Flow of Control

Chapter 3
Modified by James O’Reilly
Objectives

• Compare values of primitive types
• Compare objects such as strings
• Use the primitive type `boolean`
• Use Boolean Operators
• Use Java branching statements
  • `if, if-else, if-else if-else, etc.`
  • `switch`
• Use simple enumerations in a program
Introduction to Boolean Expressions

• The value of a boolean expression is either true or false.

• Examples
  - time < limit
  - balance <= 0
### Java Comparison Operators

**Figure 3.4 Java Comparison Operators**

<table>
<thead>
<tr>
<th>Math Notation</th>
<th>Name</th>
<th>Java Notation</th>
<th>Java Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
<td>==</td>
<td>balance == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer == 'y'</td>
</tr>
<tr>
<td>≠</td>
<td>Not equal to</td>
<td>!=</td>
<td>income != tax</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>answer != 'y'</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;</td>
<td>expenses &gt; income</td>
</tr>
<tr>
<td>≥</td>
<td>Greater than or equal to</td>
<td>&gt;=</td>
<td>points &gt;= 60</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>&lt;</td>
<td>pressure &lt; max</td>
</tr>
<tr>
<td>≤</td>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>expenses &lt;= income</td>
</tr>
</tbody>
</table>
Compound Boolean Expressions with &&

• Boolean expressions can be combined using the "and" (&&) operator.
• True if both operands are true
• False otherwise
• Example (result of comparison is a boolean value)
  
  if ((score > 0) && (score <= 100))
  ...

• Not allowed
  
  if (0 < score <= 100)
  ...
Compound Boolean Expressions with \( \| \| \)

• Boolean expressions can be combined using the "or" \( (\|\|) \) operator.
• True if either operand is true
• False otherwise
• Example
  
  \[
  \text{if } ((\text{quantity} > 5) \| (\text{cost} < 10)) \\
  \ldots
  \]

• Syntax
  
  \[
  (\text{Sub	extunderscore Expression}_1) \| (\text{Sub	extunderscore Expression}_2)
  \]
Negating a Boolean Expression

• A boolean expression can be negated using the "not" (!) operator.

• Syntax

  !(Boolean_Expression)

• Example

  (a || b) && !(a && b)
  which is exclusive or (xor)
Negating a Boolean Expression

• Figure 3.5 Avoiding the Negation Operator
• Why?

! (A Op B) Is Equivalent to (A Op B)

| !< | != |
| !<= | != |
| !> | != |
| !>= | != |
| !== | != |

Why?

Java Logical Operator examples

- Figure 3.6

<table>
<thead>
<tr>
<th>Name</th>
<th>Java Notation</th>
<th>Java Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical <em>and</em></td>
<td><code>&amp;&amp;</code></td>
<td><code>(sum &gt; min) &amp;&amp; (sum &lt; max)</code></td>
</tr>
<tr>
<td>Logical <em>or</em></td>
<td>`</td>
<td></td>
</tr>
<tr>
<td>Logical <em>not</em></td>
<td><code>!</code></td>
<td><code>!(number &lt; 0)</code></td>
</tr>
</tbody>
</table>
Boolean Operator Truth Table

- FIGURE 3.7 The Effect of the Boolean Operators \&\& (and), || (or), and ! (not) on Boolean values

| Value of $A$ | Value of $B$ | Value of $A$ \&\& $B$ | Value of $A$ || $B$ | Value of ! ($A$) |
|--------------|--------------|-----------------------|----------------|----------------|------------------|
| true         | true         | true                  | true           | true           | false            |
| true         | false        | false                 | true           | true           | false            |
| false        | true         | false                 | true           | true           | true             |
| false        | false        | false                 | false          | true           | true             |
Using `==`

- `==` is appropriate for determining if two integers or characters have the same value.
  
  ```java
  if (a == 3)
  where a is an integer type
  ```

- `==` is **not** appropriate for determining if two floating points values are equal. Use `<` and some appropriate tolerance instead because of rounding error (remember this – common pitfall for newer programmers).
  
  ```java
  if (abs(b - c) < epsilon)
  where b, c, and epsilon are floating point types
  ```
Using `==`

- `==` is **not** appropriate for determining if two Objects, anything not a primitive type, have the same value.
  - `if (s1 == s2)`, where `s1` and `s2` refer to strings, or other Objects, determines only if `s1` and `s2` refer a common memory location.
  - If `s1` and `s2` refer to strings or other objects with identical contents, but stored in different memory locations, `(s1 == s2)` is false.
  - This is (very) occasionally used for comparison but we have to *guarantee* that memory address equality implies equality.
Comparing **Strings** (short version: don’t use `==`)

- `==` compares memory addresses (whether they are the exact same Object). This works in special contexts but not in general.

- To test the equality of objects of class String, use method `equals`.
  
  ```java
  s1.equals(s2)
  or
  s2.equals(s1)
  ```

- To test for equality ignoring case, use method `equalsIgnoreCase`.
  
  ```java
  ("Hello".equalsIgnoreCase("hello"))
  ```
Comparing **Strings** (short version: don’t use `==`)

```java
System.out.println("Enter two values, a and b:");
String a = kb.nextLine(); //kb is a Scanner
String b = kb.nextLine();
System.out.println("a==b is " + (a==b));
System.out.println("a.equals(b) is " + a.equals(b));
```

```
Enter two values, a and b:

<table>
<thead>
<tr>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
</tr>
</tbody>
</table>
```

`a==b` is false
`a.equals(b)` is true
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.
  • Very useful to compare things in programming (and establish an order like with less than or greater than ( < or > )— need it to sort efficiently).
Lexicographic Order

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

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

- **Syntax**
  
  ```java
  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 if `string_2` precedes `string_1`.
The Type **boolean**

- The type **boolean** is a primitive type with only two values: **true** and **false**.

- Boolean variables can make programs more readable.

  ```java
  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.

  ```java
  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

• 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

- Figure 3.9

### 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 `||`
Precedence Rules

• In what order are the operations performed?

\[
\text{score} < \min/2 - 10 \quad || \quad \text{score} > 90 \\
\text{score} < (\min/2) - 10 \quad || \quad \text{score} > 90 \\
\text{score} < ((\min/2) - 10) \quad || \quad \text{score} > 90 \\
(score < ((\min/2) - 10)) \quad || \quad (score > 90)
\]
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 associated with an \texttt{||} is \texttt{true}, the expression is \texttt{true}.
  • If the first operand associated with an \texttt{&&} is \texttt{false}, the expression is \texttt{false}.
• This is called \textit{short-circuit} or lazy evaluation.
Short-circuit Evaluation

• 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.

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

• *Complete evaluation* can be achieved by substituting `&` for `&&` or `|` for `||`.
Input and Output of Boolean Values

• Example

```java
boolean booleanVar = false;
System.out.println(booleanVar);
System.out.println("Enter a boolean value:");
Scanner keyboard = new Scanner(System.in);
booleanVar = keyboard.nextBoolean();
System.out.println("You entered " + booleanVar);
```
Input and Output of Boolean Values

• Dialog

false
Enter a boolean value: true
true
You entered true
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. (*if* and *switch* )

• A *loop statement* repeats an action until a stopping condition occurs. (Later)
General **if** Statement Structure

```java
if (Boolean_Expression) {
    if_Statements
}
else if (Boolean_Expression) {
    else_if_Statements
}
else {
    else_statements
}
```

Exactly 1 **if**

0 to many **else ifs**

0 or 1 **elses**

--- *happens if nothing else does before*

• **Boolean_Expression**s are tried one after the other, top down, until one passes
• The **statement**(s) for test that passed are executed. (**else** entered, if reached).
• None of the other **statements** are executed
Simple **if-else** Statement

• A branching statement that chooses between two possible actions.

• Syntax

```java
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```
The **if-else** Statement

• Example

```java
if (balance >= 0)
    balance = balance + (INTEREST_RATE * balance) / 12;
else
    balance = balance - OVERDRAWN_PENALTY;
```
The **if-else** Statement

- Figure 3.1 The Action of the **if-else** Statement sample program Listing 3.1
The **if-else** Statement

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

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
Semantics of the **if-else** Statement

- Figure 3.2

```java
if (Boolean_Expression)
    Statement_1
else
    Statement_2
```
Compound (Block) Statements

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

• Syntax

```java
{
    Statement_1;
    Statement_2;
...
}
```
Compound Statements

• A compound statement can be used wherever a statement can be used.

• Example

```java
if (total > 10)
{
    sum = sum + total;
    total = 0;
}
```
Compound Statements and Ifs

• To include multiple statements in a branch, enclose the statements in braces.
  
  ```java
  if (count < 3)
  {
    total = 0;
    count = 0;
  }
  ```

• Generally, use the curly braces when making an `if`.

• Only if you absolutely sure you need a single statement should you skip it. (This is a common source of syntax errors for beginning programmers).
Omitting the `else` Part

• FIGURE 3.3 The Semantics of an `if` Statement without an `else`
Nested `if-else` Statements

• An `if-else` statement can contain any sort of statement within it.
• Nesting here means “inside”
• 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

• Syntax

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

• Each `else` is paired with the nearest unmatched `if`.
• If used properly, indentation communicates which `if` goes with which `else`.
• If used improperly, indentation confuses everyone.
• Braces can and should be used like parentheses to group statements.
Nested Statements

• Subtly different forms

First Form

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

Second Form

```java
if (a > b)
    if (c > d)
        e = f
    else
        g = h;
// oops
```
Multibranch if-else Statements

• Syntax

```java
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

- Figure 3.8 Semantics
Multibranch `if-else` Statements

• View sample program Listing 3.3

```java
class Grader {
    public static void main(String[] args) {
        int score = 85;
        Score = score;
        Grade = 'B';
    }
}
```

Enter your score:
85
Score = 85
Grade = B

Sample screen output
Multibranch if-else Statements

• Equivalent code

```java
if (score >= 90)
    grade = 'A';
else if ((score >= 80) && (score < 90))
    grade = 'B';
else if ((score >= 70) && (score < 80))
    grade = 'C';
else if ((score >= 60) && (score < 70))
    grade = 'D';
else
    grade = 'F';
```
Case Study – Body Mass Index

• Body Mass Index (BMI) is used to estimate the risk of weight-related problems
• \( \text{BMI} = \frac{\text{mass}}{\text{height}^2} \)
  • Mass in kilograms, height in meters
• Health assessment if:
  • \( \text{BMI} < 18.5 \) Underweight
  • \( 18.5 \leq \text{BMI} < 25 \) Normal weight
  • \( 25 \leq \text{BMI} < 30 \) Overweight
  • \( 30 \leq \text{BMI} \) Obese
Case Study – Body Mass Index

• Algorithm
  • Input height in feet & inches, weight in pounds
  • Convert to meters and kilograms
    • 1 lb = 2.2 kg
    • 1 inch = 0.254 meters
  • Compute BMI
  • Output health risk using if statements

View sample program Listing 3.4

class BMI
The Conditional Operator

```java
if (n1 > n2)
    max = n1;
else
    max = n2;
```
can be written as
```
max = (n1 > n2) ? n1 : n2;
```

• The `?` and `:` together are call the *conditional operator* or *ternary operator*. 
The Conditional Operator

• (very) Occasionally allows very simple `if-else` logic to be expressed succinctly.
• The conditional operator is useful with print and println statements.

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

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

• Example

```java
if (numberOfWinners == 0)
{
    System.out.println("Error: Dividing by zero.");
    System.exit(0);
}
else
{
    oneShare = payoff / numberOfWinners;
    System.out.println("Each winner will receive $" + oneShare);
}
```
Input Validation

• You should check your input to ensure that it is within a valid or reasonable range. For example, consider a program that converts feet to inches. You might write the following:

```java
int feet = keyboard.nextInt();
int inches = feet * 12;
```

• What if:
  • The user types a negative number for feet?
  • The user enters an unreasonable value like 100? Or a number larger than can be stored in an int? (2,147,483,647)
Input Validation

• Address these problems by ensuring that the entered values are reasonable:

```java
int feet = keyboard.nextInt();
if ((feet >= 0) && (feet < 10))
{
    int inches = feet * 12;
    ...
}
```
The **switch** Statement

- The **switch** statement is a multiway branch that makes a decision based on an *integral* (integer or character) expression.
  - Java 7 allows String expressions
- The **switch** statement begins with the keyword **switch** followed by an integral expression in parentheses and called the *controlling expression*. 
The **switch** Statement

- 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.
The **switch** Statement

• 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.
The `switch` Statement

• Syntax

```java
switch (Controlling_Expression) {
    case Case_Label:
        Statement(s);
        break;
    case Case_Label:
        ...
    default:
        ...
}
```
The **switch** Statement

• View **sample program** Listing 3.5

```java
class MultipleBirths {
    public static void main(String[] args) {
        int numberOfBabies;

        numberOfBabies = Integer.parseInt(JOptionPane.showInputDialog("Enter number of babies:"));

        switch (numberOfBabies) {
            case 1:
                System.out.println("Congratulations.");
                break;
            case 3:
                System.out.println("Wow. Triplets.");
                break;
            case 4:
                System.out.println("Unbelievable; 4 babies.");
                break;
            case 6:
                System.out.println("I don't believe you.");
                break;
        }
    }
}
```

Sample screen output

Enter number of babies: 1
Congratulations.

Enter number of babies: 3
Wow. Triplets.

Enter number of babies: 4
Unbelievable; 4 babies.

Enter number of babies: 6
I don't believe you.
The `switch` Statement

• 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.
Enumerations

• Consider a need to restrict contents of a variable to certain values
• An enumeration lists the values a variable can have
• Adds clarity to code. Allows us to group constants and give meaningful names.

• Example

```java
enum MovieRating {E, A, B}

MovieRating rating;
rating = MovieRating.A;
```
Enumerations

• Now possible to use in a `switch` statement

```java
switch (rating)
{
    case E: //Excellent
        System.out.println("You must see this movie!");
        break;
    case A: //Average
        System.out.println("This movie is OK, but not great.");
        break;
    case B: // Bad
        System.out.println("Skip it!");
        break;
    default:
        System.out.println("Something is wrong.");
}
```
Enumerations

• An even better choice of descriptive identifiers for the constants

```java
class MovieRating {
    enum {EXCELLENT, AVERAGE, BAD}
}

MovieRating rating = MovieRating.AVERAGE;

switch (rating) {
    case EXCELLENT:  ...
    ...
}
```