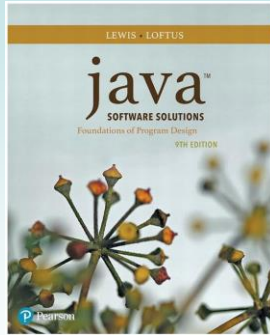


## Chapter 5

# Conditionals and Loops



Java Software Solutions  
Foundations of Program Design  
9<sup>th</sup> Edition

John Lewis  
William Loftus

PEARSON

Copyright © 2017 Pearson Education, Inc.

## Conditionals and Loops

- Now we will examine programming statements that allow us to:
  - make decisions
  - repeat processing steps in a loop
- Chapter 5 focuses on:
  - boolean expressions
  - the if and if-else statements
  - comparing data
  - while loops
  - iterators

Copyright © 2017 Pearson Education, Inc.

## Outline



### **Boolean Expressions**

**The `if` Statement**

**Comparing Data**

**The `while` Statement**

**Iterators**

**The `ArrayList` Class**

Copyright © 2017 Pearson Education, Inc.

## Flow of Control

- Unless specified otherwise, the order of statement execution through a method is linear: one after another
- Some programming statements allow us to make decisions and perform repetitions
- These decisions are based on *boolean expressions* (also called *conditions*) that evaluate to true or false
- The order of statement execution is called the *flow of control*

Copyright © 2017 Pearson Education, Inc.

## Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- They are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- The Java conditional statements are the:
  - if and if-else statement
  - switch statement
- We'll explore the switch statement in Chapter 6

Copyright © 2017 Pearson Education, Inc.

## 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 (=)

Copyright © 2017 Pearson Education, Inc.

## Boolean Expressions

- An `if` statement with its boolean condition:

```
if (sum > MAX)
    delta = sum - MAX;
```

- First, the condition is evaluated: the value of `sum` is either greater than the value of `MAX`, or it is not
- If the condition is true, the assignment statement is executed; if it isn't, it is skipped
- See `Age.java`

Copyright © 2017 Pearson Education, Inc.

## Logical Operators

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

<code>!</code>	Logical NOT
<code>&amp;&amp;</code>	Logical AND
<code>  </code>	Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

Copyright © 2017 Pearson Education, Inc.

## Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition  $a$  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

Copyright © 2017 Pearson Education, Inc.

## 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

Copyright © 2017 Pearson Education, Inc.

## Logical AND and Logical OR

- 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

Copyright © 2017 Pearson Education, Inc.

## Logical Operators

- Expressions that use logical operators can form complex conditions

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

- All logical operators have lower precedence than the relational operators
- The `!` operator has higher precedence than `&&` and `||`

Copyright © 2017 Pearson Education, Inc.

## Boolean Expressions

- Specific expressions can be evaluated using truth tables

<code>total &lt; MAX</code>	<code>found</code>	<code>!found</code>	<code>total &lt; MAX &amp;&amp; !found</code>
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Copyright © 2017 Pearson Education, Inc.

## Short-Circuited Operators

- The processing of `&&` and `||` 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.");
```

- This type of processing should be used carefully

Copyright © 2017 Pearson Education, Inc.

# Outline

Boolean Expressions

➡ The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Copyright © 2017 Pearson Education, Inc.

## The `if` Statement

- Let's now look at the `if` statement in more detail
- 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.

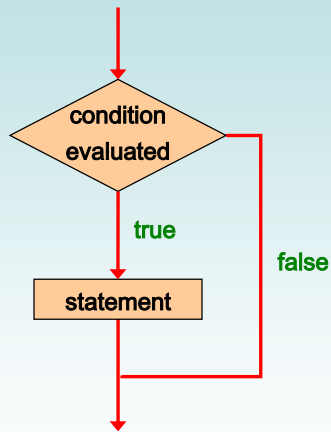
```
if ( condition )  
    statement;
```

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

Copyright © 2017 Pearson Education, Inc.



## Logic of an if statement



Copyright © 2017 Pearson Education, Inc.

## Indentation

- The statement controlled by the `if` statement is indented to indicate that relationship
- The use of a consistent indentation style makes a program easier to read and understand
- The compiler ignores indentation, which can lead to errors if the indentation is not correct

**"Always code as if the person who ends up  
maintaining your code will be a violent  
psychopath who knows where you live."**

**-- Martin Golding**

Copyright © 2017 Pearson Education, Inc.

## Quick Check

What do the following statements do?

```
if (total != stock + warehouse)
    inventoryError = true;
```

Sets the boolean variable to true if the value of `total` is not equal to the sum of `stock` and `warehouse`

```
if (found || !done)
    System.out.println("Ok");
```

Prints "Ok" if `found` is true or `done` is false

Copyright © 2017 Pearson Education, Inc.

## The if-else Statement

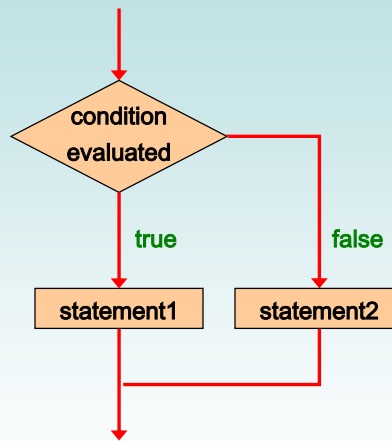
- An *else clause* can be added to an `if` statement to make an *if-else statement*

```
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
- See `Wages.java`

Copyright © 2017 Pearson Education, Inc.

## Logic of an if-else statement



Copyright © 2017 Pearson Education, Inc.

## The Coin Class

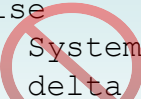
- Let's look at an example that uses a class that represents a coin that can be flipped
- Instance data is used to indicate which face (heads or tails) is currently showing
- See `CoinFlip.java`
- See `Coin.java`

Copyright © 2017 Pearson Education, Inc.

## Indentation Revisited

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

```
if (depth >= UPPER_LIMIT)
    delta = 100;
else
    System.out.println("Reseting Delta");
    delta = 0;
```



- Despite what the indentation implies, `delta` will be set to 0 no matter what

Copyright © 2017 Pearson Education, Inc.

## 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++;
}
```

Copyright © 2017 Pearson Education, Inc.

## Block Statements

- The `if` clause, or the `else` clause, or both, could govern block statements

```
if (total > MAX)
{
    System.out.println("Error!!");
    errorCount++;
}
else
{
    System.out.println("Total: " + total);
    current = total*2;
}
```

- See `Guessing.java`

Copyright © 2017 Pearson Education, Inc.

## Nested if Statements

- The statement executed as a result of an `if` 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
- See `MinOfThree.java`

Copyright © 2017 Pearson Education, Inc.

# Outline

**Boolean Expressions**

**The `if` Statement**



**Comparing Data**

**The `while` Statement**

**Iterators**

**The `ArrayList` Class**

Copyright © 2017 Pearson Education, Inc.

## 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

Copyright © 2017 Pearson Education, Inc.

## 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

Copyright © 2017 Pearson Education, Inc.

## Comparing Float Values

- To determine the equality of two floats, 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

Copyright © 2017 Pearson Education, Inc.

## Comparing Characters

- As we've discussed, 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, the character '+' is less than the character 'J' because it comes before it in the Unicode character set
- Appendix C provides an overview of Unicode

Copyright © 2017 Pearson Education, Inc.

## 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

Copyright © 2017 Pearson Education, Inc.



## Comparing Strings

- Remember that 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");
```

Copyright © 2017 Pearson Education, Inc.

## Comparing Strings

- We cannot use the relational operators to compare strings
- The `String` class contains the `compareTo` method for determining 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`

Copyright © 2017 Pearson Education, Inc.

## Comparing Strings

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

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

Copyright © 2017 Pearson Education, Inc.

## Lexicographic Ordering

- Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed
- For example, the string "Great" comes before the string "fantastic" because all of the uppercase letters come before all of the lowercase letters in Unicode
- Also, short strings come before longer strings with the same prefix (lexicographically)
- Therefore "book" comes before "bookcase"

Copyright © 2017 Pearson Education, Inc.

## Comparing Objects

- The `==` operator can be applied to objects – it returns true if the two 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
- When you write a class, you can redefine the `equals` method to return true under whatever conditions are appropriate


Copyright © 2017 Pearson Education, Inc.

## Outline

**Boolean Expressions**

**The `if` Statement**

**Comparing Data**

 **The `while` Statement**

**Iterators**

**The `ArrayList` Class**

Copyright © 2017 Pearson Education, Inc.

## Repetition Statements

- *Repetition statements* allow us to execute a statement multiple times
- Often they are referred to as *loops*
- Like conditional statements, they are controlled by boolean expressions
- Java has three kinds of repetition statements: `while`, `do`, and `for` loops
- The `do` and `for` loops are discussed in Chapter 6

Copyright © 2017 Pearson Education, Inc.

## The while Statement

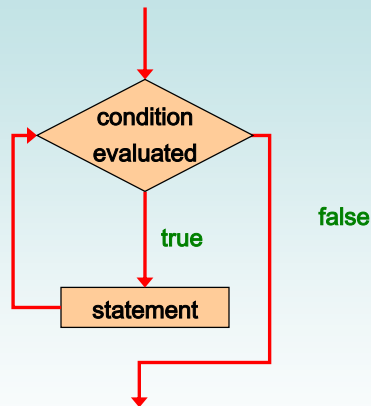
- A *while statement* has the following syntax:

```
while ( condition )  
    statement;
```

- If the **condition** is true, the **statement** is executed
- Then the condition is evaluated again, and if it is still true, the statement is executed again
- The statement is executed repeatedly until the condition becomes false

Copyright © 2017 Pearson Education, Inc.

## Logic of a while Loop



Copyright © 2017 Pearson Education, Inc.

## The while Statement

- An example of a while statement:

```
int count = 1;
while (count <= 5)
{
    System.out.println(count);
    count++;
}
```

- If the condition of a `while` loop is false initially, the statement is never executed
- Therefore, the body of a `while` loop will execute zero or more times

Copyright © 2017 Pearson Education, Inc.

## Sentinel Values

- Let's look at some examples of loop processing
- A loop can be used to maintain a *running sum*
- A *sentinel value* is a special input value that represents the end of input
- See `Average.java`

Copyright © 2017 Pearson Education, Inc.

## Input Validation

- A loop can also be used for *input validation*, making a program more *robust*
- It's generally a good idea to verify that input is valid (in whatever sense) when possible
- See `WinPercentage.java`

Copyright © 2017 Pearson Education, Inc.

## Infinite Loops

- The body of a `while` loop eventually must make the condition false
- If not, it is called an *infinite loop*, which will execute until the user interrupts the program
- This is a common logical error
- You should always double check the logic of a program to ensure that your loops will terminate normally

Copyright © 2017 Pearson Education, Inc.

## Infinite Loops

- An example of an infinite loop:

```
int count = 1;
while (count <= 25)
{
    System.out.println(count);
    count = count - 1;
}
```

- This loop will continue executing until interrupted (Control-C) or until an underflow error occurs

Copyright © 2017 Pearson Education, Inc.

## Nested Loops

- Similar to nested `if` statements, loops can be nested as well
- That is, the body of a loop can contain another loop
- For each iteration of the outer loop, the inner loop iterates completely
- See `PalindromeTester.java`

Copyright © 2017 Pearson Education, Inc.

## Quick Check

How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 10)
{
    count2 = 1;
    while (count2 < 20)
    {
        System.out.println("Here");
        count2++;
    }
    count1++;
}
```

**10 \* 19 = 190**

Copyright © 2017 Pearson Education, Inc.



## Outline

**Boolean Expressions**

**The `if` Statement**

**Comparing Data**

**The `while` Statement**



**Iterators**

**The `ArrayList` Class**

Copyright © 2017 Pearson Education, Inc.

## Iterators

- An *iterator* is an object that allows you to process a collection of items one at a time
- It lets you step through each item in turn and process it as needed
- An iterator has a `hasNext` method that returns `true` if there is at least one more item to process
- The `next` method returns the next item
- Iterator objects are defined using the `Iterator` interface, which is discussed further in Chapter 7

Copyright © 2017 Pearson Education, Inc.

## Iterators

- Several classes in the Java standard class library are iterators
- The `Scanner` class is an iterator
  - the `hasNext` method returns true if there is more data to be scanned
  - the `next` method returns the next scanned token as a string
- The `Scanner` class also has variations on the `hasNext` method for specific data types (such as `hasNextInt`)

Copyright © 2017 Pearson Education, Inc.

## Iterators

- The fact that a `Scanner` is an iterator is particularly helpful when reading input from a file
- Suppose we wanted to read and process a list of URLs stored in a file
- One scanner can be set up to read each line of the input until the end of the file is encountered
- Another scanner can be set up for each URL to process each part of the path
- See `URLDissector.java`

Copyright © 2017 Pearson Education, Inc.

## Outline

**Boolean Expressions**

**The `if` Statement**

**Comparing Data**

**The `while` Statement**

**Iterators**

 **The `ArrayList` Class**

Copyright © 2017 Pearson Education, Inc.

## The `ArrayList` Class

- An `ArrayList` object stores a list of objects, and is often processed using a loop
- The `ArrayList` class is part of the `java.util` package
- You can reference each object in the list using a numeric index
- An `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary

Copyright © 2017 Pearson Education, Inc.

## The ArrayList Class

- Index values of an `ArrayList` begin at 0 (not 1):

0	"Bashful"
1	"Sleepy"
2	"Happy"
3	"Dopey"
4	"Doc"

- Elements can be inserted and removed
- The indexes of the elements adjust accordingly

Copyright © 2017 Pearson Education, Inc.

## ArrayList Methods

- Some `ArrayList` methods:

```
boolean add(E obj)
void add(int index, E obj)
Object remove(int index)
Object get(int index)
boolean isEmpty()
int size()
```

Copyright © 2017 Pearson Education, Inc.

## The ArrayList Class

- The type of object stored in the list is established when the `ArrayList` object is created:

```
ArrayList<String> names = new ArrayList<String>();  
ArrayList<Book> list = new ArrayList<Book>();
```

- This makes use of Java *generics*, which provide additional type checking at compile time
- An `ArrayList` object cannot store primitive types, but that's what wrapper classes are for
- See `Beatles.java`

Copyright © 2017 Pearson Education, Inc.

## Summary

- Chapter 5 focused on:
  - boolean expressions
  - the if and if-else statements
  - comparing data
  - while loops
  - iterators

Copyright © 2017 Pearson Education, Inc.