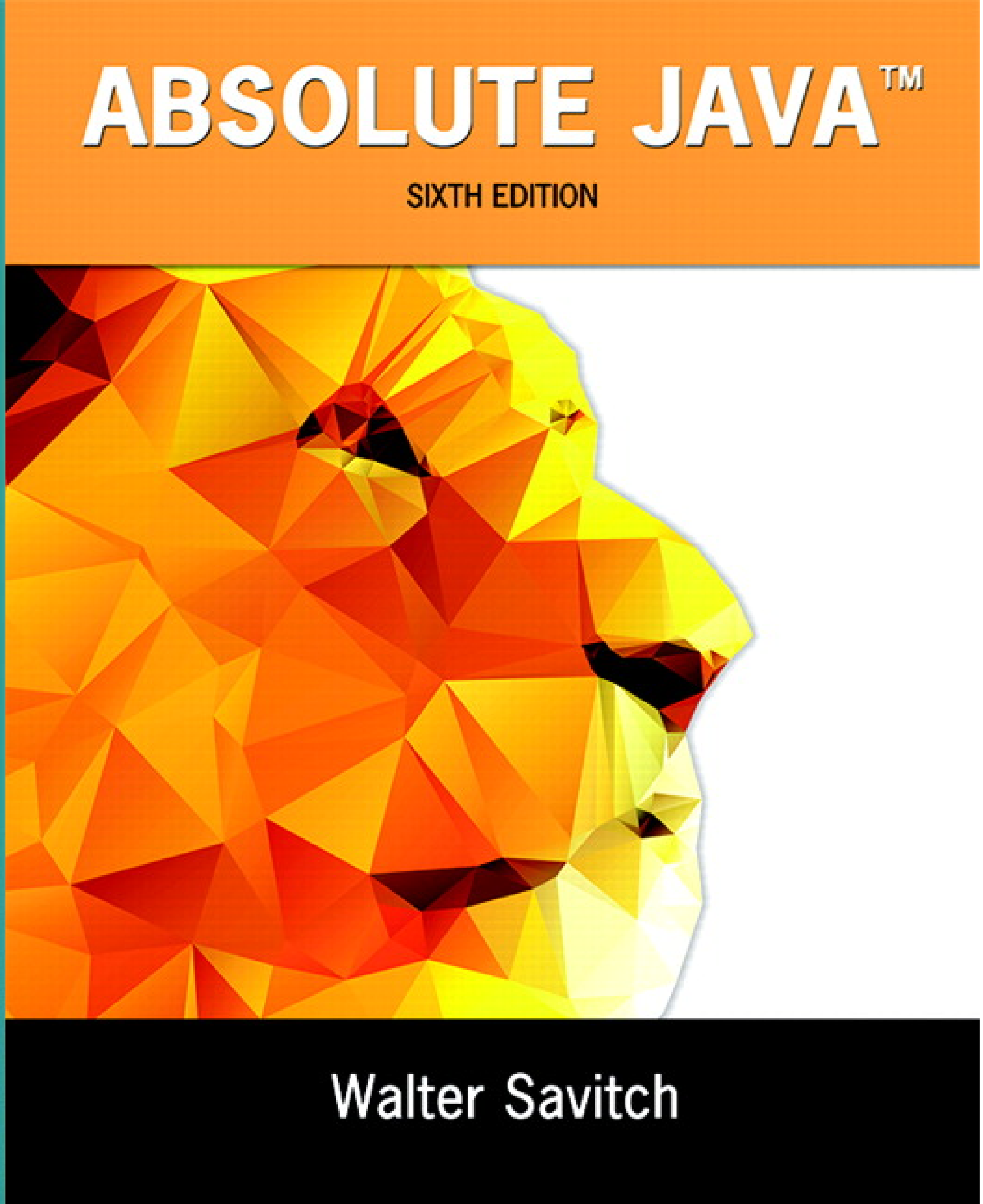


Chapter 3

Flow of Control

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

PEARSON

Flow of Control

- As in most programming languages, *flow of control* in Java refers to its *branching* and *looping* mechanisms
- Java has several branching mechanisms: **`if-else`**, **`if`**, and **`switch`** statements
- Java has three types of loop statements: the **`while`**, **`do-while`**, and **`for`** statements
- Most branching and looping statements are controlled by Boolean expressions
  - A Boolean expression evaluates to either **`true`** or **`false`**
  - The primitive type **`boolean`** may only take the values **`true`** or **`false`**

Branching with an `if-else` Statement

- An `if-else` statement chooses between two alternative statements based on the value of a Boolean expression

```
if (Boolean_Expression)
   Yes_Statement
else
   No_Statement
```

  - The `Boolean_Expression` must be enclosed in parentheses
  - If the `Boolean_Expression` is `true`, then the `Yes_Statement` is executed
  - If the `Boolean_Expression` is false, then the `No_Statement` is executed

Compound Statements

- Each **`Yes_Statement`** and **`No_Statement`** branch of an **`if-else`** can be a made up of a single statement or many statements
- *Compound Statement*: A branch statement that is made up of a list of statements
  - A compound statement must always be enclosed in a pair of braces (**`{ }`**)
  - A compound statement can be used anywhere that a single statement can be used

Compound Statements

```
if (myScore > your Score)
{
   System.out.println("I win!");
   wager = wager + 100;
}
else
{
   System.out.println
             ("I wish these were golf scores.");
   wager = 0;
}
```

Omitting the `else` Part

- The `else` part may be omitted to obtain what is often called an `if` statement

```
if (Boolean_Expression)
  Action_Statement
```

  - If the `Boolean_Expression` is true, then the `Action_Statement` is executed
  - The `Action_Statement` can be a single or compound statement
  - Otherwise, nothing happens, and the program goes on to the next statement

```
if (weight > ideal)
   calorieIntake = calorieIntake - 500;
```

Nested Statements

- `if-else` statements and `if` statements both contain smaller statements within them
  - For example, single or compound statements
- In fact, any statement at all can be used as a subpart of an `if-else` or `if` statement, including another `if-else` or `if` statement
  - Each level of a nested `if-else` or `if` should be indented further than the previous level
  - Exception: *multiway* `if-else` statements

Multiway **`if-else`** Statements

- The multiway **`if-else`** statement is simply a normal **`if-else`** statement that nests another **`if-else`** statement at every **`else`** branch
  - It is indented differently from other nested statements
  - All of the **`Boolean_Expressions`** are aligned with one another, and their corresponding actions are also aligned with one another
  - The **`Boolean_Expressions`** are evaluated in order until one that evaluates to **`true`** is found
  - The final **`else`** is optional

Multiway **`if-else`** Statement

```
if (Boolean_Expression)
   Statement_1
else if (Boolean_Expression)
   Statement_2
          .
          .
          .
else if (Boolean_Expression_n)
   Statement_n
else
  Statement_For_All_Other_Possibilities
```

The `switch` Statement

- The **`switch`** statement is the only other kind of Java statement that implements *multiway* branching
  - When a **`switch`** statement is evaluated, one of a number of different branches is executed
  - The choice of which branch to execute is determined by a *controlling expression* enclosed in parentheses after the keyword **`switch`**
    - The controlling expression must evaluate to a **`char`**, **`int`**, **`short`**, or **`byte`**

The **`switch`** Statement

- Each branch statement in a **`switch`** statement starts with the reserved word **`case`**, followed by a *constant* called a *case label*, followed by a colon, and then a sequence of statements
  - Each case label must be of the same type as the controlling expression
  - Case labels need not be listed in order or span a complete interval, but each one may appear only once
  - Each sequence of statements may be followed by a **`break`** statement (**`break;`**)

The `switch` Statement

- There can also be a section labeled **`default`**:
  - The **`default`** section is optional, and is usually last
  - Even if the case labels cover all possible outcomes in a given **`switch`** statement, it is still a good practice to include a **`default`** section
    - It can be used to output an error message, for example
- When the controlling expression is evaluated, the code for the case label whose value matches the controlling expression is executed
  - If no case label matches, then the only statements executed are those following the **`default`** label (if there is one)

The `switch` Statement

- The `switch` statement ends when it executes a `break` statement, or when the end of the `switch` statement is reached
  - When the computer executes the statements after a case label, it continues until a `break` statement is reached
  - If the `break` statement is omitted, then after executing the code for one case, the computer will go on to execute the code for the next case
  - If the `break` statement is omitted inadvertently, the compiler will not issue an error message

The **`switch`** Statement

```
switch (Controlling_Expression)
{
  case Case_Label_1:
          Statement_Sequence_1
          break;
  case Case_Label_2:
          Statement_Sequence_2
          break;
             :
             .
  case Case_Label_n:
          Statement_Sequence_n
          break;
  default:
          Default_Statement Sequence
          break;
}
```

The Conditional Operator

- The *conditional operator* is a notational variant on certain forms of the **`if-else`** statement
  - Also called the *ternary operator* or *arithmetic if*
  - The following examples are equivalent:

```
if (n1 > n2)    max = n1;
else            max = n2;
```

vs.

```
max = (n1 > n2) ? n1 : n2;
```

  - The expression to the right of the assignment operator is a *conditional operator expression*
  - If the Boolean expression is true, then the expression evaluates to the value of the first expression (**`n1`**), otherwise it evaluates to the value of the second expression (**`n2`**)

Boolean Expressions

- A Boolean expression is an expression that is either **`true`** or **`false`**
- The simplest Boolean expressions compare the value of two expressions

```
time < limit
yourScore == myScore
```

  – Note that Java uses two equal signs (**`==`**) to perform equality testing: A single equal sign (**`=`**) is used only for assignment
  – A Boolean expression does not need to be enclosed in parentheses, unless it is used in an **`if-else`** statement

Java Comparison Operators

Display 3.3 Java Comparison Operators

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	`==`	`x + 7 == 2*y` `answer == 'y'`
≠	Not equal to	`!=`	`score != 0` `answer != 'y'`
>	Greater than	`>`	`time > limit`
≥	Greater than or equal to	`>=`	`age >= 21`
<	Less than	`<`	`pressure < max`
≤	Less than or equal to	`<=`	`time <=limit`

Pitfall: Using == with Strings

- The equality comparison operator (`==`) can correctly test two values of a *primitive* type
- However, when applied to two *objects* such as objects of the `String` class, `==` tests to see if they are stored in the same memory location, not whether or not they have the same value
- In order to test two strings to see if they have equal values, use the method `equals`, or `equalsIgnoreCase`

```
string1.equals(string2)
string1.equalsIgnoreCase(string2)
```

Lexicographic and Alphabetical Order

- *Lexicographic* ordering is the same as *ASCII* ordering, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all uppercase letters come before lowercase letters
  - If `s1` and `s2` are two variables of type `String` that have been given `String` values, then `s1.compareTo(s2)` returns a negative number if `s1` comes before `s2` in lexicographic ordering, returns zero if the two strings are equal, and returns a positive number if `s2` comes before `s1`
- When performing an alphabetic comparison of strings (rather than a lexicographic comparison) that consist of a mix of lowercase and uppercase letters, use the `compareToIgnoreCase` method instead

Building Boolean Expressions

- When two Boolean expressions are combined using the *"and"* (`&&`) operator, the entire expression is true provided both expressions are true
  - Otherwise the expression is false
- When two Boolean expressions are combined using the *"or"* (`||`) operator, the entire expression is true as long as one of the expressions is true
  - The expression is false only if both expressions are false
- Any Boolean expression can be negated using the `!` operator
  - Place the expression in parentheses and place the `!` operator in front of it
- Unlike mathematical notation, strings of inequalities must be joined by `&&`
  - Use `(min < result) && (result < max)` rather than `min < result < max`

Evaluating Boolean Expressions

- Even though Boolean expressions are used to control branch and loop statements, Boolean expressions can exist independently as well
  - A Boolean variable can be given the value of a Boolean expression by using an assignment statement
- A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated
    - The only difference is that arithmetic expressions produce a number as a result, while Boolean expressions produce either **`true`** or **`false`** as their result

```
boolean madeIt = (time < limit) && (limit < max);
```

Truth Tables

Display 3.5 Truth Tables

AND		
Exp_1	*Exp_2*	*Exp_1 && Exp_2*
true	true	true
true	false	false
false	true	false
false	false	false

OR		
Exp_1	*Exp_2*	*Exp_1 || Exp_2*
true	true	true
true	false	true
false	true	true
false	false	false

NOT	
Exp	*!(Exp)*
true	false
false	true

Short-Circuit and Complete Evaluation

- Java can take a shortcut when the evaluation of the first part of a Boolean expression produces a result that evaluation of the second part cannot change
- This is called *short-circuit evaluation* or *lazy evaluation*
  - For example, when evaluating two Boolean subexpressions joined by `&&`, if the first subexpression evaluates to `false`, then the entire expression will evaluate to `false`, no matter the value of the second subexpression
  - In like manner, when evaluating two Boolean subexpressions joined by `||`, if the first subexpression evaluates to `true`, then the entire expression will evaluate to `true`

Short-Circuit and Complete Evaluation

- There are times when using short-circuit evaluation can prevent a *runtime error*
  - In the following example, if the number of `kids` is equal to zero, then the second subexpression will not be evaluated, thus preventing a *divide by zero error*
  - Note that reversing the order of the subexpressions will not prevent this

```
if ((kids !=0) && ((toys/kids) >=2)) . . .
```

- Sometimes it is preferable to always evaluate both expressions, i.e., request complete evaluation
  - In this case, use the `&` and `|` operators instead of `&&` and `||`

Precedence and Associativity Rules

- Boolean and arithmetic expressions need not be fully parenthesized
- If some or all of the parentheses are omitted, Java will follow *precedence* and *associativity* rules (summarized in the following table) to determine the order of operations
  - If one operator occurs higher in the table than another, it has *higher precedence*, and is grouped with its operands before the operator of lower precedence
  - If two operators have the same precedence, then *associativity rules* determine which is grouped first

Precedence and Associativity Rules

Display 3.6 Precedence and Associativity Rules

Highest Precedence

PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation., [ ], ( )	Left to right
++ (postfix, as in x++), -- (postfix)	Right to left
The unary operators: +, -, ++ (prefix, as in ++x), -- (prefix), and !	Right to left
Type casts (Type)	Right to left
The binary operators *, /, %	Left to right
The binary operators +, -	Left to right
The binary operators <, >, <=, >=	Left to right
The binary operators ==, ! =	Left to right
The binary operator &	Left to right
The binary operator |	Left to right
The binary operator &&	Left to right
The binary operator ||	Left to right
The ternary operator (conditional operator) ?:	Right to left
The assignment operators =, *=, /=, %=, +=, -=, & =, |=	Right to left

Lowest Precedence

Evaluating Expressions

- In general, parentheses in an expression help to document the programmer's intent
  - Instead of relying on precedence and associativity rules, it is best to include most parentheses, except where the intended meaning is obvious
- *Binding*: The association of operands with their operators
  - A fully parenthesized expression accomplishes binding for all the operators in an expression
- *Side Effects*: When, in addition to returning a value, an expression changes something, such as the value of a variable
  - The *assignment*, *increment*, and *decrement* operators all produce side effects

Rules for Evaluating Expressions

- Perform binding
  - Determine the equivalent fully parenthesized expression using the precedence and associativity rules
- Proceeding left to right, evaluate whatever subexpressions can be immediately evaluated
  - These subexpressions will be operands or method arguments, e.g., numeric constants or variables
- Evaluate each outer operation and method invocation as soon as all of its operands (i.e., arguments) have been evaluated

Loops

- *Loops* in Java are similar to those in other high-level languages
- Java has three types of loop statements: the **`while`**, the **`do-while`**, and the **`for`** statements
  - The code that is repeated in a loop is called the *body* of the loop
  - Each repetition of the loop body is called an *iteration* of the loop

`while` statement

- A **`while`** statement is used to repeat a portion of code (i.e., the loop body) based on the evaluation of a Boolean expression
  - The Boolean expression is checked *before* the loop body is executed
    - When false, the loop body is not executed at all
  - Before the execution of each following iteration of the loop body, the Boolean expression is checked again
    - If true, the loop body is executed again
    - If false, the loop statement ends
  - The loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (**`{ }`**)

`while` Syntax

```
while (Boolean_Expression)
    Statement
```

Or

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
        .
        .
        .
    Statement_Last
}
```

`do-while` Statement

- A `do-while` statement is used to execute a portion of code (i.e., the loop body), and then repeat it based on the evaluation of a Boolean expression
  - The loop body is executed at least once
    - The Boolean expression is checked *after* the loop body is executed
  - The Boolean expression is checked after each iteration of the loop body
    - If true, the loop body is executed again
    - If false, the loop statement ends
    - Don't forget to put a semicolon after the Boolean expression
  - Like the while statement, the loop body can consist of a single statement, or multiple statements enclosed in a pair of braces (`{ }`)

do-while Syntax

```
do
    Statement
while (Boolean_Expression);
```

Or

```
do
{
    Statement_1
    Statement_2
    .
    .
    .
    Statement_Last
}   while (Boolean_Expression);
```

Equivalence of while and do-while loop

Given the following structure for a `do-while` loop:

```
do
 {
    Statements;
 } while (Boolean condition);
```

The equivalent while loop is:

```
 Statements;
 while (Boolean condition)
 {
    Statements;
 }
```

Equivalence of do-while and while loop

Given the following structure for a `while` loop:

```
while (Boolean condition)
{
   Statements;
}
```

The equivalent `do-while` loop is:

```
if (Boolean condition)
{
    do
    {
       Statements;
    } while (Boolean condition);
}
```

Algorithms and Pseudocode

- The hard part of solving a problem with a computer program is not dealing with the syntax rules of a programming language
- Rather, coming up with the underlying solution method is the most difficult part
- An *algorithm* is a set of precise instructions that lead to a solution
  - An algorithm is normally written in *pseudocode*, which is a mixture of programming language and a human language, like English
  - Pseudocode must be precise and clear enough so that a good programmer can convert it to syntactically correct code
  - However, pseudocode is much less rigid than code: One needn't worry about the fine points of syntax or declaring variables, for example

The `for` Statement

- The `for` statement is most commonly used to step through an integer variable in equal increments
- It begins with the keyword `for`, followed by three expressions in parentheses that describe what to do with one or more *controlling variables*
  - The first expression tells how the control variable or variables are *initialized* or *declared* and *initialized* before the first iteration
  - The second expression determines when the loop should *end*, based on the evaluation of a Boolean expression *before* each iteration
  - The third expression tells how the control variable or variables are *updated after* each iteration of the loop body

The `for` Statement Syntax

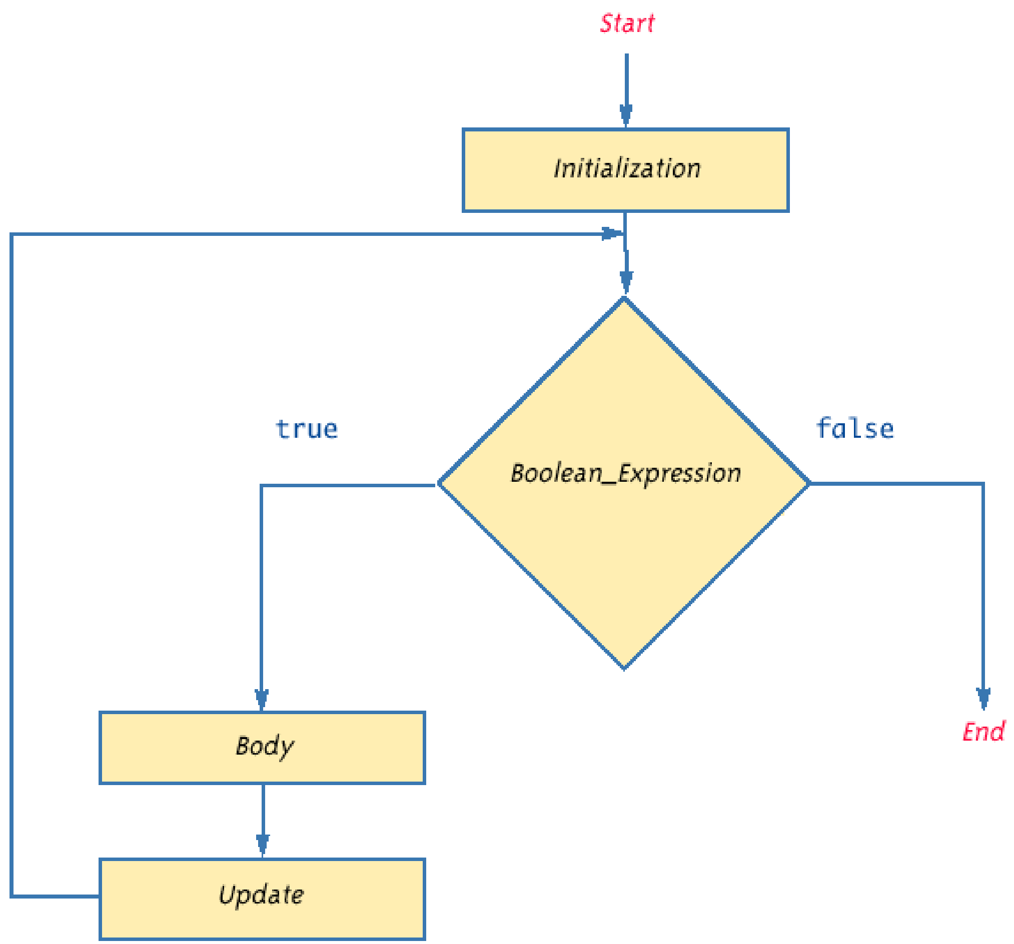
```
for (Initializing; Boolean_Expression; Update)
   Body
```

- The **`Body`** may consist of a single statement or a list of statements enclosed in a pair of braces (**`{  }`**)
- Note that the three control expressions are separated by two, not three, semicolons
- Note that there is no semicolon after the closing parenthesis at the beginning of the loop

Semantics of the `for` Statement

Display 3.9 Semantics of the for Statement

```
for (Initialization; Boolean_Expression; Update)
        Body
```



`for` Statement Syntax and Alternate Semantics

Display 3.10 **for Statement Syntax and Alternate Semantics *(Part 1 of 2)***

for Statement Syntax:

Syntax:

```
for (Initialization; Boolean_Expression; Update)
    Body
```

Example:

```
for (number = 100; number >= 0; number--)
    System.out.println(number
         + " bottles of beer on the shelf.");
```

for Statement Syntax and Alternate Semantics

Display 3.10 **for Statement Syntax and Alternate Semantics *(Part 2 of 2)***

EQUIVALENT while LOOP:

EQUIVALENT SYNTAX:

```
Initialization;
while (Boolean_Expression)
{
    Body
    Update;
}
```

EQUIVALENT EXAMPLE:

```
number = 100;
while (number >= 0)
{
    System.out.println(number
          + " bottles of beer on the shelf.");

    number--;
}
```

SAMPLE DIALOGUE

```
100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
              .
                 .
                 .
0 bottles of beer on the shelf.
```

The Comma in `for` Statements

- A `for` loop can contain multiple initialization actions separated with commas
  - Caution must be used when combining a declaration with multiple actions
  - It is illegal to combine multiple type declarations with multiple actions, for example
  - To avoid possible problems, it is best to declare all variables outside the `for` statement
- A `for` loop can contain multiple update actions, separated with commas, also
  - It is even possible to eliminate the loop body in this way
- However, a `for` loop can contain only one Boolean expression to test for ending the loop

Infinite Loops

- A `while`, `do-while`, or `for` loop should be designed so that the value tested in the Boolean expression is changed in a way that eventually makes it false, and terminates the loop
- If the Boolean expression remains true, then the loop will run forever, resulting in an *infinite loop*
  - Loops that check for equality or inequality (`==` or `!=`) are especially prone to this error and should be avoided if possible

Nested Loops

- Loops can be *nested*, just like other Java structures
  - When nested, the inner loop iterates from beginning to end for each single iteration of the outer loop

```
int rowNum, columnNum;
for (rowNum = 1; rowNum <=3; rowNum++)
{
  for (columnNum = 1; columnNum <=2;
                      columnNum++)
    System.out.print(" row " + rowNum +
                     " column " + columnNum);
  System.out.println();
}
```

The `break` and `continue` Statements

- The `break` statement consists of the keyword `break` followed by a semicolon
  - When executed, the `break` statement ends the nearest enclosing switch or loop statement
- The `continue` statement consists of the keyword `continue` followed by a semicolon
  - When executed, the `continue` statement ends the current loop body iteration of the nearest enclosing loop statement
  - Note that in a `for` loop, the `continue` statement transfers control to the *update* expression
- When loop statements are nested, remember that any `break` or continue statement applies to the innermost, containing loop statement

The Labeled **`break`** Statement

- There is a type of **`break`** statement that, when used in nested loops, can end any containing loop, not just the innermost loop
- If an enclosing loop statement is labeled with an *Identifier,* then the following version of the break statement will exit the labeled loop, even if it is not the innermost enclosing loop:

```
break someIdentifier;
```

- To label a loop, simply precede it with an *Identifier* and a colon:

```
someIdentifier:
```

The `exit` Statement

- A `break` statement will end a loop or switch statement, but will not end the program
- The `exit` statement will immediately end the program as soon as it is invoked:

  ```
  System.exit(0);
  ```

- The `exit` statement takes one integer argument
  - By tradition, a zero argument is used to indicate a normal ending of the program

Loop Bugs

- The two most common kinds of loop errors are unintended *infinite loops* and *off-by-one errors*
  – An off-by-one error is when a loop repeats the loop body one too many or one too few times
    - This usually results from a carelessly designed Boolean test expression
  – Use of **==** in the controlling Boolean expression can lead to an infinite loop or an off-by-one error
    - This sort of testing works only for characters and integers, and should never be used for floating-point

Tracing Variables

- *Tracing variables* involves watching one or more variables change value while a program is running
- This can make it easier to discover errors in a program and debug them
- Many *IDE*s (*Integrated Development Environments*) have a built-in utility that allows variables to be traced without making any changes to the program
- Another way to trace variables is to simply insert temporary output statements in a program

```
System.out.println("n = " + n);  // Tracing n
```

  - When the error is found and corrected, the trace statements can simply be commented out

General Debugging Techniques

- Examine the system as a whole – don’t assume the bug occurs in one particular place
- Try different test cases and check the input values
- Comment out blocks of code to narrow down the offending code
- Check common pitfalls
- Take a break and come back later
- DO NOT make random changes just hoping that the change will fix the problem!

Debugging Example (1 of 9)

- The following code is supposed to present a menu and get user input until either ‘a’ or ‘b’ is entered.

```
String s = "";
char c = ' ';
Scanner keyboard = new Scanner(System.in);

do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s.toLowerCase();
  c = s.substring(0,1);
}
while ((c != 'a') || (c != 'b'));
```

Debugging Example (2 of 9)

Result: Syntax error:

```
c = s.substring(0,1);    : incompatible types
found:  java.lang.String
required: char
```

- Using the “random change” debugging technique we might try to change the data type of `c` to `String`, to make the types match
- This results in more errors since the rest of the code treats `c` like a `char`

Debugging Example (3 of 9)

- First problem: substring returns a String, use charAt to get the first character:

```
String s = "";
char c = ' ';
Scanner keyboard = new Scanner(System.in);

do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') || (c != 'b'));
```

Now the program compiles, but it is stuck in an infinite loop.   Employ tracing:

Debugging Example (4 of 9)

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   System.out.println("String s = " + s);
   s.toLowerCase();
   System.out.println("Lowercase s = " + s);
   c = s.charAt(0);
   System.out.println("c = " + c);
}
while ((c != 'a') || (c != 'b'));
```

Sample output:

```
Enter 'A' for option A or 'B' for option B.
A
String s = A
Lowercase s = A
c = A
Enter 'A' for option A or 'B' for option B.
```

From tracing we can see that the string is never changed to lowercase. Reassign the lowercase string back to `s`.

Debugging Example (5 of 9)

- The following code is supposed to present a menu and get user input until either ‘a’ or ‘b’ is entered.

```
do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s = s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') || (c != 'b'));
```

However, it’s still stuck in an infinite loop. What to try next?

Debugging Example (6 of 9)

- Could try the following "patch"

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   if ( c == 'a')
        break;
   if (c == 'b')
        break;
}
while ((c != 'a') || (c != 'b'));
```

This works, but it is ugly! Considered a coding atrocity, it doesn't fix the underlying problem. The boolean condition after the while loop has also become meaningless. Try more tracing:

Debugging Example (7 of 9)

```
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   System.out.println("c != 'a' is " + (c != 'a'));
   System.out.println("c != 'b' is " + (c != 'b'));
   System.out.println("(c != 'a') || (c != 'b')) is "
                  + ((c != 'a') || (c != 'b')));
}
while ((c != 'a') || (c != 'b'));
```

Sample output:

```
Enter 'A' for option A or 'B' for option B.
A
c != 'a' is false
c != 'b' is true
(c != 'a') || (c != 'b')) is true
```

From the trace we can see that the loop's boolean expression is true because `c` cannot be not equal to `'a'` and not equal to `'b'` at the same time.

Debugging Example (8 of 9)

- Fix: We use && instead of ||

```
do
{
  System.out.println("Enter 'A' for option A or 'B' for option B.");
  s = keyboard.next();
  s = s.toLowerCase();
  c = s.charAt(0);
}
while ((c != 'a') && (c != 'b'));
```

Debugging Example (9 of 9)

- Even better: Declare a boolean variable to control the do-while loop. This makes it clear when the loop exits if we pick a meaningful variable name.

```
boolean invalidKey;
do
{
   System.out.println("Enter 'A' for option A or 'B' for option B.");
   s = keyboard.next();
   s = s.toLowerCase();
   c = s.charAt(0);
   if (c == 'a')
        invalidKey = false;
   else if (c == 'b')
        invalidKey = false;
   else
        invalidKey = true;
}
while (invalidKey);
```

Assertion Checks

- An *assertion* is a sentence that says (asserts) something about the state of a program
  - An assertion must be either true or false, and should be true if a program is working properly
  - Assertions can be placed in a program as comments
- Java has a statement that can check if an assertion is true

```
assert Boolean_Expression;
```

  - If assertion checking is turned on and the `Boolean_Expression` evaluates to `false`, the program ends, and outputs an *assertion failed error message*
  - Otherwise, the program finishes execution normally

Assertion Checks

- A program or other class containing assertions is compiled in the usual way
- After compilation, a program can run with assertion checking turned on or turned off
  - Normally a program runs with assertion checking turned off
- In order to run a program with assertion checking turned on, use the following command (using the actual **`ProgramName`**):

```
java -enableassertions ProgramName
```

Preventive Coding

- Incremental Development
  - Write a little bit of code at a time and test it before moving on
- Code Review
  - Have others look at your code
- Pair Programming
  - Programming in a team, one typing while the other watches, and periodically switch roles

Generating Random Numbers

- The Random class can be used to generate pseudo-random numbers
  - Not truly random, but uniform distribution based on a mathematical function and good enough in most cases
- Add the following import

```
import java.util.Random;
```

- Create an object of type Random

```
Random rnd = new Random();
```

Generating Random Numbers

- To generate random numbers use the nextInt() method to get a random number from 0 to n-1

```
int i = rnd.nextInt(10);   // Random number from 0 to 9
```

- Use the nextDouble() method to get a random number from 0 to 1 (always less than 1)

```
double d = rnd.nextDouble();   // d is >=0 and < 1
```

Simulating a Coin Flip

Display 3.11

```java
import java.util.Random;
public class CoinFlipDemo
{
    public static void main(String[] args)
    {
        Random randomGenerator = new Random();
        int counter = 1;

        while (counter <= 5)
        {
            System.out.print("Flip number " + counter + ": ");
            int coinFlip = randomGenerator.nextInt(2);
            if (coinFlip == 1)
                System.out.println("Heads");
            else
                System.out.println("Tails");
            counter++;
        }
    }
}
```

Sample Dialogue (output will vary)

```
Flip number 1: Heads
Flip number 2: Tails
Flip number 3: Heads
Flip number 4: Heads
Flip number 5: Tails
```