Basic Objects, Conditionals and Loops

- 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

Booleans

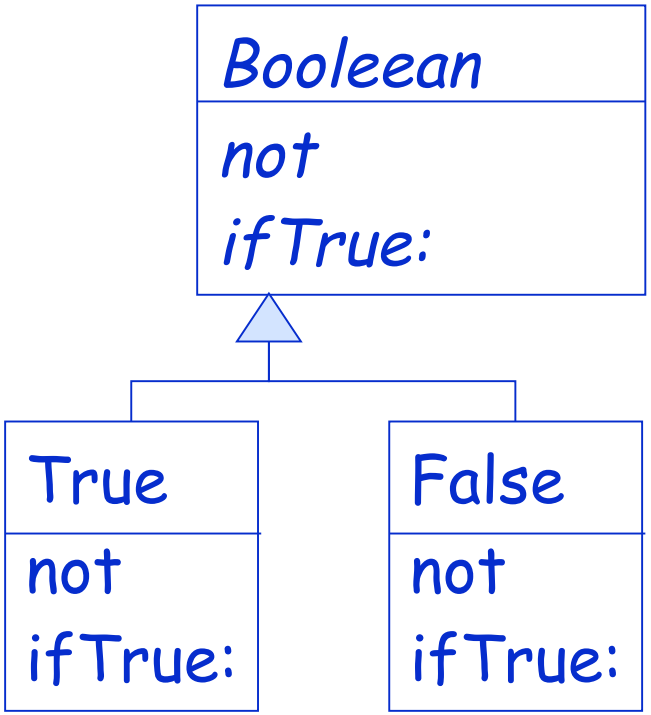
Boolean Objects

- false and true are objects described by classes Boolean, True and False
- Uniform, but optimized and inlined (macro expansion at compile time)
- Logical Comparisons &, |, xor:, not

```
aBooleanExpr comparison anotherBooleanExpr
(1 isZero) & false
```

Boolean Hierarchy

- Please open your browser and analyse it
- How to implement in OO true and false without conditional?



Boolean Lazy Logical Operators

- Lazy Logical operators

```
aBooleanExpr and: andBlock
```

andBlock will only be valued if aBooleanExpression is true

```
aBooleanExpression or: orBlock
```

orBlock will only be valued if aBooleanExpression is false

```
false and: [1 error: 'crazy']
```

PrIt-> false and not an error

Conditional: Messages to Boolean

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

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

Why 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 evaluated. Blocks are necessary to avoid evaluating both branches.

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]
```

Collections

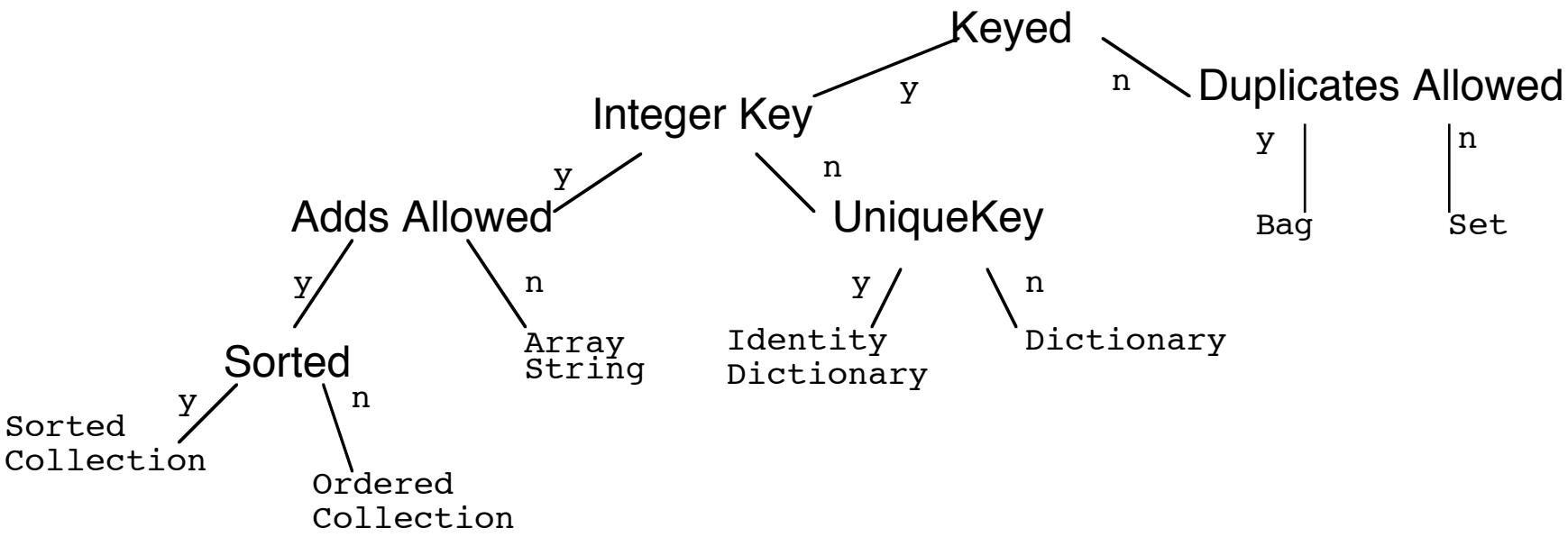
Collections

- 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: Another View



Some Collection Methods

- Will be 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

Sequenceable Specific (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

KeyedCollection Specific (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

Choose your Camp!

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

```
absolute: aCol
  |result|
  result := aCol species new: aCol size.
  1 to: aCollection size do:
        [ :each | result at: each put: (aCol at: each)
  abs].
        ^ result
```

Choose your Camp

- You could also write:

```
absolute: aCollection
  ^ aCollection 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 ]
```

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 Collections Important 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

Common Shared 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. On 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!

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

readFromString: recreating Objects

- 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,

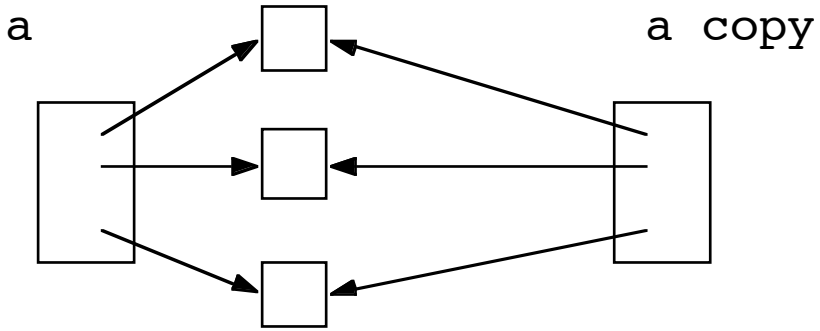
#doesNotUnderstand: aMessage,

#halt, #halt: aString,
	To invoke the debugger
	Input defaultState ifTrue:[self halt]

#shouldNotImplement
	Bad sign: 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 in VW

```
Object>>copy
        ^ self shallowCopy postCopy
Object>>postCopy
```

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