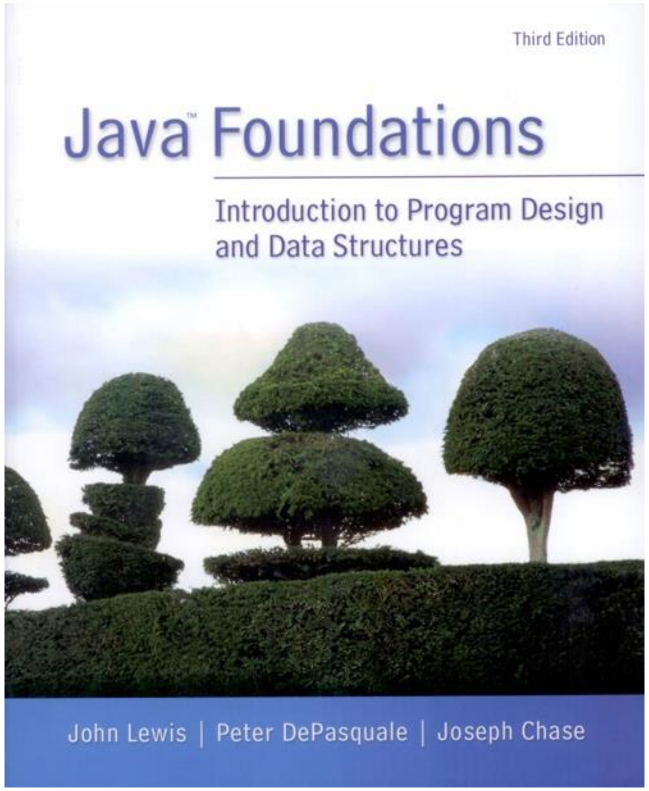

Chapter 4

Conditionals

Objectives

- How to use a debugger to examine program execution and fix errors
- How to evaluate Boolean expressions
- How to control program execution sequence using conditionals

Using a Debugger

Using a Debugger

You can examine the flow of your program and see the variable values that are declared using a ***debugger***

To start the debugger in Eclipse:

- Set a ***breakpoint*** on one of the lines of the program where you want to begin debugging
- Breakpoints are set by selecting the line where the execution should pause and then hit CTRL+SHIFT+B
- To start debugging, press F11

Controlling Program Flow Using the IF Statement

Flow of Control

Unless specified otherwise, the order of statement execution (***flow of control***) through a method is linear: one statement after another in sequence

Some programming statements allow us to

- decide whether or not to execute a particular statement
- execute a statement over and over, repetitively

These decisions are based on ***Boolean expressions*** (or ***conditions***) that evaluate to true or false

Conditional Statements

A ***conditional statement*** lets us choose which statement will be executed next

The Java conditional statements are the

- ***if*** *statement*
- ***if-else*** *statement*
- ***switch*** *statement*

The `if` Statement

The ***if statement*** has the following syntax

`if` is a Java reserved word

The *`condition`* must be a boolean expression. It must evaluate to either true or false.

```
if ( condition )
    statement;
```

If the *`condition`* is true, the *`statement`* is executed.
If it is false, the *`statement`* is skipped.

EXAMPLE: Age.java

Boolean Expressions

Boolean Expressions

A condition often uses one of Java's ***equality operators*** or ***relational operators***, which all return Boolean results

==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Note the difference between the equality operator (==) and the assignment operator (=)

Logical Operators

Boolean expressions can also use the following ***logical operators***

!	Logical NOT
&&	Logical AND
||	Logical OR

They all take Boolean operands and produce Boolean results

NOT is a unary operator (operates on one operand)

AND and OR are binary operators (two operands)

Logical NOT

The ***logical NOT* operation** is also called ***logical negation*** or ***logical complement***

If `a` is true, then `!a` is false

if `a` is false, then `!a` is true

Logical expressions can be shown using a ***truth table***

`a`	`!a`
true	false
false	true

Logical AND and Logical OR

The ***logical AND* expression**

```
a && b
```

is true if both `a` and `b` are true, and false otherwise

The ***logical OR* expression**

```
a || b
```

is true if `a` or `b` or both are true, and false otherwise

Logical Operators

A truth table shows all possible true-false combinations of the terms

Since `&&` and `||` each have two operands, there are four possible combinations of conditions `a` and `b`

`a`	**`b`**	**`a && b`**	**`a || b`**
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Logical Operators

Expressions that use logical operators can form complex conditions

```
if (total < MAX+5 && !found)
   System.out.println ("Processing…");
```

Precedence order:

All logical operators have lower precedence than the relational operators

Logical NOT has higher precedence than logical AND and logical OR

Boolean Expressions

Specific expressions can be evaluated using truth tables

`total < MAX`	`found`	`!found`	`total < MAX && !found`
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Short-Circuited Operators

The processing of logical AND and logical OR is ***short-circuited***

If the left operand is sufficient to determine the result, the right operand is not evaluated

```
if (count != 0 && total/count > MAX)
   System.out.println ("Testing…");
```

Other Conditional Statements

The `if-else` Statement

An ***else clause*** can be added to an `if` statement to make an ***if-else statement***

```
if ( condition )
   statement1;
else
   statement2;
```

If the `condition` is true, `statement1` is executed; if the `condition` is false, `statement2` is executed

One or the other will be executed, but not both

EXAMPLE: `Wages.java`

Indentation Revisited

Remember that indentation is for the human reader, and is ignored by the computer

```
if (total > MAX)
   System.out.println ("Error!!");
   errorCount++;
```

Despite what is implied by the indentation, the increment will occur whether the condition is true or not

Block Statements

Several statements can be grouped together into a ***block statement*** delimited by braces

A block statement can be used wherever a statement is called for in the Java syntax rules

```
if (total > MAX)
{
   System.out.println ("Error!!");
   errorCount++;
}
```

EXAMPLE: `Guessing.java`

Nested `if` Statements

The statement executed as a result of an `if` statement or `else` clause could be another `if` statement

These are called ***nested if statements***

An `else` clause is matched to the last unmatched `if` (no matter what the indentation implies)

Braces **{ … }** can be used to specify the `if` statement to which an `else` clause belongs

`EXAMPLE: MinOfThree.java`

Caveats with Comparing Values

Comparing Data

When comparing data using Boolean expressions, it's important to understand the nuances of certain data types

Let's examine some key situations

- comparing floating point values for equality
- comparing characters
- comparing strings (alphabetical order)
- comparing object vs. comparing object references

Comparing Float Values

You should rarely use the equality operator (`==`) when comparing two floating point values (`float` or `double`)

Two floating point values are equal only if their underlying binary representations match exactly

Computations often result in slight differences that may be irrelevant

In many situations, you might consider two floating point numbers to be “close enough” even if they aren't exactly equal

Comparing Float Values

To determine the equality of two floats, you may want to use the following technique

```
if (Math.abs(f1 - f2) < TOLERANCE)
   System.out.println ("Essentially equal");
```

If the difference between the two floating point values is less than the tolerance, they are considered to be equal

The tolerance could be set to any appropriate level, such as 0.000001

Comparing Characters

Java character data is based on the Unicode character set

Unicode establishes a particular numeric value for each character, and therefore an ordering

- We can use relational operators on character data based on this ordering
- For example, `'+' < 'J'` because it comes before it in the Unicode character set

Comparing Characters

In Unicode, the digit characters (0-9) are contiguous and in order

Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in order

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing Strings

Remember: in Java a character string is an object

The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order

The `equals` method returns a Boolean result

```
        if (name1.equals(name2))
           System.out.println ("Same name");
```

Comparing Strings

We cannot use the relational operators to compare strings

The `String` class contains a method called `compareTo` to determine if one string comes before another

A call to `name1.compareTo(name2)`

- returns zero if `name1` and `name2` are equal (contain the same characters)
- returns a negative value if `name1` is less than `name2`
- returns a positive value if `name1` is greater than `name2`

Comparing Strings

```
if (name1.compareTo(name2) < 0)
   System.out.println (name1 + "comes first");
else
   if (name1.compareTo(name2) == 0)
      System.out.println ("Same name");
   else
      System.out.println (name2 + "comes first");
```

Because comparing characters and strings is based on a character set, it is called a ***lexicographic ordering***

Lexicographic Ordering

Not strictly alphabetical when uppercase and lowercase characters are mixed

- For example, `"Great"` comes before the `"fantastic"`

Also, short strings come before longer strings with the same prefix (lexicographically)

- Therefore `"book"` comes before `"bookcase"`

Comparing Objects

The `==` operator can be applied to objects

– returns true if references are aliases of each other

The `equals` method is defined for all objects, but **unless we redefine it when we write a class, it has the same semantics as the == operator**

It has been redefined in the `String` class to compare the characters in the two strings

The Switch Statement

The `switch` Statement

The ***switch statement*** provides another way to decide which statement to execute next

The `switch` statement evaluates an expression, then attempts to match the result to one of several possible ***cases***

- Each case contains a value and a list of statements

The flow of control transfers to statement associated with the first case value that matches

The `switch` Statement

The general syntax of a `switch` statement is

`switch` and **`case`** are reserved words

```
switch ( expression )
{
   case value1 :
      statement-list1
   case value2 :
      statement-list2
   case value3 :
      statement-list3
   case  ...

}
```

If *`expression`* matches *`value2`*, control jumps to here

The `switch` Statement

Often a ***break statement*** is used as the last statement in each case's statement list

- A `break` statement causes control to transfer to the end of the `switch` statement
- If a `break` statement is not used, the flow of control will continue into the next case

The `switch` Statement

An example of a `switch` statement

```
switch (option)
{
   case 'A':
      aCount++;
      break;
   case 'B':
      bCount++;
      break;
   case 'C':
      cCount++;
      break;
}
```

The `switch` Statement

A `switch` statement can have an optional ***default case***

- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- If there is no default case, and no other value matches, control falls through to the statement after the switch

The `switch` Statement

The expression of a `switch` statement must result in an ***integral type***, meaning an integer (`byte`, `short`, `int`, `long`) or a `char`

- **It cannot be a `boolean` value or a floating point value (`float` or `double`)**

The implicit Boolean condition in a `switch` statement is equality

- **You cannot perform relational checks with a `switch` statement**

`EXAMPLE: GradeReport.java`

Summary

- A debugger can be used to examine program execution and fix errors
- The flow of control can be changed using conditional statements
- Conditional statements depend on the evaluation of Boolean expressions
- Floating point values are rarely exactly equal so equality checks should account for tolerance
- Strings should be compared using the equals method