Unit 3: Boolean Expressions, if Statements
Compound Boolean Expressions
and Comparing Objects

Adapted from:
1) Building Java Programs: A Back to Basics Approach
   by Stuart Reges and Marty Stepp
2) Runestone CSAwesome Curriculum

https://longbaonguyen.github.io
Evaluating logic expressions

Sometimes it is useful to use **nested if conditions**: if statements within if statements.

```java
// if x is odd
if(x % 2 != 0){
    // if x is positive
    if(x > 0){
        ...
    }
}
```

We can combine the above nested if conditions using **logical operators**.
• Tests can be combined using *logical operators*.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>(2 == 3) &amp;&amp; (-1 &lt; 5)</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>!(2 == 3)</td>
<td>true</td>
</tr>
</tbody>
</table>

• "Truth tables" for each, used with logical values *p* and *q*:

- **p** | **q** | **p && q** | **p || q**
  - true | true  | true       | true   
  - true | false | false      | true   
  - false| true  | false      | true   
  - false| false | false      | false  

- **p** | **!p**
  - true| false 
  - false| true
The following code

```java
// if x is odd
if (x % 2 != 0) {
    // if x is positive
    if (x > 0) {
        ...
    }
}
```

is equivalent to:

```java
// if x is odd and positive
if (x % 2 != 0 && x > 0) {
    ...
}
```
Using boolean

boolean goodAge    = age >= 21 && age < 29;
boolean goodHeight = height >= 78 && height < 84;
boolean rich       = salary >= 100000.0;

if ((goodAge && goodHeight) || rich) {
    System.out.println("Okay, let's go out!");
} else{
    System.out.println("It's not you, it's me... ");
}
Evaluating logic expressions

• Relational operators have lower precedence than math.

\[
5 \times 7 \geq 3 + 5 \times (7 - 1) \\
5 \times 7 \geq 3 + 5 \times 6 \\
35 \geq 3 + 30 \\
35 \geq 33 \\
true
\]

• Relational operators cannot be "chained" as in algebra.

\[
2 \leq x \leq 10 \\
true \leq 10 \\
(assume \ that \ x \ is \ 15) \\
error!
\]

– Instead, combine multiple tests with && or ||

\[
2 \leq x \ && x \leq 10 \\
true \ && false \\
false
\]
## Order of Operations

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest</td>
<td>**</td>
<td>exponentiation</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>negation</td>
</tr>
<tr>
<td></td>
<td>*, /, %</td>
<td>multiplication, division, modulo</td>
</tr>
<tr>
<td></td>
<td>+, -</td>
<td>adding, subtraction</td>
</tr>
<tr>
<td></td>
<td>==, !=, &lt;, &gt;, &lt;=, &gt;=</td>
<td>comparisons (relationals)</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>logical not</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>logical or</td>
</tr>
<tr>
<td>lowest</td>
<td>=</td>
<td>assignment</td>
</tr>
</tbody>
</table>
Evaluating logic expressions

AND is evaluated before OR.

```cpp
int x = 2;
int y = 4;
int z = 5;

z > 2 || x > 3 && y < 3 ;
// true if evaluate && before ||
// false if evaluate || before &&

// the correct answer is true: &&
// MUST be evaluated before ||
```
Logical questions

• What is the result of each of the following expressions?

```c
int x = 42;
int y = 17;
int z = 25;

- y < x && y <= z
- x % 2 == y % 2 || x % 2 == z % 2
- x <= y + z && x >= y + z
- !(x < y && x < z)
- (x + y) % 2 == 0 || !((z - y) % 2 == 0)
```

Answers:
- true
- false
- true
- true
- false
// Returns the larger of the two given integers.
public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    else {
        return b;
    }
}

• Methods can return different values using if/else
  – Whichever path the code enters, it will return that value.
  – Returning a value causes a method to immediately exit.
  – All paths through the code must reach a return statement.
public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    // Error: not all paths return a value
}

- The following also does not compile:

public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    else if (b >= a) {
        return b;
    }
}

- The compiler thinks if/else/if code might skip all paths, even though mathematically it must choose one or the other.
public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    else{
        return b;
    }
}

OR

public static int max(int a, int b) {
    if (a > b) {
        return a;
    }
    
    return b;
}

Write a method `quadrant` that accepts a pair of real numbers \( x \) and \( y \) and returns the quadrant for that point:

- **Example:** `quadrant(-4.2, 17.3)` returns 2

  If the point falls directly on either axis, return 0.
if/else, return answer

```java
class Quadrant {
    public static int quadrant(double x, double y) {
        if (x > 0 && y > 0) {
            return 1;
        }
        else if (x < 0 && y > 0) {
            return 2;
        }
        else if (x < 0 && y < 0) {
            return 3;
        }
        else if (x > 0 && y < 0) {
            return 4;
        }
        else { // at least one coordinate equals 0
            return 0;
        }
    }
}
```
De Morgan's Law

- **De Morgan's Law**: Rules used to negate boolean tests.
  - Useful when you want the opposite of an existing test.

\[
!(a \ & \ & \ b) = !a \ | \ | \ !b \\
!(a \ | \ | \ b) = !a \ & \ & \ !b
\]

- Example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Negated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x == 7 \ &amp; \ &amp; \ y &gt; 3) {</td>
<td>if (x != 7 \</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
De Morgan’s Law

In Java:

```java
!((age < 12) || (age >= 65))
```

In English: *It is not the case that age less than 12 or age greater than or equal to 65. !!!?*

Simplify using de Morgan’s Law:

```java
!(age < 12) && !(age >= 65)
```

The reverse the meaning of the relational expressions:

```java
(age >= 12) && (age < 65)
```

That is, *when age is at least 12 and less than 65.*
Truth Tables

Truth tables can be used to prove Boolean identities.

For example, use a truth table to prove:

\[ !(a \land b) = !a \lor !b \]

| a   | b   | !(a && b) | !a || !b |
|-----|-----|----------|---------|
| true| true| false    | false   |
| true| false| true     | true    |
| false| true| true     | true    |
| false| false| true     | true    |

Since both expressions have the same values in all cases, they are equivalent.
Java stops evaluating a test if it knows the answer.

- `&&` stops early if any part of the test is `false`
- `||` stops early if any part of the test is `true`

```java
int count = <input from user>;
int sum = <input from user>;
if(count > 0 && (double)sum / count > 10){
    System.out.println("average > 10");
}
else{
    System.out.println("count <= 0 or average <= 10");
}
```

If `count = 0` above, there is a potential to divide by 0. However, short-circuit prevents this since `count > 0` is false, it stops early and no division by zero was performed.
Write a method named `isVowel` that returns whether a `String` is a vowel (a, e, i, o, or u). Assume all letters are lowercase.

- `isVowel("q")` returns false
- `isVowel("a")` returns true
- `isVowel("e")` returns true

```java
public static boolean isVowel(String s) {
    return s.equals("a") || s.equals("e") || s.equals("i") || s.equals("o") || s.equals("u");
}
```
Change the above method into an `isNonVowel` method that returns whether a `String` is any character except a vowel.

- `isNonVowel("q")` returns true
- `isNonVowel("a")` returns false
- `isNonVowel("e")` returns false

What’s the wrong strategy?

```
// Enlightened "Boolean Zen" version
public static boolean isNonVowel(String s) {
    return !s.equals("a") && !s.equals("e") &&
           !s.equals("i") && !s.equals("o") &&
           !s.equals("u");
}
```
Boolean practice questions

Use `isVowel` to write `isNonVowel`.

```java
// Enlightened "Boolean Zen" version
public static boolean isNonVowel(String s) {
    return !isVowel(s);
}
```
Comparing Objects

Two objects are considered **aliases** when they both reference the same object. Comparing using == check whether two variables are aliases. Consider the Sprite class we discussed in Unit 2 used to represent a game character.

```java
public class Aliases {
    public static void main(String[] args) {
        Sprite player = new Sprite(30, 50);
        Sprite another = player;
        System.out.println(player == another); // true
    }
}
```

Both object references `player` and `another` points to the same address hence the same object in memory.
Comparing Objects

Two **different** objects can have the same attributes/data.

```java
public class Aliases2 {
    public static void main(String[] args) {
        Sprite player = new Sprite(30, 50);
        Sprite another = new Sprite(30, 50);
        System.out.println(player == another); // false!
        System.out.println(player != another); // true!
    }
}
```

The references `player` and `another` above are two different Sprite objects (created individually using `new`) but both are located at the same coordinate.
We saw that for String objects, `==` is used to check if the two String references point to the same object whereas the `equals` method check if they have the same characters.

```java
String a = "hi";
String b = new String("hi");
System.out.println(a == b); // false, different objects
System.out.println(a.equals(b)); // true
```
Later in Unit 5 when we write our own objects, it will be useful to implement the `equals` method for our class so check whether two different objects are equivalent (same data).

For example, consider Point objects with attributes x and y representing points on the plane. Although the following two points are distinct programmatically. They are equivalent mathematically. The `equals` method will allow us to detect this. More on this later.

```java
Point a = new Point(3,4);
Point b = new Point(3,4);
System.out.println(a == b); // false, different objects
System.out.println(a.equals(b)); // true
```
Lab 1: BMI

Create a new repl for this lab.

Formula for body mass index (BMI):

\[ BMI = \frac{weight}{height^2} \times 703 \]

Write a program that produces output like the following. Use Scanner for input.

<table>
<thead>
<tr>
<th>BMI</th>
<th>Weight class</th>
</tr>
</thead>
<tbody>
<tr>
<td>below 18.5</td>
<td>underweight</td>
</tr>
<tr>
<td>[18.5 – 25)</td>
<td>normal</td>
</tr>
<tr>
<td>[25.0 – 30)</td>
<td>overweight</td>
</tr>
<tr>
<td>30.0 and up</td>
<td>obese</td>
</tr>
</tbody>
</table>

Height (in inches): 70.0
Weight (in pounds) 194.25
BMI = 27.868928571428572
Overweight
Lab 1: BMI

Your program must include two methods: 1) the method `bmi` which takes two double parameters `height` and `weight` and returns the bmi and 2) the method `weightClass` which takes two double parameters `height` and `weight` and returns a string classifying the weight class. The `weightClass` method must call the `bmi` method!

```java
public static double bmi(double height, double weight)
{
...
}

public static String weightClass(double height, double weight)
{
...
}
```
Go to this repl (https://repl.it/@LongNguyen18/Conditional-Lab) and complete the code as requested by the comments. You will write four methods:

1) lucky7
2) mult35
3) blackjack
4) enoughBricks
References

1) Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp

2) Runestone CSAwesome Curriculum: https://runestone.academy/runestone/books/published/csawesome/index.html

For more tutorials/lecture notes in Java, Python, game programming, artificial intelligence with neural networks:

https://longbaonguyen.github.io