Smalltalk Lecture 1

Why Smalltalk?

- *Pure* object-oriented language and environment
  - "Everything is an object"

- Origin of *many innovations* in OO development
  - RDD, IDE, MVC, XUnit …

- Improves on many of its successors
  - Fully interactive and dynamic

What is Smalltalk?

> ***Pure OO language***

— Single inheritance

— Dynamically typed

> ***Language and environment***

— Guiding principle: *"Everything is an Object"*

— Graphical Smalltalk implementations have class browser, debugger, inspector, …

— Mature class library and tools

> ***Virtual machine***

— Objects exist in a persistent *image* [+ *changes*]

— Incremental compilation

Smalltalk vs. C++ vs. Java

	Smalltalk	**C++**	***Java***
Object model	Pure	Hybrid	Hybrid
Garbage collection	Automatic	Manual	Automatic
Inheritance	Single	Multiple	Single
Types	Dynamic	Static	Static
Reflection	Fully reflective	Introspection	Introspection
Concurrency	Semaphores, Monitors	Some libraries	Monitors
Modules	Categories, namespaces	Namespaces	Packages

Smalltalk: a State of Mind

> *Small and uniform language*
  - Syntax fits on one sheet of paper

> *Large library of reusable classes*
  - Basic Data Structures, GUI classes, Database Access, Internet, Graphics

> *Advanced development tools*
  - Browsers, GUI Builders, Inspectors, Change Management Tools, Crash Recovery Tools, Project Management Tools

> *Interactive virtual machine technology*
  - Truly platform-independent

> *Team Working Environment*
  - Releasing, versioning, deploying

Origins of Smalltalk

> **Project at Xerox PARC in 1970s**
>
> — Language and environment for new generation of graphical workstations (target: "Dynabook")

> **In Smalltalk-72, every object was an independent entity**
>
> — Language was designed for children (!)
>
> — Evolved towards a meta-reflective architecture

> **Smalltalk-80 is the standard**

Precursor, Innovator & Visionary

> First to be based on Graphics
>
> — Multi-Windowing Environment (Overlapping Windows)
>
> — Integrated Development Environment: Debugger, Compiler, Text Editor, Browser
>
> With a pointing device (yes, a *Mouse*)
>
> Ideas were taken over
>
> — Apple Lisa, Mac
>
> — Microsoft Windows 1.0
>
> Platform-independent Virtual Machine
>
> Garbage Collector
>
> Just-in-time Compilation
>
> Everything was there, the complete Source Code

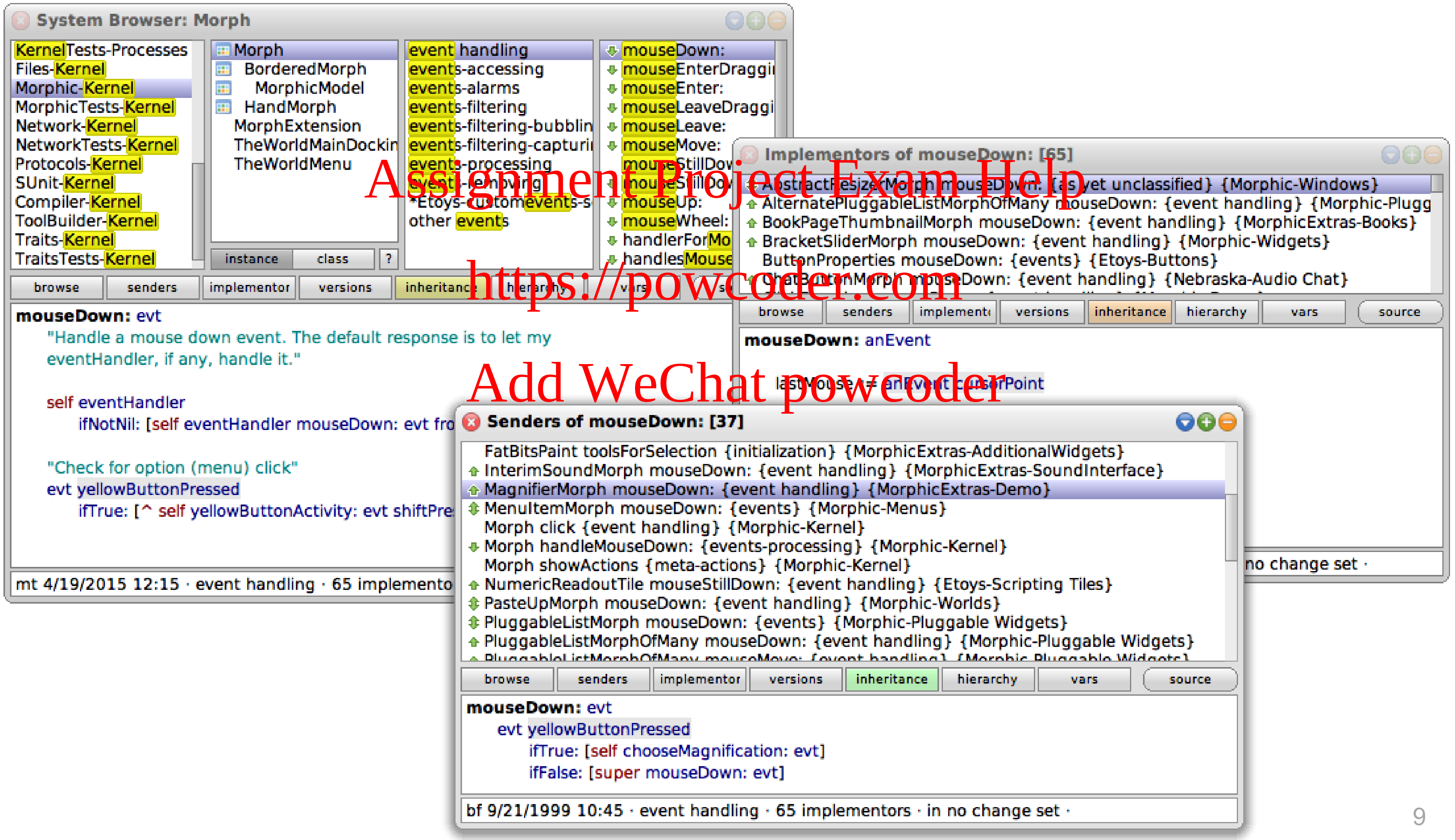
What are Squeak and Pharo?

- We will use text-based GNU Smalltalk in this course
- Encourage you to also look at graphical-based Smalltalk implementations

- Squeak is a modern, open-source, highly portable, fast, full-featured Smalltalk implementation
  - Based on original Smalltalk-80 code

- Pharo is a lean and clean fork of Squeak
  - www.pharo-project.org

Squeak desktop



System Browser: Morph

```
KernelTests-Processes      Morph                 event handling                  mouseDown:
Files-Kernel                 BorderedMorph       events-accessing                mouseEnterDraggi
Morphic-Kernel                 MorphicModel      events-alarms                   mouseEnter:
MorphicTests-Kernel          HandMorph           events-filtering                mouseLeaveDraggi
Network-Kernel               MorphExtension      events-filtering-bubblin        mouseLeave:
NetworkTests-Kernel          TheWorldMainDockin  events-filtering-capturi        mouseMove:
Protocols-Kernel             TheWorldMenu        events-processing               mouseStillDow
SUnit-Kernel                                     events-removing                 mouseStillDow
Compiler-Kernel                                  *Etoys-customevents-s           mouseUp:
ToolBuilder-Kernel                               other events                    mouseWheel:
Traits-Kernel                                                                    handlerForMo
TraitsTests-Kernel           instance  class  ?                                  handlesMouse
```

browse | senders | implementor | versions | inheritance | hierarchy | vars

```
mouseDown: evt
	"Handle a mouse down event. The default response is to let my
	eventHandler, if any, handle it."

	self eventHandler
		ifNotNil: [self eventHandler mouseDown: evt fro

	"Check for option (menu) click"
	evt yellowButtonPressed
		ifTrue: [^ self yellowButtonActivity: evt shiftPre
```

mt 4/19/2015 12:15 · event handling · 65 implemento

Implementors of mouseDown: [65]

```
AbstractResizerMorph mouseDown: {as yet unclassified} {Morphic-Windows}
AlternatePluggableListMorphOfMany mouseDown: {event handling} {Morphic-Plugg
BookPageThumbnailMorph mouseDown: {event handling} {MorphicExtras-Books}
BracketSliderMorph mouseDown: {event handling} {Morphic-Widgets}
ButtonProperties mouseDown: {events} {Etoys-Buttons}
ChatButtonMorph mouseDown: {event handling} {Nebraska-Audio Chat}
```

browse | senders | implementors | versions | inheritance | hierarchy | vars | source

```
mouseDown: anEvent

	lastMouse := anEvent cursorPoint
```

no change set ·

Senders of mouseDown: [37]

```
FatBitsPaint toolsForSelection {initialization} {MorphicExtras-AdditionalWidgets}
InterimSoundMorph mouseDown: {event handling} {MorphicExtras-SoundInterface}
MagnifierMorph mouseDown: {event handling} {MorphicExtras-Demo}
MenuItemMorph mouseDown: {events} {Morphic-Menus}
Morph click {event handling} {Morphic-Kernel}
Morph handleMouseDown: {events-processing} {Morphic-Kernel}
Morph showActions {meta-actions} {Morphic-Kernel}
NumericReadoutTile mouseStillDown: {event handling} {Etoys-Scripting Tiles}
PasteUpMorph mouseDown: {event handling} {Morphic-Worlds}
PluggableListMorph mouseDown: {events} {Morphic-Pluggable Widgets}
PluggableListMorphOfMany mouseDown: {event handling} {Morphic-Pluggable Widgets}
PluggableListMorphOfMany mouseMove: {event handling} {Morphic-Pluggable Widgets}
```

browse | senders | implementor | versions | inheritance | hierarchy | vars | source

```
mouseDown: evt
	evt yellowButtonPressed
		ifTrue: [self chooseMagnification: evt]
		ifFalse: [super mouseDown: evt]
```

bf 9/21/1999 10:45 · event handling · 65 implementors · in no change set ·

Smalltalk — Key Concepts

> *Everything is an object*
> — numbers, files, editors, compilers, points, tools, booleans …

> Everything happens by *sending messages*

> Every object is an instance of one class

> — which is also an object
> — A class defines the structure and the behavior of its instances.

> Objects have private (protected) state
> — Encapsulation boundary is the object

> Dynamic binding
> — Variables are dynamically typed and bound

Objects and Classes

> *Every object is an instance of a class*

- A class specifies the structure and the behaviour of all its instances

- Instances of a class share the same behavior and have a specific state

- *Classes are objects* that create other instances
- Metaclasses are classes that create classes as instances
- Metaclasses describe class behaviour and state (subclasses, method dictionary, instance variables...)

Messages and Methods

> Message — which action to perform

```
aWorkstation accept: aPacket
aMonster eat: aCookie
```

> Method — how to carry out the action

```
accept: aPacket [
    (aPacket isAddressedTo: self)
       ifTrue:[
           Transcript show:
              'A packet is accepted by the Workstation ',
           self name asString ]
       ifFalse: [super accept: aPacket]
]
```

Smalltalk Run-Time Architecture

> Byte-code is translated to native code by a just-in-time compiler
  — Some Smalltalks, but not Pharo

> Source and changes are not needed to interpret the byte-code.

  — Just needed for development
  — Normally removed for deployment

> An application can be delivered as byte-code files that will be executed with a VM.
  — The development image is stripped to remove the unnecessary development components.

Objects in Smalltalk

- *Everything* is an object
  - Things only happen by message passing
  - Variables are dynamically bound
- State is private to objects
  - "protected" for subclasses
- Methods are public
  - "private" methods by convention only
- (Nearly) every object is a reference
  - Unused objects are garbage-collected
- Single inheritance

Smalltalk Syntax on a Postcard

```
exampleWithNumber: x [
"A method that illustrates every part of Smalltalk method syntax
except primitives. It has unary, binary, and key word messages,
declares arguments and temporaries (but not block temporaries),
accesses a global variable (but not and instance variable),
uses literals (array, character, symbol, string, integer, float),
uses the pseudo variable true false, nil, self, and super,
and has sequence, assignment, return and cascade. It has both zero
argument and one argument blocks. It doesn't do anything useful, though"
    |y|
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #($a #a 'a' 1 1.0)
        do: [:each | Transcript
            show: (each class name);
            show: (each printString);
            show: ' '].
    ^ x < y
]
```

Language Constructs

`^`	return
"…"	comment
`#`	symbol or array
'…'	string
`[ ]`	block or byte array
`.`	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

Examples

comment:	`"a comment"`
character:	`$c $h $a $r $a $c $t $e $r $s $# $@`
string:	`'a nice string'`
symbol:	`#mac #+`
array:	`#(1 2 3 (1 3) $a 4)`
dynamic array:	`{ 1 + 2 . 3 / 4 . 5 * 6}`
Integer:	`72, 2r101`
real:	`1.5, 6.03e-34, 2.4e7`
fraction:	`1/33`
boolean:	`true, false`
pseudo variables	`self, super`
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:` ...

Messages instead of keywords

> In most languages, basic operators and control constructs are defined as language constructs and keywords

> In Smalltalk, there are only *messages* sent to objects

— `bitShift:` (>>) is just *a message sent to a number*

```
10 bitShift: 2
```

— `ifTrue:` (if-then-else) is just *a message sent to a boolean*

```
(x>1) ifTrue: [ Transcript show: 'bigger' ]
```

— `do:`, `to:do:` (loops) are just *messages to collections or numbers*

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

> Minimal parsing

> Language is *extensible*

Smalltalk Syntax

Every expression is a message send

> *Unary messages*

```
Transcript cr
5 factorial
```

> *Binary messages*

```
3 + 4
5 * 6
```

> *Keyword messages*

```
Transcript show: 'hello world'
2 raisedTo: 32
segment fromX: 1 fromY: 1
        toX: 50 toY: 100
```

Precedence

(…) > Unary > Binary > Keyword

1. Evaluate *left-to-right*
2. Unary messages have *highest precedence*
3. Next are binary messages
4. Keyword messages have *lowest precedence*

```
2 raisedTo: 1 + 3 factorial
```

```
128
```

5. Use *parentheses* to change precedence

```
1 + 2 * 3
1 + (2 * 3)
```

```
9        (!)
7
```

Binary Messages

> Syntax:
- aReceiver *aSelector* anArgument
- Where *aSelector* is made up of 1 or 2 characters from:

  `+ - / \ * ~ < > = @ % | & ! ? ,`
- Except: second character may not be `$`

> Examples:

```
2 * 3 - 5
5 >= 7
6 = 7
'hello', ' ', 'world'
(3@4) + (1@2)
2<<5
64>>5
```

More syntax

> Comments are enclosed in *double quotes*

```
"This is a comment."
```

> Use *periods* to separate expressions

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr     "NB: don't need one here"
```

> Use *semi-colons* to send a cascade of messages to the same object

```
Transcript cr; show: 'hello world'; cr
```

Variables

> Declare local variables with `|  ...  |`

```
| x y |
```

> Use `:=` to assign a value to a variable

```
x := 1
```

> Old fashioned assignment operator: $\leftarrow$ (must type “_”)

Method Return

> Use a *caret* to return a value from a method or a block

```
max: aNumber
    ^ self < aNumber
        ifTrue: [aNumber]
        ifFalse: [self]
```

```
1 max: 2
```

2

> Methods *always* return a value

— By default, methods return `self`

Block closures

> Use *square brackets* to delay evaluation of expressions

```
^ 1 < 2 ifTrue: ['smaller'] ifFalse: ['bigger']
```

```
'smaller'
```

Variables

- *Local variables* within methods (or blocks) are delimited by `|var|`
- *Block parameters* are delimited by `:var|`

```
OrderedCollection>>collect: aBlock [
    "Evaluate aBlock with each of my elements as the argument."
    | newCollection |
    newCollection := self species new: self size.
    firstIndex to: lastIndex do:
        [ :index |
        newCollection addLast: (aBlock value: (array at: index))].
    ^ newCollection
]
```

```
[:n | |x y| x := n+1. y := n-1. x * y] value: 10
```

99

Control Structures

> *Every control structure is realized by message sends*

```
|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 ]
```

Creating objects

> *Class methods*

```
OrderedCollection new
Array new: 200
Array with: 10 with: 20 with: 30
```

> *Factory methods*

```
1@2      "a Point"
1/2      "a Fraction"
```

Creating classes

> Send a message to a class (!)

```
Number subclass: #Complex

Complex instanceVariableNames: 'real imaginary'
```

Some Conventions

> Method selector is a *symbol*, e.g., `#add:`
> Method scope conventions using `>>`
(for displaying the methods of a class)

— *Instance Method* defined in the class `Node`

```
Node>>accept: aPacket
```

— *Class Method* defined in the class `Node class`
(i.e., in the class of the the class Node)

```
Node class>>withName: aSymbol
```

> `aSomething` is an instance of the class `Something`

Example: Money (abstract Smalltalk syntax)

> We define Money as a subclass of Object, with getters and setters

```
Object subclass: #Money.
Money instanceVariableNames: 'amount currency'.
```

```
Money>>amount: aNumber
   amount:= aNumber.
```

```
Money>>amount
   ^ amount
```

```
Money>>currency: aString
   currency := aString.
```

```
Money>>currency
   ^ currency
```

GNU Smalltalk syntax

- We define Money as a subclass of Object, with getters and setters

```
Object subclass: #Money.
Money instanceVariableNames: 'amount currency'.

Money extend [
    amount [ ^amount ]
    amount: aNumber [ amount := aNumber ]
    currency [ ^currency ]
    currency: aString [ currency := aString ]
].
```

Test cases

```
x := Money new currency: 'USD'; amount: 100.
y := Money new currency: 'Euro'; amount: 500.

x amount printNl.          "100"
x currency printNl.        "'USD'"
y amount printNl.          "500"
y currency printNl.        "'Euro'"

x amount: 999.
y currency: 'Yen'.

x amount printNl.          "999"
x currency printNl.        "'USD'"
y amount printNl.          "500"
y currency printNl.        "'Yen'"
```

Comparisons (binary operators)

```
Money>>= aMoney
   ^ self currency = aMoney currency
   and: [ self amount = aMoney amount ]
```

```
Money>>~= aMoney
   ^ (self = aMoney) not
```

```
Money extend [    "GNU Smalltalk syntax"
    = aMoney [
        ^ self currency = aMoney currency
          and: [ self amount = aMoney amount ]
    ]
    ~= aMoney [^ (self = aMoney) not]
].
```

More test cases

```
x := Money new currency: 'USD'; amount: 100.
y := Money new currency: 'Euro'; amount: 500.
z := Money new currency: 'USD'; amount: 100.

(x = y) printNl.          "false"
(x ~= y) printNl.         "true"
(x = z) printNl.          "true"
(x ~= z) printNl.         "false"
```

Using GNU Smalltalk

```
> gst
    "evaluates Smalltalk code line-by-line"
st> 2 + 3 * 4
20
st> "press control-D to exit interpreter"

> gst Money.st
    "runs all the Smalltalk code in given file"


> gst
st> FileStream fileIn: 'Money.st'
    "loads and runs code in file, stays in gst"
st>
```