9/28/2022

# 3.5 Review

# Operator Associativity

On the cafeteria repl.it, some students wrote code like this:

```
if (food1.isFruit() || (food2.isFruit() || (food3.isFruit() || (food4.isFruit() ||
food5.isFruit()))))) {
      …
}
```

Understandable… all our examples of && and || showed only two operands!

It is legal, however, to chain || together:

```
if (food1.isFruit() || food2.isFruit() || food3.isFruit() || food4.isFruit() || food5.isFruit()) {
      …
}
```

# Operator Associativity

&& and || are defined as having two operands. The && and || operators are left associative.

What is the meaning of left-associative?

Left-associative operators of the same precedence are evaluated in order from left to right. For example, addition and subtraction have the same precedence and they are left-associative.

In the expression `10-4+2`, the subtraction is done first because it is to the left of the addition, producing a value of `8`.

`10-4+2` is equivalent to `(10-4)+2`, not `10-(4+2)`.

```
food1.isFruit() || food2.isFruit() || food3.isFruit() || food4.isFruit() || food5.isFruit()

(food1.isFruit() || food2.isFruit()) || food3.isFruit() || food4.isFruit() || food5.isFruit()

((food1.isFruit() || food2.isFruit()) || food3.isFruit()) || food4.isFruit() || food5.isFruit()

(((food1.isFruit() || food2.isFruit()) || food3.isFruit()) || food4.isFruit()) ||
food5.isFruit()
```

# Operator Associativity and Precedence

What does this do?

```
hasTicket || onGuestList && standingInLine
```

&& has higher precedence, so this is equivalent to

```
hasTicket || (onGuestList && standingInLine)
```

but probably what you want is

```
(hasTicket || onGuestList) && standingInLine
```

… which means something very different.

Use parentheses when needed!

**Operator Precedence**

| Operators | Precedence |
|---|---|
| postfix | *expr*++ *expr*-- |
| unary | ++*expr* --*expr* +*expr* -*expr* ~ ! |
| multiplicative | * / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | \| |
| logical AND | && |
| logical OR | \|\| |
| ternary | ? : |
| assignment | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

# Returning Boolean Expressions

A lot of students wrote code like this:

```
if (hasFruit() && hasVegetable() && hasGrain() && hasDairy() && hasProtein()) {
      return true;
} else {
      return false;
}
```

This works fine, but you can just return a boolean expression:

```
return hasFruit() && hasVegetable() && hasGrain() && hasDairy() && hasProtein();
```

Odd but true: `return` doesn't require parentheses, but `if` does.

# 3.6
# Equivalent Boolean Expressions

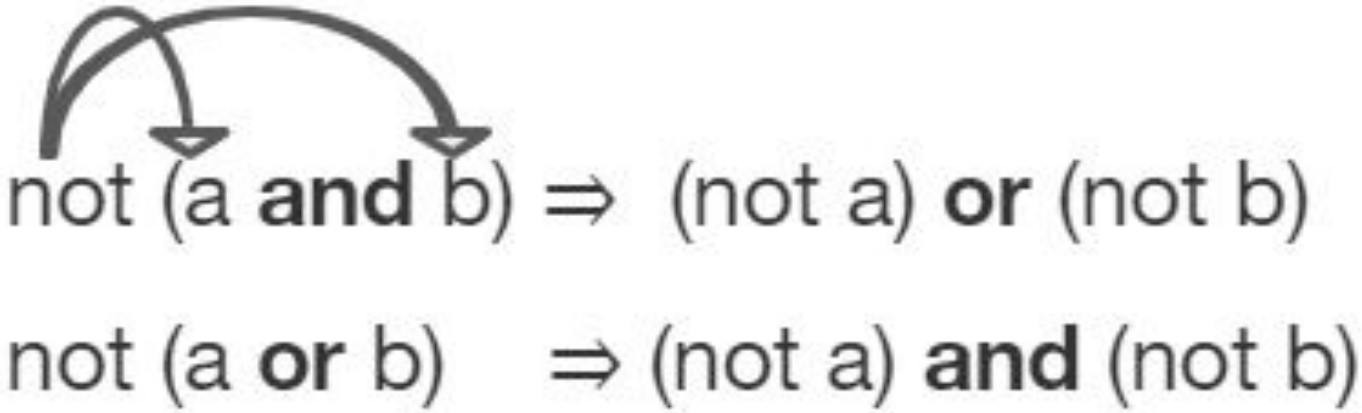
# De Morgan's Laws

Augustus De Morgan (27 June 1806 – 18 March 1871) was a British mathematician and logician. He formulated De Morgan's Laws.

$$(P \wedge Q) \iff \neg(\neg P \vee \neg Q),$$
$$(P \vee Q) \iff \neg(\neg P \wedge \neg Q).$$

# DeMorgan's Laws

Rules by which we can simplify Booleans to make them easier to read or interpret



# DeMorgan's Laws

In Java:

- `!(a && b)` is equivalent to `!a || !b`
- `!(a || b)` is equivalent to `!a && !b`

# DeMorgan's Laws

`!(a && b)` is equivalent to `!a || !b`

There are not **both cats and dogs here**.

`!(cats && dogs)`

There are **not cats here**, **not dogs here,** or both.

`!cats || !dogs`

| | |
|---|---|
| cats | true |
| dogs | true |
| !(cats && dogs) | false |
| !cats \|\| !dogs | false |

| | |
|---|---|
| cats | true |
| dogs | false |
| !(cats && dogs) | true |
| !cats \|\| !dogs | true |

| | |
|---|---|
| cats | false |
| dogs | true |
| !(cats && dogs) | true |
| !cats \|\| !dogs | true |

| | |
|---|---|
| cats | false |
| dogs | false |
| !(cats && dogs) | true |
| !cats \|\| !dogs | true |

# DeMorgan's Laws

`!(a || b)` is equivalent to `!a && !b`

There are not **cats here or dogs here**.

`!(cats || dogs)`

There are **not cats here**, and there are **not dogs here.**

`!cats && !dogs`

| | |
|---|---|
| cats | true |
| dogs | true |
| !(cats \|\| dogs) | false |
| !cats && !dogs | false |

| | |
|---|---|
| cats | true |
| dogs | false |
| !(cats \|\| dogs) | false |
| !cats && !dogs | false |

| | |
|---|---|
| cats | false |
| dogs | true |
| !(cats \|\| dogs) | false |
| !cats && !dogs | false |

| | |
|---|---|
| cats | false |
| dogs | false |
| !(cats \|\| dogs) | true |
| !cats && !dogs | true |

# Negated Relational Expressions

For negated relational expressions, **you can flip the operator and remove the !**

- !(c == d) is equivalent to (c != d)
- !(c != d) is equivalent to (c == d)
- !(c < d) is equivalent to (c >= d)
- !(c > d) is equivalent to (c <= d)
- !(c <= d) is equivalent to (c > d)
- !(c >= d) is equivalent to (c < d)

# DeMorgan's Laws

**Truth Tables** are a way to show that two boolean expressions are equivalent.

You can kind of think of this as a proof, but it also a tool to let you think about booleans more easily.

Let's use truth tables to demonstrate DeMorgan's Laws:

- !(a && b) is equivalent to !a || !b
- !(a || b) is equivalent to !a && !b

| a | b | !(a && b) | !a \|\| !b |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

# DeMorgan's Laws: Exercise

On your own, on pen and paper, use truth tables to check if the following are equivalent:

1. !(x == 0 || x >= 1) ⇔ !(x == 0) && !(x >= 1)

2. !(x == 0 || x >= 1) ⇔ x != 0 && x < 1

Can you also demonstrate that the expressions are equivalent using DeMorgan's Laws?

!(a && b) ⇔ !a || !b
!(a || b) ⇔ !a && !b

# DeMorgan's Laws: Exercise

Use DeMorgan's Laws and flipping operators to simplify the following expressions:

1. !(x > 2 && y < 4)
2. !(x == 2 && y > 4)
3. !(x!=5 && y!=7)
4. !(x<= 5 && y > 7)

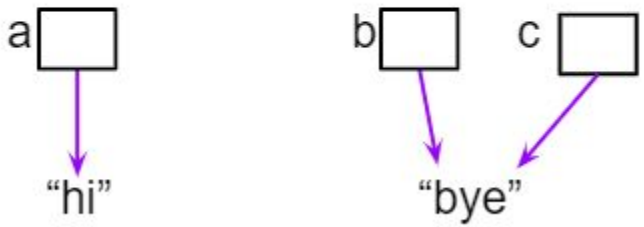
Cheat Sheet:

| | | |
|---|---|---|
| !(a && b) | ⇔ | !a \|\| !b |
| !(a \|\| b) | ⇔ | !a && !b |
| !(c == d) | ⇔ | (c != d) |
| !(c != d) | ⇔ | (c == d) |
| !(c < d) | ⇔ | (c >= d) |
| !(c > d) | ⇔ | (c <= d) |
| !(c <= d) | ⇔ | (c > d) |
| !(c >= d) | ⇔ | (c < d) |

# 3.7
# Object Equality

# Object/String Equality

```
String a = new String("hi");
String b = new String("bye");
String c = b;        // c is now an alias for b
```




```
System.out.println(c);
System.out.println(b == c);
System.out.println(b.equals(c));
```

# Object/String Equality

```
String s1 = new String("Hello");

String s2 = new String("Hello");

System.out.println(s1 == s2);

System.out.println(s1.equals(s2));
```

*What's going on in memory here?*

What will this code print out?

# Object/String Equality

The .equals() method is a special method that all *classes* can define that decide whether or not an object of that class is equal to something else.

The default implementation of .equals is Object.equals. The documentation says:

> The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

This is a lot of fancy talk that basically means Object.equals is this:

```
public boolean equals(Object object) {
    return this == object;
}
```

# Reference Variables

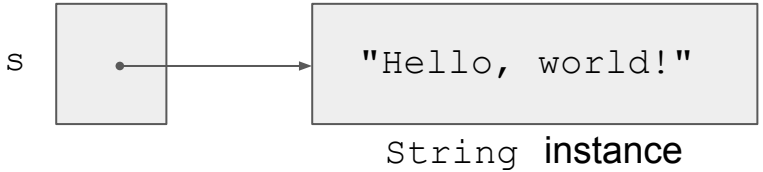
# Nope! It's a String reference to null!

`s` is a reference variable of type `String`.

A `String` reference, like all object references, either points to a `String` object instance, or is `null`

```
String s;
```

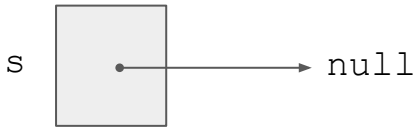

`s = "Hello, world!";` ← `s` started out `null`, but now points to a legit string




# null

**null** means *nothing*, no object is pointed to.

You can use **null** to check if a variable points to an object instance or not.




```
String s;

if (s == null) {

    System.out.println("s doesn't exist!");

}
```

*What's going on in memory here?*

# null

If you try usings methods or attributes of a `null` object reference, Java throws `NullPointerException`:

```
String s;

if (s.indexOf("a") >= 0) {

    System.out.println(s + " contains an a"); ←throws NullPointerException

}
```

Some exceptions are not bugs in your program, like an exception thrown if the network is down.

But a `NullPointerException` usually means you need to fix your code.

# null

How can you get around this? One way is to use Java's optimization of **short-circuiting** boolean expressions!

```
if (s != null && s.indexOf("a") >= 0) {
    System.out.println(s + " contains an a");
}
```

If the left side of a `&&` expression is `false`, Java will immediately cut out ahead of time. (Same for an `||` expression when the left side evaluates to `true`)

```
if (a && b) { … }
```

```
if (a || b) { … }
```

# null checks

Code like this that checks for null in Java is common.

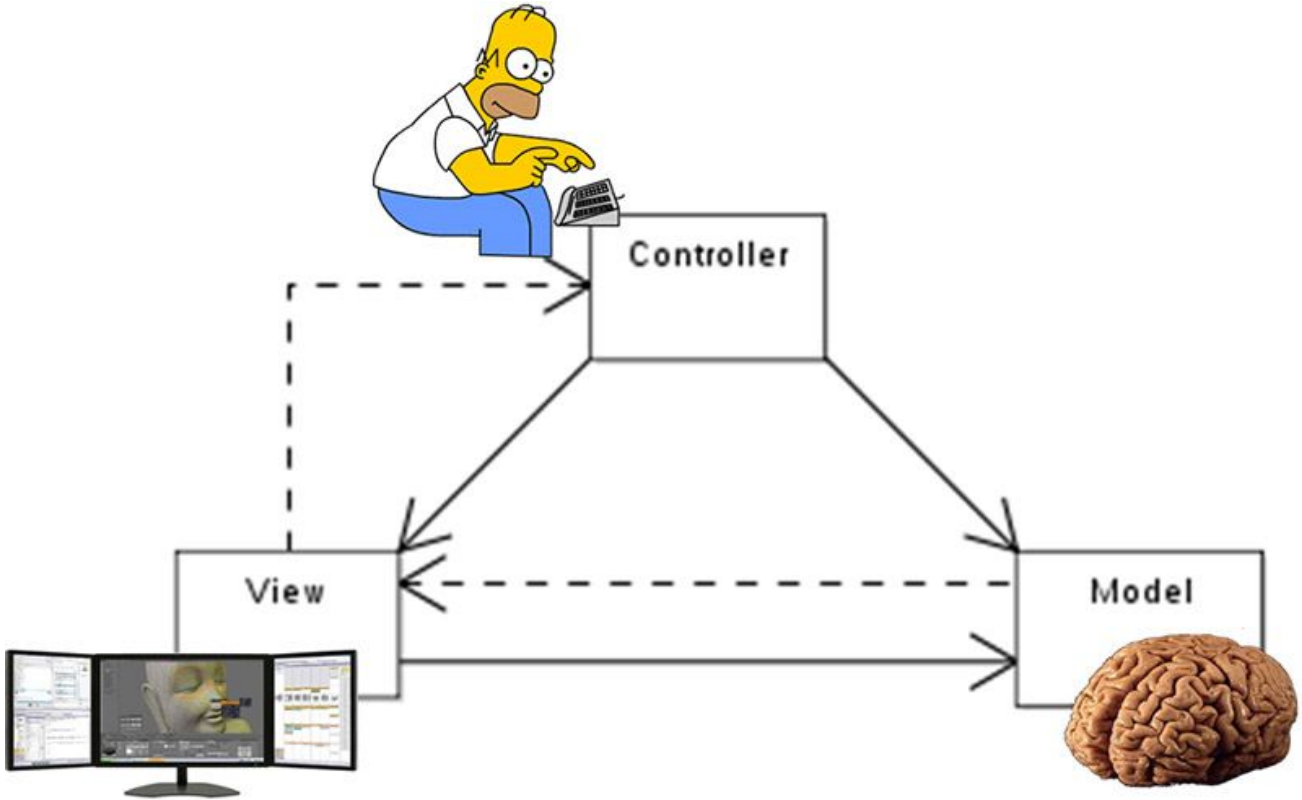
```
if (s != null && s.indexOf("a") >= 0) {
    System.out.println(s + " contains an a");
}
```

However, your code should not check for `null` if there isn't a legitimate reason for a variable to be `null`. Reason about whether it should ever be `null`.

Don't litter your code with unnecessary null checks! This is actually what NullPointerException is for.

"Coding defensively" is good, but it is possible to code TOO defensively.

# Concept in today's Replit: Model-View-Controller



# Practice!

Today's Repl.it will give you some else-if practice, and some boolean logic practice:

Repl: [truthGame]

```
//
// This is an incomplete implementation of the game
// "Two Truths and a Lie."
// The user will be prompted for three statements about themselves,
// two truths and one lie.
// The computer will then guess which statement is the lie.
// If the computer is wrong, it should make another guess.
//
// This game is built using a design pattern called Model-View-Controller,
// or MVC. (https://en.wikipedia.org/wiki/Model-view-controller)
// This is a very popular pattern used by many frameworks
// used to build Web sites and mobile and desktop applications.
// All of the state (variables) of the game and the game logic
// are in the TruthGameModel class, which you will be implementing.
//
// The TruthGameModel class is not finished, but it comes with
// a test suite, TruthGameModelTest, which will verify that it is
// correctly implemented. This test suite will run automatically
// when the program starts, and the game won't run until the self-test
// passes.
//
```