Flow of Control

Chapter 3

Objectives

- learn about Java branching statements
- learn about loops
- learn about the type `boolean`

Flow of Control

- *Flow of control* is the order in which a program performs actions.
  - Up to this point, the order has been sequential.
- A *branching statement* chooses between two or more possible actions.
- A *loop statement* repeats an action until a stopping condition occurs.

Branching Statements: Outline

- The `if-else` Statement
- Introduction to Boolean Expressions
- Nested Statements and Compound Statements
- Multibranch `if-else` Statements
- The `switch` Statament
- (optional) The Conditional Operator

The `if-else` Statement

- A branching statement that chooses between two possible actions.
- syntax

```
if (Boolean_Expression)
  Statement_1
else
  Statement_2
```


The `if-else` Statement, cont.

- example

```
if (count < 3)
    total = 0;
else
    total = total + count;
```

The `if-else` Statement, cont.

- `class BankBalance`

```
import java.util.*;
public class BankBalance
{
    public static final double OVERDRAWN_PENALTY = 8.00;
    public static final double INTEREST_RATE = 0.02;//2% annually
    public static void main(String[] args)
    {
        double balance;
        System.out.print("Enter your checking account balance: $");
        Scanner keyboard = new Scanner(System.in);
        balance = keyboard.nextDouble();
        System.out.println("Original balance $" + balance);
        if (balance >= 0)
            balance = balance + (INTEREST_RATE * balance)/12;
        else
            balance = balance - OVERDRAWN_PENALTY;
        System.out.println("After adjusting for one month");
        System.out.println("of interest and penalties,");
        System.out.println("your new balance is $" + balance);
    }
}
```

Sample Screen Dialog 1

```
Enter your checking account balance: $505.67
Original balance $505.67
After adjusting for one month
of interest and penalties,
your new balance is $506.51278
```

Sample Screen Dialog 2

```
Enter your checking account balance: $-15.53
Original balance $-15.53
After adjusting for one month
of interest and penalties,
your new balance is $-23.53
```

Display 3.1
A Program Using if-else

Compound Statements

- To include multiple statements in a branch, enclose the statements in braces.

```
if (count < 3)
{
    total = 0;
    count = 0;
}
```


Omitting the `else` Part

- If the `else` part is omitted and the expression after the `if` is false, no action occurs.
- syntax

```
if (Boolean_Expression)
    Statement
```

- example

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


Introduction to Boolean Expressions

- The value of a *boolean expression* is either `true` or `false`.
- examples

```
time < limit
balance <= 0
```


Java Comparison Operators

Math Notation	Name	Java Notation	Java Examples
=	Equal to	==	`balance == 0` `answer == 'y'`
≠	Not equal to	!=	`income != tax` `answer != 'y'`
>	Greater than	>	`expenses > income`
≥	Greater than or equal to	>=	`points >= 60`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`expenses <= income`

Display 3.2
Java Comparison Operators

Compound Boolean Expressions

- Boolean expressions can be combined using the “and” (`&&`) operator.
- example

```
if ((score > 0) && (score <= 100))
...
```

- not allowed

```
if (0 < score <= 100)
...
```

Compound Boolean Expressions, cont.

- syntax

```
(Sub_Expression_1) && (Sub_Expression_2)
```

- Parentheses often are used to enhance readability.
- The larger expression is true only when both of the smaller expressions are true.

Compound Boolean Expressions, cont.

- Boolean expressions can be combined using the "or" (`||`) operator.
- example

```
if ((quantity > 5) || (cost < 10))
...
```

- syntax

```
(Sub_Expression_1) || (Sub_Expression_2)
```


Compound Boolean Expressions, cont.

- The larger expression is true
  - when either of the smaller expressions is true
  - when both of the smaller expressions are true.
- "or" in Java is *inclusive or*
  - either or both to be true.
- *exclusive or*
  - one or the other, but not both to be true.

Negating a Boolean Expression

- Boolean negation
  - "not" (`!`) operator.
- syntax

```
!Boolean_Expression
```

- Example:

```
Boolean walk = false;
System.out.println(!walk);
```


Truth Tables

&& (and)

Value of *A*	Value of *B*	Resulting Value of *A* && *B*
true	true	true
true	false	false
false	true	false
false	false	false

|| (or)

Value of *A*	Value of *B*	Resulting Value of *A* || *B*
true	true	true
true	false	true
false	true	true
false	false	false

! (not)

Value of *A*	Resulting Value of !(*A*)
true	false
false	true

Display 3.15
Truth Tables for Boolean Operators

Primary Logical Operators

- Primary logical operators: and, or, not
- **Any** logical expression can be composed
- Example: *exclusive or*

```
(a || b) && !(a && b)
```

- Either work or play:

```
(work || play) && !(work && play)
```

- `^` is exclusive-or in Java
  - `work ^ play`
  - not a logical operator in most languages

Using ==

- == is appropriate for determining if two integers or characters have the same value.
  `if (a == 3)`
  where `a` is an integer type
- == is not appropriate for determining if two floating point values are equal.
  - Use < and some appropriate tolerance instead.
    `if (Math.abs(b - c) < epsilon)`
  - `b`, `c`, and `epsilon` are of floating point type

[www.cs.fit.edu/~pkc/classes/cse1001/FloatEquality.java]

Using ==, cont.

- == is not appropriate for determining if two objects have the same value.
  - `if (s1 == s2)`
    - determines only if s1 and s2 are at the same memory location.
  - If `s1` and `s2` refer to strings with identical sequences of characters, but stored in different memory locations
    - `(s1 == s2)` is false.

Using ==, cont.

- To test the equality of objects of class String, use method `equals`.
  `s1.equals(s2)`
  or
  `s2.equals(s1)`
  www.cs.fit.edu/~pkc/classes/cse1001/StringEqual.java
- To test for equality ignoring case, use method `equalsIgnoreCase`.
  `("Hello".equalsIgnoreCase("hello"))`

`equals` and `equalsIgnoreCase`

- syntax
  `String``.equals(`*`Other_String`*`)`
  `String``.equalsIgnoreCase(`*`Other_String`*`)`

Testing Strings for Equality

- `class StringEqualityDemo`

```
import java.util.*;

public class StringEqualityDemo
{
    public static void main(String[] args)
    {
        String s1, s2;

        System.out.println("Enter two lines of text:");

        Scanner keyboard = new Scanner(System.in);
        s1 = keyboard.nextLine();
        s2 = keyboard.nextLine();

        if (s1.equals(s2))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");

        if (s2.equals(s1))
            System.out.println("The two lines are equal.");
        else
            System.out.println("The two lines are not equal.");

        if (s1.equalsIgnoreCase(s2))
            System.out.println(
                        "But the lines are equal, ignoring case.");
        else
            System.out.println(
                        "Lines are not equal, even ignoring case.");
    }
}
```

There two invocations of the method equals are equivalent.

Sample Screen Dialog

```
Enter two lines of text:
Java is not coffee.
Java is NOT COFFEE.
The two lines are not equal.
The two lines are not equal.
But the lines are equal, ignoring case.
```

Display 3.3
Testing Strings for Equality

Lexicographic Order

- Lexicographic order is similar to alphabetical order, but is it based on the order of the characters in the ASCII (and Unicode) character set.
  - All the digits come before all the letters.
  - All the uppercase letters come before all the lower case letters.

Lexicographic Order, cont.

- Strings consisting of alphabetical characters can be compared using method `compareTo` and method `toUpperCase` or method `toLowerCase`.

```
String s1 = "Hello";
String lowerS1 = s1.toLowerCase();
String s2 = "hello";
if (lowerS1.compareTo(s2) == 0)
    System.out.println("Equal!");

//or use s1.compareToIgnoreCase(s2)
```


Method `compareTo`

- syntax

  String_1`.compareTo(`*String_2*`)`
- Method compareTo returns
  - a negative number if *String_1* precedes *String_2*
  - zero if the two strings are equal
  - a positive number of *String_2* precedes *String_1*
  - Tip: Think of `compareTo` is subtraction

Comparing Numbers vs. Comparing Strings

Integer and floating-point values	String objects
== !=	`equals( )` `equalsIgnoreCase( )`
> < >= <=	`compareTo( )` `[lexicographical ordering]`

Nested Statements

- An `if-else` statement can contain any sort of statement within it.
- In particular, it can contain another `if-else` statement.
  - An `if-else` may be nested within the "if" part.
  - An `if-else` may be nested within the "else" part.
  - An `if-else` may be nested within both parts.

Nested Statements, cont.

- syntax

```
if (Boolean_Expression_1)
  if (Boolean_Expression_2)
    Statement_1
  else
    Statement_2
else
  if (Boolean_Expression_3)
    Statement_3
  else
    Statement_4
```


Nested `if` Example

```
if (temperature > 90)  // int temperature
  if (sunny)           // boolean sunny
    System.out.println("Beach");
  else
    System.out.println("Movie");
else
  if (sunny)
    System.out.println("Tennis");
  else
    System.out.println("Volleyball");
```

Nested Statements, cont.

- Each `else` is paired with the **nearest unmatched** `if`.
- Indentation can communicate which `if` goes with which `else`.
- Braces are used to group statements.

Nested Statements, cont.

- Different indentation

first form

```
if (a > b)
  if (c > d)
    e = f;
else
  g = h;
```

second form

```
if (a > b)
  if (c > d)
    e = f;
  else
    g = h;
```

Same to the compiler!

Nested Statements, cont.

- Are these different?

first form

```
if (a > b)
{
  if (c > d)
    e = f;
}
else
    g = h;
```

second form

```
if (a > b)
  if (c > d)
    e = f;
else
  g =h;
```


Nested Statements, cont.

- Proper indentation and nested if-else statements

"else" with outer "if"

```
if (a > b)
{
  if (c > d)
    e = f;
}
else
    g = h;
```

"else" with inner "if"

```
if (a > b)
  if (c > d)
    e = f;
  else
    g =h;
```


Compound Statements

- When a list of statements is enclosed in braces (`{}`), they form a single *compound statement.*
- syntax

```
{
    Statement_1;
    Statement_2;
   …
}
```


Compound Statements, cont.

- A compound statement can be used wherever a statement can be used.
- example

```
if (total > 10)
{
   sum = sum + total;
   total = 0;
}
```

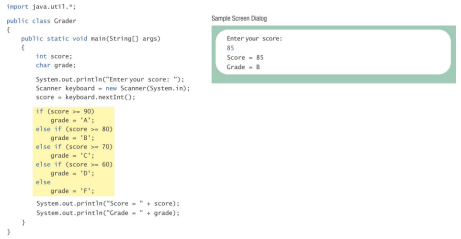
Multibranch `if-else` Statements

- syntax

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
else if (Boolean_Expression_3)
    Statement_3
else if …
else
    Default_Statement
```


Multibranch `if-else` Statements, cont.

- `class Grader`



Display 3.4
Multibranch if-else Statement

Multibranch `if-else` Statements, cont.

- equivalent logically

```
if (score >= 90)
  grade = 'A';
if ((score >= 80) && (score < 90))
  grade = 'B';
if ((score >= 70) && (score < 80))
  grade = 'C';
if ((score >= 60) && (score < 70))
  grade = 'D';
if (score < 60)
  grade = 'F';
```


`switch` Statement

- The `switch` statement is a multiway branch that makes a decision based on an *integral* (integer or character) expression.
- The `switch` statement begins with the keyword `switch` followed by an integral expression in parentheses and called the *controlling expression.*

`switch` Statement, cont.

- A list of cases follows, enclosed in braces.
- Each case consists of the keyword `case` followed by
  - a constant called the *case label*
  - a colon
  - a list of statements.
- The list is searched for a case label matching the controlling expression.

`switch` Statement, cont.

- The action associated with a matching case label is executed.
- If no match is found, the case labeled `default` is executed.
  - The `default` case is optional, but recommended, even if it simply prints a message.
- Repeated case labels are not allowed.

`switch` Statement, cont.

- `class MultipleBirths`

`switch` Statement, cont.

- The action for each case typically ends with the word `break`.
- The optional `break` statement prevents the consideration of other cases.
- The controlling expression can be anything that evaluates to an integral type (integer or character).

The `switch` Statement, cont.

- syntax

```
switch (Controlling_Expression)
{
   case Case_Label:
         Statement(s);
         break;
   case Case_Label:
   …
   default:
   …
}
```


Switch with `char` Type

```
char grade = 'A';
switch(grade)
{
    case 'A':
    case 'B':
    case 'C':
    case 'D':
        System.out.println("Pass");
        break;
    case 'W':
        System.out.println("Withdraw");
        break;
    case 'I':
        System.out.println("Incomplete");
        break;
    default:
        System.out.println("Fail");
}
```


Conditional Operator

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

can be written as

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

- The `?` and `:` together is called the *conditional operator* (a *ternary* operator).
- Note `(n1 > n2) ? n1 : n2` is an *expression* that has a value unlike the "normal" if statement

Conditional Operator, cont.

- The conditional operator can be useful with print statements.

```
System.out.print("You worked " + hours + " " +
   ((hours > 1) ? "hours" : "hour"));
```

Summary of branching

- `if` statement (1 or 2 branches)
- Multi-branch `if-else-if` statement (3 or more branches)
- Multi-branch `switch` statement
- Conditional operator `? :`

Loop Statements

- A portion of a program that repeats a statement or a group of statements is called a *loop.*
- The statement or group of statements to be repeated is called the *body* of the loop.
- A loop could be used to compute grades for each student in a class.
- There must be a means of exiting the loop.

Loop Structure

1. Control of loop: ***ICU***
   1. ***Initialization***
   2. ***Condition for termination (continuing)***
   3. ***Updating the condition***
2. Body of loop

Loop Statements

- the `while` Statement
- the `do-while` Statement
- the `for` Statement

`while` Statement

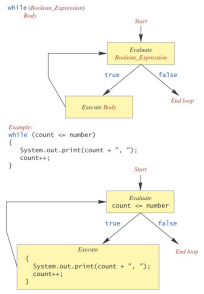
- also called a `while` loop
- a controlling boolean expression
  - True -> repeats the statements in the loop body
  - False -> stops the loop
  - Initially false (the very first time)
    - loop body will not even execute once

`while` Statement, cont.

- syntax

```
while (Boolean_Expression)
   Body_Statement
or
while (Boolean_Expression)
{
   First_Statement
   Second_Statement
   …
}
```

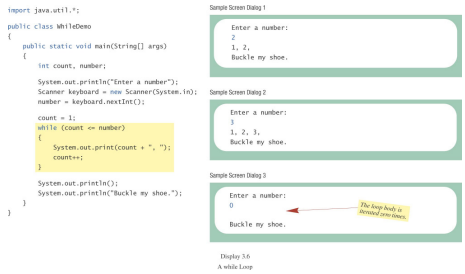
while Statement, cont.



Display 3.7
Semantics of the while Statement

while Statement, cont.

- `class WhileDemo`



Display 3.6
A while Loop

do-while Statement

- also called a `do-while` loop (repeat-until loop)
- similar to a `while` statement
  - except that the loop body is executed at least once
- syntax

```
do
    Body_Statement
while (Boolean_Expression);
```

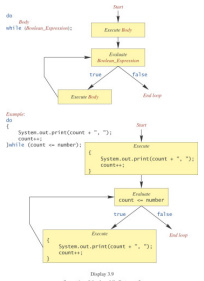
  - **don't forget the semicolon at the end!**

do-while Statement, cont.

- First, the loop body is executed.
- Then the boolean expression is checked.
  - As long as it is true, the loop is executed again.
  - If it is false, the loop exits.
- equivalent `while` statement

```
Statement(s)_S1
while (Boolean_Condition)
    Statement(s)_S1
```

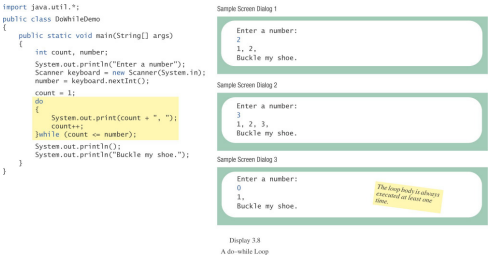

do-while Statement, cont.



Display 3.9
Semantics of the do-while Statement

do-while Statement, cont.

- `class DoWhileDemo`



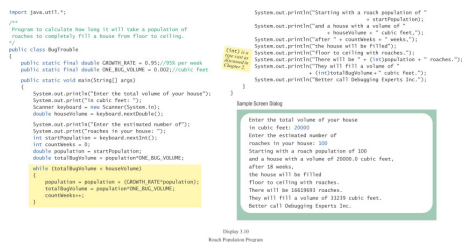
Display 3.8
A do-while Loop

Programming Example: Bug Infestation

- given
  - volume of a roach: 0.0002 cubic feet
  - starting roach population
  - rate of increase: 95%/week
  - volume of a house
- find
  - number of weeks to exceed the capacity of the house
  - number and volume of roaches

Programming Example: Bug Infestation, cont.

- `class BugTrouble`



Infinite Loops

- A loop which repeats without ever ending
- the controlling boolean expression (condition to continue)
  - never becomes false
- A negative growth rate in the preceding problem causes `totalBugVolume` always to be less than `houseVolume`
  - the loop never ends.

`for` Statement

- A `for` statement executes the body of a loop a fixed number of times.
- example

```
for (count = 1; count < 3; count++)
     System.out.println(count);
System.out.println("Done");
```


`for` Statement, cont.

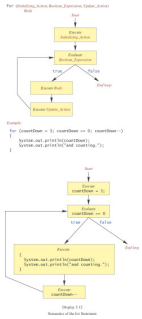
- syntax

```
for (Initialization; Condition; Update)
   Body_Statement
```

  - *Body_Statement*
    - a simple statement or
    - a compound statement in `{}`.
- corresponding `while` statement

```
Initialization
while (Condition)
   Body_Statement_Including_Update
```


`for` Statement, cont.



`for` Statement, cont.

- `class ForDemo`

```
public class ForDemo
{
    public static void main(String[] args)
    {
        int countDown;
        for (countDown = 3; countDown >= 0; countDown  )
        {
            System.out.println(countDown);
            System.out.println("and counting.");
        }
        System.out.println("Blast off!");
    }
}
```

Screen Output

```
3
and counting.
2
and counting.
1
and counting.
0
and counting.
Blast off!
```

Display 3.11
A for Statement

Multiple Initialization, etc.

- example

```
for (n = 1, p = 1; n < 10; n++)
    p = p * n
```

- Only one boolean expression is allowed, but it can consist of `&&`s, `||`s, and `!`s.
- Multiple update actions are allowed, too.

```
for (n = 1, p = 100; n < p; n++, p -= n)
```

- rarely used

Choosing a Loop Statement

- If you know how many times the loop will be iterated, use a `for` loop.
- If you don't know how many times the loop will be iterated, but
  - it could be zero, use a `while` loop
  - it will be at least once, use a `do-while` loop.
- Generally, a `while` loop is a safe choice.

Summary of loop statements

- `while` loop
- `do-while` loop
- `for` loop

`break` Statement in Loops: NOT recommended

- A `break` statement can be used to end a loop immediately.
- The `break` statement ends only the **innermost** loop that contains the `break` statement.
- `break` statements make loops **more difficult** to understand:
  - Loop could end at different places (multiple possible exit points), harder to know where.
- Always try to end a loop at only one place--makes debugging easier (only one possible exit point)

Misuse of `break` Statements in loops (p. 177)

- "Because of the complications they introduce, `break` statements in loops should be avoided.
- Some authorities contend that a `break` statement should never be used to end a loop,
- but virtually all programming authorities agree that they should be used at most sparingly."

`exit` Method

- Sometimes a situation arises that makes continuing the program pointless.
- A program can be terminated normally by `System.exit(0).`
- example

```
if (numberOfWinners == 0)
{
   System.out.println("/ by 0");
   System.exit(0);
}
```


Programming with Loops: Outline

- The Loop Body
- Initializing Statements
- Ending a Loop
- Loop Bugs
- Tracing Variables

Loop Body

- To design the loop body, write out the actions the code must accomplish.
- Then look for a repeated pattern.
  - The pattern need not start with the first action.
  - The repeated pattern will form the body of the loop.
  - Some actions may need to be done after the pattern stops repeating.

Initializing Statements

- Some variables need to have a value before the loop begins.
  - Sometimes this is determined by what is supposed to happen after one loop iteration.
  - Often variables have an initial value of zero or one, but not always.
- Other variables get values only while the loop is iterating.

Ending a Loop

- If the number of iterations is known before the loop starts, the loop is called a *count-controlled loop.*
  - use a `for` loop.
- Asking the user before each iteration if it is time to end the loop is called the *ask-before-iterating technique.*
  - appropriate for a small number of iterations
  - Use a `while` loop or a `do-while` loop.

Ending a Loop, cont.

- For large input lists, a *sentinel value* can be used to signal the end of the list.
  - The sentinel value must be different from all the other possible inputs.
  - A negative number following a long list of nonnegative exam scores could be suitable.

```
90
0
10
-1
```

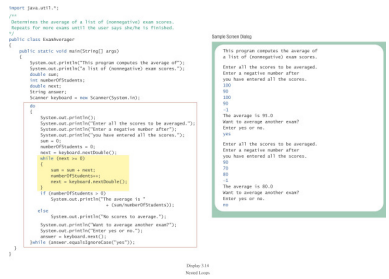
Ending a Loop, cont.

- example - reading a list of scores followed by a sentinel value

```
int next = keyboard.nextInt();
while (next  >= 0)
{
   Process_The_Score
   next = keyboard.nextInt();
}
```


Ending a Loop, cont.

- `class ExamAverager`



Nested Loops

- The body of a loop can contain any kind of statements, including another loop.
- In the previous example
  - the average score was computed using a `while` loop.
  - This `while` loop was placed inside a `do-while` loop so the process could be repeated for other sets of exam scores.

Nested Loops

- The body of a loop can have any kind of statements, including another loop.

```
for (line = 0; line < 4; line++)
{
   for (star = 0; star < 5; star++)
      System.out.print('*');
   System.out.println();
}
```

body of outer loop

body of inner loop

- Each time the outer loop body is executed, the inner loop body will execute 5 times.
- 20 times total

Output:

```
*****
*****
*****
*****
```


Declaring Variables Outside Loop Bodies

- Declaration of variables **inside** a loop body is repeated with each execution of the loop body--can be inefficient
- Declaration of variables can generally be moved outside the loop body.

```
while (…)
{
   int x;
   …
}
```


Loop Bugs

- common loop bugs
  - unintended infinite loops
  - off-by-one errors
  - testing equality of floating-point numbers
- subtle infinite loops
  - The loop may terminate for some input values, but not for others.
  - For example, you can't get out of debt when the monthly penalty exceeds the monthly payment.

Off-by-One Errors

- The loop body is repeated one too many times or one too few times.
- examples
  - < is used when <= should be used or <= is used when < should be used
  - using the index of the last character of a string instead of the length of the string (or vice versa)
- easy to overlook

Off by One

```
int i = 0;
while (i <= 10)
{
  System.out.println(i);
  i++;
}
```


Subtle Infinite Loops

- Verify that the monthly payment exceeds the penalty, for example, before entering a loop to determine the number of payments needed to get out of debt.

```
if (payment <= penalty)
   System.out.println("payment is too
   small");
else
{
  ...
```


Empty `for` Statement

- What is printed by

```
int product = 1, number;
for (number = 1; number <= 10; number++);
    product = product * number;
System.out.println(product);
```

- The last semicolon in

```
for (number = 1; number <= 10; number++);
```

produces an empty `for` statement.

Empty `while` Statement

```
int product = 1, number = 1;
while (number <= 10);
{
    product = product * number;
    number++;
}
System.out.println(product);
```

- The last semicolon in

```
while (number <= 10);
```

produces an empty `while` loop body.

Testing Equality of Floating-point Numbers

- == works satisfactorily for integers and characters.
- == is not reliable for floating-point numbers (which are approximate quantities).
  - Can cause infinite loops
  - Use <= or >= rather than == or !=.
- `www.cs.fit.edu/~pkc/classes/cse1001/FloatEquality.java`

Tracing Variables

- *Tracing variables* means watching the variables change while the program is running.
  - Simply insert temporary output statements in your program to print of the values of variables of interest
  - or, learn to use the debugging facility that may be provided by your system.

Tracing Variables, cont.

```
float creditCardBalance = 9000.0;
while (creditCardBalance > 0)
{
  … // input payment
  creditCardBalance -= payment;

  … // calculate penalty
  creditCardBalance += penalty;

  system.out.println(creditCardBalance);
}
```


Type `boolean`

- Boolean Expressions and Variables
- Truth Tables and Precedence Rules
- Input and Output of Boolean Values

Type `boolean`, cont.

- The type `boolean` is a primitive type with only two values: `true` and `false`.
- Boolean variables can make programs more readable.

```
if (systemsAreOK)
```

instead of

```
if((temperature <= 100) && (thrust >= 12000)
  && (cabinPressure > 30) && …)
```


Boolean Expressions and Variables

- Variables, constants, and expressions of type `boolean` all evaluate to either `true` or `false`.
- A boolean variable can be given the value of a boolean expression by using an assignment operator.

```
boolean isPositive = (number > 0);
...
if (isPositive) ...
```


Naming Boolean Variables

- Choose names such as `isPositive` or `systemsAreOk`.
- Avoid names such as `numberSign` or `systemStatus`.

Precedence Rules

- Parentheses should be used to indicate the order of operations.
- When parentheses are omitted, the order of operation is determined by *precedence rules.*

Precedence Rules, cont.

- Operations with *higher precedence* are performed before operations with *lower precedence.*
- Operations with *equal precedence* are done left-to-right (except for unary operations which are done right-to-left).

Precedence Rules, cont.

Highest Precedence

First: the unary operators +, –, ++, – –, and !
Second: the binary arithmetic operators *, /, %
Third: the binary arithmetic operators +, –
Fourth: the boolean operators <, >, <=, >=
Fifth: the boolean operators ==, !=
Sixth: the boolean operator &
Seventh: the boolean operator |
Eighth: the boolean operator &&
Ninth: the boolean operator ||

Lowest Precedence

Comparison operators:
<, >, <=, >=
==, !=

Logical operators:
&
|
&&
||

Display 3.16
Precedence Rules

Precedence Rules, cont.

- In what order are the operations performed?

```
score < min/2 – 10 || score > 90
score < (min/2) – 10 || score > 90
score < ((min/2) – 10) || score > 90
(score < ((min/2) – 10)) || score > 90
(score < ((min/2) – 10)) || (score > 90)
```


Short-circuit Evaluation

- Sometimes only *part* of a boolean expression needs to be evaluated to determine the value of the entire expression.
  - If the first operand of `||` is `true`
    - entire expression is `true`
  - If the first operand of `&&` is `false`
    - entire expression is `false`
- This is called *short-circuit* or *lazy* evaluation.

Short-circuit Evaluation, cont.

- Short-circuit evaluation is not only efficient, sometimes it is essential!
- A run-time error can result, for example, from an attempt to divide by zero.

```
if ((number != 0) && (sum/number > 5))
```

- *Complete evaluation* can be achieved by substituting `&` for `&&` or `|` for `||`.

Short-circuit Evaluation

```
int count = 1;
…
if ( …  && (++count < 10) )
{
  …
}
System.out.println(count);
```


Input and Output of Boolean Values

- example

```
boolean boo = false;
System.out.println(boo);
System.out.print("Enter a boolean value: ");
Scanner keyboard = new Scanner (System.in);
boo = keyboard.nextBoolean();
System.out.println(boo);
```


Input and Output of Boolean Values, cont.

- dialog

```
false
Enter a boolean value: true
true
```


Using a Boolean Variable to End a Loop

- example

```
boolean numbersLeftToRead = true;
while (numbersLeftToRead)
{
   next = keyboard.nextInt();
   if (next < 0)
         numbersLeftToRead = false;
   else
         Process_Next_Number
}
```


Using a Boolean Variable to End a Loop, cont

- class BooleanDemo

```
import java.util.*;

/**
  Illustrates the use of a boolean variable to control loop ending.
*/
public class BooleanDemo
{
    public static void main(String[] args)
    {
        System.out.println("Enter nonnegative numbers.");
        System.out.println("Place a negative number at the end");
        System.out.println("to serve as an end marker.");

        int next, sum = 0;
        boolean numbersLeft = true;
        Scanner keyboard = new Scanner(System.in);
        while (numbersLeft)
        {
            next = keyboard.nextInt();
            if (next < 0)
                numbersLeft = false;
            else
                sum = sum + next;
        }
        System.out.println("The sum of the numbers is " + sum);
    }
}
```

Sample Screen Dialog

```
Enter nonnegative numbers.
Place a negative number at the end
to serve as an end marker.
1 2 3 -1
The sum of the numbers is 6
```

Display 3.17
Use of a Boolean Variable to End a Loop

Summary

- You have learned about Java branching statements.
- You have learned about loops.
- You have learned about the type `boolean`.