Chapter 4

Basic Control Structures

4.1 Performing Comparisons

- Most programs need the ability to test conditions and make decisions based on the outcomes of those tests.
- The primary tool for testing conditions is the `if` statement, which tests whether a `boolean` expression has the value `true` or `false`.
- Most of the conditions in a program involve comparisons, which are performed using the relational operators and the equality operators.

Relational Operators

- The ***relational operators*** test the relationship between two numbers, returning a `boolean` result.
  - `<` Less than
  - `>` Greater than
  - `<=` Less than or equal to
  - `>=` Greater than or equal to
- Examples:
  - `5 < 3` ⇒ *false*
  - `5 > 3` ⇒ *true*
  - `3 > 3` ⇒ *false*
  - `5 <= 3` ⇒ *false*
  - `5 >= 3` ⇒ *true*
  - `3 >= 3` ⇒ *true*

Relational Operators

- The relational operators don't require operands to have identical types.
- If an `int` value is compared with a `double` value, the `int` value will be converted to `double` before the comparison is performed.
- The arithmetic operators take precedence over the relational operators, so Java would interpret

```
a - b * c < d + e
```

as

```
(a - b * c) < (d + e)
```


Equality Operators

- Testing whether two values are equal or not equal is done using the ***equality operators:***
  - `==` Equal to
  - `!=` Not equal to
- The equality operators have lower precedence than the relational operators.
- Examples of the equality operators:
  - `6 == 2` ⇒ *false*
  - `6 != 2` ⇒ *true*
  - `2 == 2` ⇒ *true*
  - `2 != 2` ⇒ *false*

Equality Operators

- As with the relational operators, the types of the operands don't need to be the same.
- If an `int` operand is compared to a `double` operand, the `int` value is converted automatically to `double` type before the comparison is performed:
  - `2 == 2.0` ⇒ *true*
  - `2 == 2.1` ⇒ *false*

Testing Floating-Point Numbers for Equality

- Because of round-off error, floating-point numbers that seem as though they should be equal may not be.
- For example, the condition

  `1.2 - 1.1 == 0.1`

  is false, because the value of `1.2 - 1.1` is 0.09999999999999987, not 0.1.
- One way to avoid problems with round-off error is to test whether floating-point numbers are close enough, rather than testing whether they're equal.

Testing Objects for Equality

- If `x` and `y` are two object variables of the same type, the expression

  `x == y`

  tests whether `x` and `y` refer to the same object (or both `x` and `y` have the value `null`).
- The expression

  `x != y`

  tests whether `x` and `y` refer to different objects (or just one of `x` and `y` has the value `null`).
- In either case, the references in `x` and `y` are being compared, not the objects that `x` and `y` refer to.

The **`equals`** Method

- The `equals` method is used to test whether two objects contain matching data.
- The value of `x.equals(y)` is `true` if the objects that `x` and `y` represent are "equal."
- 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`.

Comparing Strings

- Strings are objects, so the == operator should not be used to test whether two strings are equal.
- Instead, use `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.
- For example, if `str1` is `"hotjava"` and `str2` is `"HotJava"`, the value of `str1.equalsIgnoreCase(str2)` is `true`.

Comparing Strings

- `String` is one of the few classes in the Java API that supports relational operations.
- To compare the strings `str1` and `str2`, the `compareTo` method is used:

  `str1.compareTo(str2)`
- `compareTo` 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.

Comparing Strings

- `compareTo` looks for the first position in which the strings are different.
- For example, `"aab"` is considered to be less than `"aba"`.
- If the characters in the strings match, then `compareTo` considers the shorter of the two strings to be smaller.
- For example, `"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.

4.2 Logical Operators

- The ***logical operators*** are used to combine the results of comparisons. Example:

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

- There are three logical operators:

`!` Logical *not*
`&&` Logical *and*
`||` Logical *or*

- `!` is a unary operator. `&&` and `||` are binary operators.
- All logical operators expect `boolean` operands and produce `boolean` results.

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.

Performing the *Not* Operation

- When applied to a false value, the `!` (“not”) operator returns `true`. When applied to a true value, it returns `false`.
- The value of `9 < 11` is `true`, but the value of `!(9 < 11)` is `false`.
- The `!` operator is often used to test whether objects (including strings) are not equal:

```
!str1.equals(str2)
```

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

- The `!` operator takes precedence over the `&&` operator, which in turn takes precedence over the `||` operator.
- 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 `!` operator can often be simplified by applying one of ***de Morgan's Laws:***

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

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

  expr1 and *expr2* are arbitrary `boolean` expressions.

Simplifying `boolean` Expressions

- Using de Morgan's Laws, the expression

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

could be rewritten as

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

and then simplified to

```
i < 1 || i > 10
```

- This version has fewer operators and avoids using the `!` operator, which can make `boolean` expressions hard to understand.

Testing for Leap Years

- `boolean` expressions often get complicated, as in the case of testing whether a year is a leap year.
- A leap year must be a multiple of 4. However, if a year is a multiple of 100, then it must also be a multiple of 400 in order to be a leap year.
- 2000 is a leap year, but 2100 is not.
- A `boolean` expression that tests for a leap year:

```
(year % 4 == 0) &&
(year % 100 != 0 || year % 400 == 0)
```


4.3 Simple `if` Statements

- The `if` statement allows a program to test a condition.
- Form of the `if` statement:

  `if (` *expression* `)`
     statement

  expression must have `boolean` type.
- When the `if` statement is executed, the condition is evaluated. If it's true, then *statement* is executed. If it's false, *statement* is not executed.

An Example

- Example of an `if` statement:

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

- Be careful not to use the = operator in an `if` statement's condition:

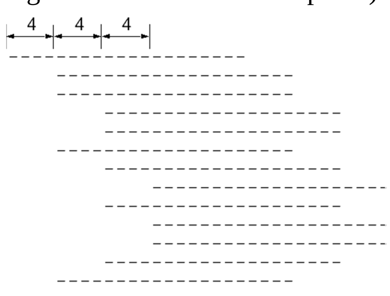
```
if (i = 0) …  // WRONG
```


Indentation

- Each `if` statement contains an "inner statement"—the one to be executed if the condition is true.
- To make it clear that this statement is inside the `if` statement, programmers normally indent the inner statement.
- The amount of indentation should be the same as the amount used to indent a method within a class, or a statement within a method.

Indentation

- The appearance of a properly nested program (assuming an indentation of four spaces):



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.

Statement Nesting

- Because of blocks, statements are often deeply nested:

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

- It's important to use visual cues to show nesting:
  - Increasing the indentation for each new nesting level.
  - Aligning statements at the same level of nesting.

Layout of Statements

- To avoid "indentation creep," it's customary to align braces with the statement that encloses them:

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

- To conserve vertical space, many programmers put the left curly brace at the end of the previous line:

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


4.4 `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;
```

- Layout conventions:
  - Align `else` with `if`.
  - Put each assignment on a separate line.
  - Indent each assignment.

`if` Statement Layout

- If the statements inside an `if` statement are very short, programmers will sometimes put them on the same line as the `if` or `else`:

```
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

- Many programs need to 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:

```
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");
```


Simplifying Cascaded `if` Statements

- A simplified version of the "letter grade" `if` statement:

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


Program: Flipping a Coin

- Using `if` statements, it's easy to write a program that asks the user to guess the outcome of a simulated coin flip.
- The program will indicate whether or not the user guessed correctly:

```
Enter heads or tails: tails
Sorry, you lose.
```

- If the user's input isn't `heads` or `tails` (ignoring case), the program will print an error message and terminate.

The `Math.random` Method

- Simulating a coin flip can be done using the `Math.random` method.
- This method returns a "random" number (technically, a ***pseudorandom*** number) that's greater than or equal to 0.0 and less than 1.0.
- If `Math.random` returns a number less than 0.5, the program will consider the outcome of the coin flip to be heads.

`CoinFlip.java`

```
// Asks the user to guess a coin flip

import java.util.Scanner;

public class CoinFlip {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt user to guess heads or tails
    System.out.print("Enter heads or tails: ");
    String userInput = input.nextLine();
    if (!userInput.equalsIgnoreCase("heads") &&
        !userInput.equalsIgnoreCase("tails")) {
      System.out.println("Sorry, you didn't enter heads " +
                         "or tails; please try again.");
      return;
    }
```



```
    // Choose a random number
    double randomNumber = Math.random();

    // Determine whether user guessed correctly
    if (userInput.equalsIgnoreCase("heads") &&
        randomNumber < 0.5)
      System.out.println("You win!");
    else if (userInput.equalsIgnoreCase("tails") &&
             randomNumber >= 0.5)
      System.out.println("You win!");
    else
      System.out.println("Sorry, you lose.");
  }
}
```

- After reading the user's input, the program ***validates*** it, to make sure that it meets the requirements of the program.

The “Dangling `else`” Problem

- When one `if` statement contains another, the “dangling `else`” problem can sometimes occur.
- Example:

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

- When this statement is executed, the `sum` variable doesn’t change if `n` is larger than `max`, an unexpected outcome.

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.

Ambiguity

- Ambiguity is common in programming languages.
- An expression such as `a + b * c` could mean either `(a + b) * c` or `a + (b * c)`.
- Java resolves ambiguity in expressions by adopting rules for precedence and associativity.
- Ambiguity in expressions can be avoided by using parentheses, just as ambiguity in `if` statements can be avoided by using braces.

4.5 The **`boolean`** Type

- `boolean` expressions are used in `if` statements, but the `boolean` type has other uses.
- Variables and parameters can have `boolean` type, and methods can return `boolean` values.

Declaring `boolean` Variables

- `boolean` variables are ideal for representing data items that have only two possible values.
- In the `CoinFlip` program, the user’s choice (heads or tails) could be recorded in a `boolean` variable:

```
boolean headsWasSelected;
```

- Good names for `boolean` variables often contain a verb such as “is,” “was,” or “has.”

Assigning to a `boolean` Variable

- `boolean` variables can be assigned either `true` or `false`:

```
headsWasSelected = true;
```

- The value assigned to a `boolean` variable often depends on the outcome of a test:

```
if (userInput.equalsIgnoreCase("heads"))
  headsWasSelected = true;
else
  headsWasSelected = false;
```

- There's a better way to get the same effect:

```
headsWasSelected =
  userInput.equalsIgnoreCase("heads");
```


Testing a `boolean` Variable

- An `if` statement can be used to test whether a `boolean` variable is true:

```
if (headsWasSelected) …
```

- Comparing the variable with `true` is unnecessary:

```
if (headsWasSelected == true) …
```

- To test whether `headsWasSelected` is false, the best technique is to write

```
if (!headsWasSelected) …
```

rather than

```
if (headsWasSelected == false) …
```


Displaying the Value of a `boolean` Variable

- Both the `System.out.print` and the `System.out.println` methods are capable of displaying a `boolean` value:

```
System.out.println(headsWasSelected);
```

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

Program: Flipping a Coin (Revisited) `CoinFlip2.java`

```
// Asks the user to guess a coin flip

import java.util.Scanner;

public class CoinFlip2 {
  public static void main(String[] args) {
    boolean headsWasSelected = false;

    Scanner input = new Scanner(System.in);
    // Prompt user to guess heads or tails
    System.out.print("Enter heads or tails: ");
    String userInput = input.nextLine();
    if (userInput.equalsIgnoreCase("heads"))
      headsWasSelected = true;
    else if (!userInput.equalsIgnoreCase("tails")) {
      System.out.println("Sorry, you didn't enter heads " +
                         "or tails; please try again.");
      return;
    }
```


Program: Flipping a Coin (Revisited)

```
    // Choose a random number
    double randomNumber = Math.random();

    // Determine whether user guessed correctly
    if (headsWasSelected && randomNumber < 0.5)
      System.out.println("You win!");
    else if (!headsWasSelected && randomNumber >= 0.5)
      System.out.println("You win!");
    else
      System.out.println("Sorry, you lose.");
  }
}
```


4.6 Loops

- Most algorithms contain steps that require an action to be repeated more than once.
- The recipe for Hollandaise sauce in Chapter 1 contained the following step:

  5. Beat the yolks with a wire whisk until they begin to thicken. Add: 1 tablespoon boiling water.

- The first sentence consists of an *action* to be repeated ("beat the yolks with a wire whisk") and a *condition* to be checked ("[the yolks] begin to thicken").

Loop Terminology

- A language construct that repeatedly performs an action is called a ***loop.***
- In Java, every loop has a statement to be repeated (the ***loop body***) and a condition to be checked (the ***controlling expression***).
- Each time the loop body is executed, the controlling expression is checked. If the expression is true, the loop continues to execute. If it's false, the loop terminates.
- A single cycle of the loop is called an ***iteration.***

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

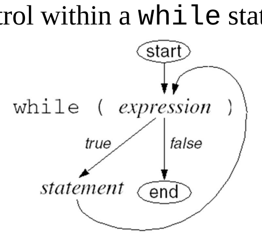
- The `while` statement is the simplest of Java's loop statements.
- Form of the `while` statement:

  `while (` *expression* `)`
  statement
- The controlling expression must have `boolean` type. The loop body can be any statement.
- Example:

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


The `while` Statement

- When a `while` statement is executed, the controlling expression is evaluated first. If it has the value `true`, the loop body is executed and the expression is tested again.
- Flow of control within a `while` statement:



The `while` Statement

- When a `while` statement terminates, the controlling expression is guaranteed to be false. (In other words, the logical negation of the controlling expression must be true.)
- The body of a `while` statement is not executed at all if the controlling expression is false to begin with.

Blocks as Loop Bodies

- Most loops will need more than one statement within the loop body, so the body will have to be a block.
- Consider the problem of finding the greatest common divisor (GCD).
- The 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.
  - The variable can be used only within the block.
  - A variable declared in a 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 constructor should compute the GCD of the fraction's numerator and denominator and then divide both the numerator and the denominator by the GCD.
- The constructor should also adjust the fraction so that the denominator is never negative.

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.

4.7 Counting Loops

- Many loops require a ***counter:*** a variable whose value increases or decreases systematically each time through the loop.
- A loop that reads 10 numbers entered by the user and sums them would need a variable that keeps track of how many numbers the user has entered so far.

A “Countdown” Loop

- Consider the problem of writing a loop that displays a countdown:

```
T minus 10 and counting
T minus 9 and counting
T minus 8 and counting
T minus 7 and counting
T minus 6 and counting
T minus 5 and counting
T minus 4 and counting
T minus 3 and counting
T minus 2 and counting
T minus 1 and counting
```


A “Countdown” Loop

- The countdown loop will need a counter that's assigned the values 10, 9, …, 1:

```
int i = 10;
while (i > 0) {
  System.out.println("T minus " + i +
                     " and counting");
  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.

Counting Up

- A loop that displays the numbers from 1 to `n` along with their squares:

```
int i = 1;
while (i <= n) {
  System.out.println(i + " " + i * i);
  i += 1;
}
```

- Output of the loop if `n` is 5:

```
1 1
2 4
3 9
4 16
5 25
```


Counter Variables

- Variables used as counters should be integers, not floating-point numbers.
- Counters often have names like `i`, `j`, or `k` (but not `l`).
- Using short names for counters is a tradition. Also, there’s often no more meaningful name for a counter than `i`.

Increment and Decrement Operators

- Most loops that have a counter variable will either ***increment*** the variable (add 1 to it) or ***decrement*** the variable (subtract 1 from it).
- 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

- Java has a special operator for incrementing a variable: `++`, the ***increment operator.***
- There are two ways to use `++` to increment a variable:

```
++i;
i++;
```

- When placed before the variable to be incremented, `++` is a ***prefix*** operator.
- When placed after the variable, `++` is a ***postfix*** operator.

Increment and Decrement Operators

- Java also has an operator for decrementing a variable: `--`, the ***decrement operator.***
- The `--` operator can go in front of the variable or after the variable:

```
--i;
i--;
```


Increment and Decrement Operators

- When `++` and `--` are used in isolation, it doesn't matter whether the operator goes before or after the variable.
- When `++` and `--` are used within some other type of statement, 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.

Side Effects

- The `++` and `--` operators don't behave like normal arithmetic operators.
- Evaluating the expression `i + j` doesn't change `i` or `j`. Evaluating `++i` causes a permanent change to `i`, however.
- The `++` and `--` operators are said to have a ***side effect,*** because these operators do more than simply produce a result.

Using the Increment and Decrement Operators in Loops

- The increment and decrement operators are used primarily to update loop counters.
- A modified version of the countdown loop:

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

  Using `--i` instead of `i--` would give the same result.

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;
Scanner input = new Scanner(System.in);
while (i <= 8) {
  System.out.print("Enter Program " + i +
                   " score: ");
  programTotal += input.nextDouble();
  i++;
}
```


Program: Counting Coin Flips

- The `CountFlips` program will flip an imaginary coin any number of times.
- After the user enters the number of flips desired, `CountFlips` will print the number of heads and the number of tails:

```
Enter number of flips: 10
Number of heads: 6
Number of tails: 4
```


A Coin-Flipping Loop

- There are several ways to write a loop that flips a coin `n` times.
- Two possibilities:

Technique 1	***Technique 2***
`int i = 1;`	
`while (i <= n) {`	`while (n > 0) {`
`…`	`…`
`i++;`	`n--;`
`}`	`}`

- The first technique preserves the value of `n`. The second avoids using an additional variable.

`CountFlips.java`

```
// Counts the no. of heads/tails in a series of coin flips

import java.util.Scanner;

public class CountFlips {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt user to enter number of flips
    System.out.print("Enter number of flips: ");
    int flips = input.nextInt();

    // Flip coin for specified number of times
    int heads = 0, tails = 0;
    while (flips > 0) {
      if (Math.random() < 0.5)
        heads++;
      else
        tails++;
      flips--;
    }
```



```
    // Display number of heads and tails
    System.out.println("Number of heads: " + heads);
    System.out.println("Number of tails: " + tails);
  }
}
```


A Possible Modification

- An alternate version of the `while` loop:

```
while (flips-- > 0)
  if (Math.random() < 0.5)
    heads++;
  else
    tails++;
```

4.8 Exiting from a Loop: The `break` Statement

- Java's `break` statement allows a loop to terminate at any point, not just at the beginning.
- Form of the `break` statement:

```
break;
```

- Executing a `break` statement inside the body of a loop causes the loop to terminate immediately.
- Each `break` statement is usually nested inside an `if` statement, so that the enclosing loop will terminate only when a certain condition has been satisfied.

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

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:

```
Scanner input = new Scanner(System.in);
while (true) {
  System.out.print("Enter a number (enter 0 to stop): ");
  int n = input.nextInt();
  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.

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

```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 java.util.Scanner;

public class SSNInfo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt the user to enter an SSN and trim the input
    System.out.print("Enter a Social Security number: ");
    String ssn = input.nextLine().trim();
```



```java
// 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;
  System.out.print("\nPlease re-enter number: ");
  ssn = input.nextLine().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.