Booleans, Conditionals and Loops

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.iam.unibe.ch/~ducasse/

Outline

- Booleans
- Basic Loops
- Overview of the Collection hierarchy— more than 80 classes: (Bag, Array, OrderedCollection, SortedCollection, Set, Dictionary...)
- Loops and Iteration abstractions
- Common object behavior

Boolean Messages

- Logical Comparisons: &, |, xor:, not
- ***aBooleanExpr*** comparison ***aBooleanExpr***
  - (1 isZero) & false
  - Date today isRaining not
- Uniform, but optimized and inlined (macro expansion at compile time)

Boolean Lazy Logical Operators

- ***Lazy*** Logical operators
  aBooleanExpr ***and:*** andBlock

  andBlock will ***only*** be evaluated if aBooleanExpression is true

- aBooleanExpression ***or:*** orBlock
  orBlock will ***only*** be evaluated if aBooleanExpression is false

  false and: [1 error: 'crazy']
  PrIt -> false and not an error

Conditional: messages to booleans

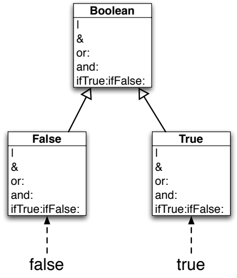
- aBoolean **ifTrue:** aTrueBlock **ifFalse:** aFalseBlock
- aBoolean **ifFalse:** aFalseBlock **ifTrue:** aTrueBlock
- aBoolean **ifTrue:** aTrueBlock
- aBoolean **ifFalse:** aFalseBlock

(thePacket isAddressedTo: self)
    ifTrue: [self print: thePacket]
    ifFalse: [super accept: thePacket]

- Hint: Take care — true is the boolean value and True is the class of true, its unique instance!

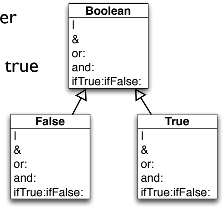
Boolean Objects

- false and true are objects described by classes Boolean, True and False



Boolean Hierarchy

- How to implement in OO true and false without conditional?
- Late binding: Let the receiver decide!
- Same message on false and true produces different results



Not

false not -> true
true not -> false

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

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

| (Or)

- **true** | true -> **true**
- **true** | false -> **true**
- **true** | anything -> **true**

- false | **true** -> **true**
- false | **false** -> **false**
- false | **anything** -> **anything**

Boolean>> | aBoolean

```
Boolean>> | aBoolean
  "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
  "Evaluating disjunction (OR) -- answer with the
  argument, aBoolean."

  ^ aBoolean
```

True>> | aBoolean

true | true -> **true**
true | false -> **true**
true | anything -> **true**

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

  ^ self
```

Boolean>>xor:

```
Boolean>>xor: aBoolean
  "Exclusive OR.  Answer true if the receiver is not
  equivalent to aBoolean."
  ^(self == aBoolean) not
```

Just implementing not on False and True defines xor: !

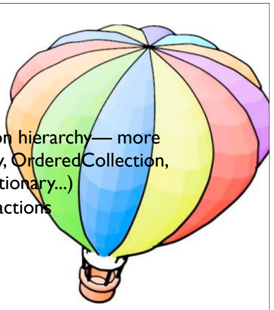
Block Use in Conditional?

- Why do conditional expressions use blocks?
- Because, when a message is sent, the receiver and the arguments of the message are *always* evaluated. Blocks are necessary to avoid evaluating both branches.

To do

- Please open your browser and analyze it
- Have a look at the ifTrue:ifFalse: implementation

Roadmap



- Booleans
- ***Basic Loops***
- Overview of the Collection hierarchy— more than 80 classes: (Bag, Array, OrderedCollection, SortedCollection, Set, Dictionary...)
- Loops and Iteration abstractions
- Common object behavior

Some Basic Loops

- aBlockTest whileTrue
- aBlockTest whileFalse
- aBlockTest whileTrue: aBlockBody
- aBlockTest whileFalse: aBlockBody
- anInteger timesRepeat: aBlockBody
- [x<y] whileTrue: [x := x + 3]
- 10 timesRepeat: [Transcript show: 'hello'; cr]

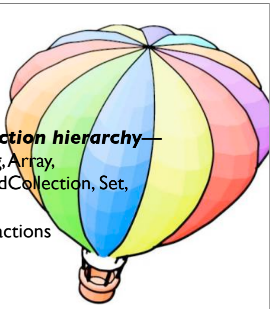
For the Curious...

```
BlockClosure>>whileTrue: aBlock
  ^ self value
        ifTrue:[ aBlock value.
              self whileTrue: aBlock ]

BlockClosure>>whileTrue
  ^ [ self value ] whileTrue:[]
```

For the Curious…

```
Integer>>timesRepeat: aBlock
    "Evaluate the argument, aBlock, the number of
    times represented by the receiver."

   | count |
   count := 1.
   [count <= self] whileTrue:
         [aBlock value.
       count := count + 1]
```

Roadmap



- Booleans
- Basic Loops
- ***Overview of the Collection hierarchy***— more than 80 classes: (Bag, Array, OrderedCollection, SortedCollection, Set, Dictionary...)
- Loops and Iteration abstractions
- Common object behavior

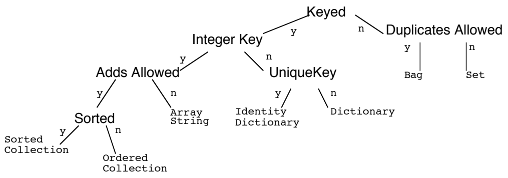
Collections

- Some criteria to identify them
  - Access: indexed, sequential or key-based.
  - Size: fixed or dynamic.
  - Element type: any or well-defined type.
  - Order: defined, defineable or none.
  - Duplicates: possible or not

Essential Collection

Sequenceable	ordered
ArrayedCollection	fixed size + key = integer
Array	any kind of elements
CharacterArray	elements = character
String	
IntegerArray	
Interval	arithmetique progression
LinkedList	dynamic chaining of the element
OrderedCollection	size dynamic + arrival order
SortedCollection	explicit order
Bag	possible duplicate + no order
Set	no duplicate + no order
IdentitySet	identification based on identity
Dictionary	element = associations + key based
IdentityDictionary	key based on identity

Essential Collections



Some Collection Methods

- Are defined, redefined, optimized or forbidden in the subclasses
- Accessing: size, capacity, at: anInteger, at: anInteger put: anElement
- Testing: isEmpty, includes: anElement, contains: aBlock, occurencesOf: anElement
- Adding: add: anElement, addAll: aCollection
- Removing: remove: anElement, remove:anElement ifAbsent: aBlock, removeAll: aCollection
- Enumerating (See generic enumerating): do: aBlock, collect: aBlock, select: aBlock, reject: aBlock, detect:, detect: aBlock ifNone: aNoneBlock, inject: avalue into: aBinaryBlock
- Converting: asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: aBlock
- Creation: with: anElement, with:with:, with:with:with:, with:with:with:with:, with:All: aCollection

Array

```
|arr|
arr := #(calvin hates suzie).
arr at: 2 put: #loves.
arr
PrIt-> #(#calvin #loves #suzie)
```

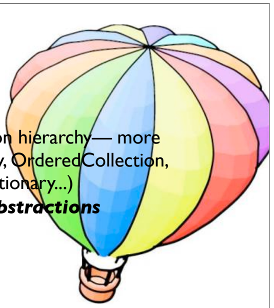
- Accessing: first, last, atAllPut: anElement, atAll: anIndexCollection: put: anElement
- Searching (*: + ifAbsent:): indexOf: anElement, indexOf: anElement ifAbsent: aBlock
- Changing: replaceAll: anElement with: anotherElement
- Copying: copyFrom: first to: last, copyWith: anElement, copyWithout: anElement

Dictionary

```
|dict|
dict := Dictionary new.
dict at: 'toto' put: 3.
dict at: 'titi' ifAbsent: [4]. -> 4
dict at: 'titi' put: 5.
dict removeKey: 'toto'.
dict keys -> Set ('titi')
```

- Accessing: at: aKey, at: aKey ifAbsent: aBlock, at: aKey ifAbsentPut: aBlock, at: aKey put: aValue, keys, values, associations
- Removing: removeKey: aKey, removeKey: aKey ifAbsent: aBlock
- Testing: includeKey: aKey
- Enumerating: keysAndValuesDo: aBlock, associationsDo: aBlock, keysDo: aBlock

Roadmap



- Booleans
- Basic Loops
- Overview of the Collection hierarchy— more than 80 classes: (Bag, Array, OrderedCollection, SortedCollection, Set, Dictionary...)
- ***Loops and Iteration abstractions***
- Common object behavior

Choose your Camp!

To get all the absolute values of numbers you could write:

```
|result|
aCol :=  ( 2 -3 4 -35 4 -11).
result := aCol species new: aCol size.
1 to: aCollection size do:
      [ :each | result
                        at: each put: (aCol at: each) abs].
result
```

Choose your Camp (II)

- You could also write:

```
( 2 -3 4 -35 4 -11) collect: [ :each | each abs ]
```

- Really important: Contrary to the first solution, the second solution works well for indexable collections and also for sets.

Iteration Abstraction: do:/collect:

aCollection do: aOneParameterBlock
aCollection collect: aOneParameterBlock
aCollection with: anotherCollection do: aBinaryBlock

```
(15 10 19 68) do:
      [:i | Transcript show: i printString ; cr ]

(15 10 19 68) collect: [:i | i odd ]
      PrIt->  (true false true false)

(1 2 3) with:  (10 20 30)
      do: [:x :y| Transcript show: (y ** x) printString ; cr ]
```

Opening the Box

Iterators are messages sent to collection objects
Collection is responsible of its traversal

```
SequenceableCollection>>do: aBlock
   "Evaluate aBlock with each of the receiver's elements
   as the argument."

   1 to: self size do: [:i | aBlock value: (self at: i)]
```

select:/reject:/detect:

aCollection select: aPredicateBlock
aCollection reject: aPredicateBlock
aCollection detect: aOneParameterPredicateBlock
aCollection
detect: aOneParameterPredicateBlock
ifNone: aNoneBlock

```
(15 10 19 68) select: [:i|i odd] ->  (15 19)
(15 10 19 68) reject: [:i|i odd] ->   (10 68)
(12 10 19 68 21) detect: [:i|i odd] PrIt-> 19
(12 10 12 68) detect: [:i|i odd] ifNone:[1] PrIt-> 1
```

inject:into:

aCollection inject: aStartValue into: aBinaryBlock

```
   | acc |
   acc := 0.
   (1 2 3 4 5) do: [:element | acc := acc + element].
   acc
   -> 15
Is equivalent to
   (1 2 3 4 5)
      inject: 0
      into: [:acc :element| acc + element]
   -> 15
```

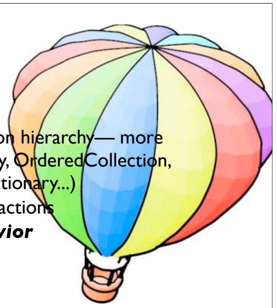
Do not use it if the resulting code is not crystal clear!

Other Collection Methods

aCollection includes: anElement
aCollection size
aCollection isEmpty
aCollection contains: aBooleanBlock

```
(1 2 3 4 5) includes: 4 -> true
(1 2 3 4 5) size -> 5
(1 2 3 4 5) isEmpty -> false
(1 2 3 4 5) contains: [:each | each isOdd] -> true
```

Roadmap



- Booleans
- Basic Loops
- Overview of the Collection hierarchy— more than 80 classes: (Bag, Array, OrderedCollection, SortedCollection, Set, Dictionary...)
- Loops and Iteration abstractions
- ***Common object behavior***

Common Shared Behavior

- Object is the root of the inheritance tree
- Defines the common and minimal behavior for all the objects in the system.

- Comparison of objects: ==, ~~, =, =~, isNil, notNil

Identity vs. Equality

- = anObject returns true if the structures are equivalent (the same hash number)
- (Array with: 1 with: 2) = (Array with:1 with:2) PrIt-> true

- == anObject returns true if the receiver and the argument point to the same object. == should never be overridden.

```
Object>>= anObject
   ^ self == anObject

~= is: not =
~~  is: not ==

(Array with: 1 with: 2 ) == (Array with: 1 with:2) PrIt-> false
(Array with: 1 with: 2 ) = (Array with: 1 with:2) PrIt-> true
```

- Take care when redefining = . One should override hash too!

Identity vs. Equality

- Remember pizza story?

- Identity
  - I want to eat your pizza
  - I eat it!

- Equality
  - I want to eat your pizza
  - I get a new that is like your pizza

Common Behavior: Printing

- Print and store objects: printString, printOn: aStream. printString calls printOn: aStream

```
(123 1 2 3) printString
        -> ' (123 1 2 3)'
Date today printString
        -> 'October 5, 1997'
```

Storing

- storeString, storeOn: aStream.
- storeString calls storeOn: aStream

```
Date today storeString
 -> '(Date readFromString: ''10/5/1997'')'
OrderedCollection new add: 4 ; add: 3 ; storeString
-> '((OrderedCollection new) add: 4; add: 3; yourself)'
```

- You need the compiler, so for a deployment image this is not convenient

Recreating objects from strings

- Create instances from stored objects: class methods readFrom: aStream, readFromString: aString

- Object readFromString: '((OrderedCollection new) add: 4; yourself)'
- -> OrderedCollection (4)

Notifying the Programmer

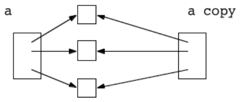
- error: aString,
- doesNotUnderstand: aMessage,
- halt, halt: aString,
  - To invoke the debugger
  - Input defaultState ifTrue:[self halt]

- shouldNotImplement
  - Sign of bad design: subclassing

- subclassResponsibility
  - Abstract method

Copying

- Copying of objects: shallowCopy, copy
- shallowCopy : the copy shares instance variables with the receiver.
- default implementation of copy is shallowCopy



Copying

```
Object>>copy
   ^ self shallowCopy postCopy

Object>>postCopy
   ^ self
```

- postCopy is a hook method
- copy is a template method

About responsibility

- No super magic global copy mechanism, just an object-oriented one
- The original object passes the control to its copy
- The copied object is in charge of its copy
- It decides which part should be copied and how

Summary

Booleans are objects too

Late binding

- gives responsibility to the receiver to decide how to treat a message
- the same message on different receiver produces different results
- performs a dispatch, but the programmer does not do it, the execution does it!

Collections are objects too

Collections provides traversal abstractions

Objects share a common behavior