Questions?

- Yesterday:
Today in COMP 110

- Lecture
  - More Branching Statements
  - Loops

- Lab
  - Open Lab Time
  - No New Lab Today
More About Branching Statements

- Short-Circuit v. Complete Evaluation
- switch Statement
- Ternary Operator
- System.exit(0);
More about && and ||

- && and || are short-circuiting operators
  - Consider these expressions
    - Expression_A && Expression_B
    - Expression_C || Expression_D
  - If Expression_A is false, does the value of Expression_B matter?
  - If Expression_C is true, does the value of Expression_D matter?
More about && and ||

- && and || are short-circuiting operators
- Consider these expressions
  - Expression_A && Expression_B
  - Expression_C || Expression_D

- If Expression_A is false, then Expression_B will not be evaluated
- If Expression_C is true, then Expression_D will not be evaluated
Short-Circuit Evaluation

- Can be useful to prevent runtime errors, consider:
  
  ```java
  if ((count > 0) && ((totalScore / count) >= 80)) {
    System.out.println("The average is at least 80");
  }
  ```

- If the value of count were zero, then the right expression would have produced an error (divide by zero)
  
  - Short-circuiting ensures that the right expression is not evaluated in this case
Complete Evaluation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Short-Circuit Evaluation</th>
<th>Complete Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Complete evaluation is not usually needed
- Both provide the same result in the absence of possible errors
Boolean Operator: $\wedge$ (XOR)

- XOR: Exclusive Or
- Evaluates to true if only one of the operands is true
- Requires complete evaluation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a OR b</th>
<th>a AND b</th>
<th>a XOR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Small if/elses

Consider the following code fragment

```java
if (altitude <= 100)
{
    System.out.println("Too Low!");
}
else
{
    System.out.println("It's ok.");
}
```

There is a single-statement alternative

```java
System.out.println((altitude <= 100) ? "Too Low!" : "It's ok");
```
The Conditional Operator: ?:

- Ternary Operator (3 operands)
  - Not a statement!

- Syntax
  - Boolean_Expression ? Expression_1 : Expression_2

- Expression_1 and Expression_2 must each evaluate to the same type
  - The result of the ?: operator evaluates to that type

- Recommend surrounding each expression with parentheses for clarity
Another ?: Example

Equivalent Code Fragments

- `double pay;
  if (hoursWorked <= 40)
  {
    pay = hoursWorked * PAY_RATE;
  }
  else
  {
    pay = 40 * PAY_RATE + (hoursWorked - 40) * 1.5 * PAY_RATE;
  }

- `double pay = (hoursWorked <= 40) ?
  (hoursWorked * PAY_RATE) :
  (40 * PAY_RATE + (hoursWorked - 40) * 1.5 * PAY_RATE);`
The Conditional Operator: ? :

- Problem
  - Often makes code harder to read

- Useful for
  - Simple yet similar conditional assignment statements
  - Complicated conditional method invocations that differ in only one small part
Updated Order of Operations

- From highest to lowest precedence
  - Unary: +, -, ++, --, !
  - Unary: (Type)
  - Binary: *, /, %
  - Binary: +, -
  - Binary: <, >, <=, >=
  - Binary: ==, !=
  - Binary: &
  - Binary: ^
  - Binary: |
  - Binary: &&
  - Binary: ||
  - Ternary: ?: 
  - Binary: All assignment operators (=, +=, ...)

6/27/2012
if (year == 1)
{
    System.out.println("freshman");
}
else if (year == 2)
{
    System.out.println("sophomore");
}
else if (year == 3)
{
    System.out.println("junior");
}
else if (year == 4)
{
    System.out.println("senior");
}
else
{
    System.out.println("unknown");
}
The switch statement

```java
switch(year) {
    case 1:
        System.out.println("freshman");
        break;
    case 2:
        System.out.println("sophomore");
        break;
    case 3:
        System.out.println("junior");
        break;
    case 4:
        System.out.println("senior");
        break;
    default:
        System.out.println("unknown");
        break;
}
```
The switch Statement

- Syntax
  - switch (Expression)
    {
      case Case_Literal_1:
        List_Of_Statements_1
        break;
      
      case Case_Literal_2:
        List_Of_Statements_2
        break;
        
      // ...
      
      case Case_Literal_n:
        List_Of_Statements_n
        break;
      
      default:
        List_Of_Default_Statements
        break;
    }

- The evaluated type of Expression and the type of the Case_Literals must be the same
  - Java 6 & 5\textsuperscript{th} Edition Textbook
    - Only int, char, and enum types are supported
  - Java 7 & 6\textsuperscript{th} Edition Textbook
    - Only int, char, enum types, and Strings are supported
    - Strings behave as if using the equals method

- The number of cases can be zero or more

- The default case is optional
The switch Statement

Syntax

switch (Expression)
{
    case Case_Literal_1:
        List_Of_Statements_1
        break;

    case Case_Literal_2i:
    case Case_Literal_2ii:
    case Case_Literal_2iii:
        List_Of_Statements_2
        break;

    // ...

    case Case_Literal_n:
        List_Of_Statements_n
        break;

    default:
        List_Of_Default_Statements
        break;
}

The number of literals assigned to a List_Of_Statements can be one or more

- If any of the literals match the value of the evaluated expression, then that list of statements will be executed.
The break Statement

- Syntax

```java
switch (Expression)
{
    case Case_Literal_1:
        List_Of_Statements_1
    break;

    case Case_Literal_2:
        List_Of_Statements_2
        break;

    // ...

    case Case_Literal_n:
        List_Of_Statements_n
        break;

    default:
        List_Of_Default_Statements
        break;
}
```

- The break statement returns the flow of the program to the statements that follow the switch statement.

- If omitted, execution falls through to the next case’s List_Of_Statements.

- In the example at left, if the expression evaluates to the first case literal value, then List_Of_Statements_1 is executed.

- Then List_Of_Statements_2 is executed.

- If the expression instead evaluates to the second case literal value, then List_Of_Statements_2 is executed.
Special Values as Cases

- Example:
  - `int dayOfWeek = 2; // Wednesday`

```
switch(dayOfWeek)
{
    case 0:
        System.out.println("It's Monday!");
        break;
    case 1:
        System.out.println("It's Tuesday!");
        break;
    case 2:
        System.out.println("It's Wednesday!");
        break;
    // ...
    case 5:
    case 6:
        System.out.println("It's the weekend!");
}
```
Enumerate Data Types

- Replaces the need for special numeric values with names

```java
public enum DayOfWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}

DayOfWeek dayOfWeek = DayOfWeek.WEDNESDAY;

switch(dayOfWeek) {
    case MONDAY:
        System.out.println("It's Monday!");
        break;
    case TUESDAY:
        System.out.println("It's Tuesday!");
        break;
    case WEDNESDAY:
        System.out.println("It's Wednesday!");
        break;
    //...
    case SATURDAY:
    case SUNDAY:
        System.out.println("It's the weekend!");
        break;
}
```
Enumerate Data Types

- Basic Syntax:
  - public enum EnumTypeName {
    VALUE_NAME_1,
    VALUE_NAME_2,
    VALUE_NAME_3,
    // ...
    VALUE_NAME_n
  }

- Location:
  - In separate EnumTypeName.java file
  - Inside an existing class, but outside of a method

- Use constant style naming scheme
  - Possible values of an enum type are constants

- Can contain zero or more VALUE_NAMEs

- enum types are classes
System.exit(0);

- A special method
  - Most methods return control of the program to the next statement when they finish
  - `System.exit(0)` causes the program to stop executing

- When using GUI code, `System.exit(0);` is required at the end of your `main` method for the program to terminate

- Also useful for situations where continued executing is pointless or not desired
Questions?
Logistics

- Next: Loops
- Later: Open Lab Time (No New Lab)
- Tomorrow: More Loops