

# **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

# Evaluating logic expressions

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

```
// 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**.

# Logical operators

- Tests can be combined using *logical operators*:

| Operator | Description | Example              | Result |
|----------|-------------|----------------------|--------|
| &&       | and         | (2 == 3) && (-1 < 5) | false  |
|          | or          | (2 == 3)    (-1 < 5) | true   |
| !        | not         | !(2 == 3)            | true   |

- "Truth tables" for each, used with logical values  $p$  and  $q$ :

| <b>p</b> | <b>q</b> | <b>p &amp;&amp; q</b> | <b>p    q</b> |
|----------|----------|-----------------------|---------------|
| true     | true     | true                  | true          |
| true     | false    | false                 | true          |
| false    | true     | false                 | true          |
| false    | false    | false                 | false         |

| <b>p</b> | <b>!p</b> |
|----------|-----------|
| true     | false     |
| false    | true      |

# Combining Tests

The following code

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

is equivalent to:

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

# Using boolean

- Why is type `boolean` useful?
  - Can capture a complex logical test result and use it later
  - Can write a method that does a complex test and returns it
  - Makes code more readable
  - Can pass around the result of a logical test (as param/return)

```
int age = 21, height = 88;  
double salary = 100000;
```

```
boolean goodAge      = age >= 12 && age < 29; //true  
boolean goodHeight   = height >= 78 && height < 84; //false  
boolean rich         = salary >= 100000.0; //true
```

# 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 * 7 >= 3 + 5 * (7 - 1)
```

```
5 * 7 >= 3 + 5 * 6
```

```
35 >= 3 + 30
```

```
35 >= 33
```

```
true
```

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

```
2 <= x <= 10
```

```
true <= 10
```

```
error!
```

(assume that x is 15)

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

```
2 <= x && x <= 10
```

```
true && false
```

```
false
```

# Order of Operations

| <b>Precedence</b> | <b>Operator</b>      | <b>Operation</b>                 |
|-------------------|----------------------|----------------------------------|
| highest           | **                   | exponentiation                   |
|                   | -                    | negation                         |
|                   | *, /, %              | multiplication, division, modulo |
|                   | +, -                 | adding, subtraction              |
|                   | ==, !=, <, >, <=, >= | comparisons(relationals)         |
|                   | not                  | logical not                      |
|                   | and                  | logical and                      |
|                   | or                   | logical or                       |
| lowest            | =                    | assignment                       |



# Evaluating logic expressions

AND is evaluated before OR.

```
int x = 2;
```

```
int y = 4;
```

```
int z = 5;
```

```
boolean a = 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?

```
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

# if/else with return

*// 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.

# All paths must return

```
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.

# Correction

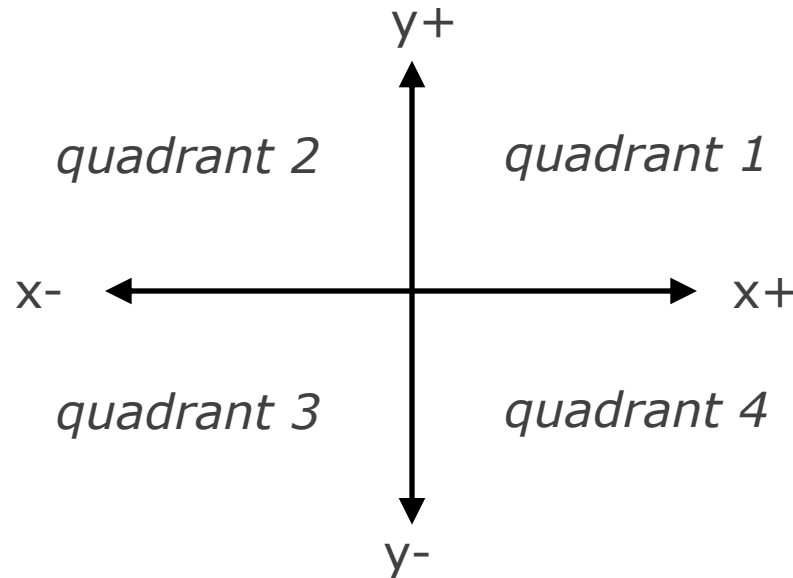
```
public static int max(int a, int b) {  
    if (a > b) {  
        return a;  
    } else{  
        return b;  
    }  
}
```

**OR**

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

# if/else, return question

- 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

```
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:

| Original Code  | Negated Code   |
|--|--|
| <pre>if (x == 7 &amp;&amp; y &gt; 3) {<br/>    ...<br/>}</pre> | <pre>if (x <b>!=</b> 7 <b>  </b> y <b>&lt;=</b> 3) {<br/>    ...<br/>}</pre> |



# De Morgan's Law

In 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:

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

The reverse the meaning of the relational expressions:

```
(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:

$$\neg (a \ \&\& \ b) = \neg a \ || \ \neg b$$

| <b>a</b> | <b>b</b> | <b>!(a &amp;&amp; b)</b> | <b>!a    !b</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.

# "Short-circuit" evaluation

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`

```
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.

# Boolean practice questions

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`

```
public static boolean isVowel(String s) {  
    return s.equals("a") || s.equals("e") ||  
           s.equals("i") || s.equals("o") ||  
           s.equals("u");  
}
```

OR alternatively, use `indexOf`:

```
public static boolean isVowel(String s) {  
    return s.length() == 1 && "aeiou".indexOf(s) != -1;  
}
```

# Boolean practice questions

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`.

```
// 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.

```
public class Main{  
    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.

```
public class Main{  
    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.



# equals

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.

```
String a = "hi";  
String b = new String("hi");  
System.out.println(a == b); // false, different objects  
System.out.println(a.equals(b)); // true
```

**Note: DO NOT USE `==` for Strings! Always use `.equals`**

# equals

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.

```
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):

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

| BMI         | Weight class |
|-------------|--------------|
| below 18.5  | underweight  |
| [18.5 – 25) | normal       |
| [25.0 – 30) | overweight   |
| 30.0 and up | obese        |

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

```
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!**

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

```
public static String weightClass(double height,
    double weight)
{...}
```

# 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>