Smalltalk
Lecture 3

Control Constructs

- All control constructs in Smalltalk are implemented by message passing
  - No keywords
  - Open, extensible
  - Built up from Booleans and Blocks

Blocks

- A Block is a *closure*
  - A function that captures variable names in its lexical context
    - *I.e., a lambda abstraction*
  - First-class value
    - *Can be stored, passed, evaluated*
- Use to delay evaluation
- Syntax:

```
[ :arg1 :arg2 | |temp1 temp2| expression. expression ]
```

  - Returns last expression of the block
  - Can have any number of arguments

Block Example

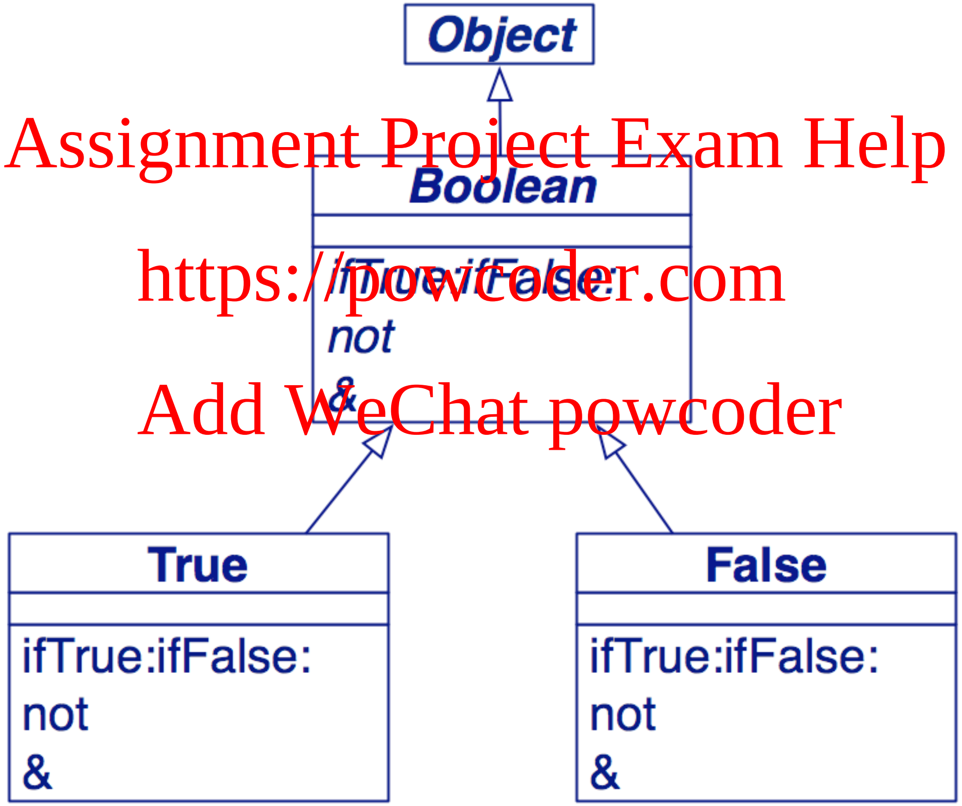
```
|sqr|
sqr := [:n | n*n ].
sqr value: 5
```

```
25
```

Block evaluation messages

```
[2 + 3 + 4 + 5] value
[:x | x + 3 + 4 + 5 ] value: 2
[:x :y | x + y + 4 + 5] value: 2 value: 3
[:x :y :z | x + y + z + 5] value: 2 value: 3 value: 4
[:x :y :z :w | x + y + z + w] value: 2 value: 3 value: 4 value: 5
```

Booleans


Object
Boolean
ifTrue:ifFalse:
not
&
True
ifTrue:ifFalse:
not
&
False
ifTrue:ifFalse:
not
&

True and False

```
True>>ifTrue: trueBlock ifFalse: falseBlock
   "Evaluate the trueBlock."

   ^ trueBlock value
```

```
False>>ifTrue: trueBlock ifFalse: falseBlock
   "Evaluate the falseBlock."

   ^ falseBlock value
```

```
x<y ifTrue: [min:=x] ifFalse: [min:=y]
```

How would you implement `not`, `&`, `|`, *…?*

true and false

- `true` and `false` are unique instances of `True` and `False`
  - Optimized and inlined

- Lazy evaluation (short-circuiting) with `and:` and `or:`

```
false and: [1/0]
```

```
false
```

```
false & (1/0)
```

`ZeroDivide error!`

Various kinds of Loops

```
|n|
n:= 10.
[n>0] whileTrue: [ Transcript display: n; cr. n:=n-1]

1 to: 10 do: [:n | Transcript display: n; cr ]

(1 to: 10) do: [:n | Transcript display: n; cr ]

10 timesRepeat: [ Transcript show: 'hi'; cr ]
```

In each case, what is the target object of the iterator message?

Why is the receiver of the whileTrue: message a block (rather than a boolean)?

whileTrue:

```
BlockContext>>whileTrue: aBlock
   "First evaluate the receiver block (self).
   If true, then evaluate the body, and recurse.
   Otherwise exit the loop."

   ^ self value
         ifTrue: [aBlock value.
                  self whileTrue: aBlock]
         ifFalse: [nil]
```

How would you implement `whileFalse:`?

Exceptions

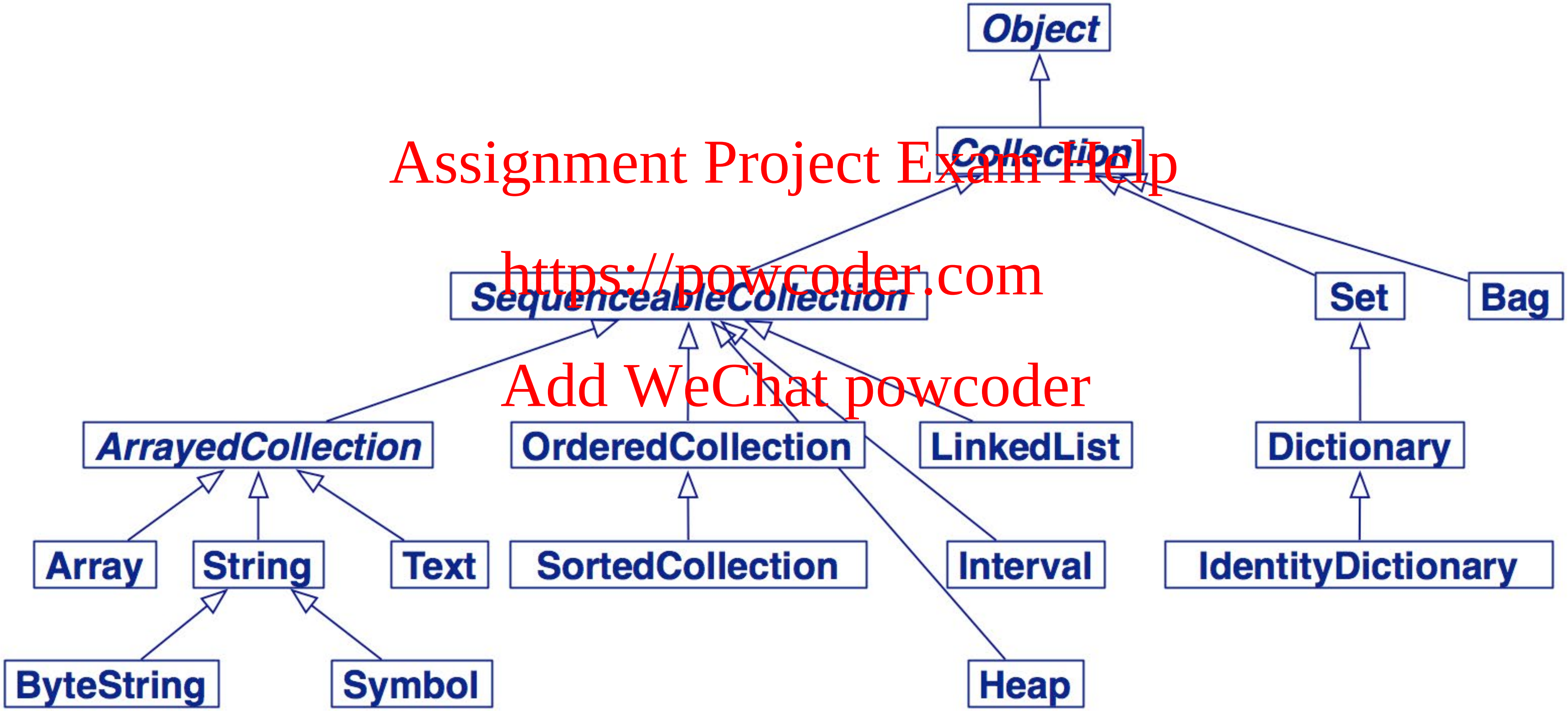
```
-1 factorial
```

```
Error!
```

```
[:n |
   [n factorial]
      on: Error
      do: [0]
] value: -1
```

```
0
```

Collections



Each implementation of Smalltalk might provide a slightly different predefined class hierarchy

Collections

> The Collection hierarchy offers many of the most useful classes in the Smalltalk system

— Resist the temptation to program your own collections!

— Except as useful exercises for educational purposes

> Classification criteria:

— Access: indexed, sequential or key-based.

— Size: fixed or dynamic.

— Element type: fixed or arbitrary type.

— Order: defined, definable, or none.

— Duplicates: possible or not

Kinds of Collections

`Sequenceable`	ordered
`ArrayedCollection`	fixed size + index = integer
`Array`	any kind of element
`String`	elements = character
`IntegerArray`	elements = integers
`Interval`	arithmetic 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

Some Collection Methods

> Are defined, redefined, optimized, or forbidden (!) in subclasses

Accessing	size, capacity, at: *anIndex*, at: *anIndex* put: *anElement*
Testing	isEmpty, includes: *anElement*, contains: *aBlock*, occurrencesOf: *anElement*
Adding	add: *anElement*, addAll: *aCollection*
Removing	remove: *anElement*, removeAll: *aCollection*, remove: *anElement* ifAbsent: *aBlock*
Enumerating	do: *aBlock*, collect: *aBlock*, select: *aBlock*, reject: *aBlock*, inject: *aValue* into: *aBinaryBlock*, detect: *aBlock*, detect: *aBlock* ifNone: *aNoneBlock*
Converting	asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: *aBlock*
Creation	with: *anElement*, with:with:, with:with:with:, with:with:with:with:, withAll: *aCollection*

Array example

```
|life|
life := #(#calvin #hates #suzie).
life at: 2 put: #loves.
life
```

```
(#calvin #loves #suzie)
```

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

Dictionary example

```
|dict|
dict := Dictionary new.
dict at: 'foo' put: 3.
dict at: 'bar' ifAbsent: [4].
dict at: 'bar' put: 5.
dict removeKey: 'foo'.
dict keys
```

a Set('bar')

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*

Common messages

```
#(1 2 3 4) includes: 5                         false
#(1 2 3 4) size                                4
#(1 2 3 4) isEmpty                             false
#(1 2 3 4) contains: [:some | some < 0 ]       false
#(1 2 3 4) do:
   [:each | Transcript show: each ]            1234
#(1 2 3 4) with: #(5 6 7 8)
   do: [:x : y | Transcript show: x+y; cr]     6
                                               8
                                               10
                                               12
#(1 2 3 4) select: [:each | each odd ]         #(1 3)
#(1 2 3 4) reject: [:each | each odd ]         #(2 4)
#(1 2 3 4) detect: [:each | each even ]        2
#(1 2 3 4) collect: [:each | each even ]       {false.true.false.true}
#(1 2 3 4) inject: 0
   into: [:sum :each | sum + each]             10
```

Converting

- Send `asSet, asBag, asSortedCollection` etc. to convert between kinds of collections
- Send `keys, values` to extract collections from dictionaries
- Use various factory methods to build new kinds of collections from old

```
d := Dictionary from: {1->#a. 2->#b. 3->#c}
```

```
Dictionary (
        1->#a
        2->#b
        3->#c
)
```

```
d keys
```

```
Set (1 2 3 )
```

```
d values
```

```
(#a #b #c )
```

Iteration — the hard road and the easy road

How to get absolute values of a collection of integers?

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

```
#(2 3 4 35 4 11)
```

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

```
#(2 3 4 35 4 11)
```

***NB:** The second solution also works for indexable collections and sets.*

Functional programming style

```
|factorial|
factorial :=
  [:n |
    (1 to: n)
      inject: 1 into:
      [:product :each | product * each ]].

factorial value: 10
```

3628800