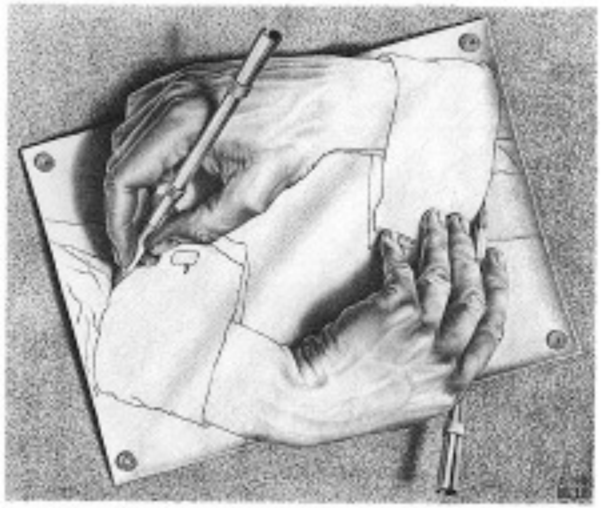
3. Standard Classes

Birds-eye view

Reify everything — by reifying its entire implementation model, Smalltalk succeeds in being open, and extensible. New features can be added without changing the syntax!



Roadmap

- > Object
- > Numbers, Characters, Strings and Arrays
- > Variables
- > Blocks and Control structures
- > Collections

Selected material courtesy Stéphane Ducasse

Roadmap

> **Object**
> Numbers, Characters, Strings and Arrays
> Variables
> Blocks and Control structures
> Collections

Review — Objects in Smalltalk

> *Everything* is an object
- — Things only happen by message passing
- — Variables are dynamically bound

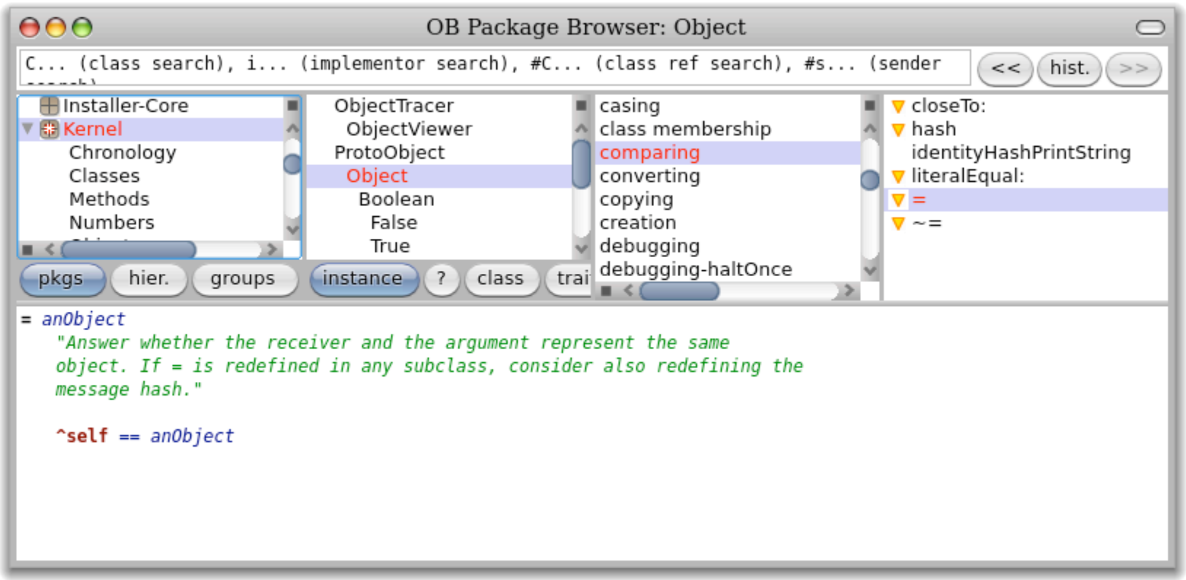
> Each object is an instance of one class
- — A class defines the structure and the behavior of its instances.
- — Single inheritance
- — A class is an instance of a metaclass

> 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: $).
```

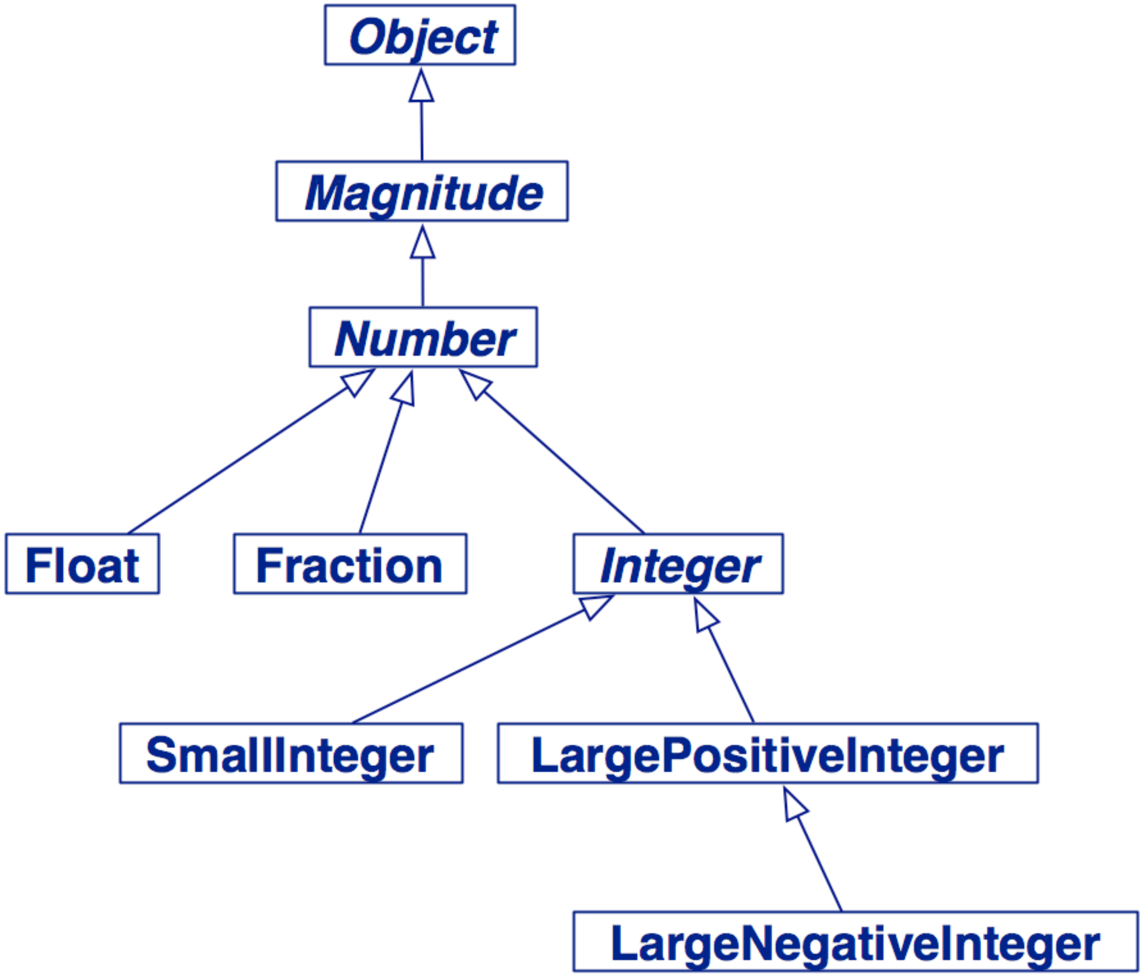
Object methods to support the programmer

`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.

Roadmap

- Object
- **Numbers, Characters, Strings and Arrays**
- Variables
- Blocks and Control structures
- Collections

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`	**3.3**
`1 class`	**SmallInteger**
`1 class maxVal class`	**SmallInteger**
`(1 class maxVal + 1) class`	**LargePositiveInteger**
`(1/3) + (2/3)`	**1**
`1000 factorial / 999 factorial`	**1000**
`2/3 + 1`	**(5/3)**

Browse the hierarchy to see how coercion works.

Try this in Java!

```
1000 factorial
```

```
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404
800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779
505950995276120874975462497043601418278094646496291056393887437886487337119181045825783647849977012
476632889835955735432513185323958463075557409114262417474349347553428646576611667797396668820291207
379143853719588249808126867838374559731746136085379534524221586593201928090878297308431392844403281
231558611036976801357304216168747609675871348312025478589320767169132448426236131412508780208000261
683151027341827977704784635868170164365024153691398281264810213092761244896359928705114964975419909
342221566832572080821333186116811553615836546984046708975602900950537616475847728421889679646244945
160765353408198901385442487984959953319101723355556602139450399736280750137837615307127761926849034
352625200015888535147331611702103968175921510907788019393178114194545257223865541461062892187960223
838971476088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779
911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153181617210
465832036786906117260158783520751516284225540265170483304226143974286933061690897968482590125458327
168226458066526769958652682272807075781391858178889652208164348344825993266043367660176999612831860
788386150279465955131156552036093988180612138558600301435694527224206344631797460594682573103790084
024432438465657245014402821885252470935190620929023136493273497565513958720559654228749774011413346
962715422845862377387538230483865688976461927383814900140767310446640259899490222221765904339901886
018566526485061799702356193897017860040811889729918311021171229845901641921068884387121855646124960
798722908519296819372388642614839657382291123125024186649353143970137428531926649875337218940694281
434118520158014123344828015051399694290153483077644569099073152433278288269864602789864321139083506
217095002597389863554277196742822248757586765752344220207573630569498825087968928162753848863396909
959826280956121450994871701244516461260379029309120889086942028510640182154399457156805941872748998
094254742173582401063677404595741785160829230135358081840096996372524230560855903700624271243416909
004153690105933983835777939410970027753472000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Characters

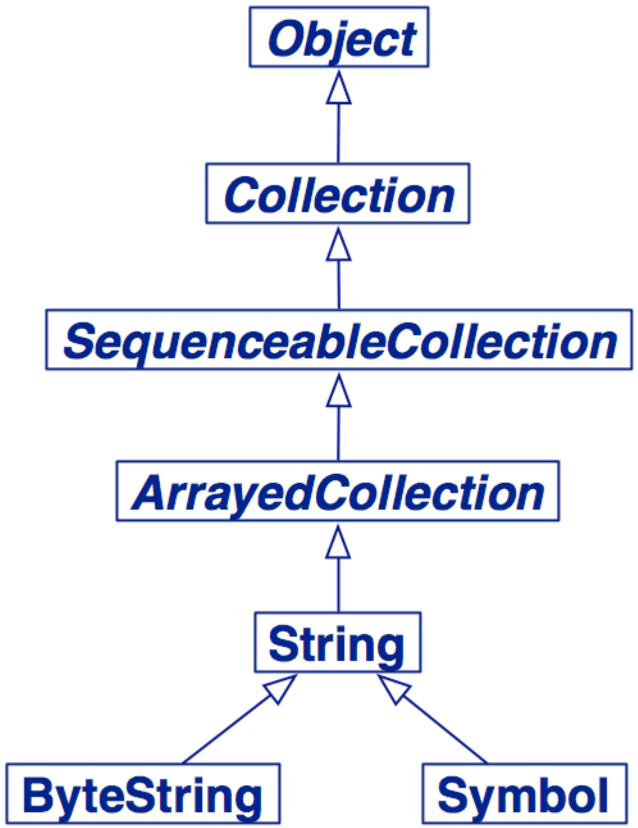
- Characters:

```
$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.

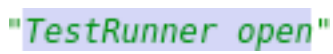
Comments and Tips

> 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"
```



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'
'calvin' == 'calvin'
'cal','vin' = 'calvin'
'cal','vin' == 'calvin'

#calvin = #calvin
#calvin == #calvin
#cal,#vin = #calvin
#cal,#vin == #calvin
#cal,#vin
(#cal,#vin) asSymbol == #calvin
```

```
true
true
true
false

true
true
true
false
'calvin'
true
```

!

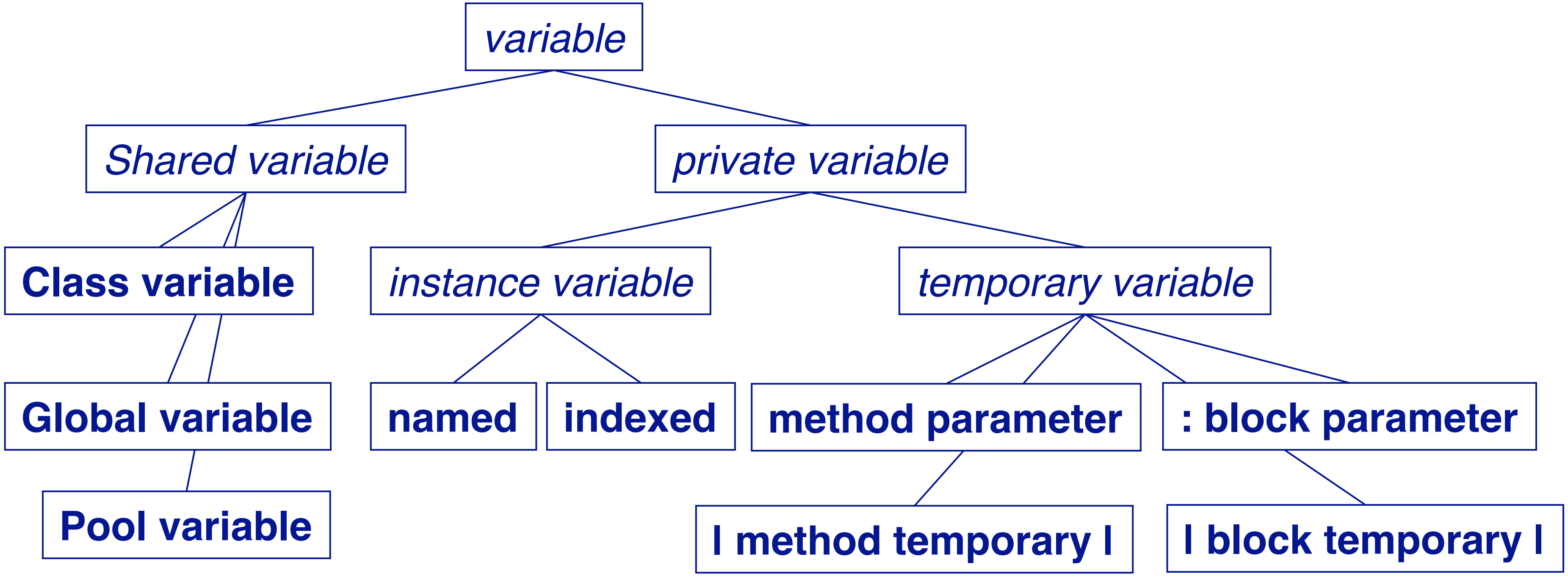
NB: Comparing strings is slower than comparing symbols by a factor of 5 to 10. However, converting a string to a symbol is more than 100 times more expensive.

Roadmap

- Object
- Numbers, Characters, Strings and Arrays
- **Variables**
- Blocks and Control structures
- Collections

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

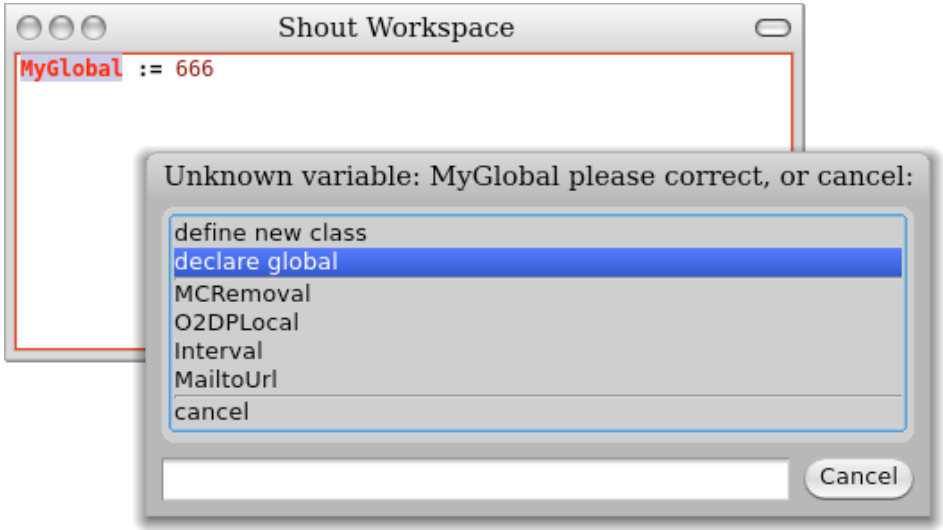
- Assignment binds a name to an object reference
  - Not done by message passing!
  - Method arguments cannot be assigned to!
    - *Use a temporary instead*
  - Different names can point to the same object!
    - *Watch out for unintended side effects*

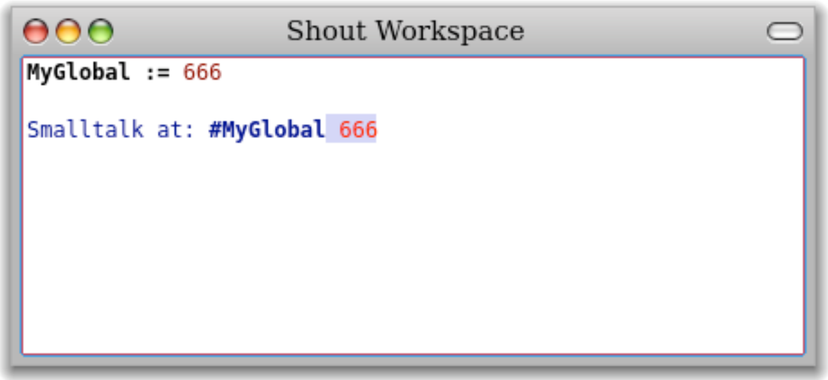
```
|p1 p2|
p1 := 3@4.
p2 := p1.
p1 setX: 5 setY: 6.
p2
```

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

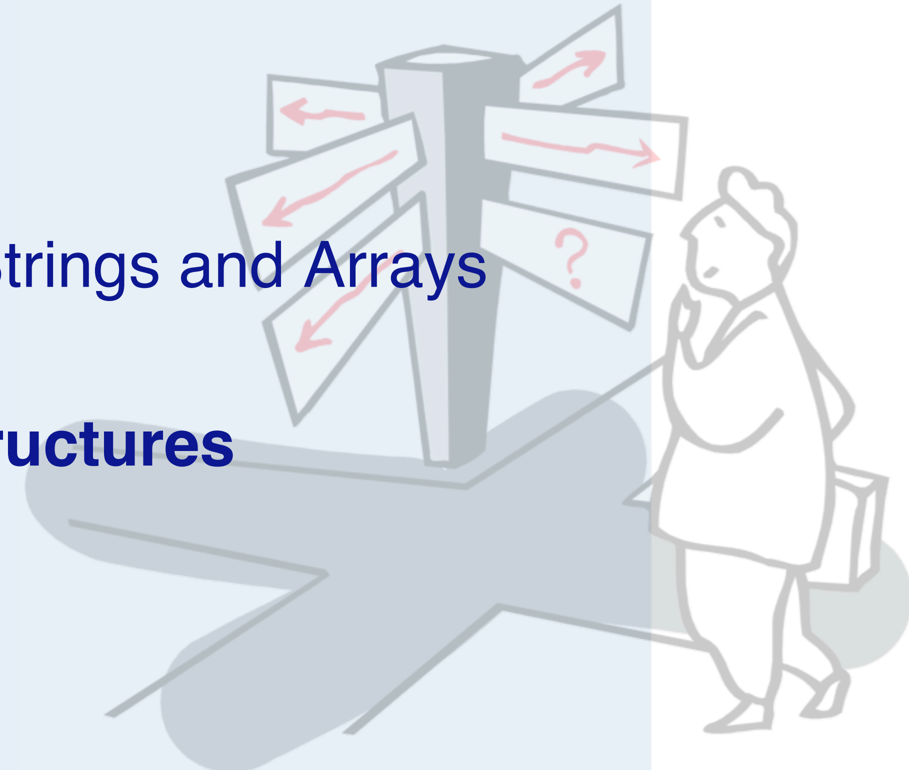
- Private to an object
  - Visible to methods of the defining class and subclasses
  - Has the same lifetime as the object
  - Define accessors (getters and setters) to facilitate initialization
    - *Put accessors in a private category!*

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

Roadmap



- Object
- Numbers, Characters, Strings and Arrays
- Variables
- **Blocks and Control structures**
- Collections

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

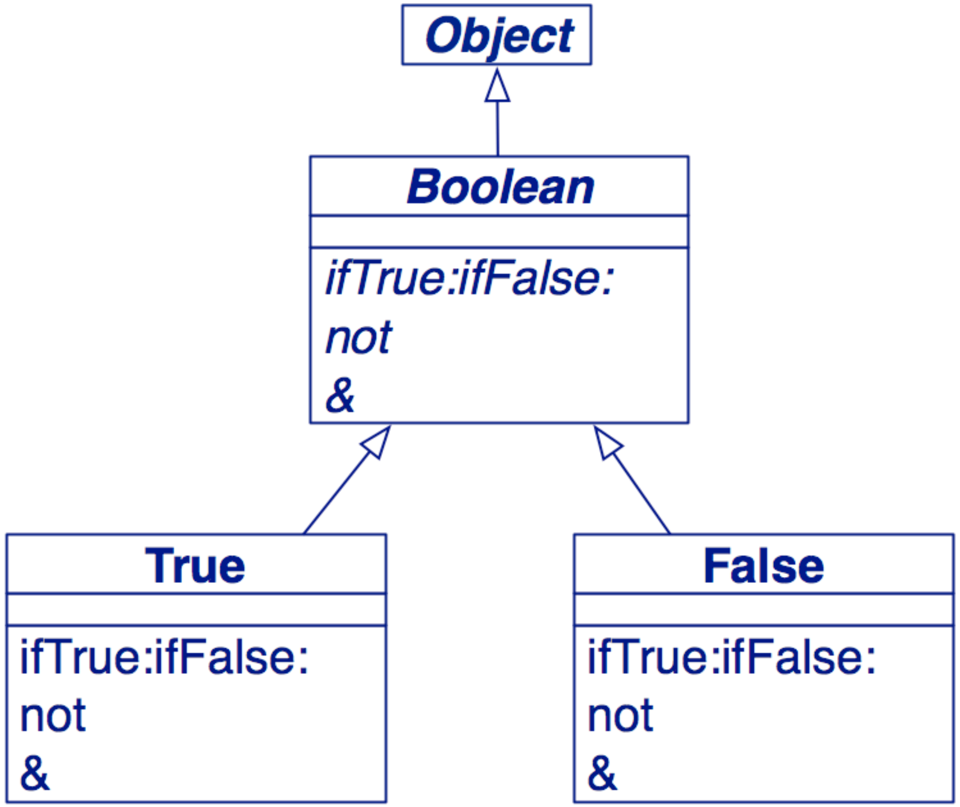
```
|sqr|
sqr := [:n | n*n ].
sqr value: 5
```

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!
```

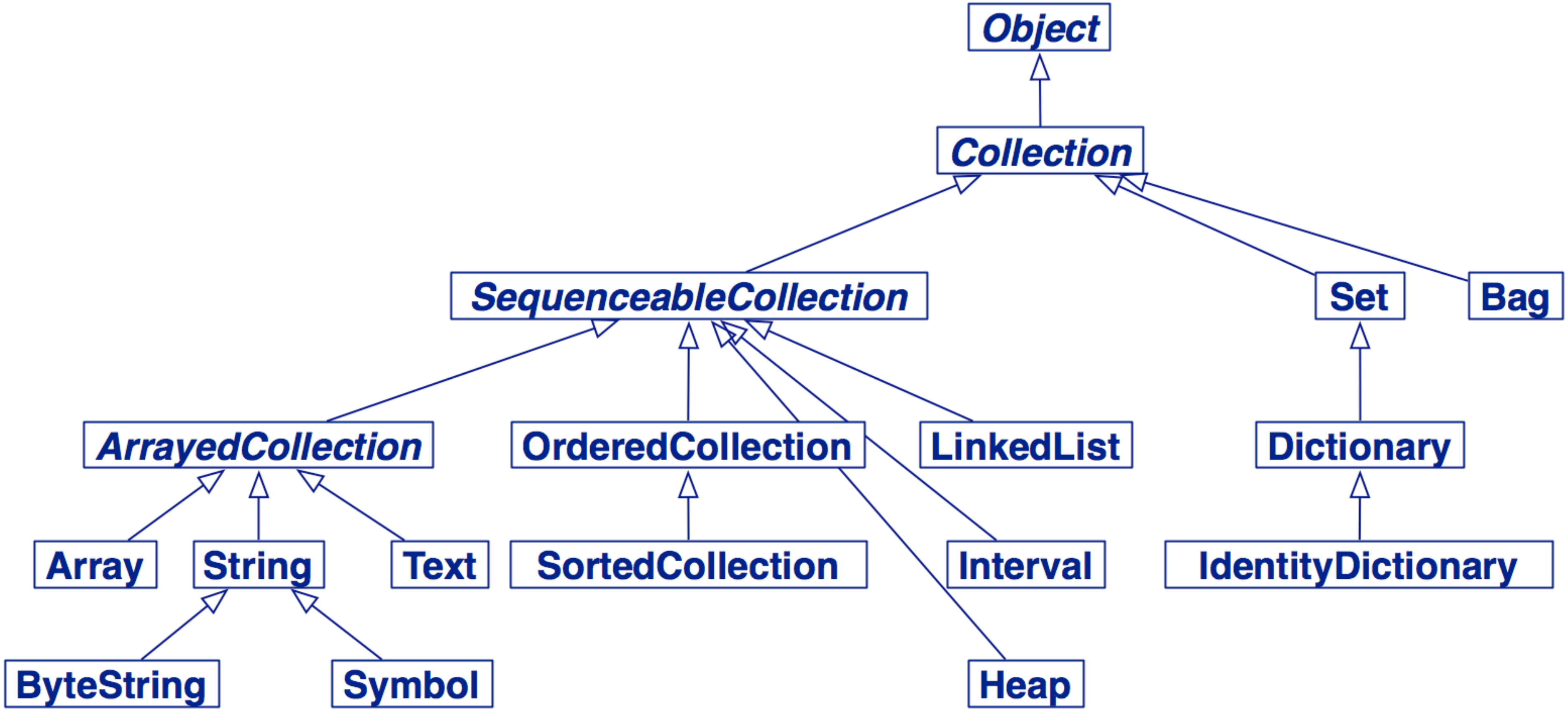
```
[:n |
   [n factorial]
      on: Error
      do: [0]
] value: -1
```

```
0
```

Roadmap

> Object
> Numbers, Characters, Strings and Arrays
> Variables
> Blocks and Control structures
> **Collections**

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
```

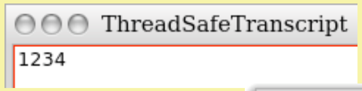
a Set('bar')

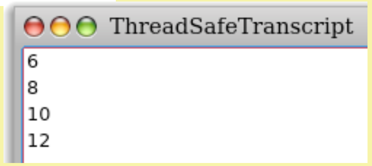
Accessing	at: *aKey*, at: *aKey* ifAbsent: *aBlock*, at: *aKey* ifAbsentPut: *aBlock*, at: *aKey* put: *aValue*, keys, values, associations
Removing	removeKey: *aKey*, removeKey: *aKey* ifAbsent: *aBlock*
Testing	includeKey: *aKey*
Enumerating	keysAndValuesDo: *aBlock*, associationsDo: *aBlock*, keysDo: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

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/