To the Roots of Dispatch and Objects

Deeply understanding the essence of method dispatch

Stéphane Ducasse and Damien Cassou

http://stephane.ducasse.free.fr/
stephane.ducasse@inria.fr

Table Of Content

A cornerstone point

Message passing, "method invocation", method dynamic selection are the heart of object-oriented programming. The lecture will rethink this essential and fundamental aspect of object-oriented programming using some simple examples. After this lecture you will never look the same your programs.

Motto

- Let’s open our eyes, look, understand, and deeply understand the underlying design aspects of object-oriented programming.

Booleans in Pharo

Booleans

```
3 > 0
   ifTrue: ['positive']
   ifFalse: ['negative']
—> 'positive'
```

- Conceptually `ifTrue:ifFalse:` is a message sent to an object: a boolean!
- `ifTrue:ifFalse:` is in fact radically optimized by the compiler but you can implement another one such as `siAlors:sinon:` and try it at home.

Booleans

In Pharo, Booleans have nothing special, just a superb implementation!

- & | not
- or: and: (lazy)
- xor:
- ifTrue:ifFalse:
- ifFalse:ifTrue:
- =>
- ....

Exercices

Three Exercises

- 1 Implement `not`
- 2 Implement `|` (or)
- 3 Why such exercises? What these exercises want to show us?

Exercise 1: Implement not

Exercise 1: Implement not

- Propose an implementation of `not` in a world where you do not have Booleans implemented yet.
- You only have objects and messages.

```
false not
—> true

true not
—> false
```

Exercise 2: Implement | (Or) ifTrue: ifFalse:

Exercise 2: Implement | (Or)

- Propose an implementation of `or` (named `|` in Pharo) in a world where you do not have Booleans.
- You only have objects and messages.

```
true | true −> true
true | false −> true
true | anything −> true

false | true −> true
false | false −> false
false | anything −> anything
```

Exercise 2: Variation - Implement ifTrue:ifFalse:

- Propose an implementation of `ifTrue:ifFalse:` in a world where you do not have Booleans.
- You only have objects, messages and closures.

```
false ifTrue: [ 3 ] ifFalse: [ 5 ]
−> 5

true ifTrue: [ 3 ] ifFalse: [ 5 ]
−> 3
```

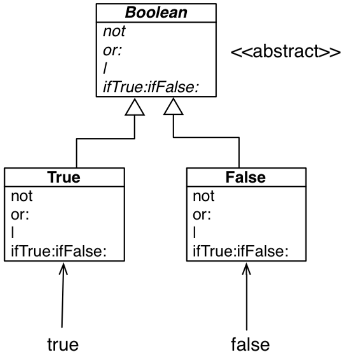
Boolean Implementation

Booleans Implementation Hint One

- The solution does not use conditionals
- Else we would obtain a recursive definition of `ifTrue:ifFalse:`

Boolean Implementation Hint Two

- The solution uses three classes: `Boolean`, `True` and `False`
- `false` and `true` are unique instances described by their own classes
- `false` is an instance of the class `False`
- `true` is an instance of the class `True`



How do we express choice in OOP?

- We send messages to objects

```
...
...
x color
—> Color red
```

- where `x` can be a button, a pane, a window, a magic card, a bird
- Let’s the receiver decide
- Do not ask, tell

Boolean not implementation

- Class `Boolean` is an abstract class that implements behavior common to true and false. Its subclasses are `True` and `False`.
- Subclasses must implement methods for logical operations `&`, `not`, and controls `and:`, `or:`, `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, `ifFalse:ifTrue:`

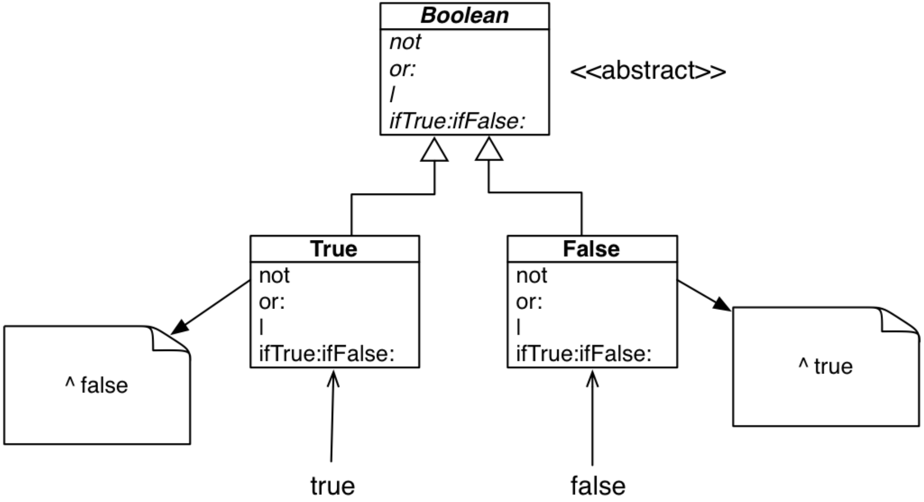
```
Boolean>>not
    "Abstract method. Negation: Answer true if the receiver is false, answer false if the receiver is true."
    self subclassResponsibility
```

Not implementation in two methods

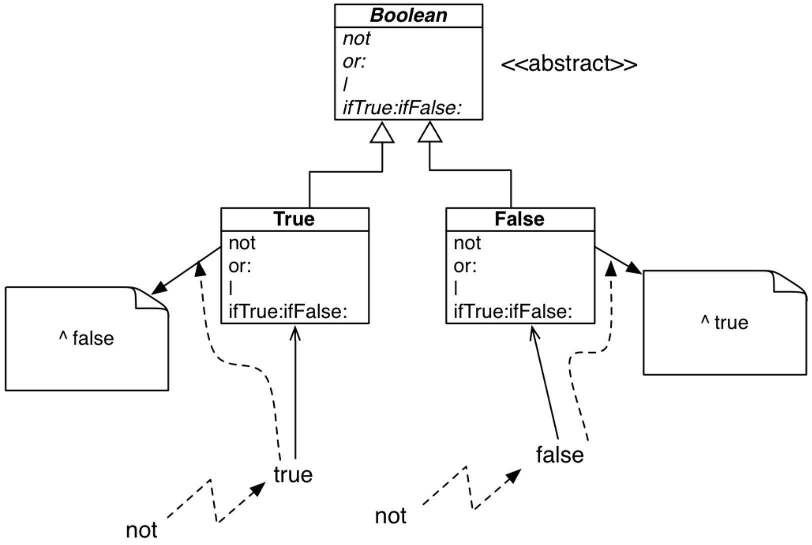
```
False>>not
   "Negation -- answer true since the receiver is false."
   ^ true
```

```
True>>not
   "Negation--answer false since the receiver is true."
   ^ false
```

Not implementation in two methods



Not implementation in two methods



| (Or)

```
true | true –> true
true | false –> true
true | anything –> true

false | true –> true
false | false –> false
false | anything –> anything
```

Boolean» | aBoolean

```
Boolean>> | aBoolean
    "Abstract method. Evaluating disjunction (OR): Evaluate the argument. Answer true if either the receiver
     or the argument is true."
    self subclassResponsibility
```

False» | aBoolean

```
false | true −> true
false | false −> false
false | anything −> anything
```

False» | aBoolean

```
false | true –> true
false | false –> false
false | anything –> anything
```

```
False >> | aBoolean
   "Evaluating disjunction (OR) –– answer with the argument, aBoolean."
   ^ aBoolean
```

True» | aBoolean

```
true | true –> true
true | false –> true
true | anything –> true
```

True» | aBoolean

```
true | true –> true
true | false –> true
true | anything –> true
```

```
True>> | aBoolean
    "Evaluating disjunction (OR) –– answer true since the receiver is true."
    ^ true
```

True» | aBoolean

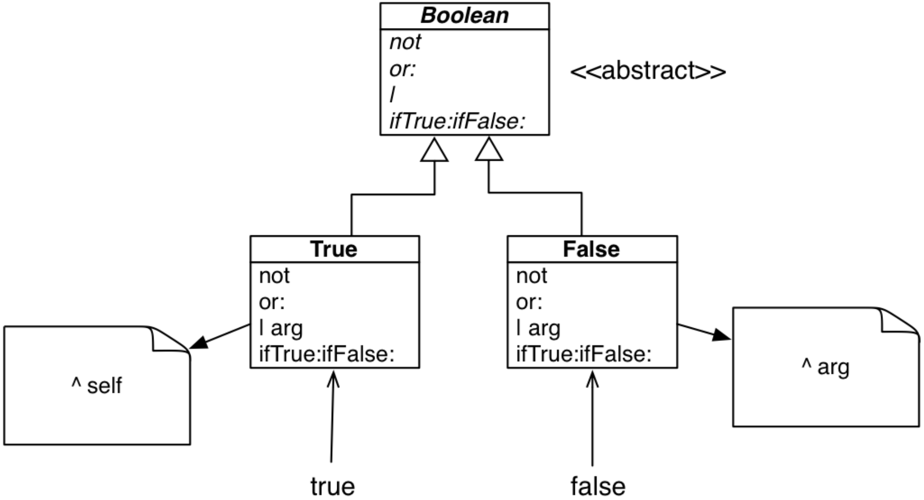
```
true | true –> true
true | false –> true
true | anything –> true
```

```
True>> | aBoolean
    "Evaluating disjunction (OR) –– answer true since the receiver is true."
    ^ true
```

- The object `true` is indeed the receiver of the message!

```
True>> | aBoolean
    "Evaluating disjunction (OR) –– answer true since the receiver is true."
    ^ self
```

Or implementation in two methods



So what ?

Ok so what?

- You will probably not implement `Booleans` in the future
- So is it really that totally useless?
- What is the lesson to learn?

Message sends act as case statements

- Message sends act as case statements
- But with messages, the case statements is dynamic in the sense that it depends on the classes loaded and the objects to which the message is sent.

Sending a message is making a choice

- The execution engine will select the right method depending on the class of the receiver.
- Each time you send a message, the system will select the method corresponding to the receiver.
- Sending a message is a choice operator.

Question

- Could we have been able to implement the same implementation in only one class?

Question

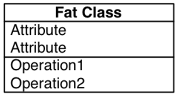
- Could we have been able to implement the same implementation in only one class?
- NO NO NO

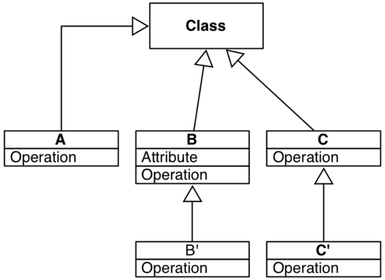
Class play case roles

- To have the possibility to activate the choice operator you must have choices = classes
- If we would have said that the `Boolean` would be composed of only one class, we could not have use dynamic binding.

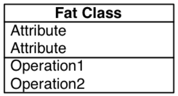
A Class Hierarchy is a Skeleton for Dynamic Dispatch

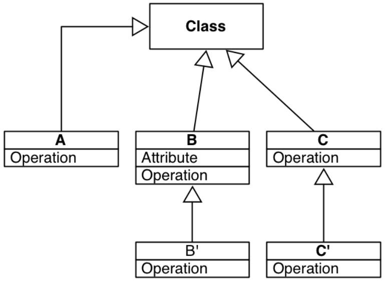
- A class hierarchy is the exoskeleton for dynamic binding.
- Compare the solution with one class vs. a hierarchy.





Advantages of small class hierarchy





- The hierarchy provides a way to specialize behavior.
- It is also more declarative in the sense that you only focus on one class.
- It is more modular in the sense that you can package different classes in different packages.
- You can also load classes separately.

Do not ask, tell

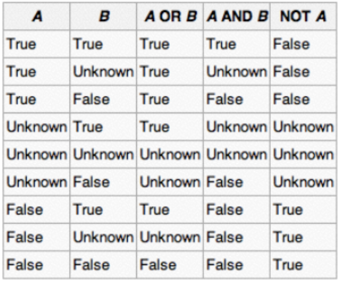
- Sending a message let the receiver decide.
- The client does not have to decide.
- Client code is not fixed. Different receivers may be substitued dynamically

Avoid Conditionals

- Use objects and messages, when you can.
- The execution engine acts as a conditional switch: Use it!
- Check the AntiIfCampaign.

Follow-up: Implement ternary logic

- Boolean: `true`, `false`, `unknown`



A	*B*	*A* OR *B*	*A* AND *B*	NOT *A*
True	True	True	True	False
True	Unknown	True	Unknown	False
True	False	True	False	False
Unknown	True	True	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
False	True	True	False	True
False	Unknown	Unknown	False	True
False	False	False	False	True

- Implementing in your own classes.

Summary

Summary

- Tell, do not ask
- Let the receiver decide
- Message sends as potential dynamic conditional
- Class hiearchy builds a skeleton for dynamic dispatch
- Avoid conditional