Java Programming Week 10
Decision Making

Orange Coast College
Computer Science 170
Spring 2004

The Plan For Week 10
- Logical Operators: combining boolean expressions
- Boolean II: dealing with complex boolean expressions
- Multiple Choice: nested and ladder-style if-else
- More Selection: switch and the conditional operator
- More Methods: factory and overloaded methods
- Homework: the Guess’OMatic steals your Zippoids
- Reading: Chapter 5 (187-194 booleans), Chapter 6 (223-233 selection), Chapter 7 (218-289 method review)

Lesson A: The Logical Operators
- What are the logical operators?
- The NOT operator
- The two AND operators
- The two OR operators
- Pitfalls: impossible and unavoidable conditions
- Short-circuit evaluation
- Exercise 10A
Many times, you need to test several values at once. Check if an input value falls between 1 and 100, for instance. Java's logical operators are used to combine several tests. Logical operators work on boolean values. Just like arithmetic operators work on numeric values. One unary logical operator: NOT (!). Four binary logical operators: Two versions of AND (&&, &), Two versions of OR (| |, |).

What are Logical Operators?

The NOT Operator

The unary NOT symbol is the exclamation point! It appears in front of its boolean operand. It produces the reverse of the expression it operates on.

```
10 == 5 + 3 + 2;  // true
! (10 == 5 + 3 + 2); // false
! 10 == 5 + 3 + 2;  // illegal
```

The Binary Logical Operators

Unlike NOT, all of these are binary operators. Each requires two boolean operands. Each produces a boolean result.

```
boolean value  1  boolean value
   &&          boolean value
```

What are Logical Operators?

The NOT Operator

The unary NOT symbol is the exclamation point! It appears in front of its boolean operand. It produces the reverse of the expression it operates on.

```
10 == 5 + 3 + 2;  // true
! (10 == 5 + 3 + 2); // false
! 10 == 5 + 3 + 2;  // illegal
```

The Binary Logical Operators

Unlike NOT, all of these are binary operators. Each requires two boolean operands. Each produces a boolean result.

```
boolean value  1  boolean value
   &&          boolean value
```
The AND Operators
- The symbol used by Java for AND is && or &
- Corresponds to normal use of word "and"
  - Both conditions must be true for a true result
  - If either condition (or both) is false, result is false

```java
if ( units >= 123 && GPA >= 1.5 )
    youGraduate();
```

The OR Operators
- The symbol used by Java for OR is || or | |
  - No space between characters ||
- Corresponds to natural understanding of OR
  - If either condition is true, the result is true
  - Both conditions must be false for a false result

```java
if ( quiz > 75 || homework > 95 )
    finalGrade = 'B';
```

Pitfall 1: The Impossible Condition
- When no combination of operands could ever be true
  - "Give a 25% discount to children who are under 13 and to patrons who are 65 and older"
  - A "literal" reading results in code like this:

```java
if ( age < 13 && age > 65 )
    discount = .25;
else
    discount = 0;
```
- No possible value for age where this could be true
### Pitfall 2: The Unavoidable Condition
- When no conceivable condition could ever be false
  - "If the patron's age is greater than 12 or less than 66, charge 8.75 for the ticket. Everyone else pays 5.75."
- A "literal" translation results in code like this:
  ```java
  if (age > 12 || age < 66)
      ticketPrice = 8.75;
  else
      ticketPrice = 5.75;
  ```
- For every possible value of age, the condition is true

### Pitfall Solutions
- These are caused by confusing AND and OR
- First example could look like this:
  ```java
  if (age < 13 || age > 65) discount = .25;
  else discount = 0;
  ```
- Second example could look like this:
  ```java
  if (age > 12 && age < 66) ticketPrice = 8.75;
  else ticketPrice = 5.75;
  ```

### Short-circuit Evaluation I
- Sometimes, you "know" the answer to an expression without evaluating the entire thing.
  - For instance, what is the value of 0 * x?
  - You know the value is 0, whatever value x has
- Expressions using logical operators are similar
  - The value of 2 + 2 == 5 & a > b is false
  - The values stored in a and b are irrelevant
Short-circuit Evaluation II

- When using the `||` and `&&` operators, Java stops evaluating the expression once it can determine the result.
- This is called **short-circuit evaluation**.
  - The expression is evaluated from left to right.
  - Evaluation stops when the result is assured.
- For `&&` evaluation stops when a `false` value is encountered.
- For `||` evaluation stops when a `true` value is encountered.
- The `|` and `&` operators evaluate both sides regardless.

Short-circuit Evaluation III

- Some examples using short-circuit and regular evaluation.

```
if ( (a != 0) && (b / a > 12) ) // OK
if ( (a != 0) & (b/a > 12)) // NOT OK
if ( (a > 10) || (b++ > 7)) // Problem?
if ( (a > 10) | (b++ > 7)) // OK
```

Precedence

- Relational operators appear below the arithmetic operators.
  - `isTrue = a > 3 * 5; // OK, parens not required`
- Logical operators appear below relational operators.
  - `isTrue = a > 3 || a < 1; // parens not required`
- Logical AND is higher than logical OR.
  - `isTrue = 10 < 15 || 5 > 8 & & 3 > 5;`
- Introduction to Exercise 10A
Lesson B: Simpler Boolean Expressions

- Why simplify Boolean expressions?
- Simplifying expressions using NOT
- Alternatives to NOT
- Using De Morgan's law
- Truth tables: AND, OR, and NOT
- Evaluating an expression with a truth table
- Exercise 10B

Why Simplify Boolean Expressions?

- Just tell me what this means:
  "It's not true that some people won't be confused by logical expressions with NOT operators applied to AND and OR expressions."
- This is true: we often have a hard time deciphering AND and OR expressions, especially when combined with NOT
- Sometimes, rearranging terms can make things clearer
- Best way to write correct boolean expressions is to write simpler boolean expressions

Expressions Using NOT

- NOT can reverse the sense of any boolean expression
  - Not always the clearest way to write the code

```java
boolean overtime = hours > 40;
boolean straightTimeOnly = ! (hours > 40);
```

- Most of us have to think about this to understand it:
  - "If hours > 40 is true, (overtime), then NOT hours greater than 40 is false, so straightTimeOnly is false. If hours greater than 40 is false, (no overtime), then NOT false is true, so straightTimeOnly is true."
Alternatives to NOT

Most of us find this way of writing the example clearer:

```java
boolean straightTimeOnly = hours <= 40;
```

With the relational operators you can avoid NOT like this:

<table>
<thead>
<tr>
<th>NOT Expression</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>! (a == b)</td>
<td>a != b</td>
</tr>
<tr>
<td>! (a &gt; b)</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td>! (a &lt; b)</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>! (a &lt;= b)</td>
<td>a &gt; b</td>
</tr>
<tr>
<td>! (a &gt;= b)</td>
<td>a &lt; b</td>
</tr>
</tbody>
</table>

NOT, AND, and OR

NOT becomes even more complex when combined with the other logical operators, AND and OR.

Test for those who can legally drink but who don't yet qualify for Social Security:

```java
if (age >= 21 && age < 65) ...
```

Now consider its negation:

```java
if (! (age >= 21 && age < 65)) ...
```

Again, most of us have to think about it.

De Morgan's Law

Designed to simplify these kinds of expressions.

Two versions: (one for AND, one for OR)

- ! (A & B) is equivalent to !A || !B
- ! (A || B) is equivalent to !A & !B

Convert our previous example like this:

```java
if (! (age >= 21) || ! (age < 65)) ...
```

then, using alternatives to NOT, we get:

```java
if (age < 21 || age >= 65) ...
```
Truth Tables I

- Boolean expressions must be either true or false
- That means a binary expression can have $2^2 = 4$ values
- A truth-table is a "brute force" technique used to write out every possible value that an expression can take on

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Truth Tables II

- Here is the truth table for an OR expression

| A  | B  | A || B |
|----|----|------|
| true | true | true |
| true | false| true |
| false| true | true |
| false| false| false |

- Here is the truth table for NOT

<table>
<thead>
<tr>
<th>A</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Using a Truth Table I

- Step 1: Extract the boolean expressions

```java
boolean ans = x <= y && y >= z || x < z;
```

- Let a stand for $x <= y$
- Let b stand for $y >= z$
- Let c stand for $x < z$
**Using a Truth Table II**

- Step 2: Construct a table - one expression per column

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

**Using a Truth Table III**

- Step 3: Apply precedence to find the sub-expressions

```java
boolean ans = ((x <= y) && (y >= z)) || x < z;
```

- Means we'll have two "results" columns
  - a && b
  - (a && b) || c
- Add columns to the truth table, "fill in the blanks"

**Using a Truth Table IV**
**Interpreting the Results**

- Use the table to evaluate the expression when
  - \( x = 5 \), \( z = 12 \), \( y = 4 \)
- After evaluating each sub-expression
  - \( x \leq y \) is false
  - \( y \geq z \) is false
  - \( x < z \) is true
- Look up false, false, true in the table
  - The expression is true
- Introduction to Exercise 10B

**Lesson C : Multiple Choice**

- Creating branches with more than two paths
- Method 1: Nested if statements
  - Nesting example: Counting Words
- Method 2: Ladder-style if else if else statements
- Exercise 10C

**Nested Logic**

- Another way to combine boolean expressions is nesting
  - A nested if statements appears in the body of another if

```java
if (x > 0)
{
    if (y <= 0)
        quadrant = SIRIUS;
    else
        quadrant = OMEGA;
}
```

- The nested if can appear in the else part as well
Counting Words

- Let's look at an application of nested if statements
  - We'll count the words in a sentence
  - We'll assume that every word is separated by a space
- Our basic algorithm goes like this:
  - Retrieve the text and store it in a String (sentence)
  - Examine each character in the sentence
    - We'll use a loop, which you'll study in the next unit
  - Classify each character: separator or not?
  - Use classification to keep track of number of words
- Download WordCount.java starter program

Counting Words II

- Variables already defined by the starter program
  - Each character is stored in char variable ch
  - boolean variable inAWord represents state
    - Are we inside a word or in between words?
  - Remember, we assume that spaces separate words
- Start by classifying the character using if-else
  - Is the character a separator? (space)
  - Or, is the character a non-separator? (a word character)

Counting Words III

- Suppose the character you read is not a space
  - That means you are now inside a word
  - Now there are two possibilities
- If the previous character was a space then
  - You have just gone from outside to inside a word
  - Count the word you just entered
  - Set inAWord to true
- If the previous character was not a space then
  - You have just read another character from the same word
  - Don't do anything
Counting Words III

- Suppose that the current character is a space
  - This means we are between words
  - You have two options here as well
- If the previous character was also a space then...
  - You are still between words, don't do anything
- If the previous character was not a space then
  - You have just gone from inside a word to outside
  - Set inAWord to false, but don't count the word

The Dangling else

- Nested if-else statements can cause problems when every if statement doesn't have a corresponding else
  - This is called the "dangling else" as shown here

```java
if (x > y)
  if (y > z)
    print("x > y and y > z");
  else
    print("x is not > y"); // false
```

- The else gets attached to the closest unmatched if
  - You can use braces to fix this or add another else

Deeply Nested if Statements

- Nesting can also be hard to understand if it is "deep"
  - I've left off the braces here

```java
if (clicked == btnOne)
  // handle btnOne
else
  if (clicked == btnTwo)
    // handle btnTwo
  else
    if (clicked == btnThree)
      // handle btnThree
    else
      if (clicked == btnFour) ...
```
Ladder-style if Statements

- A more readable style of formatting is ladder-style

```java
if (clicked == btnOne)
    // handle btnOne
else if (clicked == btnTwo)
    // handle btnTwo
else if (clicked == btnThree)
    // handle btnThree
else if (clicked == btnFour) ...
```

- Introduction to Exercise 10C

Lesson D: More Selection

- Using the `switch` statement
  - What type is used for the `switch` selector?
  - What can be used for `case` labels?
  - How do `case` blocks end?
  - What if no label matches the selector?
  - When to use `if-else` and when to `switch`
- The conditional operator
  - How to add selection to expressions
- Exercise 10D

The `switch` Statement

- With ladder-style `if-else-if`, you might sometimes find yourself writing code that looks like this:

```java
int choice = getUserChoice();
    if (choice == 1 ) ...
    else if (choice == 2) ...
    else if (choice == 3) ...
    else ...
```

- Java has a more efficient way to do this: the `switch`
What is the switch?

- A selection statement based on integer evaluation
- Named after the switches used by the telephone company
- Dial a number and get connected to your party
- Think also, of a menu or vending machine

The parts of a switch statement are:
- The test condition or switch selector
- A body surrounded by braces (required)
- A set of labeled code fragments called case blocks
- A "flow-of-control" statement called break

The switch Syntax I

- The switch selector can be any integer expression
- Includes int, short, byte, and char
- Can't use boolean, long, float, double, or String
- Placed in parentheses following keyword switch

- Each case block begins with the keyword case
- Followed by an integer constant and a colon
- Called the case label
- Can use literals or static final constants
- Cannot use variables or ranges

The switch Syntax II
### The switch Syntax III
- When a case label matches the switch selector
  - Java "jumps" to the code immediately following label
  - Code does not need to be in braces
- Continues executing all the remaining code in the switch
  - This is called "fall through"
  - Usually put a break statement at the end of every case
- If the switch selector doesn't match any case?
  - Jump to code past switch or provide a default case
  - Use keyword default followed by a colon

### The Conditional Operator I
- You can't use if-else in a formula
  - An if is a statement, not an expression
  - Means that you often have to create an additional variable
- The conditional operator is like if-else as an expression
  - Sometimes called the ternary or tertiary operator
  - Works similar to the =IF() function in Excel

### The Conditional Operator II
- The first operand must be a boolean condition
  - Followed by a ?
- The true and false parts are separated by a colon
  - Both parts must be compatible types
  - (amt > 0) ? 10 : 30; // OK numbers
  - (amt > 0) ? 10 : "Hi"; // No, different types
- You can "nest" conditions, but quickly becomes unreadable
- Introduction to Exercise 10D
Lesson E : Method Review

- Writing factory methods
  - Why create factory methods?
  - The Button factory example
- Passing arguments to methods
- Overloaded methods
  - How type-matching works
  - Building better Button factory methods with overloading
- Exercise 10E

Why Factory Methods?

- What is a factory method?
  - A static method that creates objects
  - Used instead of a constructor
- Here are several reasons why they are useful
  - Can be simpler than calling a constructor
  - Can be used when you don’t know what constructor to call
  - Toolkit.getDefaultToolkit()
  - Can create different kinds of objects instead of just one
    - NumberFormat.getCurrencyInstance();

Why a Button Factory?

- Suppose on your next big job, the boss decides that the company forms look too plain: he wants big, blue, buttons.
- Here’s one way you could do that:
  ```java
  Font fnt = new Font("Serif", Font.BOLD, 14);
  Button okBtn = new Button("OK");
  okBtn.setFont(fnt);
  okBtn.setForeground(Color.blue);
  ```
- The only problem is, you have to do this for every button
- Now imagine that next week, he wants them all red!
Create a `ButtonFactory` class to "house" your method(s)

1. Define static method that returns a `Button`
2. Create a local `Button` in method, modify as desired
3. Return a reference to the `Button` just like a primitive

Let's start with Step 1 and the method header:

```java
public static Button getBigBlueButton()
{
    // create and customize the button
}
```

Step 2A: Manufacture the object

Create a local variable of the type you're manufacturing
Normally you'll call the constructor

```java
public static Button getBigBlueButton()
{
    Button b = new Button("Big, bad, & blue");
}
```

Step 2B: Customize the object "to taste"

Make those buttons big and blue

```java
public static Button getBigBlueButton()
{
    Button b = new Button("Big, bad, & blue");
    b.setFont(new Font("Serif", Font.BOLD, 14);
    b.setForeground(Color.blue);
}
```
Creating a Button Factory IV

- Step 3: Deliver the results
- Use the return statement to deliver your button

```java
public static Button getBigBlueButton()
{
    Button b = new Button("Big, bad, & blue");
b.setFont(new Font("Serif", Font.BOLD, 14));
b.setForeground(Color.blue);
return b;
}
```

Using the Button Factory

- Use your Button factory instead of new

```java
Button b1 = ButtonFactory.getBigBlueButton();
barney.setLabel("Barney");
```

- See BlueButtons.java in class examples [run it]
- Real advantage comes when corporate colors change from blue to red. Only need to rewrite method.
- Would be more useful if we could specify text

Argument Review

- Most methods require additional information to carry out their work.
- A method that computes square roots, for instance, requires the number to operate on
- A method like getBigBlueButton() would be much more useful if you could specify the text to display
- This information is sent to your method via "arguments"
A formal argument is like a local variable, designed to hold information sent to a method. A local variable is defined like this:

```
public void myMethod()
{
    int myLocalVar = 32;
}
```

Variable `myLocalVar` is an "int-shaped bucket" that can hold only an int value.

Formal argument is just like a local variable, but:
- It is declared inside the argument list
- It is not initialized with the assignment operator, but it is given a value when the method is called

Here is `myMethod()` with a formal argument:

```
public void myMethod(double x)
{
    int myLocalVar = 32;
}
```

Actual arguments: values supplied when calling
- When you pass a variable, the value in the variable is copied to the formal argument.
- Called "pass by value"
Arguments for getBigBlueButton()

- Step 1: Write the method header (type and name)
  ```java
  public static Button getBigBlueButton(String text)
  ```

- Step 2: Use argument name inside the method body
  ```java
  Button b = new Button(text);
  ```

- Step 3: Call your method, passing a value
  ```java
  btn = ButtonFactory.getBigBlueButton("OK");
  ```

A More Generic Factory

- For a more flexible factory, provide several args
  - For each arg, supply type and name
  - See Buttons.java and [run the applet]

```java
public static Button getButton(String text, int fontSize, Color bColor) {
    Button b = new Button(text);
    b.setFont(new Font("Dialog", Font.BOLD, fontSize));
    b.setForeground(bColor);
    return b;
}
```

Overloaded Methods I

- Two methods in the same class can have the same name, if:
  - The number of arguments they require differs
  - The type or order of their arguments differ
  - The combination of name, arg type, number, and order, is called the method signature
  - It acts like "DNA" or fingerprint to uniquely identify a particular method
**Overloaded Methods II**

- Why bother? Allows more convenient coding
  - Suppose 99% of your buttons use a blue 24 pt font, 5% require a 30 pt font, and the remainder require fonts in varying sizes and colors.
- Your options include:
  - Using `getButton(String, 24, Color.blue);`
  - Writing multiple methods like `get24ptButton()` and `get30ptButton()`
  - Write three methods, call all three `getButton()`

---

**Overloaded Methods III**

- The overloaded `getButton()` methods:
  ```java
  getButton(String s) {}
  getButton(String s, int fSize) {}
  getButton(String s, int fSize, Color bClr) {}
  ```
- To use the methods, pass different actual arguments:
  ```java
  Button a = getButton("Normal");
  Button b = getButton("Bigger", 30);
  Button c = getButton("Custom", 6, Color.red);
  ```
- Introduction to Exercise 10E

---

**Finish Up**

- Assignments due on April 27
- Quiz 10, Homework 9 [ A SuperSodaPop ]
  - Remember to send up all of the files when you post
  - Remember that Tuesday is a holiday if you work at school
- Homework 10 [ Guess'0'Matic ] due May 4
  - Get an early start
  - A “guess the number” game
  - Adds a little bit of risk (you can lose actual Zippoids)
  - Will need to use if-else-if
  - Remember not to “hide” fields with local variables