The Object-Oriented Programming Paradigm

Smalltalk

Bent Thomsen

Slides mainly based on material
by Prof. O. Nierstrasz, U. Bern and
Stephane Ducasse, INRIA

SmallTalk -- Everything is an object

- One single model
- Single inheritance
- Public methods
- Protected attributes
- Classes are simply objects too
- Class is instance of another class
- One unique method lookup
  - look in the class of the receiver

Language Constructs

^	return
"…"	comment
#	symbol or array
'…'	string
[ ]	block or byte array (VisualWorks)
.	statement separator
;	message cascade
|…|	local or block variable
:=	assignment (also _ or ←)
$_	character
:	end of selector name
_e_ _r_	number exponent or radix
!	file element separator (used in change sets)
<primitive: ...>	for VM primitive calls

Syntax in a Nutshell (I)

comment:	“a comment”
character:	$c $h $a $r $a $c $t $e $r $s $# $@
string:	‘a nice string’ ‘lulu’ ‘l’’idiot’
symbol:	#mac #+
array:	#(1 2 3 (1 3) $a 4)
byte array:	#[1 2 3]
integer:	1, 2r101
real:	1.5, 6.03e-34,4, 2.4e7
float:	1/33
boolean:	true, false
point:	10@120

Note that @ is not an element of the syntax, but just a message sent to a number. This is the same for /, bitShift, ifTrue:, do: ...

4

Syntax in a Nutshell (II)

assigment: var := aValue
block: [:var ||tmp| expr...]

temporary variable:	|tmp|
block variable:	:var
unary message:	receiver selector
binary message:	receiver selector argument
keyword based:	receiver keyword1: arg1 keyword2: arg2...
cascade:	message ; selector ...
separator:	message . message
result:	^
parenthesis:	(...)

5

Messages instead of a predefined syntax

In Java, C, C++, Ada constructs like >>, if, for, etc. are hardcoded into the grammar

In Smalltalk there are just messages defined on objects

(>>) bitShift: is just a message sent to numbers

```
10 bitShift: 2
```

(if) ifTrue: is just messages sent to a boolean

```
(1> x) ifTrue:
```

(for) do:, to:do: are just messages to collections or numbers

```
#(a b c d) do: [:each | Transcript show: each ; cr]
1 to: 10 do: [:i | Transcript show: each printString; cr]
```

Minimal parsing

Language is extensible

6

Blocks

Anonymous method

Passed as method argument or stored Functions

```
ML: fun fct(x)= x*x+3, fct(2).
            fct = fn x => x*x+3
            fct 2
Scheme: (define fct (lambda (x) (* x (+ x 3))))
            (fct 2)
ST:  fct :=[:x| x * x + 3].
     fct value: 2
```

```
Integer>>factorial
        tmp:= 1.
        2 to: self do: [:i| tmp := tmp * i]

#(1 2 3) do: [:each | Transcript show: each printString ; cr]
```

How to Define a Class?

Class Definition: A message sent to another class

Object **subclass:** #Tomagoshi

instanceVariableNames: ‘tummy hunger dayCount’

classVariableNames: ''

poolDictionaries: ''

category: ‘Monster Inc’

Instance variables are instance-based protected (not visible by clients)

8

Named Instance Variables

> Instance variables:
- Begin with a *lowercase letter*
- Must be explicitly declared: a list of instance variables
- Name should be *unique* in the inheritance chain
- Default value of instance variable is `nil`
- *Private to the instance*, not the class (in contrast to Java)
- Can be accessed by *all the methods of the class and its subclasses*
- Instance variables *cannot be accessed by class methods*.
- The clients must use *accessors* to access an instance variable.

Design Hint:

- Do not directly access instance variables of a superclass from subclass methods. This way classes are not strongly linked.

Instance Creation

1, ‘abc’

Basic class creation messages are

new, new:,

basicNew, basicNew:

Monster new

Class specific message creation
(messages sent to classes)

Tomagoshi withHunger: 10

10

How to Define a Method?

Normally defined in a browser or (by directly invoking the compiler)

Methods are public

Always return self – if no return (^) is defined

```
Tomagoshi>>digest
    "Digest slowly: every two cycles, remove one from the
     tummy”

     (dayCount isDivisibleBy: 2)
          ifTrue: [ tummy := tummy -1]
```


11

Messages and their Composition

Three kinds of messages

Unary: Node new

Binary: 1 + 2, 3@4

Keywords: aTomagoshi eat: #cooky furiously: true

Message Priority

(Msg) > unary > binary > keywords

Same Level from left to right

Example:

(10@0 extent: 10@100) bottomRight

s isNil ***ifTrue:*** [self halt]

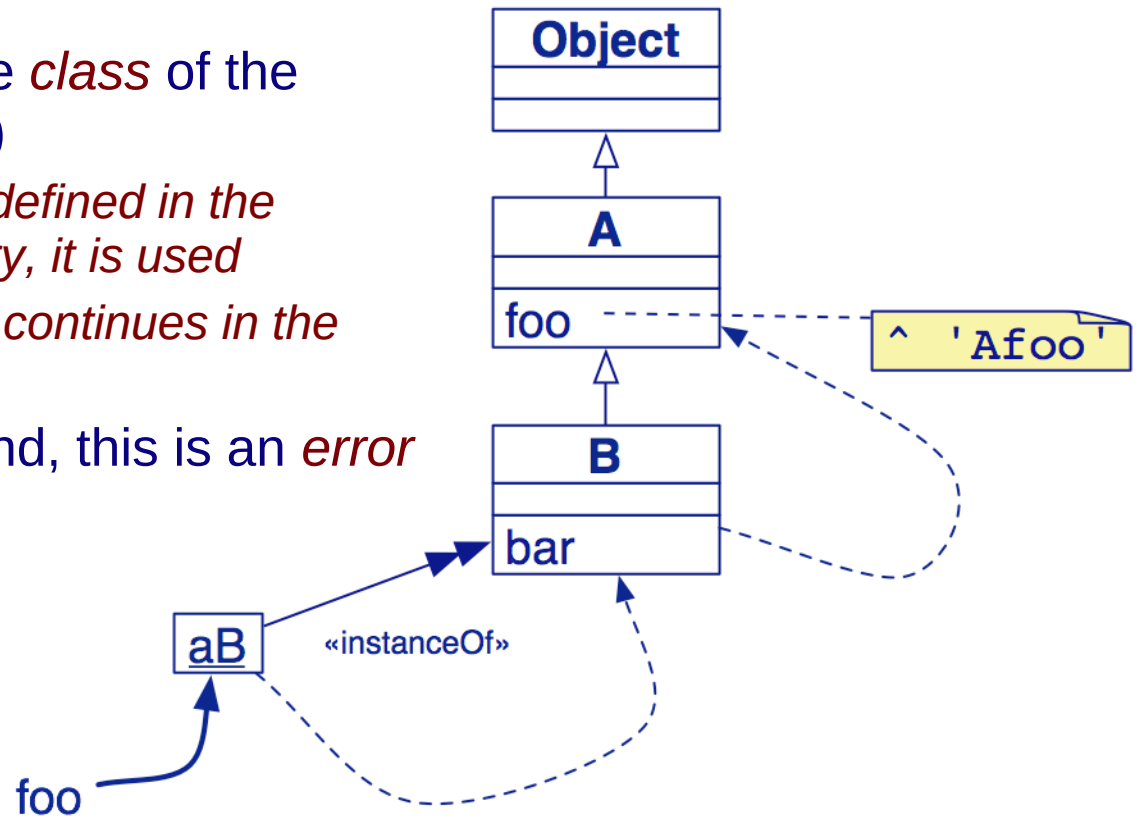
12

Normal method lookup

Two step process:

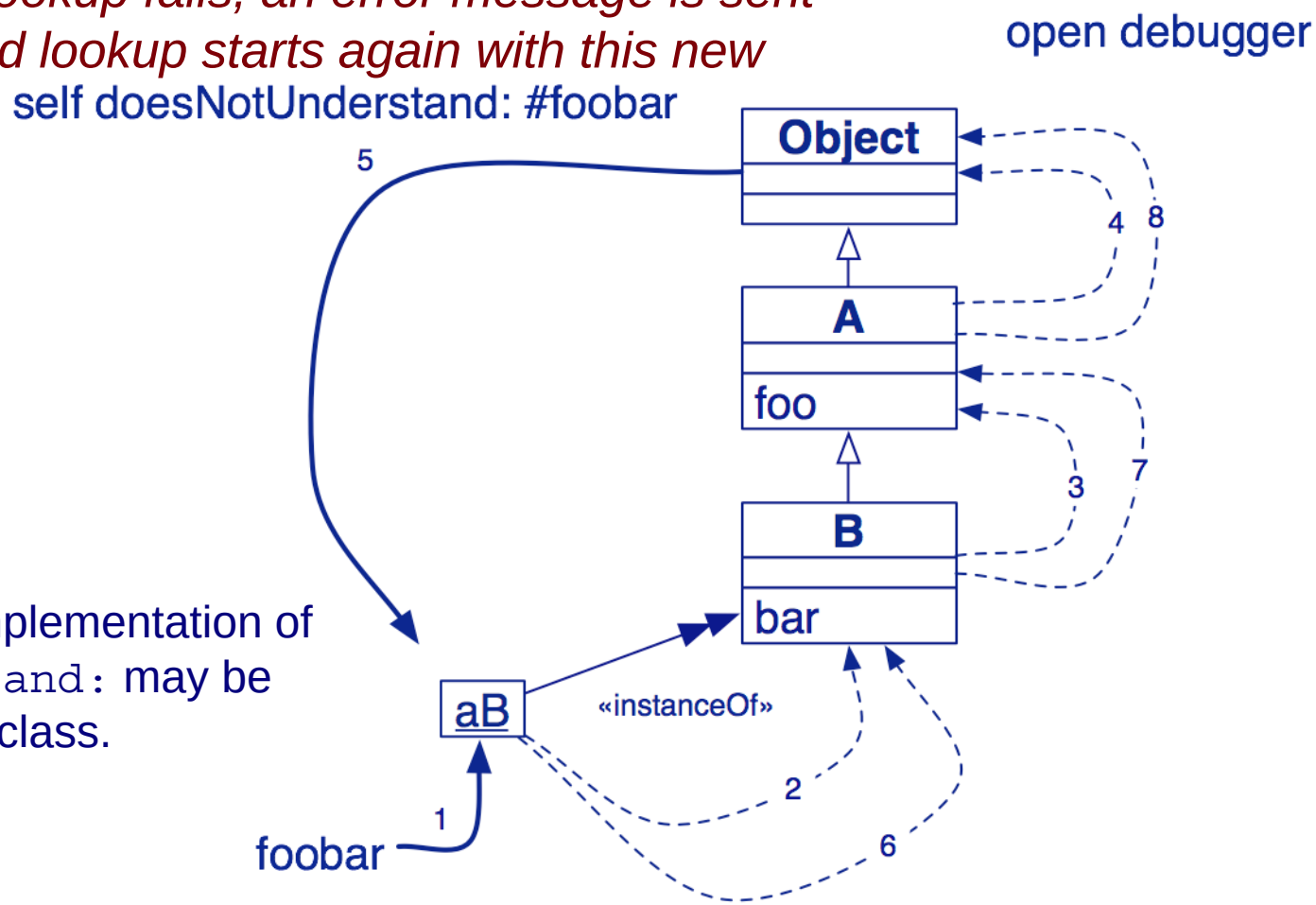
- Lookup starts in the *class* of the *receiver* (an object)
  1. *If the method is defined in the method dictionary, it is used*
  2. *Else, the search continues in the superclass*
- If no method is found, this is an *error*

…



Message not understood

When method lookup fails, an error message is sent to the object and lookup starts again with this new message.



NB: The default implementation of `doesNotUnderstand:` may be overridden by any class.

Self and Super

- Sending to self is always dynamic
- Sending to super is always static
  - Super modifies the usual method lookup to *start in the superclass of the class whose method sends to super*

Cascade

How do you format multiple messages to the same receiver?

> Use a Cascade. Separate the messages with a semicolon. Put each message on its own line and indent one tab. Only use Cascades for messages with zero or one argument.

Yourself

How can you use the value of a Cascade if the last message doesn't return the receiver of the message?

> Append the message `yourself` to the Cascade.

About yourself

- The effect of a cascade is to send all messages to the receiver of the first message in the cascade
  - `self new add: FirstSquare new; …`
- But the value of the cascade is the value returned by the last message sent

```
(OrderedCollection with: 1) add: 25; add: 35
```

```
35
```

- To get the *receiver* as a result we must send the additional message `yourself`

```
(OrderedCollection with: 1) add: 25; add: 35; yourself
```

```
an OrderedCollection(1 25 35)
```

Yourself implementation

> The implementation of `yourself` is trivial, and occurs just once in the system:

```
Object>>yourself
  ^ self
```

Other stuff you should know about

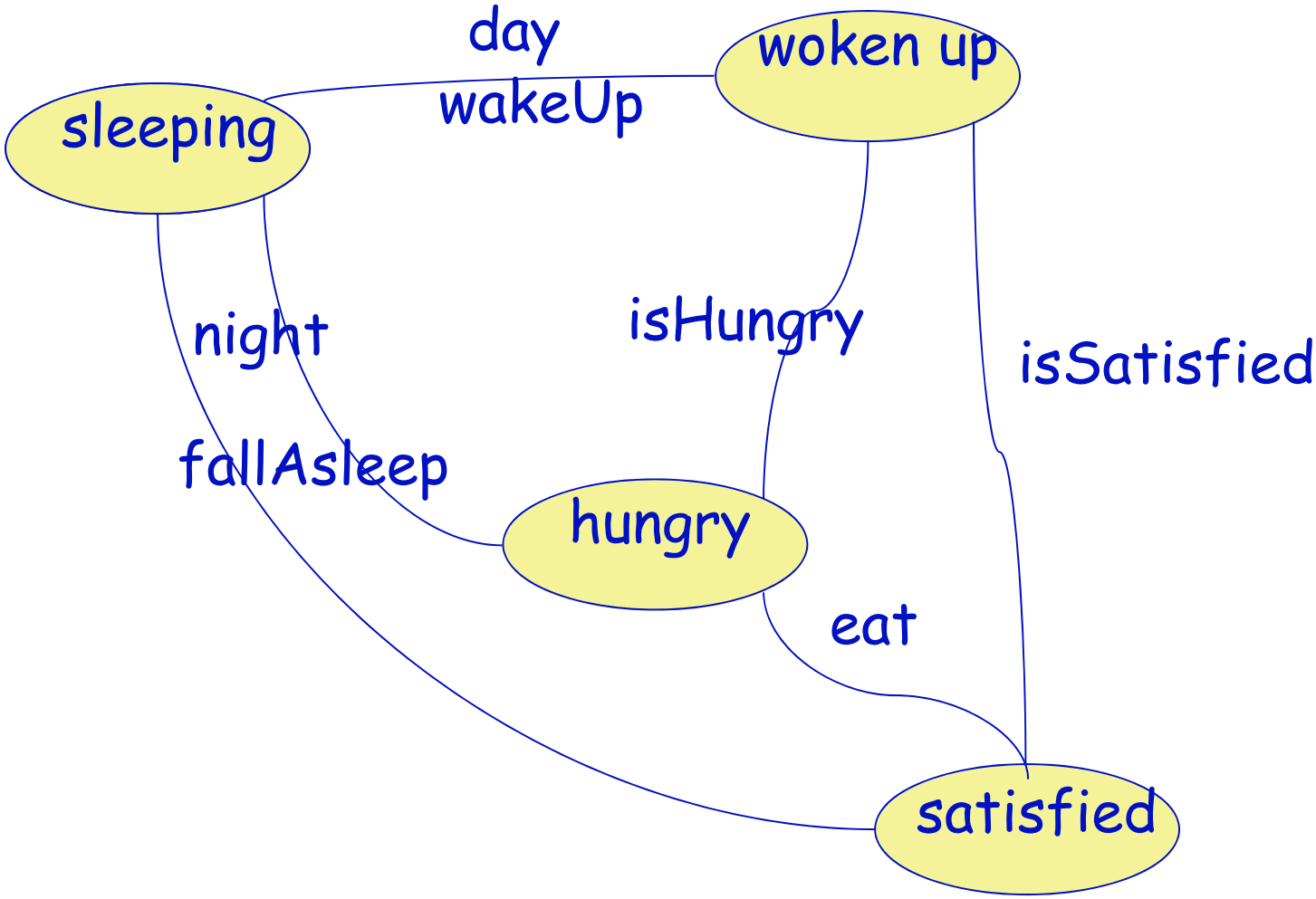
- Categories, Packages and Protocols
  - Are not Objects!
  - They are a convenience introduced by the browser to limit the amount of information that needs to be shown in each pane
  - A category is simply a collection of related classes in a Smalltalk image.
  - A package is a collection of related classes and extension methods that may be versioned using the Monticello versioning tool.
  - Categories of methods are called “protocols
- Traits
  - A trait is a collection of methods that can be included in the behaviour of a class without the need for inheritance
  - Traits may contain methods, but no instance variables

Tamagotchi

- Small entity
  - Its own night and day cycle
  - Eating, sleeping, been hungry, been satisfied
  - Changing color to indicate its mood

21

Tomagotchi



22

How to Define a Class

Fill the template:

NameOfSuperclass **subclass:** *#NameOfClass*

instanceVariableNames: '*instVarName1*'

classVariableNames: '*ClassVarName1 ClassVarName2*'

poolDictionaries: ''

category: '*category name*'

23

Tomagoshi

- For example to create the class Tomagoshi

Morph subclass: #Tomagoshi

instanceVariableNames: ‘**tummy hunger dayCount isNight state**'

classVariableNames: ''

poolDictionaries: ''

category: ’TOMA'

24

Class Comment!

I represent a tomagoshi. A small virtual animal that have its own life.

dayCount <Number> represents the number of hour (or tick) in my day and night.

isNight <Boolean> represents the fact that this is the night.

tummy <Number> represents the number of times you feed me by clicking on me.

hunger <Number> represents my appetite power.
I will be hungry if you do not feed me enough, but I'm selfish so as soon as I' satisfied I fall asleep because I do not have a lot to say.

25

How to define a method?

message selector and argument names

"comment stating purpose of message"

| temporary variable names |

statements

26

Initializing

```
Tomagoshi>>initializeToStandAlone
  "Initialize the internal state of a newly created tomagoshi"

  super initializeToStandAlone.
  tummy := 0.
  hunger := 2 atRandom + 1.
  self dayStart.
  self wakeUp
```


27

dayStart

```
Tomagoshi>>dayStart

    night := false.
    dayCount := 10
```


28

Step

```
step
  "This method is called by the system at regurlar time interval. It
defines the tomagoshi behavior."
      self timePass.
      self isHungry
                ifTrue: [self color: Color red].
      self isSatisfied
                ifTrue:
                      [self color: Color blue.
                      self fallAsleep].
      self isNight
                ifTrue:
                      [self color: Color black.
                      self fallAsleep]
```


29

Time Pass

```
Tomagoshi>>timePass
 "Manage the night and day alternance"

    Smalltalk beep.
     dayCount := dayCount -1.
     dayCount isZero
        ifTrue:[ self nightOrDayEnd.
                                dayCount := 10].
     self digest

Tomagoshi>>nightOrDayEnd
  "alternate night and day"
     night := night not
```


30

Digest

Tomagoshi>>digest

"Digest slowly: every two cycle, remove one from the tummy”

```
(dayCount isDivisibleBy: 2)
        ifTrue: [ tummy := tummy -1]
```


31

Testing

```
Tomagoshi>>isHungry
    ^ hunger > tummy

Tomagoshi>>isSatisfied
    ^self isHungry not

Tomagoshi>>isNight
    ^ isNight
```


32

State

```
Tomagoshi>>wakeUp
    self color: Color green.
    state := self wakeUpState

Tomagoshi>>wakeUpState
  "Return how we codify the fact that I sleep"
    ^ #sleep

Tomagoshi>> isSleeping
    ^ state = self wakeUpState
```


33

Eating

```
Tomagoshi>>eat
        tummy := tummy + 1
```


34

Time and Events

```
Tomagoshi>>stepTime
    "The step method is executed every steppingTime ms"
     ^ 500

Tomagoshi>>handlesMouseDown: evt
    "true means that the morph can react when the mouse down over
    it"
    ^ true

Tomagoshi>>mouseDown: evt
   self eat
```


35

Instantiating…

- To create a tomagoshi:
- Tomagoshi newStandAlone openInWorld

36

> Demo

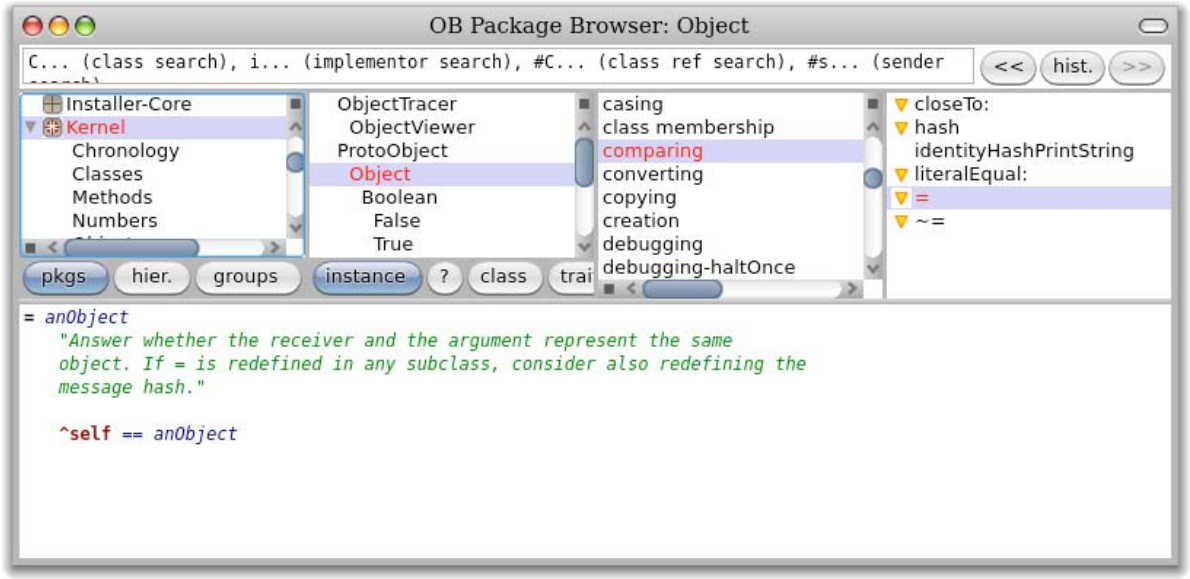
Standard Classes

Review — Objects in Smalltalk

- *Everything* is an object
  - Things only happen by message passing
  - Variables are dynamically bound
- Each object is an instance of one class
  - A class defines the structure and the behavior of its instances.
  - Single inheritance
  - A class is an instance of a metaclass
- Methods are public
  - private methods by convention in “private” protocol
- Objects have private state
  - Encapsulation boundary is the object

Object

> `Object` is the root of the inheritance tree (well, almost)
  - Defines the common and minimal behavior for all the objects in the system.
  - Comparison of objects:
    - *==, ~~, =, =~, isNil, notNil*
  - Printing
    - *`printString`, `printOn: aStream`*



Identity vs. Equality

- == tests Object identity
  - Should never be overridden
- = tests Object value
  - Should normally be overridden
    - *Default implementation is == !*
  - You should override hash too!

```
'foo','bar' = 'foobar'
'foo','bar' == 'foobar'
```

```
true
false
```

Printing

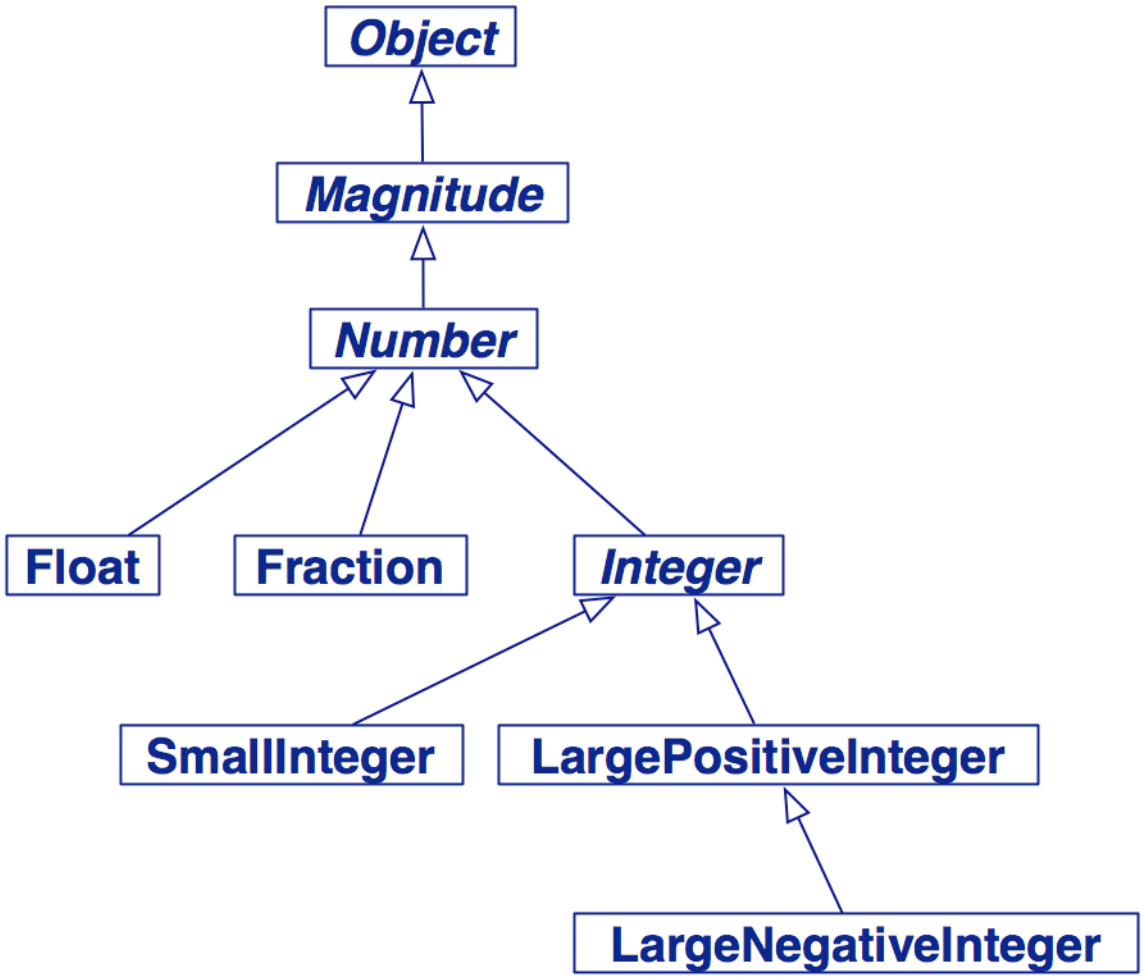
> Override `printOn:` to give your objects a sensible textual representation

```
Fraction>>printOn: aStream
  aStream nextPut: $(.
  numerator printOn: aStream.
  aStream nextPut: $/.
  denominator printOn: aStream.
  aStream nextPut: $).
```

Object methods to support the programmer

`error: aString`	Signal an error
`doesNotUnderstand: aMessage`	Handle unimplemented message
`halt, halt: aString,` `haltIf: condition`	Invoke the debugger
`subclassResponsibility`	The sending method is abstract
`shouldNotImplement`	Disable an inherited method
`deprecated: anExplanationString`	Warn that the sending method is deprecated.

Numbers


Object
Magnitude
Number
Float
Fraction
Integer
SmallInteger
LargePositiveInteger
LargeNegativeInteger

Abstract methods in Smalltalk

```
Number>>+ aNumber
   "Answer the sum of the receiver and aNumber."

   self subclassResponsibility
```

Abstract methods (part 2)

```
Object>>subclassResponsibility
   "This message sets up a framework for the behavior of the
   class' subclasses. Announce that the subclass should have
   implemented this message."

   self error: 'My subclass should have overridden ',
      thisContext sender selector printString
```

Automatic coercion

```
1 + 2.3
1 class
1 class maxVal class
(1 class maxVal + 1) class
(1/3) + (2/3)
1000 factorial / 999 factorial
2/3 + 1
```

```
3.3
SmallInteger
SmallInteger
LargePositiveInteger
1
1000
(5/3)
```

Browse the hierarchy to see how coercion works.

Try this in Java!

```
1000 factorial
```

```
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404
800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779
505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012
476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207
379143853719588249808126867838374559731746136085379534542221586593201928090878297308431392844403281
231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261
683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909
342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945
160765353408198901385442487984959953319101723355556602139450399736280750137837615307127761926849034
352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223
838971476088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779
911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153181617210
465832036786906117260158783520751516284225540265170483304226143974286933061690897968482590125458327
168226458066526769958652682272807075781391858178889652208164348344825993266043367660176999612831860
788386150279465955131156552036093988180612138558600301435694527224206344631797460594682573103790084
024432438465657245014402821885252470935190620929023136493273497565513958720559654228749774011413346
962715422845862377387538230483865688976461927383814900140767310446640259899490222221765904339901886
018566526485061799702356193897017860040811889729918311021171229845901641921068884387121855646124960
798722908519296819372388642614839657382291123125024186649353143970137428531926649875337218940694281
434118520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506
217095002597389863554277196742822248757586765752344220207573630569498825087968928162753848863396909
959826280956121450994871701244516461260379029309120889086942028510640182154399457156805941872748998
094254742173582401063677404595741785160829230135358081840096996372524230560855903700624271243416909
004153690105933983835777939410970027753472000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Characters

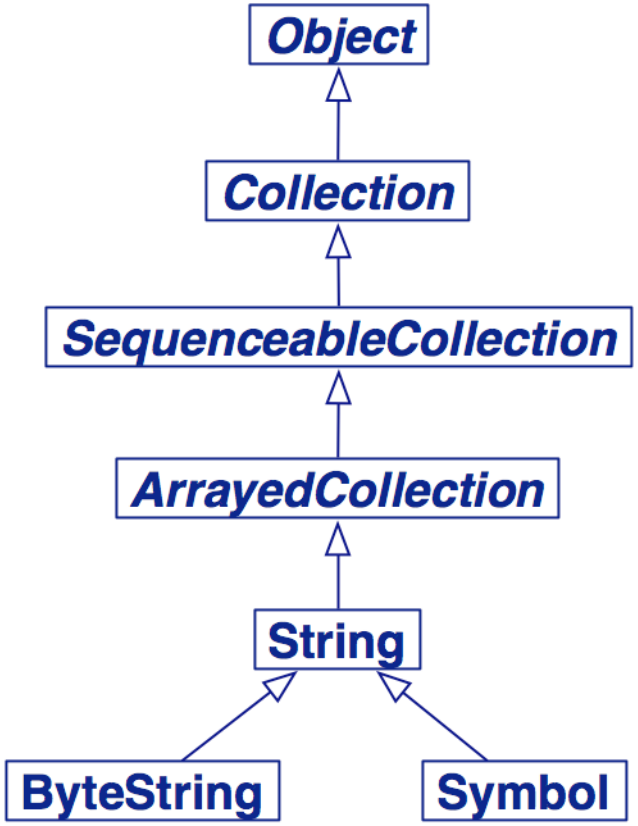
> Characters:

```
$a $B $$ $_ $1
```

> Unprintable characters:

```
Character space, Character tab, Character cr
```

Strings


Object
Collection
SequenceableCollection
ArrayedCollection
String
ByteString
Symbol

Strings

```
#mac asString
12 printString
String with: $A
'can''t' at: 4
'hello', ' ', 'world'
```

```
'mac'
'12'
'A'
$'
'hello world'
```

> To introduce a single quote inside a string, just double it.

Comments and Tips

> A comment can span several lines.
  - Avoid putting a space between the " and the first character.
  - When there is no space, the system helps you to select a commented expression. You just go after the " character and double click on it: the entire commented expression is selected. After that you can printIt or doIt, etc.

```
"TestRunner open"
```

```
"TestRunner open"
```

Literal Arrays

```
#('hello' #(1 2 3))

#(a b c)
```

```
#('hello' #(1 2 3))

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

Arrays and Literal Arrays

> Literal Arrays and Arrays only differ in creation time
  — Literal arrays are known at compile time, Arrays at run-time.

> A literal array with two symbols (not an instance of `Set`)

```
#(Set new)
```

```
#(#Set #new)
```

> An array with one element, an instance of `Set`

```
Array with: (Set new)
```

```
an Array(a Set())
```

Arrays with {} in Pharo

> { … } is a shortcut for `Array new` …

```
#(1 + 2 . 3 )

{ 1 + 2 . 3 }

Array with: 1+2 with: 3
```

```
#(1 #+ 2 #. 3)

#(3 3)

#(3 3)
```

Symbols vs. Strings

> Symbols are used as method selectors and unique keys for dictionaries
  - Symbols are read-only objects, strings are mutable
  - A symbol is unique, strings are not

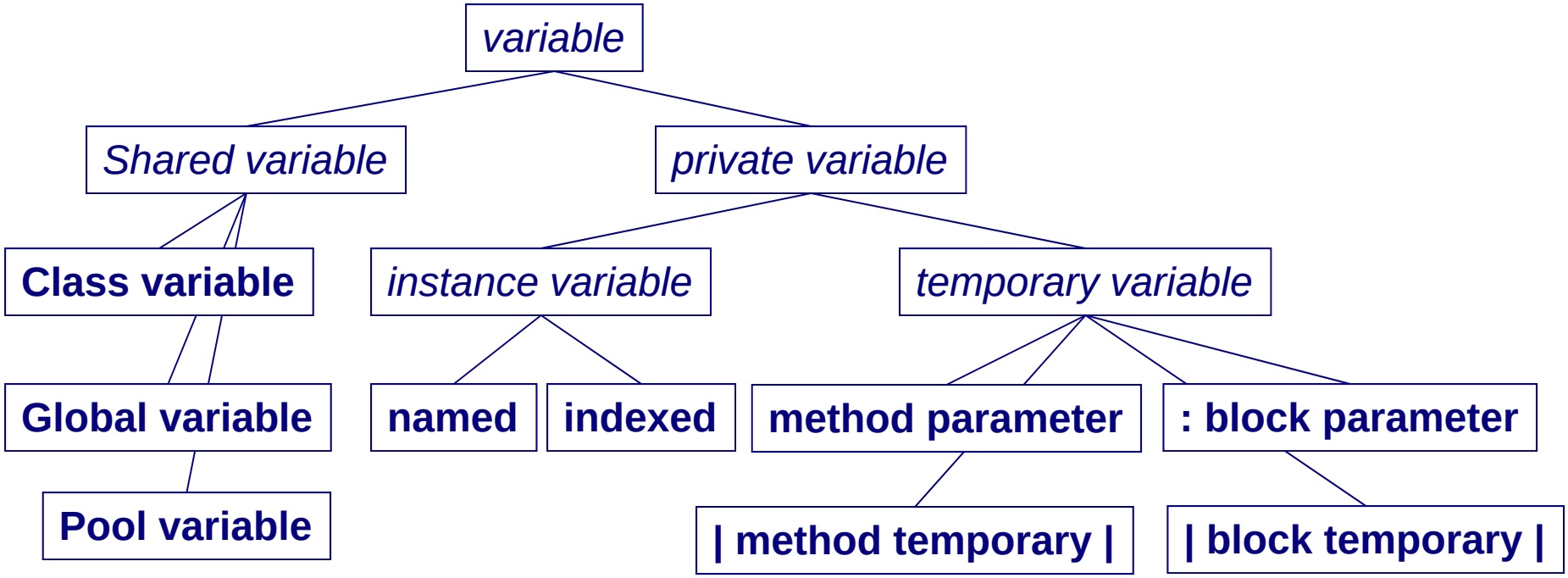
```
'calvin' = 'calvin'                    true
'calvin' == 'calvin'                   true
'cal','vin' = 'calvin'                 true
'cal','vin' == 'calvin'                false

#calvin = #calvin                      true
#calvin == #calvin                     true
#cal,#vin = #calvin                    true
#cal,#vin == #calvin                   false
#cal,#vin                              'calvin'
(#cal,#vin) asSymbol == #calvin        true
```

!

NB: Comparing strings is slower than comparing symbols by a factor of 5 to 10. However, converting a string to a symbol is more than 100 times more expensive.

Variables

> A variable maintains a reference to an object
  - Dynamically typed
  - Can reference different types of objects
  - Shared (initial uppercase) or local (initial lowercase)



Assignment

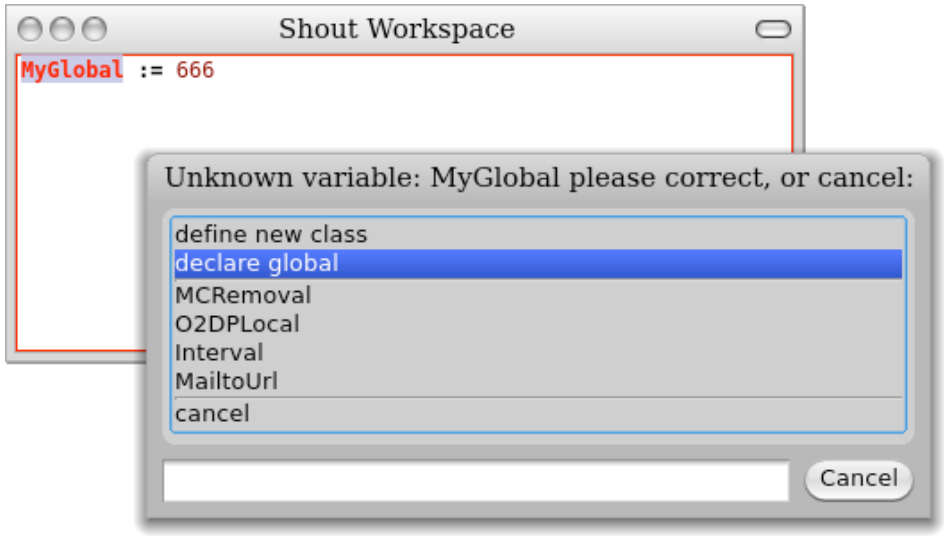
> Assignment binds a name to an object reference
- Not done by message passing!
- Method arguments cannot be assigned to!
  - *Use a temporary instead*
- Different names can point to the same object!
  - *Watch out for unintended side effects*

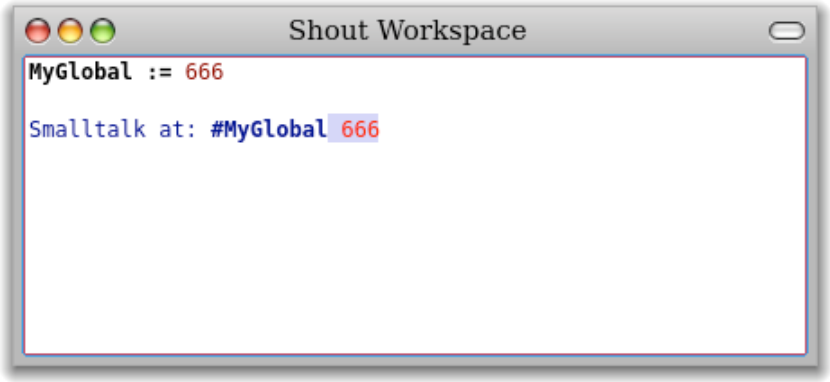
```
|p1 p2|
p1 := 3@4.
p2 := p1.
p1 setX: 5 setY: 6.
p2
```

5@6

Global Variables

> Always Capitalized (convention)
  - If unknown, Smalltalk will prompt you to create a new Global
  - Stored in the Smalltalk System Dictionary





> Avoid them!

Global Variables

> To remove a global variable:

```
Smalltalk removeKey: #MyGlobal
```

> Some predefined global variables:

`Smalltalk`	Classes & Globals
`Undeclared`	A PoolDictionary of undeclared variables accessible from the compiler
`Transcript`	System transcript
`ScheduledControllers`	Window controllers
`Processor`	A ProcessScheduler list of all processes

Instance Variables

> Private to an object
  - Visible to methods of the defining class and subclasses
  - Has the same lifetime as the object
  - Define accessors (getters and setters) to facilitate initialization
    - *Put accessors in a private category!*

Six Pseudo-Variables

> The following pseudo-variables are hard-wired into the Smalltalk compiler.

`nil`	A reference to the UndefinedObject
`true`	Singleton instance of the class True
`false`	Singleton instance of the class False
`self`	Reference to this object Method lookup starts from object's class
`super`	Reference to this object (!) Method lookup starts from the superclass
`thisContext`	Reification of execution context

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

Block Example

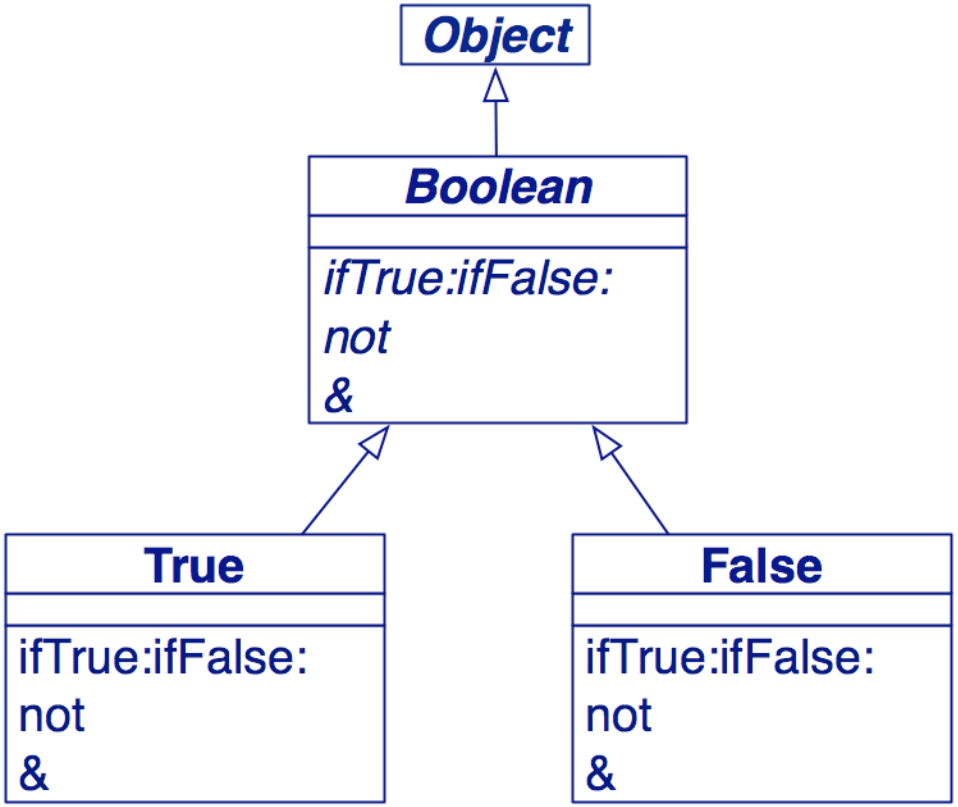
```
|sqr|
sqr := [:n | n*n ].
sqr value: 5
```

```
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

```
True>>ifTrue: trueBlock ifFalse: falseBlock
   "Answer with the value of trueBlock.
   Execution does not actually reach here
   because the expression is compiled in-line."

   ^ trueBlock value
```

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

true and false

- `true` and `false` are unique instances of `True` and `False`
  - Optimized and inlined
- Lazy evaluation 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 show: n; cr. n:=n-1]

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

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

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

In each case, what is the target object?

Exceptions

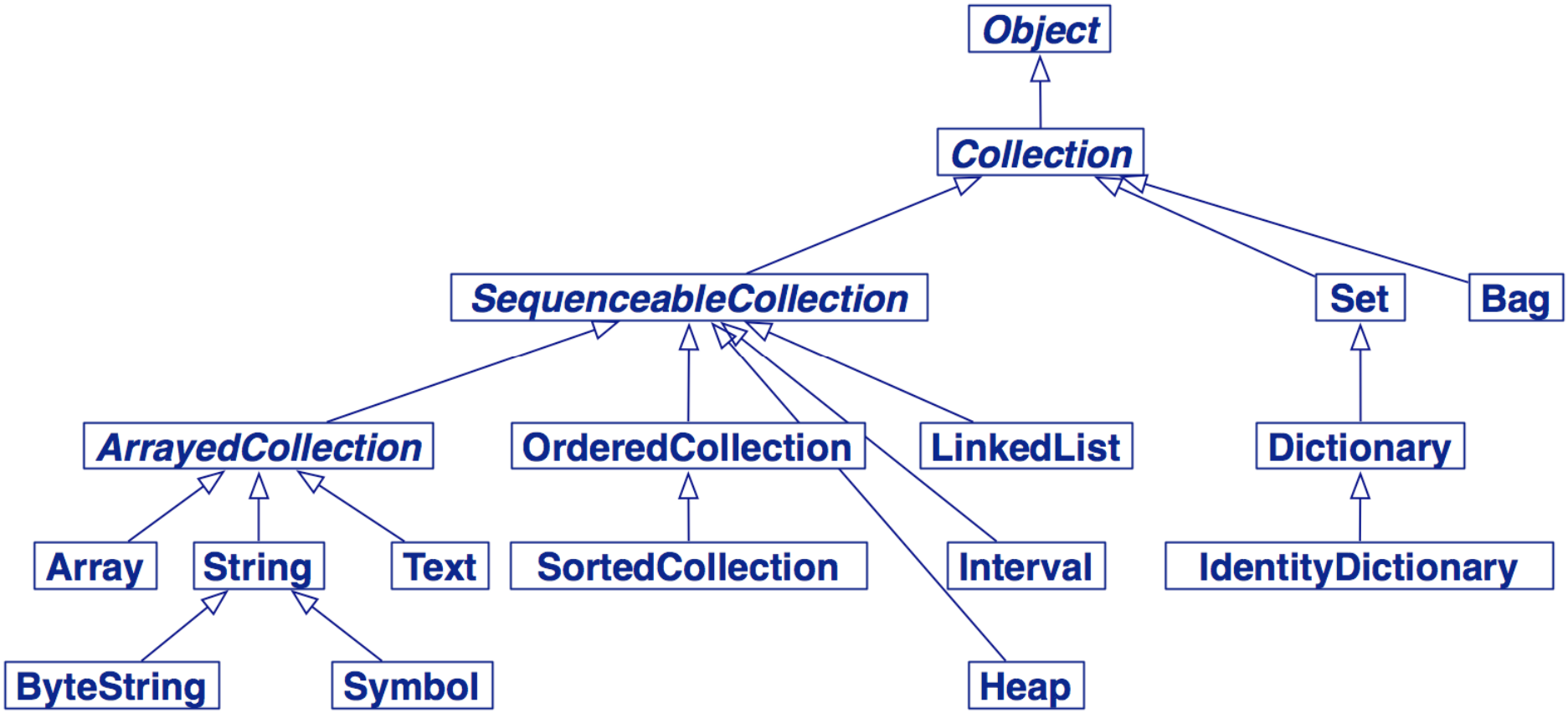
```
-1 factorial
```

```
Error!
```

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

```
0
```

Collections



Collections

- The Collection hierarchy offers many of the most useful classes in the Smalltalk system
  - Resist the temptation to program your own collections!

- Classification criteria:
  - Access: indexed, sequential or key-based.
  - Size: fixed or dynamic.
  - Element type: fixed or arbitrary type.
  - Order: defined, defineable 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, remove: anElement ifAbsent:` `aBlock, removeAll: aCollection`

Enumerating	`do: aBlock, collect: aBlock, select: aBlock, reject:`
	`aBlock, detect: aBlock, 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:, 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
```

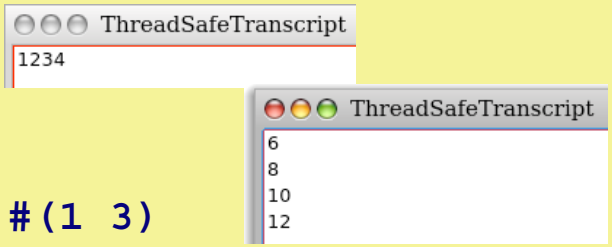
a Set('bar')

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

```
false
4
false
false
```




```
#(1 3)
#(2 4)
1
{false.true.false.true}

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

```
Dictionary newFrom: {1->#a. 2->#b. 3->#c}
```

Iteration — the hard road and the easy road

How to get absolute values of a collection of integers?

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

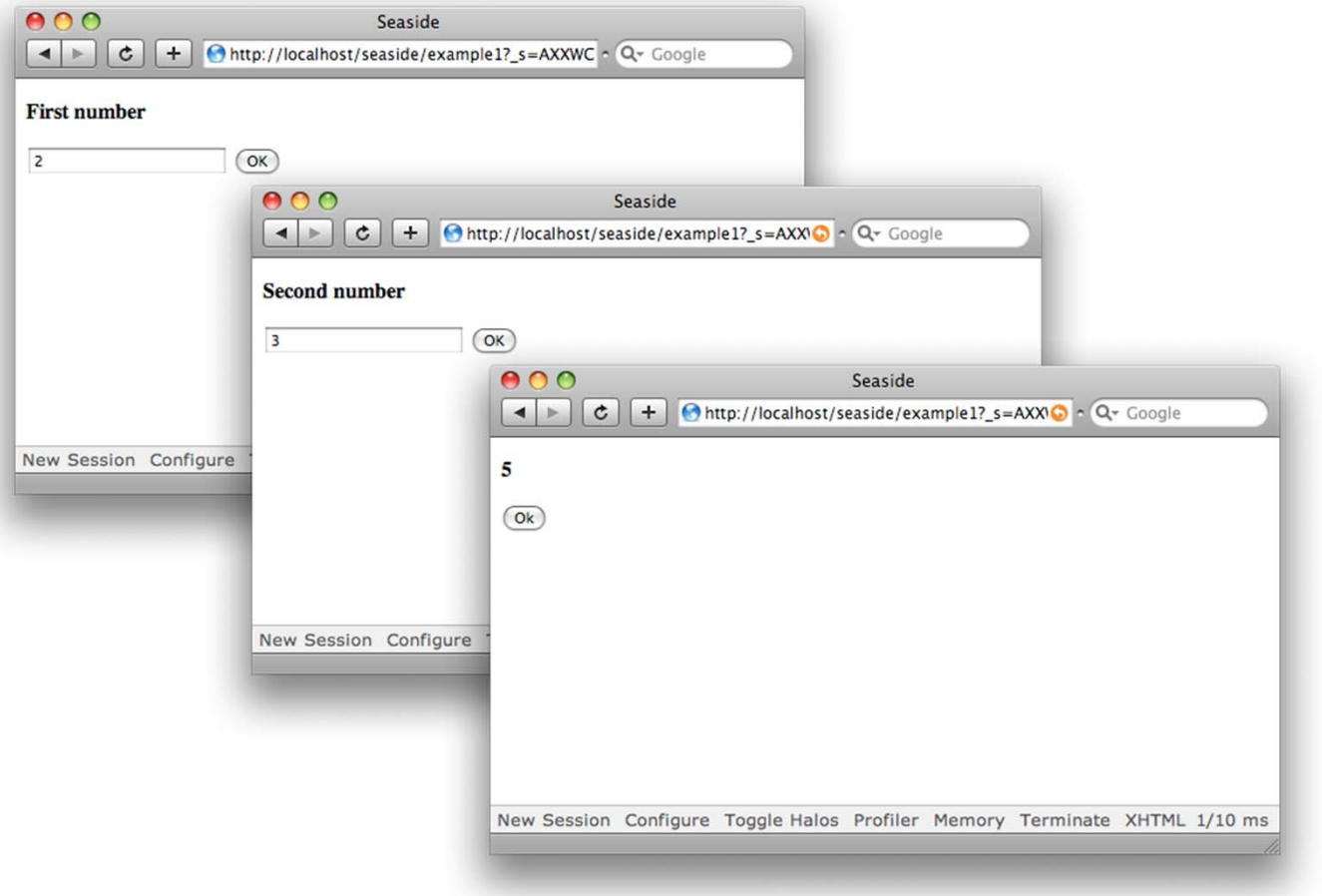
Seaside

Birds-eye view

Model your domain with objects — model domain components as objects. Compose objects, not text. Strive for fluent interfaces. Build applications by *scripting components*.

Introduction: Web Applications

Example: Adding two numbers



What is going on?

```
<form action="second.html">
   <input type="text" name="value1">
   <input type="submit" name="OK">
</form>
```

first.html

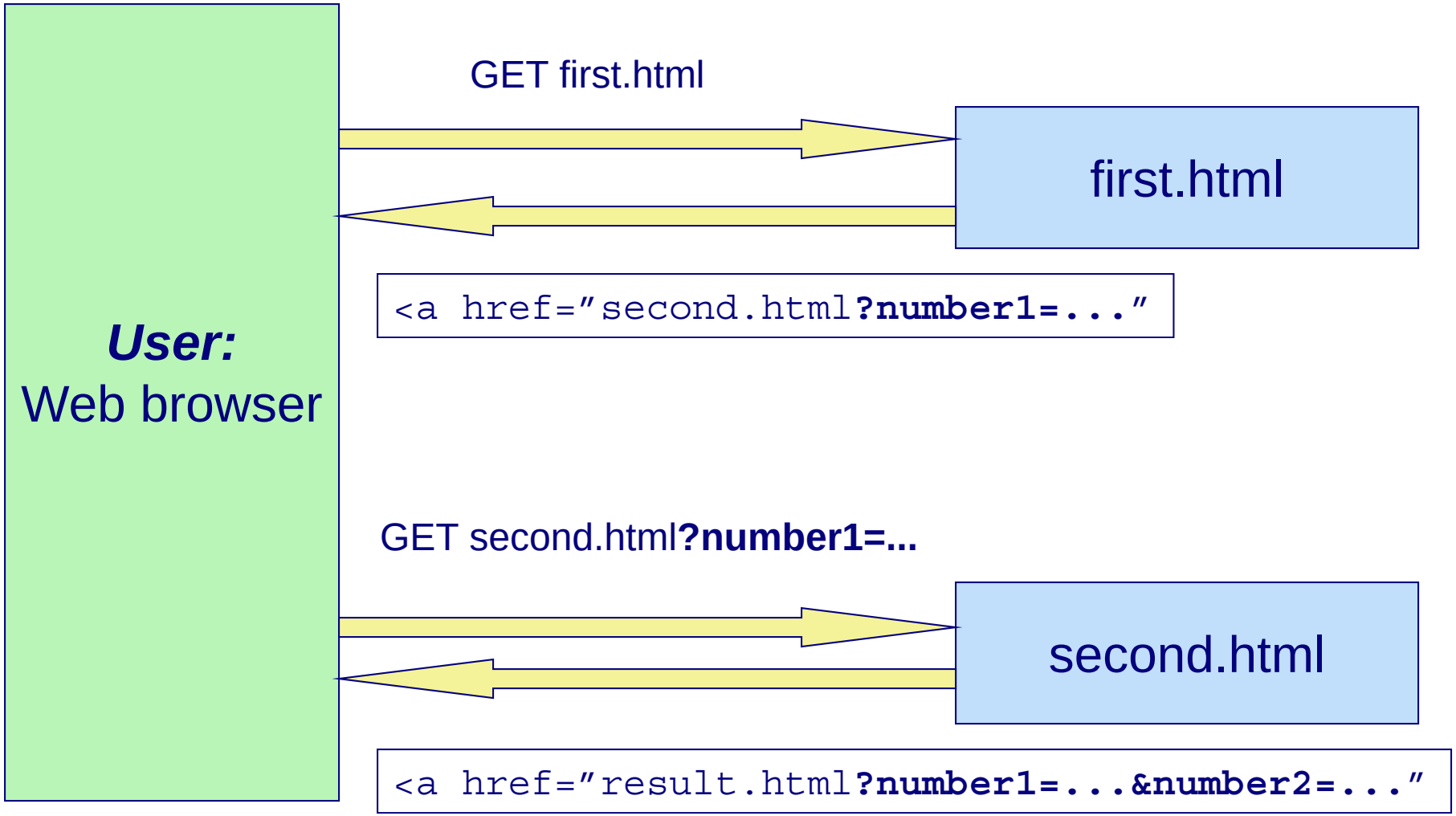
```
<form action="result.html">
   <input type="hidden"
                  name="value1" value="<% value1 %>">
</form>
```

second.html

```
<p>
   <% value1 + value2 %>
</p>
```

result.html

Control Flow:
HTTP request-response



Something is wrong…

- Control-flow quite arcane
  - Remember GOTO?
  - We do not care about HTTP!
- How to debug that?
- And what about
  - Back button?
  - Copy of URL (second browser)?

What we want

> Why not this?

```
go
   |number1 number2 |

   number1 := self request: 'First Number'.
   number2 := self request: 'Second Number'.

   self inform: 'The result is ',
                (number1 + number2) asString
```

And why not

```
> 'http://www.pharo.org' asUrl retrieveContents
```

Seaside: Features

- Sessions as continuous piece of code
- XHTML/CSS building
- Callback based event model
- Composition and reuse
- Debugging and Development tools

XHTML Building

```
html div id: 'title'; with: 'Title'
html div id: 'list'; with: [
    html span class: 'item'; with: 'Item 1'.
    html span class: 'item'; with: 'Item 2'.
]
```

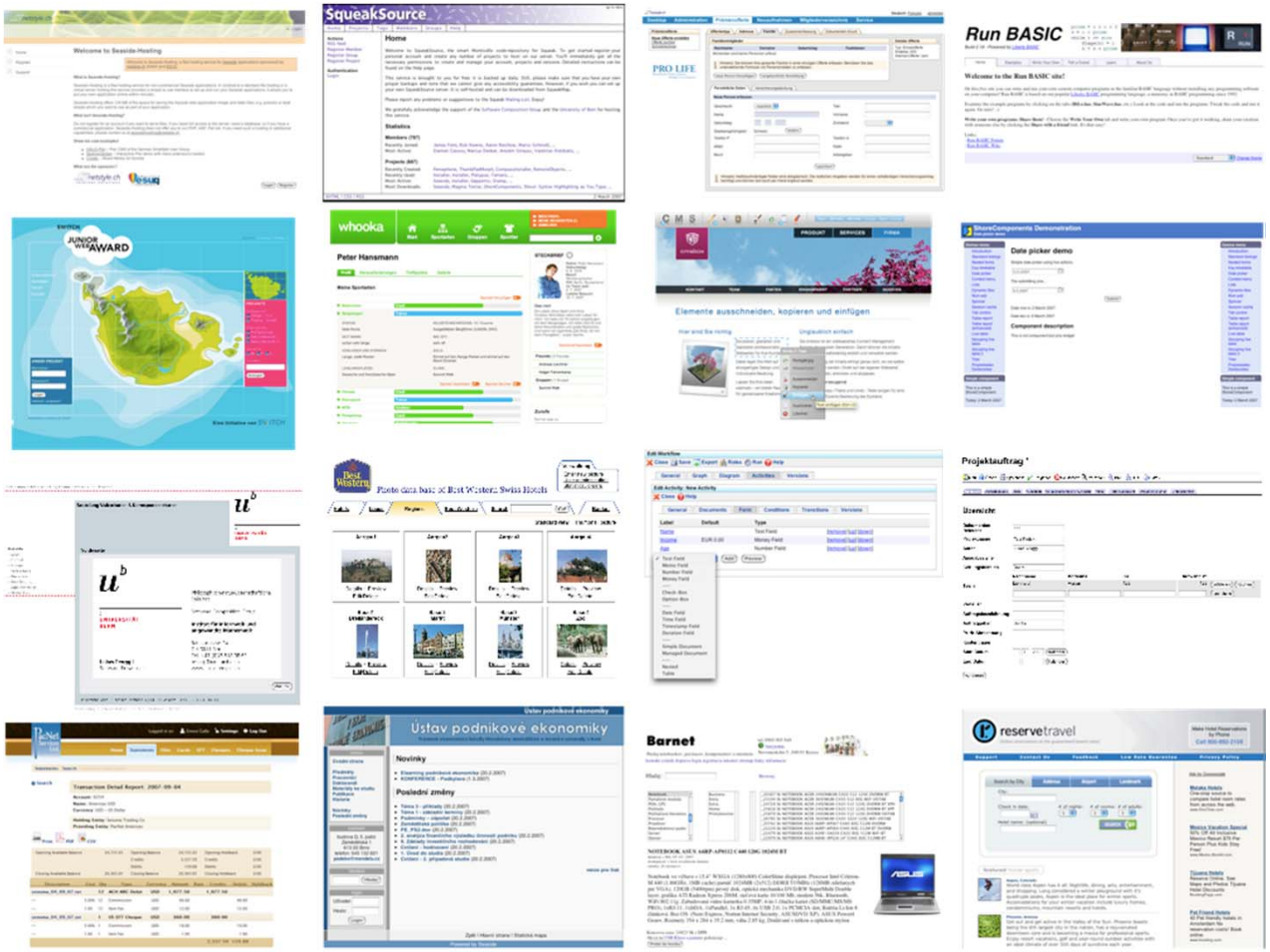
```
<div id="title">Title</div>
<div if="list">
    <span class="item">Item 1</span>
    <span class="item">Item 2</span>
</div>
```

Callback Event Model

```
Example3>>renderContentOn: html
          html form: [
          html submitButton
              callback: [ self inform: 'Hello' ];
              text: 'Say Hello' ]
```

```
....
<form action="/seaside/example2" method="post">
<input type="hidden" name="_s" value="JBbTXBnPaTLOjcjI" class="hidden"/>
<input type="hidden" name="_k" value="FFQrpnBg" class="hidden" />
<input type="submit" name="1" value="Say Hello" class="submit" />
</form>
....
```

Examples

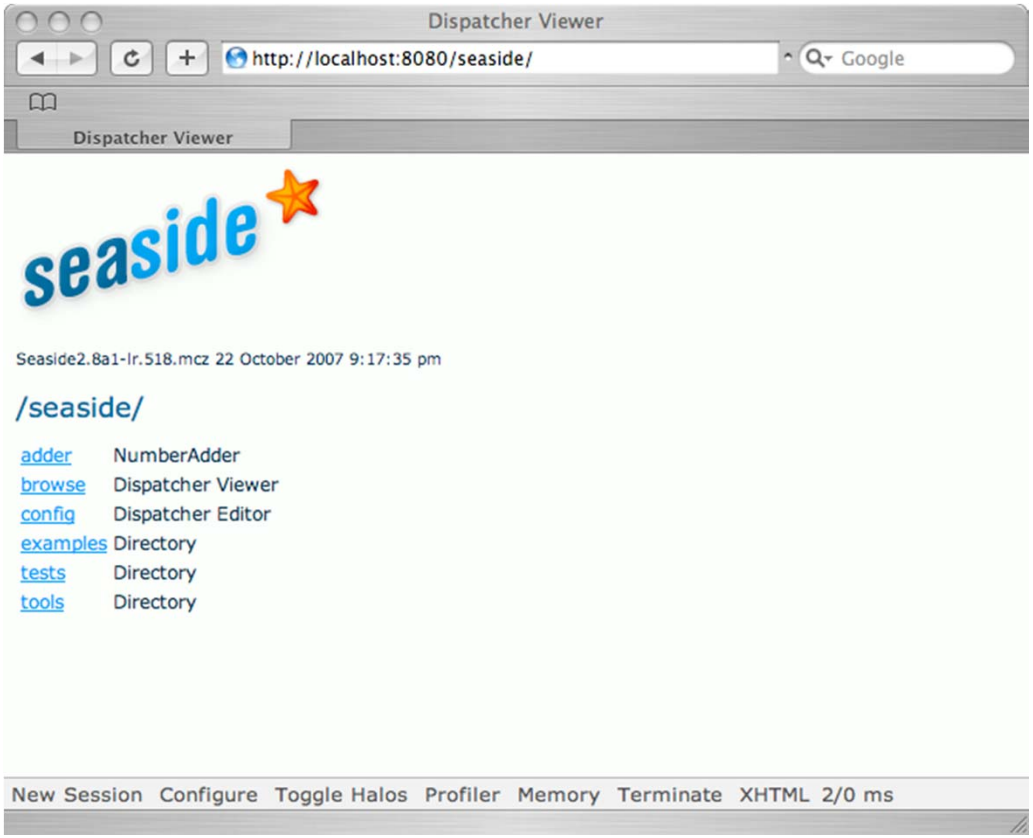

SqueakSource
Run BASIC
whooka
Peter Hansmann
Elemente ausschneiden, kopieren und einfügen
Projektauftrag
Ústav podnikové ekonomiky
Novinky
Poslední změny
Barnet
reservetravel

Installing Seaside

Download the one-click image from http://seaside.st/

Seaside

http://localhost:8080/seaside



Literature

- Dynamic Web Development with Seaside
  - http://book.seaside.st/book
- HPI Seaside Tutorial:
  - http://www.swa.hpi.uni-potsdam.de/seaside/tutorial
- Articles:
  - “Seaside — a Multiple Control Flow Web Application Framework.”
  - “Seaside: A Flexible Environment for Building Dynamic Web Applications”
  - http://scg.unibe.ch/scgbib?query=seaside-article
- …. more at http://seaside.st

What you should know!

- *How are abstract classes defined in Smalltalk?*
- *What's the difference between a* `String` *and a* `Symbol`*?*
- *Where are class names stored?*
- *What is the difference between* `self` *and* `super`*?*
- *Why do we need* `Blocks`*?*
- *How is a* `Block` *like a lambda?*
- *How would you implement* `Boolean>>and:`*?*
- *What does* `inject:into:` *do?*

Can you answer these questions?

- *How are Numbers represented internally?*
- *Is it an error to instantiate an abstract class in Smalltalk?*
- *Why isn't the assignment operator considered to be a message?*
- *What happens if you send the message* `#new` *to* `Boolean`*? To* `True` *or* `False`*?*
- *Is* `nil` *an object? If so, what is its class?*
- *Why does* `ArrayedCollection>>add:` *send itself the message* `shouldNotImplement`*?*

License

http://creativecommons.org/licenses/by-sa/3.0/