Variables in Pharo

Marcus Denker, Inria

http://marcusdenker.de

Plan for an interactive Exploration

- These Slides where done as an outline / plan for an interactive exploration
- They might therefore be not exactly the same as the content of the Demo

Variables in ST80

- Temporary Variable
- Instance Variable, Class Instance Variable
- Class Variable (and Pool Variable)
- Globals
- Pseudo Variables: self, super, thisContext

Instance Variables

- Defined by the Class (or Trait)
- Can be read via the object:
- `instVarNamed:(put:)`, `#instVarAt:(put:)`
- Instance Variables have an offset in the Object
- Defined by the order of the defined vars in the Hierarchy

```
1@2 instVarNamed: 'x'
```

Temporary Variable

- Defined by a method or Block
    - Arguments are temps, too
- Can be read via the context
- `#tempNamed:, tempNamed:put:`

```
[| temp |  temp := 1. thisContext tempNamed:  'temp' ] value
```

- With Closures this is more complex than you ever want to know!

Globals

- Entries in the “Smalltalk globals” Dictionary
- Contain the value

```
Smalltalk globals at: #Object.
Object binding value.
```

- Can be read via the global Dictionary
- Access via #value / value: on the Association
- Class Vars and Pool Vars are just Associations from other Dictionaries

“Everything is an Object”

- For Variables… not really

Globals/Class Vars

- Here we have at least the Association (#binding):

```
Object binding
```

- But there is no “GlobalVariable” class
    - No API other than #value:/#value
    - Classes define just names of variables

Instance Variables

- The class just knows the names

```
Point allInstVarNames
```

- There is no Object representing instance variables
- Classes define just names of variables
- Bytecode accesses by offset

Temporary Variables

- The methods know nothing. Even to know the variable name we need the compiler (and the source)

- There is no object representing temp Variables

- Reflective read and write is *hard* -> compiler needs to create extensive meta-data

Why Not Do Better?

- Of course memory was a concern in 1980, but today we should be able to do better!
- Why not have objects (and a class Hierarchy) that describes all Variables in the system?

Variables

- Every defined Variable is described a meta object
- Class Hierarchy: Variable
  - GlobalVariable
  - ClassVariable
  - Temporary Variables
  - Instance Variables (aka Slots)

The Hierarchy

- Variable
  - LiteralVariable
    - ClassVariable
    - GlobalVariable
    - UndeclaredVariable
    - WorkspaceVariable

- LocalVariable
  - ArgumentVariable
  - TemporaryVariable
- ReservedVariable
  - SelfVariable
  - SuperVariable
  - ThisContextVariable
- Slot

Example: vars of a class

- Get all Variables of a class
- Inspect it
- #usingMethods

```
Point instanceVariables
```

Instance Variable

- Read x in a Point

```
(Point instanceVariables first) read: (5@4)
```

- Write

```
point := 5@4.
(Point instanceVariables first) write: 100 to: point.
```

- read/write without sending a message to the object

Globals

- Object binding class
- Object binding read
- We keep the Association API so the Global Variables can play the role of associations in the global dictionary.

Object binding usingMethods

Temporary Variables

- There are too many to allocate them all
- They are created on demand (with the AST)

```
((LinkedList>>#do:) temporaryVariableNamed: 'aLink')
```

#lookupVar:

- Every variable knows the scope is was defined in
- Every scope know the outer scope

```
(Point slotNamed: #x ) scope outerScope
```

- #lookupVar: looks up names along the scope

```
[ | temp |thisContext lookupVar: 'temp' ] value.

[ | temp |thisContext lookupVar: ‘Object' ] value
```

Debugger: Read Vars

- In the Debugger we to be able to read Variables from a DoIt.

- lookupVar, then readInContext works for all Variables!

```
[ | temp | temp :=1 . (thisContext lookupVar: 'temp')
          readInContext: thisContext] value
```

- DoItIn: uses this:

```
Context>>readVariableNamed: aName
     ^ (self lookupVar: aName) readInContext: self
```

Variables as AST Annotations

- Pharo uses the RB AST
- RBVariableNode instance for every use of a Variable
- Annotated with subclasses of Variables:

```
(Point>>#x) ast variableNodes first variable == (Point slotNamed: #x)
```

OCASTSemanticAnalyzer

- OCASTSemanticAnalyzer is the visitor that does the name analysis
- Adds a scope for each block/the method
- Adds defined variables to the scope
- Every RBVariableNode use will get annotated with the variable that #lookUpVar: finds

Variables and Bytecode

- Compiler just delegates to the Variable
- InstanceVariableSlot>>#emitStore:

```
emitStore: methodBuilder
    "generate store bytecode"
    methodBuilder storeInstVar: index
```

- emitStore/emitValue:

Does that mean…

- If variables are defined by a class, could we not make a subclass?
  - And even override the code generation methods ?!

Now let’s create our own kind of Variable!

Lazy Variables

- Two ways to initialize instance state in ST80
  - implement #initialize method (#new calls it)
  - use accessors and lazy init pattern

- Can we not do better? Can the Variable not initialize itself?

Lazy Variables

```
InstanceVariableSlot subclass: #LazyInitializedInstanceVariable
   instanceVariableNames: 'default'
   classVariableNames: ''
   package: ‘CompilerTalk'
```

```
printOn: aStream
   aStream
      store: self name;
      nextPutAll: ' => ';
      nextPutAll: self class name;
      nextPutAll: ' default: '.
   default printOn: aStream
```

Lazy Variables

```
read: anObject
   "if the value is nil, we write the default value "
   ^ (super read: anObject) ifNil: [
          self write: default to: anObject]

emitValue: aMethodBuilder
   "generate bytecode for '<varname> ifNil: [<varname> := default]'"
   aMethodBuilder
      pushInstVar: index;
      pushDup;
      pushLiteral: nil;
      send: #==;
      jumpAheadTo: #target if: false;
      popTop;
      pushLiteral: default;
      storeInstVar: index;
      jumpAheadTarget: #target
```

Let’s use it

```
Object subclass: #MyClass
   instanceVariableNames: 'var'
   classVariableNames: ''
   package: ‘CompilerTalk'
```

?

How can we make ‘var’ to be a LazyInitializedInstanceVariable?

Class Definition

- We need a new way to define classes: Fluid Class Definition
  - uses cascade for extensibility
  - no string, but {} arrays for variables

Fluid Class Definition

```
Object << #Point
    slots: { #x . #y };
    tag: 'BasicObjects';
    package: 'Kernel'
```

Fluid Class Definition

- Pharo9: Default is the ST80 style class definition
  - Fluid can be enabled
  - It is used automatically when needed (when using a self defined Variable, for example)
  - Goal: Default for Pharo 10

Let’s use it

```
Object << #MyClass
    slots: { #var };
    package: ‘CompilerTalk'
```

#notation for normal instance Variables

Let’s use it

```
Object << #MyClass
   slots: { #var => LazyInitializedInstanceVariable default: 5 };
   package: ‘CompilerTalk'
```

For defining other variables: use =>

```
MyClass new var.
(MyClass new var: 8) var
```

Inspect method to see bytecode
put halt in read: method of Slot

`thisProcess`

- To get the current Process we use a message send to a global variable. But we could use a variable like `thisProcess`

- This avoids a message send (and possible interrupt check) as we can emit a bytecode

self, thisContext…

- (Object lookupVar: ‘thisContext’) usingMethods
- See classes
  - ThisContextVariable
  - SelfVariable
  - SuperVariable

The Code

```
ReservedVariable << #ThisProcessVariable
	slots: {};
	tag: 'Variables';
	package: 'Kernel'

emitValue: methodBuilder
	methodBuilder pushThisProcess

======
class side:
variableName
	^ 'thisProcess'

======
Smalltalk globals resetReservedVariables
```

Compatibility

- Fully backward compatible: we can load ST80 style class definitions (and Pharo9 just shows this view by default)
- Reflective API is compatible: “instVarAt:”… still exist
- If you want to restrict yourself to ST80, a checker could be easily created

Compatibility

- But if you start to use your own kind of Variables, you code will not be “ST80” compatible anymore
- But you will be able to use the power of the new abstraction provided
- Was this not the original idea behind Smalltalk? That a Programming System is a Medium?

Where do we use it?

- We are careful! We use Pharo ourselves…
  - We need a stable system to work with
  - We need to learn about how to use Variables best
- Variables are used by the Compiler internally
- Every instance variable is an InstanceVariableSlot (Globals, class vars)
- Variables are used by the Debugging infrastructure to read/write
  - replaced the DebuggerMethodMap
- The Spec UI Framework uses ObservableSlot

Next Steps Variables

- DoitVariable for nicer code in #DoItIn: methods
- Undeclared Variables
    - programmer interaction in read/write, not compile!
    - better behaviour for test-first development
- Implement ThisProcessVariable in Pharo10
- Use WeakSlot to simplify some code

Next Steps Fluid Classes

- Finish the last problems (see issue tracker), make it the default
- Experiments with Meta Data for Classes
  - Tag abstract classes
  - Experiment with Pragmas for classes
  - Compiler, compiler plugging, compiler options

More..

- Extend the MetaLink Model to allow MetaLinks on Variables (first code is there already)

- More Experiments about Slot Composition

  - Implement Default value once, use it on ClassVariable, InstanceVariableSlot and WeakSlot

  - First prototype, but it turned out to be too complex

Thanks…

- This is the work on *many* contributors from the Pharo Community
- Thanks for lots of interesting discussions, ideas, and code!

Help Wanted

- We are always interested in improvements!
- Pharo 10 is under active development
  - 30-40 Pull Requests integrated per week
- Your Improvements are Welcome!

https://github.com/pharo-project/pharo

Questions?