operators are left associative.
- Java would interpret

```
a < b && c >= d && e == f
```

as

```
((a < b) && (c >= d)) && (e == f)
```

Simplifying **`boolean`** Expressions

- `boolean` expressions that contain the `!` can be simplified by ***de Morgan's Laws:***

  `!(`*expr1* `&&` *expr2*`)` is equivalent to `!(`*expr1*`) || !(`*expr2*`)`

  `!(`*expr1* `||` *expr2*`)` is equivalent to `!(`*expr1*`) && !(`*expr2*`)`

  expr1 and *expr2* are `boolean` expressions.

```
!(i >= 1 && i <= 10)

!(i >= 1) || !(i <= 10)

i < 1 || i > 10
```

`if` Statements

`if (` *expression* `)` `//` *expression* is `boolean` type.
    statement

The *expression* is evaluated. true, then *statement* is executed. false, *statement* is not executed.

```
if (score > 100)
  score = 100;
```

- = operator in an `if` statement's condition:

```
if (i = 0) …   // WRONG
```

- `if` statement has an "inner statement"—to be executed if the condition is true. programmers normally indent the inner statement

Increment and Decrement Operators

- `++`, ***increment operator,*** `--`, the ***decrement operator***

  `i++; i--` // a ***postfix*** operator

  `++i; --i` // a ***prefix*** operator
- Evaluating the expression `i + j` doesn't change `i` or `j`. Evaluating `++i` causes a permanent change to `i`, however.
- Used in isolation, no difference before or after the variable.
- Used within other statements it usually does make a difference:

```
System.out.println(++i);
// Increments i and then prints the new
// value of i
System.out.println(i++);
// Prints the old value of i and then
// increments i
```

Exercise: Write a Program

- Test the following, what will be the output.

```
String s1="ab", s2="ab";
if(s1==s2) System.out.println("equal");
if(s1.equals(s2)) System.out.println("equal");

int n=0; double m=0;
if ((n==0) && (++m > 0))
                    System.out.println("m = " + m);
if ((n==0) && (m-- > 0))
                    System.out.println("m = " + m);
```

The Empty Statement

- Putting a semicolon after the test condition in an `if` statement is wrong:

```
if (score > 100);   // WRONG
  score = 100;
```

- The compiler treats the extra semicolon as an ***empty statement,*** however, so it doesn't detect an error:

```
if (score > 100)
  ;  // Empty statement--does nothing
score = 100;
```

Blocks

- An `if` statement can contain only one inner statement.
- In order to have an `if` statement perform more than one action, a ***block*** can be used.
- General form of a block:

```
{
   statements
}
```

- A block is considered to be one statement, even though it may contain any number of statements.

Blocks

- Example:

```
if (score > 100)
  {
    System.out.println("** Error: Score exceeds 100 **");
    score = 100;
  }
```

- Each of the statements inside the block ends with a semicolon, but there's no semicolon after the block itself.
- Curly brace {} can be at different location, increasing the indentation for each new nesting level.
  - Aligning statements at the same level of nesting.

`if` Statements with `else` Clauses

- The `if` statement is allowed have an `else` clause:

```
if ( expression )
    statement
else
    statement
```

- There are now two inner statements.
  - The first is executed if the expression is true.
  - The second is executed if the expression is false.

`if` Statement Layout

- An example of an `if` statement with an `else` clause:

```
if (a > b)
  larger = a;
else
  larger = b;
```

```
Or
```

```
if (a > b) larger = a;
  else larger = b;
```

`if` Statement Layout

- Recommended layout when the inner statements are blocks:

```
if (…) {
  …
} else {
  …
}
```

- Other layouts are also common. For example:

```
if (…) {
  …
}
else {
  …
}
```

Nested `if` Statements

- The statements nested inside an `if` statement can be other `if` statements.
- An `if` statement that converts an hour expressed on a 24-hour scale (0–23) to a 12-hour scale:

```
if (hour <= 11)
  if (hour == 0)
    System.out.println("12 midnight");
  else
    System.out.println(hour + " a.m.");
else
  if (hour == 12)
    System.out.println("12 noon");
  else
    System.out.println((hour - 12) + " p.m.");
```

Nested `if` Statements

- For clarity, it' s probably a good idea to put braces around the inner `if` statements:

```
if (hour <= 11) {
  if (hour == 0)
    System.out.println("12 midnight");
  else
    System.out.println(hour + " a.m.");
} else {
  if (hour == 12)
    System.out.println("12 noon");
  else
    System.out.println((hour - 12) + " p.m.");
}
```

Cascaded `if` Statements

- Test a series of conditions, one after the other, until finding one that's true.
- This situation is best handled by nesting a series of `if` statements in such a way that the `else` clause of each is another `if` statement.
- This is called a ***cascaded*** `if` statement.

Cascaded `if` Statements

- A cascaded `if` statement that prints a letter grade:

```
if (score >= 90)
  System.out.println("A");
else
  if (score >= 80 && score <= 89)
    System.out.println("B");
  else
    if (score >= 70 && score <= 79)
      System.out.println("C");
    else
      if (score >= 60 && score <= 69)
        System.out.println("D");
      else
        System.out.println("F");
```

Cascaded `if` Statements

- To avoid "indentation creep," programmers customarily put each `else` underneath the original `if`:

```
if (score >= 90)
  System.out.println("A");
else if (score >= 80 && score <= 89)
  System.out.println("B");
else if (score >= 70 && score <= 79)
  System.out.println("C");
else if (score >= 60 && score <= 69)
  System.out.println("D");
else
  System.out.println("F");
```

Cascaded `if` Statements

- General form of a cascaded `if` statement:

```
if ( expression )
   statement
else if ( expression )
   statement
…
else if ( expression )
   statement
else
   statement
```

- The `else` clause at the end may not be present.

Simplifying Cascaded `if` Statements

- A cascaded `if` statement can often be simplified by removing conditions that are guaranteed (because of previous tests) to be true.
- The "letter grade" example has three such tests:

Exercise: Write a Program

1. Write a program called StudentGrade

- Generate two random scores in the range of 0 and 100
  - `Math.random` method returns a "random" number greater than or equal to 0.0 and less than 1.0.
- Output a "letter grade " for the score

A >=90

B 80 – 90

C 70 – 80

D 60 – 70

F <60

The "Dangling `else`" Problem

- When one `if` statement contains another, the "dangling `else`" problem can sometimes occur.
- If `n` is in [0, `max`], add `n` to `sum`, if `n > max`, add `max` to `sum`

```
if (n <= max)
  if (n > 0)
    sum += n;
else
  sum += max;
```

The "Dangling `else`" Problem

- The problem is ***ambiguity***. There are two ways to read the `if` statement:

Interpretation 1

```
if (n <= max) {
  if (n > 0)
    sum += n;
} else
  sum += max;
```

Interpretation 2

```
if (n <= max) {
  if (n > 0)
    sum += n;
  else
    sum += max;
}
```

- When `if` statements are nested, Java matches each `else` clause with the nearest unmatched `if`, leading to Interpretation 2.

The "Dangling `else`" Problem

- To force Interpretation 1, the inner statement will need to be made into a block by adding curly braces:

```
if (n <= max) {
  if (n > 0)
    sum += n;
} else
  sum += max;
```

- Always using braces in `if` statements will avoid the dangling `else` problem.

`boolean` Type

- Variables and parameters can have `boolean` type, and methods can return `boolean` values.
  - ideal for representing data items that have only two possible values.
  - Good names often contain a verb such as "is," "was," or "has."
  - `boolean jobWasDone = false;`
  - `if (jobWasDone)` …
  - `if (jobWasDone == true)` …
  - `System.out.println(jobWasDone);`

  Either the word `true` or the word `false` will be displayed.

Types of Loops

- Java has three loop statements:
  - `while`
  - `do`
  - `for`
- All three use a `boolean` expression to determine whether or not to continue looping.
- All three require a single statement as the loop body. This statement can be a block, however.

Types of Loops

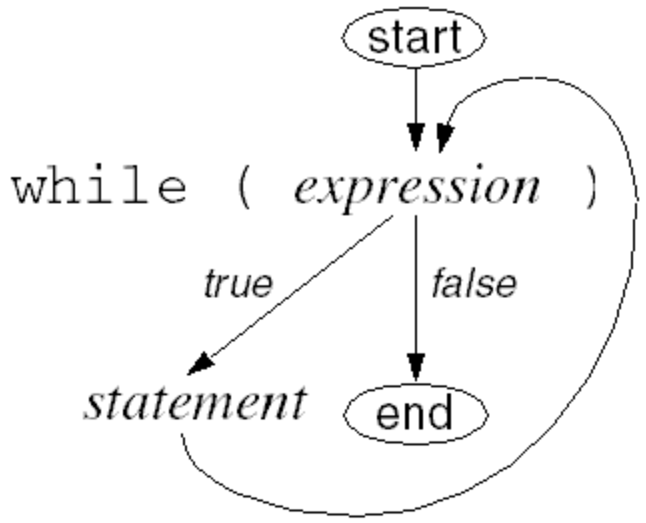
- Which type of loop to use is mostly a matter of convenience.
  - The `while` statement tests its condition *before* executing the loop body.
  - The `do` statement tests its condition *after* executing the loop body.
  - The `for` statement is most convenient if the loop is controlled by a variable whose value needs to be updated each time the loop body is executed.

The `while` Statement

```
while ( expression )
       statement
```

- The expression is evaluated first. If `true`, loop body is executed and the expression is tested again.

```
while (i < 20) // Controlling expression
    i *= 2;    // Loop body
```



Blocks as Loop Bodies

- The loop body can be a block within {}.
- The greatest common divisor (GCD) of two integers is the largest integer that divides both numbers evenly, with no remainder. For example, the GCD of 15 and 35 is 5.

Blocks as Loop Bodies

- Euclid's algorithm for computing the GCD:

  1. Let m and n be variables containing the two numbers.

  2. If n is 0, then stop: m contains the GCD.

  3. Divide m by n. Save the divisor in m, and save the remainder in n.

  4. Repeat the process, starting at step 2.

- The algorithm will need a loop of the form

```
while (n != 0) {
   …
}
```

Blocks as Loop Bodies

- A possible (but incorrect) body for the loop:

```
m = n;         // Save divisor in m
n = m % n;     // Save remainder in n
```

- Writing the loop body correctly requires the use of a ***temporary variable:*** a variable that stores a value only briefly.

```
while (n != 0) {
  r = m % n;   // Store remainder in r
  m = n;       // Save divisor in m
  n = r;       // Save remainder in n
}
```

Blocks as Loop Bodies

- Be careful to use braces if the body of a loop contains more than one statement.
- Neglecting to do so may accidentally create an infinite loop:

```
while (n != 0)   // WRONG; braces needed
  r = m % n;
  m = n;
  n = r;
```

- An ***infinite loop*** occurs when a loop's controlling expression is always true, so the loop can never terminate.

Blocks as Loop Bodies

- A table can be used to show how the variables change during the execution of the GCD loop:

	Initial value	*After iteration 1*	*After iteration 2*	*After iteration 3*	*After iteration 4*
`r`	?	30	12	6	0
`m`	30	72	30	12	6
`n`	72	30	12	6	0

- The GCD of 30 and 72 is 6, the final value of `m`.

Declaring Variables in Blocks

- A temporary variable can be declared inside a block:

```
while (n != 0) {
  int r = m % n; // Store remainder in r
  m = n;         // Save divisor in m
  n = r;         // Save remainder in n
}
```

- Any block may contain variable declarations, not just a block used as a loop body.

Declaring Variables in Blocks

- Java prohibits a variable declared inside a block from having the same name as a variable (or parameter) declared in the enclosing method.
- Declaring a variable inside a block isn't always a good idea.
  - can be used only within the block.
  - is created each time the block is entered and destroyed at the end of the block, causing its value to be lost.

Example: Improving the `Fraction` Constructor

- The original version of the `Fraction` class provides the following constructor:

```
public Fraction(int num, int denom) {
  numerator = num;
  denominator = denom;
}
```

- This constructor doesn't reduce fractions to lowest terms. Executing the statements

```
Fraction f = new Fraction(4, 8);
System.out.println(f);
```

will produce 4/8 as the output instead of 1/2.

Example: Improving the `Fraction` Constructor

- An improved version of the `Fraction` constructor:

```
public Fraction(int num, int denom) {
  // Compute GCD of num and denom
  int m = num, n = denom;
  while (n != 0) {
    int r = m % n;
    m = n;
    n = r;
  }

  // Divide num and denom by GCD; store results in instance
  // variables
  if (m != 0) {
    numerator = num / m;
    denominator = denom / m;
  }
```

Example: Improving the **`Fraction`** Constructor

```
    // Adjust fraction so that denominator is never negative
    if (denominator < 0) {
      numerator = -numerator;
      denominator = -denominator;
    }
  }
```

- If the GCD of `num` and `denom` is 0, the constructor doesn't assign values to `numerator` and `denominator`. Java automatically initializes these variables to 0 anyway, so there is no problem.

A “Countdown” Loop

- The countdown loop will need a counter that's updated:

```
int i = 10;
while (i > 0) {
  System.out.println(“i=" + i);
  i -= 1;
}
```

- `i != 0` could be used instead of `i > 0` as the controlling expression. However, `i > 0` is more descriptive, since it suggests that `i` is decreasing.
- Using short names for counters is a tradition.

Increment and Decrement Operators

- One way to increment or decrement a variable is to use the + or – operator in conjunction with assignment:

```
i = i + 1;   // Increment i
i = i - 1;   // Decrement i
```

- Another way is to use the += and –= operators:

```
i += 1;      // Increment i
i -= 1;      // Decrement i
```

Increment and Decrement Operators

- `++`, ***increment operator,*** `--`, the ***decrement operator***

  `i++; i--` // a ***postfix*** operator

  `++i; --i` // a ***prefix*** operator
- Evaluating the expression `i + j` doesn't change `i` or `j`. Evaluating `++i` causes a permanent change to `i`, however.
- Used in isolation, no difference before or after the variable.
- Used within other statements it usually does make a difference:

```
System.out.println(++i);
// Increments i and then prints the new
// value of i
System.out.println(i++);
// Prints the old value of i and then
// increments i
```

Increment and Decrement Operators

- `++` and `--` can be used in conjunction with other operators:

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

  `i` is now 2 and `j` is now 3.

- The outcome is different if the `++` operator is placed after `i`:

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

  Both `i` and `j` are now 2.

Using the Increment and Decrement Operators in Loops

- A modified version of the "squares" example:

```
while (i <= n) {
  System.out.println(i + " " + i * i);
  i++;
}
```

- `++` and `--` can sometimes be used to simplify loops, including the countdown loop:

```
while (i > 0) {
  System.out.println("T minus " + i-- +
                     " and counting");
}
```

The braces are no longer necessary.

Using the Increment and Decrement Operators in Loops

- The `CourseAverage` program of Section 2.11 would benefit greatly from counting loops.
- In particular, a loop could be used to read the eight program scores and compute their total:

```
String userInput;
double programTotal = 0.0;
int i = 1;
while (i <= 8) {
  SimpleIO.prompt("Enter Program " + i +
                  " score: ");
  userInput = SimpleIO.readLine();
  programTotal += Convert.toDouble(userInput);
  i++;
}
```

Uses of the `break` Statement

- The `break` statement has several potential uses:
  - Premature exit from a loop
  - Exit from the middle of a loop
  - Multiple exit points within a loop
- `break` statement is usually nested inside an `if` statement, so that the enclosing loop will terminate only when a certain condition has been satisfied

Premature Exit from a Loop

- The problem of testing whether a number is prime illustrates the need for premature exit from a loop.
- The following loop divides `n` by the numbers from 2 to `n` – 1, breaking out when a divisor is found:

```
int d = 2;
while (d < n) {
  if (n % d == 0)
    break;  // Terminate loop; n is not a prime
  d++;
}
if (d < n)
  System.out.println(n + " is divisible by " + d);
else
  System.out.println(n + " is prime");
```

Loops with an Exit in the Middle

- Loops in which the exit point is in the middle of the body are fairly common.
- A loop that reads user input, terminating when a particular value is entered:

```
while (true) {
  SimpleIO.prompt("Enter a number (enter 0 to stop): ");
  String userInput = SimpleIO.readLine();
  int n = Integer.parseInt(userInput);
  if (n == 0)
    break;
  System.out.println(n + " cubed is " + n * n * n);
}
```

- Using `true` as the controlling expression forces the loop to repeat until the `break` statement is executed.

Exercise: Write a Program, StudentGrade2

- Ask the user to input a score using Scanner class and check the input score. If the input score is negative, ask the user to re-input until a non-negative score is typed in.
- Output a "letter grade " for the score

A >=90

B 80 – 90

C 70 – 80

D 60 – 70

F <60

4.9 Case Study: Decoding Social Security Numbers

- The first three digits of a Social Security Number (SSN) form the "area number," which indicates the state or U.S. territory in which the number was originally assigned.
- The `SSNInfo` program will ask the user to enter an SSN and then indicate where the number was issued:

```
Enter a Social Security number: 078-05-1120
Number was issued in New York
```

Input Validation

- `SSNInfo` will partially validate the user's input:
  - The input must be 11 characters long (not counting any spaces at the beginning or end).
  - The input must contain dashes in the proper places.
- There will be no check that the other characters are digits.

Input Validation

- If an input is invalid, the program will ask the user to re-enter the input:

```
Enter a Social Security number: 078051120
Error: Number must have 11 characters

Please re-enter number: 07805112000
Error: Number must have the form ddd-dd-dddd

Please re-enter number: 078-05-1120
Number was issued in New York
```

Design of the `SSNInfo` Program

- An overall design for the program:

  1. Prompt the user to enter an SSN and trim spaces from the input.

  2. If the input isn't 11 characters long, or lacks dashes in the proper places, prompt the user to re-enter the SSN; repeat until the input is valid.

  3. Compute the area number from the first three digits of the SSN.

  4. Determine the location corresponding to the area number.

  5. Print the location, or print an error message if the area number isn't legal.

Design of the `SSNInfo` Program

- A pseudocode version of the loop in step 2:

```
while (true) {
   if (user input is not 11 characters long) {
      print error message;
   else if (dashes are not in the right places) {
      print error message;
   else
      break;
   prompt user to re-enter input;
   read input;
}
```

Design of the `SSNInfo` Program

- The input will be a single string, which can be trimmed by calling the `trim` method.
- The first three digits of this string can be extracted by calling `substring` and then converted to an integer by calling `Integer.parseInt`.
- This integer can then be tested by a cascaded `if` statement to see which location it corresponds to.

SSNInfo.java

```
// Program name: SSNInfo
// Author: K. N. King
// Written: 1999-06-18
//
// Prompts the user to enter a Social Security number and
// then displays the location (state or territory) where the
// number was issued. The input is checked for length (should
// be 11 characters) and for dashes in the proper places. If
// the input is not valid, the user is asked to re-enter the
// Social Security number.

import jpb.*;

public class SSNInfo {
  public static void main(String[] args) {
    // Prompt the user to enter an SSN and trim the input
    SimpleIO.prompt("Enter a Social Security number: ");
    String ssn = SimpleIO.readLine().trim();
```

```
// If the input isn't 11 characters long, or lacks dashes
// in the proper places, prompt the user to re-enter
// the SSN; repeat until the input is valid.
while (true) {
  if (ssn.length() != 11) {
    System.out.println("Error: Number must have 11 " +
                       "characters");
  } else if (ssn.charAt(3) != '-' ||
             ssn.charAt(6) != '-') {
    System.out.println(
     "Error: Number must have the form ddd-dd-dddd");
  } else
    break;
  SimpleIO.prompt("\nPlease re-enter number: ");
  ssn = SimpleIO.readLine().trim();
}

// Get the area number (the first 3 digits of the SSN)
int area = Integer.parseInt(ssn.substring(0, 3));
```

```
// Determine the location corresponding to the area number
String location;
if      (area == 0)   location = null;
else if (area <= 3)   location = "New Hampshire";
else if (area <= 7)   location = "Maine";
else if (area <= 9)   location = "Vermont";
else if (area <= 34)  location = "Massachusetts";
else if (area <= 39)  location = "Rhode Island";
else if (area <= 49)  location = "Connecticut";
else if (area <= 134) location = "New York";
else if (area <= 158) location = "New Jersey";
else if (area <= 211) location = "Pennsylvania";
else if (area <= 220) location = "Maryland";
else if (area <= 222) location = "Delaware";
else if (area <= 231) location = "Virginia";
else if (area <= 236) location = "West Virginia";
else if (area <= 246) location = "North Carolina";
else if (area <= 251) location = "South Carolina";
else if (area <= 260) location = "Georgia";
else if (area <= 267) location = "Florida";
else if (area <= 302) location = "Ohio";
else if (area <= 317) location = "Indiana";
else if (area <= 361) location = "Illinois";
else if (area <= 386) location = "Michigan";
else if (area <= 399) location = "Wisconsin";
else if (area <= 407) location = "Kentucky";
else if (area <= 415) location = "Tennessee";
else if (area <= 424) location = "Alabama";
else if (area <= 428) location = "Mississippi";
else if (area <= 432) location = "Arkansas";
else if (area <= 439) location = "Louisiana";
else if (area <= 448) location = "Oklahoma";
else if (area <= 467) location = "Texas";
else if (area <= 477) location = "Minnesota";
else if (area <= 485) location = "Iowa";
else if (area <= 500) location = "Missouri";
else if (area <= 502) location = "North Dakota";
else if (area <= 504) location = "South Dakota";
else if (area <= 508) location = "Nebraska";
else if (area <= 515) location = "Kansas";
else if (area <= 517) location = "Montana";
else if (area <= 519) location = "Idaho";
else if (area <= 520) location = "Wyoming";
else if (area <= 524) location = "Colorado";
else if (area <= 525) location = "New Mexico";
else if (area <= 527) location = "Arizona";
else if (area <= 529) location = "Utah";
else if (area <= 530) location = "Nevada";
else if (area <= 539) location = "Washington";
else if (area <= 544) location = "Oregon";
else if (area <= 573) location = "California";
else if (area <= 574) location = "Alaska";
else if (area <= 576) location = "Hawaii";
else if (area <= 579) location = "District of Columbia";
else if (area <= 580) location = "Virgin Islands";
else if (area <= 584) location = "Puerto Rico";
else if (area <= 585) location = "New Mexico";
else if (area <= 586) location = "Pacific Islands";
else if (area <= 588) location = "Mississippi";
else if (area <= 595) location = "Florida";
```

```
    else if (area <= 599) location = "Puerto Rico";
    else if (area <= 601) location = "Arizona";
    else if (area <= 626) location = "California";
    else if (area <= 645) location = "Texas";
    else if (area <= 647) location = "Utah";
    else if (area <= 649) location = "New Mexico";
    else if (area <= 653) location = "Colorado";
    else if (area <= 658) location = "South Carolina";
    else if (area <= 665) location = "Louisiana";
    else if (area <= 675) location = "Georgia";
    else if (area <= 679) location = "Arkansas";
    else if (area <= 680) location = "Nevada";
    else                  location = null;

    // Print the location, or print an error message if the
    // area number isn't legal
    if (location != null)
      System.out.println("Number was issued in " + location);
    else
      System.out.println("Number is invalid");
  }
}
```

4.10 Debugging

- When a program contains control structures such as the `if` and `while` statements, debugging becomes more challenging.
- It will be necessary to run the program more than once, with different input data each time.
- Each set of input data is called a ***test case.***

Statement Coverage

- Make sure that each statement in the program is executed by at least one test case. (This testing technique is called ***statement coverage.***)
- Check that the controlling expression in each `if` statement is true in some tests and false in others.
- Try to test each `while` loop with data that forces the controlling expression to be false initially, as well as data that forces the controlling expression to be true initially.

Debugging Loops

- Common types of loop bugs:
  - ***"Off-by-one" errors.*** *Possible cause:* Using the wrong relational operator in the loop's controlling expression (such as `i` < `n` instead of `i` <= `n`).
  - ***Infinite loops.*** *Possible causes:* Failing to increment (or decrement) a counter inside the body of the loop. Accidentally creating an empty loop body by putting a semicolon in the wrong place.
  - ***Never-executed loops.*** *Possible causes:* Inverting the relational operator in the loop's controlling expression (`i` > `n` instead of `i` < `n`, for example). Using the == operator in a controlling expression.

Debugging Loops

- A debugger is a great help in locating loop-related bugs. By stepping through the statements in a loop body, it’s easy to locate an off-by-one error, an infinite loop, or a never-executed loop.
- Another approach: Use `System.out.println` to print the value of the counter variable (if the loop has one), plus any other important variables that change during the execution of the loop.