

3. Flow of Control

Harald Gall, Michael Würsch

Institut für Informatik

Universität Zürich

<http://seal.ifi.uzh.ch>



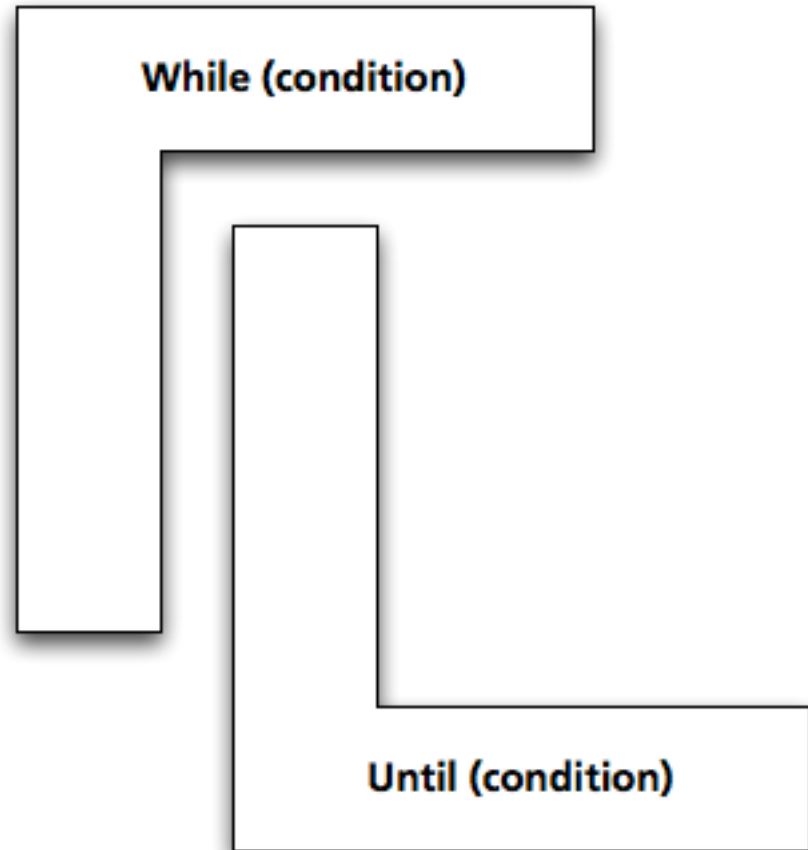
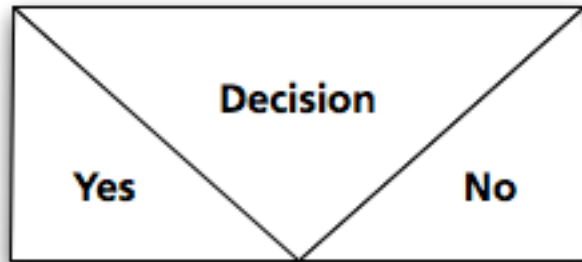
University of Zurich
Department of Informatics



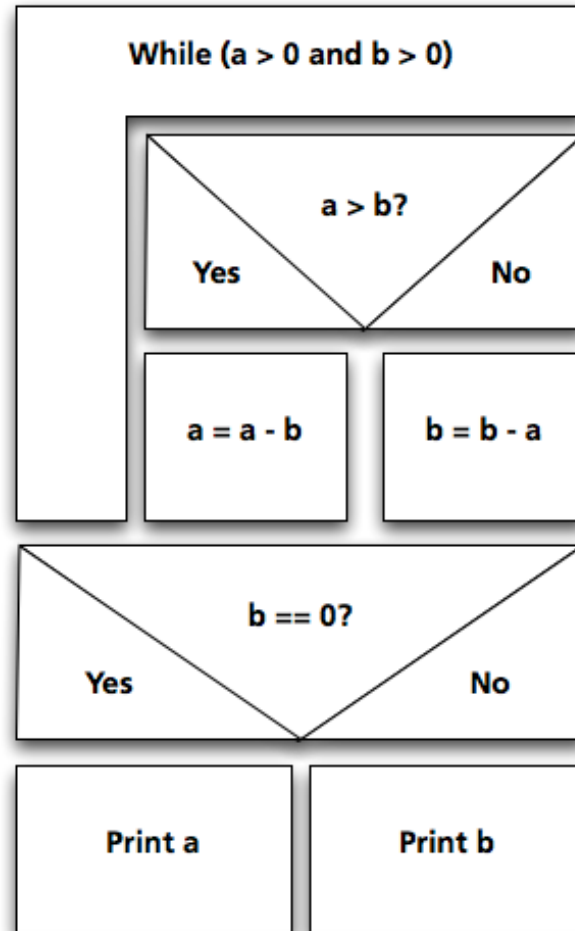
Flow of Control

- *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.

Visualizing the Flow of Control



Example: Euclid's Algorithm



Outline

- The Type `boolean` and boolean Expressions
- The `if-else` Statement
- The `switch` statement



The Type boolean

- True or False
- Example use case:

“The order can only be completed if the customer is already registered and has entered a valid credit card number.”

- $\text{Order}_{\text{ok}} = \text{Account}_{\text{exists}} \text{ AND } \text{CreditCard}_{\text{valid}}$



The Type `boolean`

- 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)  
    && ...)
```

Naming Boolean Variables

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

Input and Output of Boolean Values

- Example

```
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`

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) ...
```

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	==	<code>balance == 0</code> <code>answer == 'y'</code>
≠	Not equal to	!=	<code>income != tax</code> <code>answer != 'y'</code>
>	Greater than	>	<code>expenses > income</code>
≥	Greater than or equal to	>=	<code>points >= 60</code>
<	Less than	<	<code>pressure < max</code>
≤	Less than or equal to	<=	<code>expenses <= income</code>

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 points values are equal. Use < and some appropriate tolerance instead.

```
if (abs(b - c) < epsilon)
```

where **b**, **c**, and **epsilon** are floating point types

Using ==, cont.

- == is not appropriate for determining if two objects have the same value.
 - `if (s1 == s2)`, where `s1` and `s2` refer to strings, determines only if `s1` and `s2` refer to a common 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 ==

- To test the equality of objects of class String, use method `equals`.

```
s1.equals(s2)
```

or

```
s2.equals(s1)
```

- To test for equality ignoring case, use method `equalsIgnoreCase`.

```
("Hello".equalsIgnoreCase("hello"))
```

equals and equalsIgnoreCase

- Syntax

String.equals(Other_String)

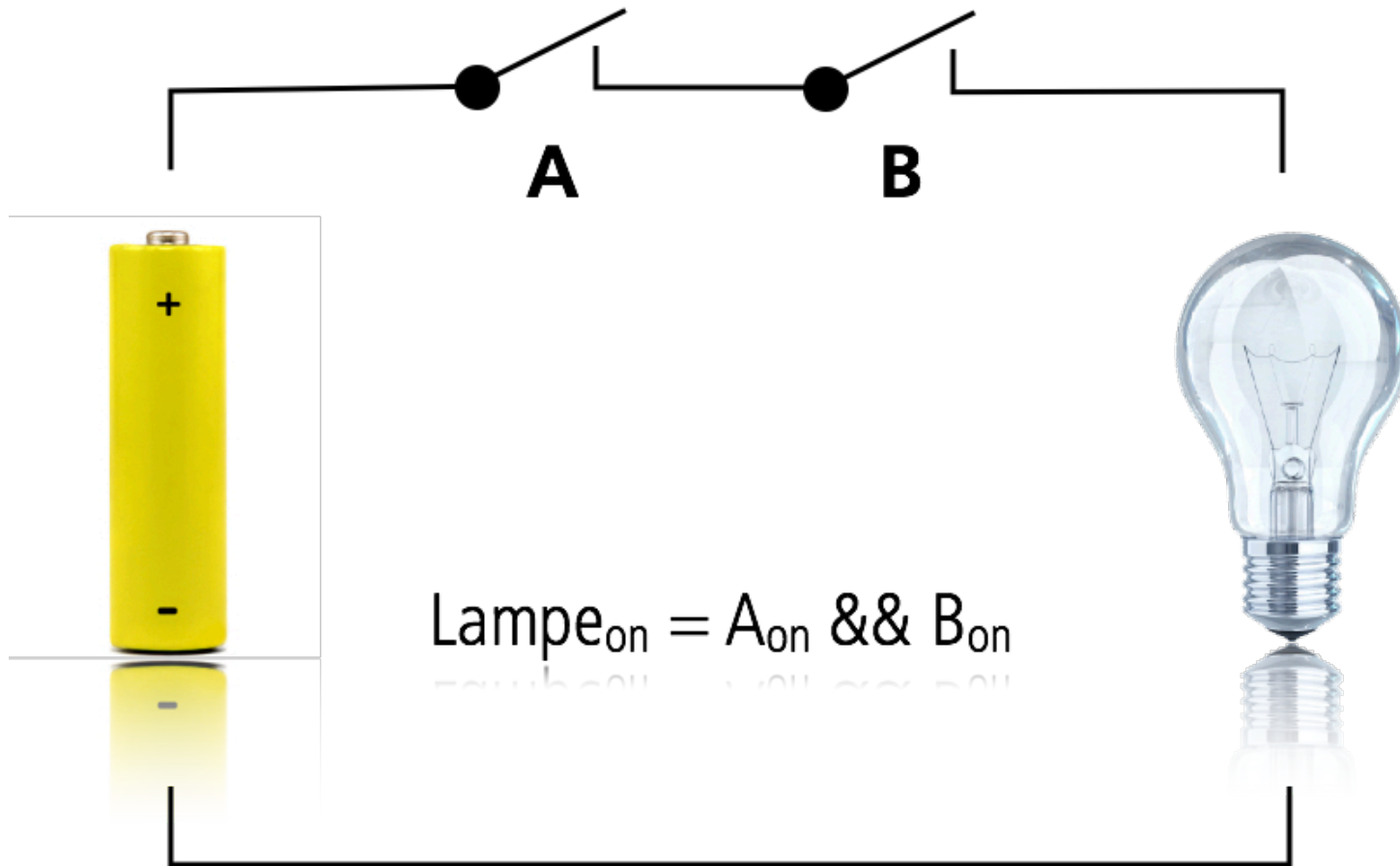
String.equalsIgnoreCase(Other_String)

Java Logical Operators

- Figure 3.6

Name	Java Notation	Java Examples
Logical <i>and</i>	&&	<code>(sum > min) && (sum < max)</code>
Logical <i>or</i>		<code>(answer == 'y') (answer == 'Y')</code>
Logical <i>not</i>	!	<code>!(number < 0)</code>

Boolean Expressions: AND



Compound Boolean Expressions

- Boolean expressions can be combined using the "and" (&&) operator.

- Example

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

```
...
```

- Not allowed

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

```
...
```

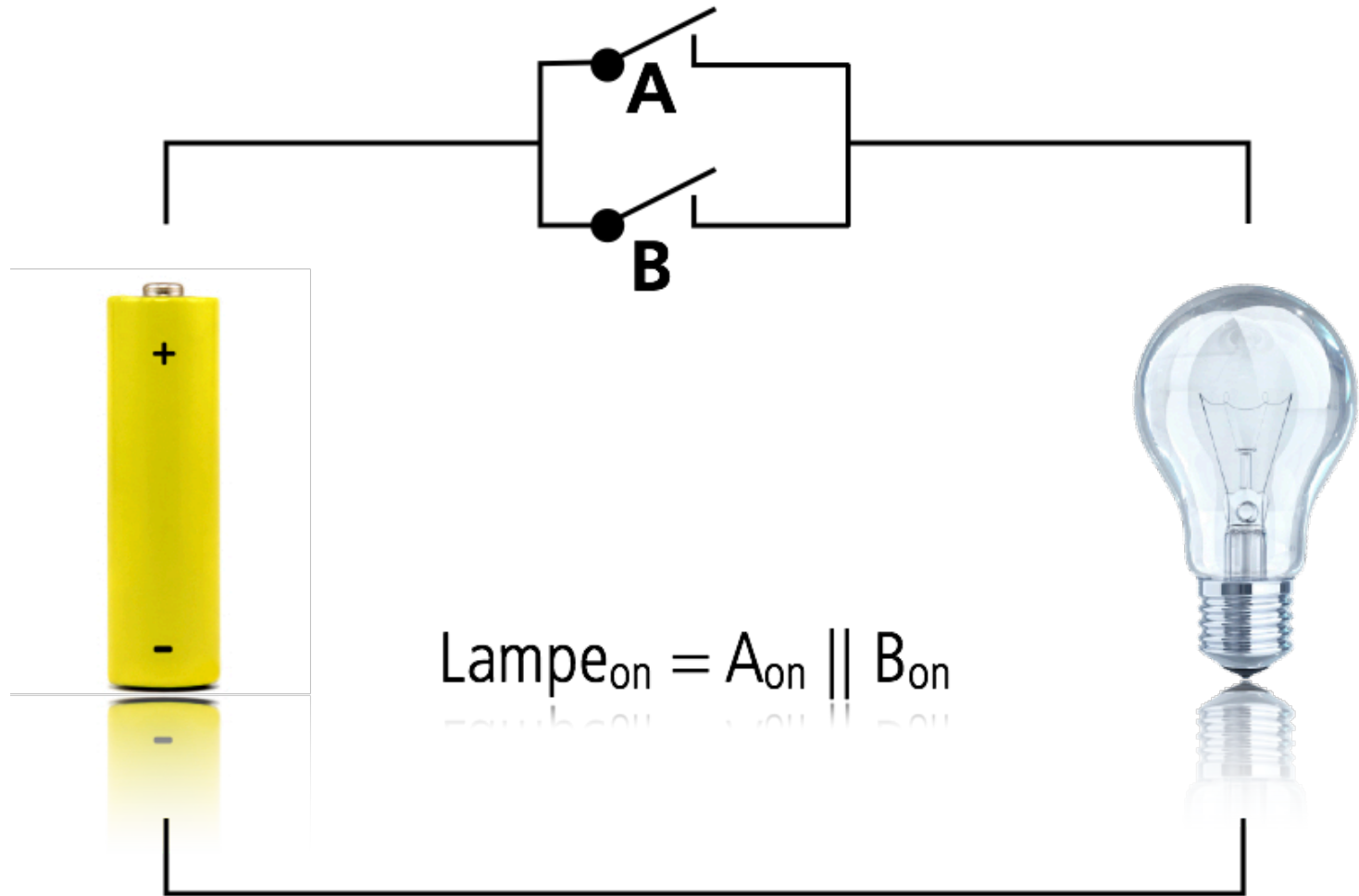
Compound Boolean Expressions

- 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.

Boolean Expressions: OR



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.
- The Java version of “or” is the *inclusive or* which allows either or both to be true.
- The *exclusive or* allows one or the other, but not both to be true.

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 `||` is **true**, the expression is **true**.
 - If the first operand associated with an `&&` is **false**, the expression is **false**.
- This is called *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.

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

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

Negating a Boolean Expression

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

- Syntax

!(Boolean_Expression)

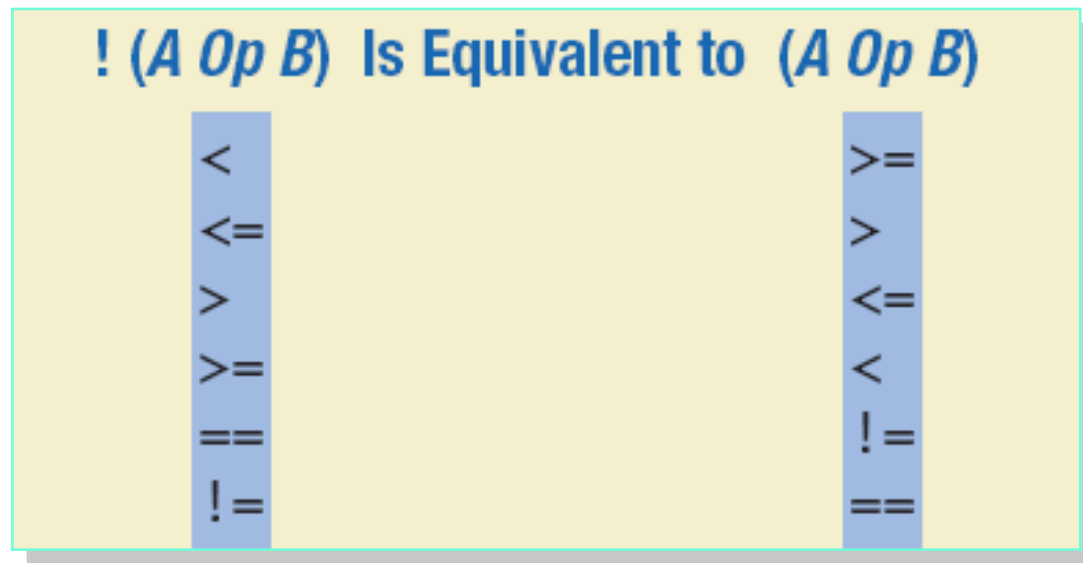
- Example

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

which is the *exclusive or*

Negating a Boolean Expression

- Figure 3.5 Avoiding the Negation Operator



Boolean Operators

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

Value of <i>A</i>	Value of <i>B</i>	Value of <i>A && B</i>	Value of <i>A B</i>	Value of <i>! (A)</i>
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

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 $||$

Lowest Precedence



Precedence Rules

- 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)`

The `if-else` Statement

- A branching statement that chooses between two possible actions.
- **syntax**
 - `if (Boolean_Expression)`
 - `Statement_1`
 - `else`
 - `Statement_2`

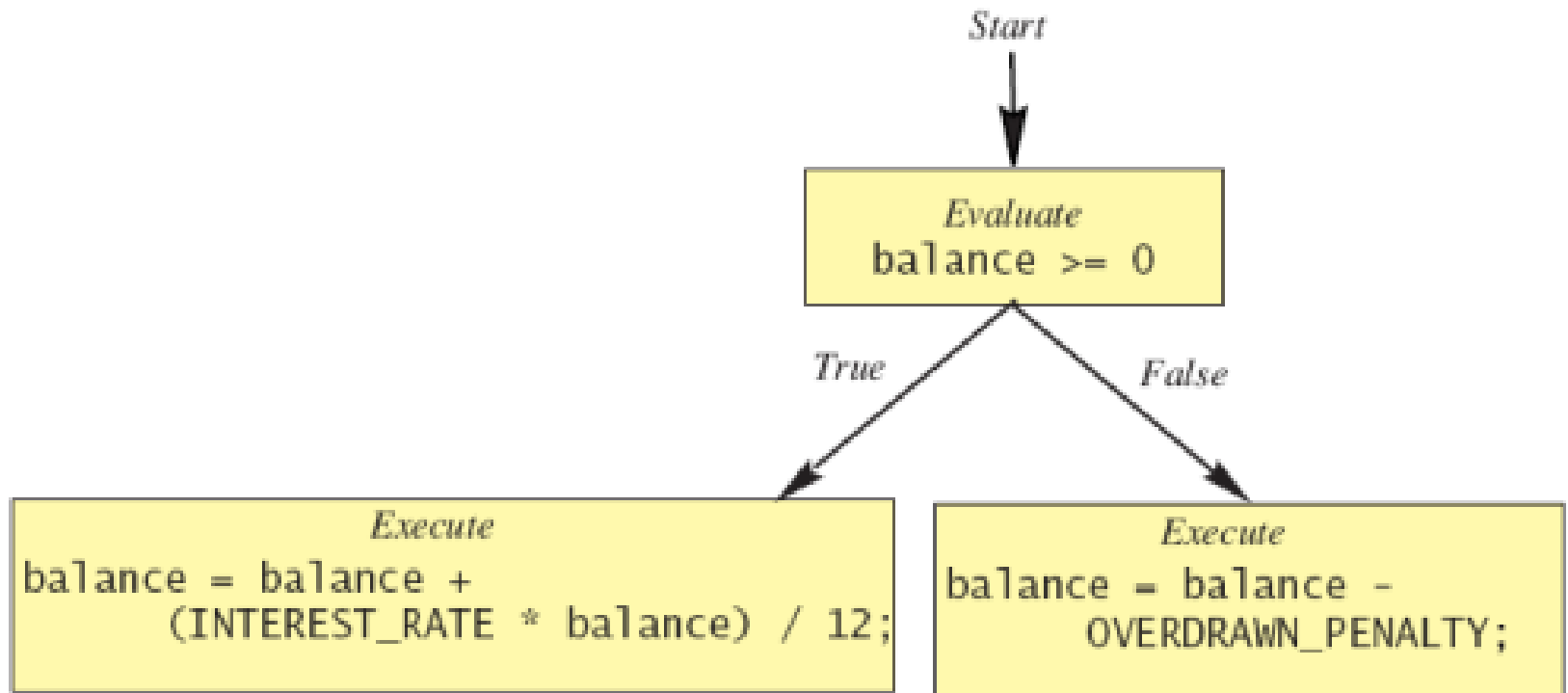
The `if-else` Statement, cont.

- Example

```
if (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

Sample
screen
output

```
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
```

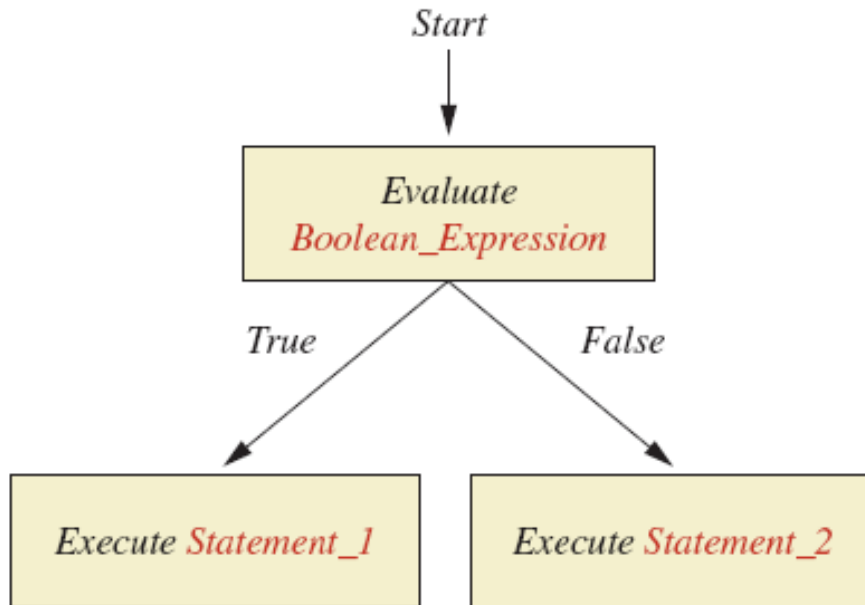
```
After adjusting for one month of interest and penalties,  
your new balance is $-23.53
```



Semantics of the `if-else` Statement

■ Figure 3.2

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



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

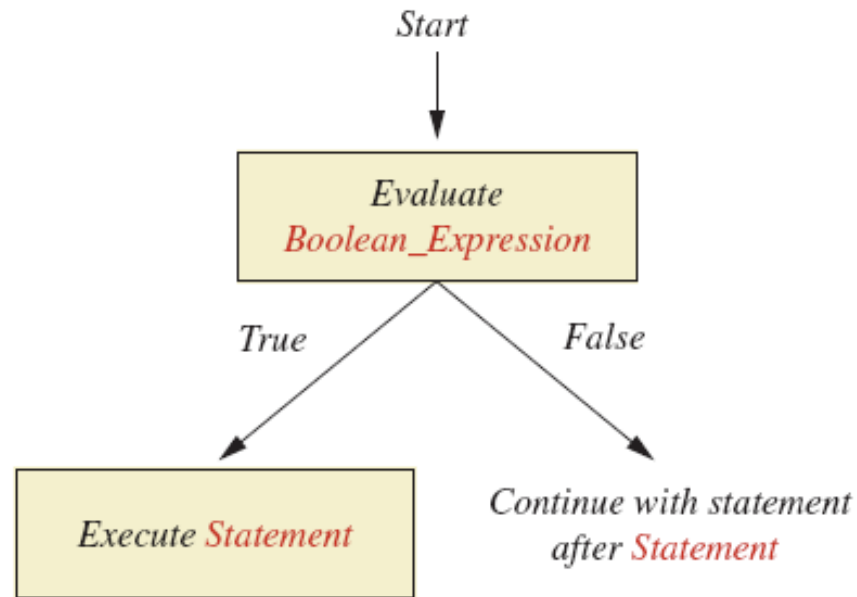
```
Enter your checking account balance: $-15.53
Original balance $-15.53
After adjusting for one month
of interest and penalties,
your new balance is $-23.53
```

Display 3.1
A Program Using `if-else`

Omitting the `else` Part

- The Semantics of an `if` Statement without an `else`

`if` (*Boolean_Expression*)
Statement



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;
```

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

- 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 (s1.compareTo(s2) == 0)
```

```
    System.out.println("Equal!");
```

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 if `String_2` precedes `String_1`.

Compound Statements

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

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

Compound Statements

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

```
{  
    Statement_1;  
    Statement_2;  
    ...  
}
```

Compound Statements

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

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

Nested `if-else` 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

- 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 Statements

- Each **else** is paired with the nearest unmatched **if**.
- **If used properly**, indentation communicates which **if** goes with which **else**.
- Braces can be used like parentheses to group statements.

Nested Statements

- Subtly different forms

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;

// oops
```

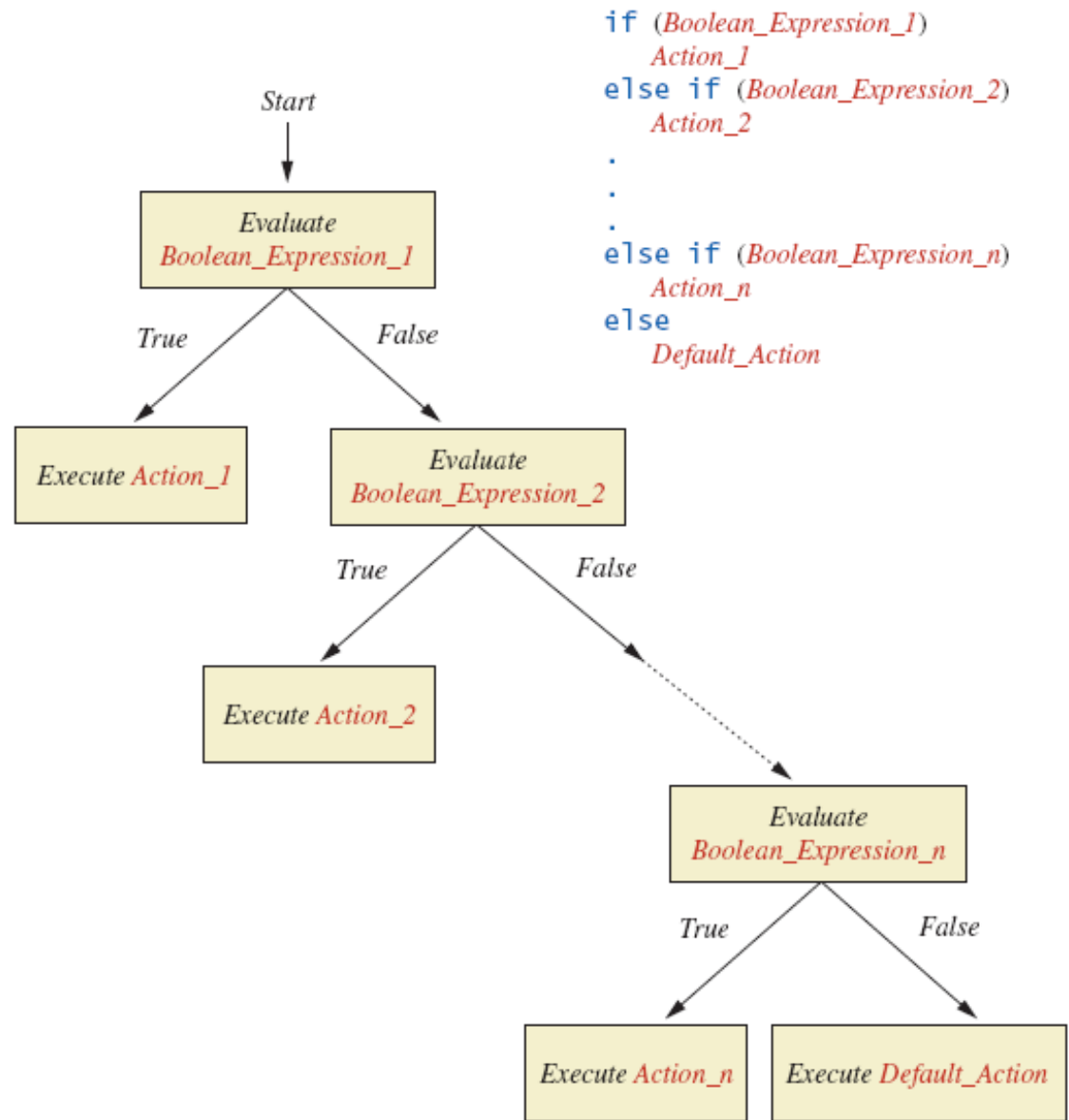
Multibranch `if-else` Statements

- 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

■ Figure 3.8 Semantics



Multibranch `if-else` Statements

- View [sample program](#) Listing 3.3
`class Grader`

Enter your score:

85

Score = 85

Grade = B

Sample
screen
output

Multibranch `if-else` Statements

- Equivalent code

```
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';
```

The `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*.

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

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



The `switch` Statement

- View [sample program](#) Listing 3.4
`class MultipleBirths`

```
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.
```

Sample
screen
output



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
- Example

```
enum MovieRating {E, A, B}  
MovieRating rating;  
rating = MovieRating.A;
```

Enumerations

- Now possible to use in a **switch** statement

```
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

```
enum MovieRating
    {EXCELLENT, AVERAGE, BAD}
rating = MovieRating.AVERAGE;

case EXCELLENT:    ...
```


The Conditional Operator

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

can be written as

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

- The **?** and **:** together are called the *conditional operator* or *ternary operator*.

The Conditional Operator

- The conditional operator is useful with print and println statements.

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

Summary

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