# Control flow : Blocks

**Blocks are closures similar to lambda-expressions!**

- `(block (`*formals*`)` *expressions*`)`
- Special syntax for parameterless blocks: `[`*expressions*`]`

**Blocks are objects**

- Cannot apply a block, but can send it the `value` message

```
-> (val twice (block (n) (+ n n)))
<Block>
-> (value twice 3)
6
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Control flow : Booleans

Booleans work in continuation-passing style

- message to Boolean gives action for true, false

Example: minimum

```
-> (val x 10)
-> (val y 20)
-> (ifTrue:ifFalse: (<= x y) [x] [y])
10
```

Note: `[x]` is shorthand notation for parameterless block `(block ( ) x)`

Comparison produces Boolean, which receives message and two blocks

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Protocol for Booleans

| Message | Description |
|---|---|
| `ifTrue:ifFalse: trueBlock falseBlock` | Full conditional |
| `ifTrue: trueBlock` | Partial conditional (for side effect only) |
| `ifFalse: falseBlock` | Partial conditional (for side effect only) |
| `not` | Logical negation |
| `& aBoolean` | Conjunction (and) |
| `\| aBoolean` | Disjunction (or) |
| `eqv: aBoolean` | Logical equality |
| `xor: aBoolean` | Logical inequality |
| `and: aBlock` | Short-circuit conjunction |
| `or: aBlock` | Short-circuit disjunction |

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Implementation of Booleans

**Booleans work by having *classes* `True` and `False`, each with 1 value**

**Need 2 classes so we can have 2 method definitions:**

```
(class True Boolean ( )
  (method ifTrue:ifFalse: (trueBlock falseBlock)
     (value trueBlock))
)
(class False Boolean ( )
  (method ifTrue:ifFalse: (trueBlock falseBlock)
     (value falseBlock))
)
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Implementation of Booleans (continued)

**All other methods implementable in terms of `ifTrue:ifFalse:`**

```
(class Boolean Object ( )
  (method ifTrue:ifFalse: (trueBlock falseBlock)
     (subclassResponsibility self))
  (method ifTrue: (trueBlock)
    (ifTrue:ifFalse: self trueBlock [ ]))
  (method ifFalse: (falseBlock)
    (ifTrue:ifFalse: self [ ] falseBlock))
  (method not ( )
    (ifTrue:ifFalse: self [false] [true]))
  ...
)
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Blocks: more control flow

## Can also send block a message requiring looping:

```
-> (val x 10)
-> (val y 20)
-> (whileTrue: [(<= x (* 10 y))] [(set x (* x 3))])
nil
-> x
270
```

## Protocol for blocks:

| | |
|---|---|
| `value` *arguments* | Evaluate, answer value of body |
| `whileTrue: bodyBlock` | Send `value` to the receiver, and if answer is true, send `value` to `bodyBlock` and repeat. |
| `whileFalse: bodyBlock` | Send `value` to the receiver, and if answer is false, send `value` to `bodyBlock` and repeat. |

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Class Hierarchies

Key to successful object-oriented programming is good class hierarchy

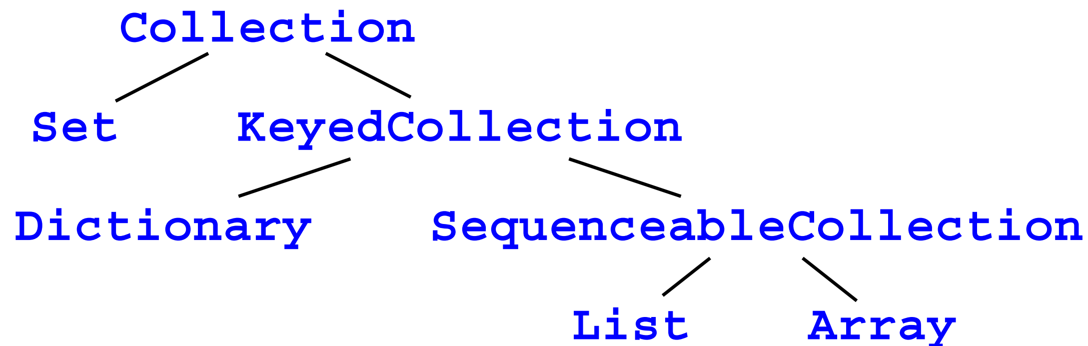
Learning real Smalltalk is 20% learning the language, and 80% learning the libraries

Famous Smalltalk browser helps navigate library

Many classes are abstract rather than concrete, and must be subclassed to be useful

© Copyright 2006 Norman Ramsey. All Rights Reserved.

# The “Collection Hierarchy”




| | |
|---|---|
| `Collection` | container for things |
| `Set` | objects are accessed in no particular order |
| `KeyedCollection` | objects are accessible by keys |
| `Dictionary` | keys can be anything |
| `SequenceableCollection` | keys must be consecutive integers |
| `List` | size of container can grow and shrink |
| `Array` | fixed size, fast access |

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Collection protocol

**Mutators:**

`add: newObject` Add argument value to collection

`addAll: aCollection` Add every element of argument

`remove: oldObject` Remove argument, error if absent

`remove:ifAbsent: oldObject exnBlock` Remove argument, evaluate `exnBlock` if absent

`removeAll: aCollection` Remove every element of argument

**Observers:**

`isEmpty` Is it empty?

`size` How many elements?

`includes: anObject` Does receiver contain argument?

`occurrencesOf: anObject` How many times is it present?

`detect: aBlock` Find and answer an element satisfying `aBlock`, essentially $\mu$Scheme's `exists?` function

`detect:ifNone: aBlock exnBlock` Detect, recover if none

`asSet` Return a set that contains the receiver's elements

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# More collection protocol

**Iterators:**

`do: aBlock` For each element `x` in collection, evaluate `(value aBlock x)`

`inject:into: thisValue binaryBlock` Essentially $\mu$Scheme's `foldl`

`select: aBlock` Essentially $\mu$Scheme's `filter`

`reject: aBlock` Filter for *not* satisfying `aBlock`, opposite of `select:`

`collect: aBlock` Essentially $\mu$Scheme's `map`

© Copyright 2006 Norman Ramsey. All Rights Reserved.

# Implementing Collections

**Subclasses need only provide `do:`, `add:`, `remove:ifAbsent:` and private method `species`**

```
(class Collection Object
  ( ) ; abstract
  (method do:  (aBlock)
     (subclassResponsibility self))
  (method add: (newObject)
     (subclassResponsibility self))
  (method remove:ifAbsent (oldObject exnBlock)
     (subclassResponsibility self))
  (method species ( ) ; will return owner class
     (subclassResponsibility self))
  ⟨other methods of class Collection⟩
)
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Implementing Collections (continued)

All other methods may be defined in terms of `do:`, `add:`, `remove:ifAbsent:` and `species`

Here are some examples:

```
⟨other methods of class Collection⟩=
(method addAll: (aCollection)
   (do: aCollection (block (x) (add: self x)))
   aCollection)
(method size ( ) (locals temp)
  (set temp 0)
  (do: self (block (_) (set temp (+ temp 1))))
  temp)
```

These methods work correctly for all subclasses

But subclasses often override (redefine) with more efficient versions

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Implementing Collections (continued)

**Use `species` to create a "collection like the receiver."**

**Example: filtering**

```
⟨other methods of class Collection⟩=
(method select: (aBlock) (locals temp)
  (set temp (new (species self)))
  (do: self (block (x)
     (ifTrue: (value aBlock x)
        [(add: temp x)])))
  temp)
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Sets (by delegation to Lists)

```
(class Set Collection
  (members) ; list of elements
  (classMethod new ( ) (initSet (new super)))
  (method initSet ( ) ; private method
     (set members (new List))
     self)
  (method do: (aBlock) (do: members aBlock))
  (method remove:ifAbsent: (item exnBlock)
    (remove:ifAbsent: members item exnBlock))
  (method add: (item)
    (ifFalse: (includes: members item)
      [(add: members item)])
    item)
  (method species ( ) Set)
  (method asSet   ( ) self) ; extra efficient
)
```

**Most subclass methods work by delegating all or part of work to list** `members`

**Note that** `Set` **is a** ***client*** **of** `List`**,** ***not*** **a subclass of** `List`**!**

©Copyright 2006 Norman Ramsey. All Rights Reserved.

# Double Dispatch

**Typical object-orientation:**

**Code you execute depends on class of the receiver (first argument)**

**What if you need to choose code based on both receiver and argument?**

**Solution: use method name to encode both operation and type of argument**

**Example: mixed arithmetic**

```
addIntegerTo:
addFloatTo:
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.