CS 535 Object-Oriented Programming & Design
Fall Semester, 2008
Doc 17 Some Parsing
Nov 20 2008

References

Domain Specific Languages, http://en.wikipedia.org/wiki/Domain-specific_programming_language

Example - Turtle Graphics

Turtle Graphics - used help teach programming

Program Turtle to

- Move across screen
- Draw patterns

Operations

- move
- turn
- penUp
- penDown

Sample Program

```
penDown
move 5
turn 90 left
move 10
turn 90 left
move 5
turn 90 left
move 10
```

How to parse Turtle Program

As String

```
turtleProgram := 'penDown
move 5
turn 90 left
move 10
turn 90 left
move 5
turn 90 left
move 10'.
lines := turtleProgram tokensBasedOn: Character cr.
aLine := lines first.
parts := aLine words
```

How to parse Turtle Program

Using Stream

```
turtleProgram := 'penDown
move 5
turn 90 left
move 10
turn 90 left
move 5
turn 90 left
move 10'.

commandStream := ReadStream on: turtleProgram.
command := commandStream upto: Character cr.
token := commandStream upto: Character space
```

TurtleStream

Possible Operations

nextToken
nextCommand
commandArguments

Executing Turtle Program/Command

TurtleInterpreter class

- Responsibilities
    - Analyze and execute turtle programs
- Collaborations
    - Turtle
    - TurtleStream

Turtle class

- Responsibilities
    - Draw on screen
    - Perform operations

TurtleInterpreter

Instance variables

- turtle - instance of Turtle
- source - instance of TurtleStream

```
TurtleInterpreter on: aProgramString
    Initializes turtle and source

        turtle := Turtle new.
        source := TurtleStream on: aProgramString
```

```
TurtleInterpreter>>evaluate
    [source atEnd]
        whileFalse: [self evaluateCommand]
```

Simple Solution

```
TurtleInterpreter>>evaluateCommand
    | command |
    command := source nextToken.
    command asLowercase = 'penUp'
        ifTrue: [^self penUp].
    command asLowercase = 'move'
        ifTrue: [^self move].
    etc.

  TurtleInterpreter>>penUp
      turtle penUp

  TurtleInterpreter>>move
      | distance |
      distance := source nextToken.
      turtle move: distance
```

Smalltalk Magic - perform

Execute symbols or strings as methods

```
'CAT' perform: #asLowercase
'CAT' perform: 'asLowercase' asSymbol
'Cat dog' perform: #tokensBasedOn: with: Character space

'CAT' perform: 'asLowerase' asSymbol
```

Dangerous Solution

```
TurtleInterpreter>>evaluateCommand
    | command |
    command := source nextToken.
    self perform: command asSymbol


TurtleInterpreter>>penUp
    turtle penUp

TurtleInterpreter>>move
    | distance |
    distance := source nextToken.
    turtle move: distance
```

Some What Better Solution

```
TurtleInterpreter>>initialize
    commandMap := Dictionary new.
    commandMap
        at: 'penup' put: #penUp;
        at: 'move' put: #move;
        etc.
```

```
TurtleInterpreter>>evaluateCommand
    | command |
    command := source nextToken.
    (commandMap containsKey: command asLowercase)
        ifTrue: [self perform: (commandMap at: command)]
        ifFalse: [deal with bad command here]
```

Command Objects

Create a Command Class for each command in language

Command knows how to

- Execute the command
- Undo the command

Allows stepping through the program and undoing operations

MoveCommand

```
Smalltalk defineClass: #MoveCommand
     superclass: #{Core.Object}
     instanceVariableNames: 'turtle amount '



 MoveCommand>>execute
     turtle move: amount

 MoveCommand>>undo
     turtle
          left: 180;
          move: amount;
          left: 180
```

Parsing

```
TurtleInterpreter>>parse
    [source atEnd]
        whileFalse: [self parseCommand]
```

```
TurtleInterpreter>>parseCommand
    | command |
    command := source nextToken.
    command asLowercase = 'penUp'
        ifTrue: [^self penUp].
    command asLowercase = 'move'
        ifTrue: [^self move].
    etc.
```

```
TurtleInterpreter>>penUp
    commands
        add: (PenUpCommand on: turtle).

TurtleInterpreter>>move
    | distance |
    distance := source nextToken.
    commands
        add: (MoveCommand turtle: turtle distance: distance)
```

Running

```
TurtleInterpreter>>run
    commands do: [:each | each execute]
```

Build a Compiler

AT Parser Compiler

The parser compiler classes make it easier to write compilers in Smalltalk

SmaCC

Smalltalk Compiler-Compiler

More Smalltalk Magic - evaluate

Compiler evaluate: aString

Compiles and executes the Smalltalk code in aString

```
Compiler evaluate: ' 1 + 2'.

Compiler evaluate: 'Transcript show: (1 + 2) printString'
```

```
| userScript |
userScript := Dialog
            request: 'Write a Smalltalk expression'
            initialAnswer: '1 + 2'.
Compiler evaluate: userScript.
```

Evaluating Blocks

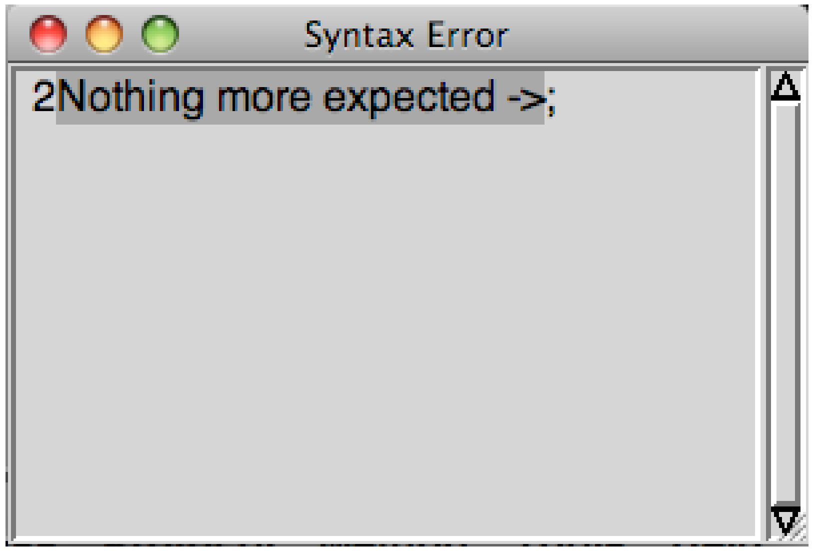
```
| script |
script := Compiler evaluate:  '[1 + 2]'.
script value
```

Embedding code in a Block

```
| userScript compiledCode |
userScript := Dialog
            request: 'Write a Smalltalk expression'
            initialAnswer: '1 + 2'.
compiledCode := Compiler evaluate: '[' , userScript , ']'.
compiledCode value
```

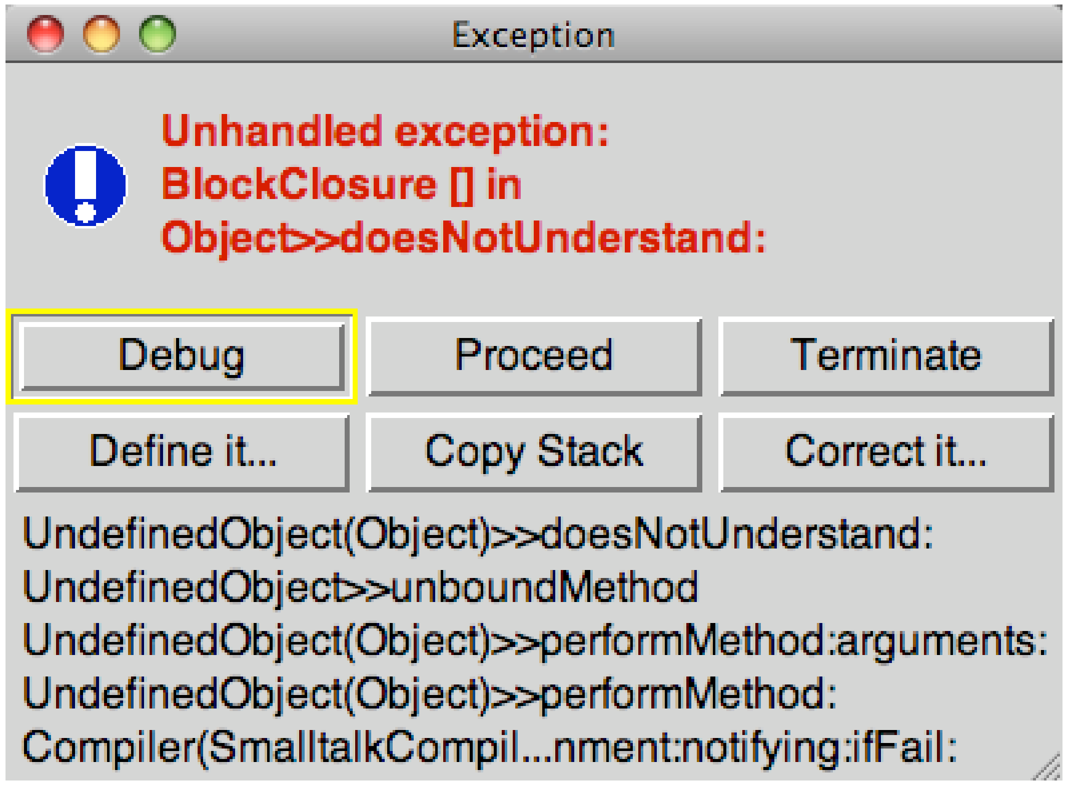
There are problems

Compiler evaluate: '2;'



```
2Nothing more expected ->;
```

Compiler evaluate: 'bar + 3'



Unhandled exception: BlockClosure [] in Object>>doesNotUnderstand:

```
UndefinedObject(Object)>>doesNotUnderstand:
UndefinedObject>>unboundMethod
UndefinedObject(Object)>>performMethod:arguments:
UndefinedObject(Object)>>performMethod:
Compiler(SmalltalkCompil...nment:notifying:ifFail:
```

Obvious Solution

If the default action is not correct for your situation then

on:do: can be used to catch the errors

```
[Compiler evaluate: '2;']
   on: Notification
   do: [:error | error handling code]

[Compiler evaluate: 'foo + 2']
   on: Notification
   do: [:error | error handling code]
```

External Variables in the Script

Ways to provide scripts access to existing variables

- Use block variables
- Use evaluate:for:logged:

Using Blocks

```
| scriptString scriptBlockString scriptBlock |
scriptString := 'price > 10
            ifTrue:[ ''expensive'']
            ifFalse:[ ''cheap'']'.
scriptBlockString := '[:price | ' , scriptString , ' ]'.
scriptBlock := Compiler evaluate: scriptBlockString.
scriptBlock value: 12
```

In the string literal assigned to scriptString, contains code that is to have a string literal ('expensive'), the inner string literals need to be quoted with two single quotes. If the script is not created from a string literal the double single quotes are not needed.

evaluate:for:logged:

Evaluates code as if it were part of an object

Used primarily for tools like debugger

Violates information hiding should be avoided

Can be used to add methods to objects

```
Smalltalk.CS535 defineClass: #SampleClass
   superclass: #{Core.Object}
   instanceVariableNames: 'age '


SampleClass>>age: anInteger
   name := anInteger
```

Script

```
| dataObject |
dataObject := SampleClass new.
dataObject age: 10.
script := ' age + 5 '.
Compiler
   evaluate: script
   for: dataObject
   logged: false
```

Since the script is run as part of the object dataObject it can access instance variable 'age'
If the logged: parameter is true the execution of the code is recorded in the change file

Undefined Variables

Evaluate the following twice

```
Compiler evaluate: 'foobar'
```

The first time you will see in the transcript:

```
UndefinedObject #DoIt - foobar is undeclared
```

The second time this message will not appear.

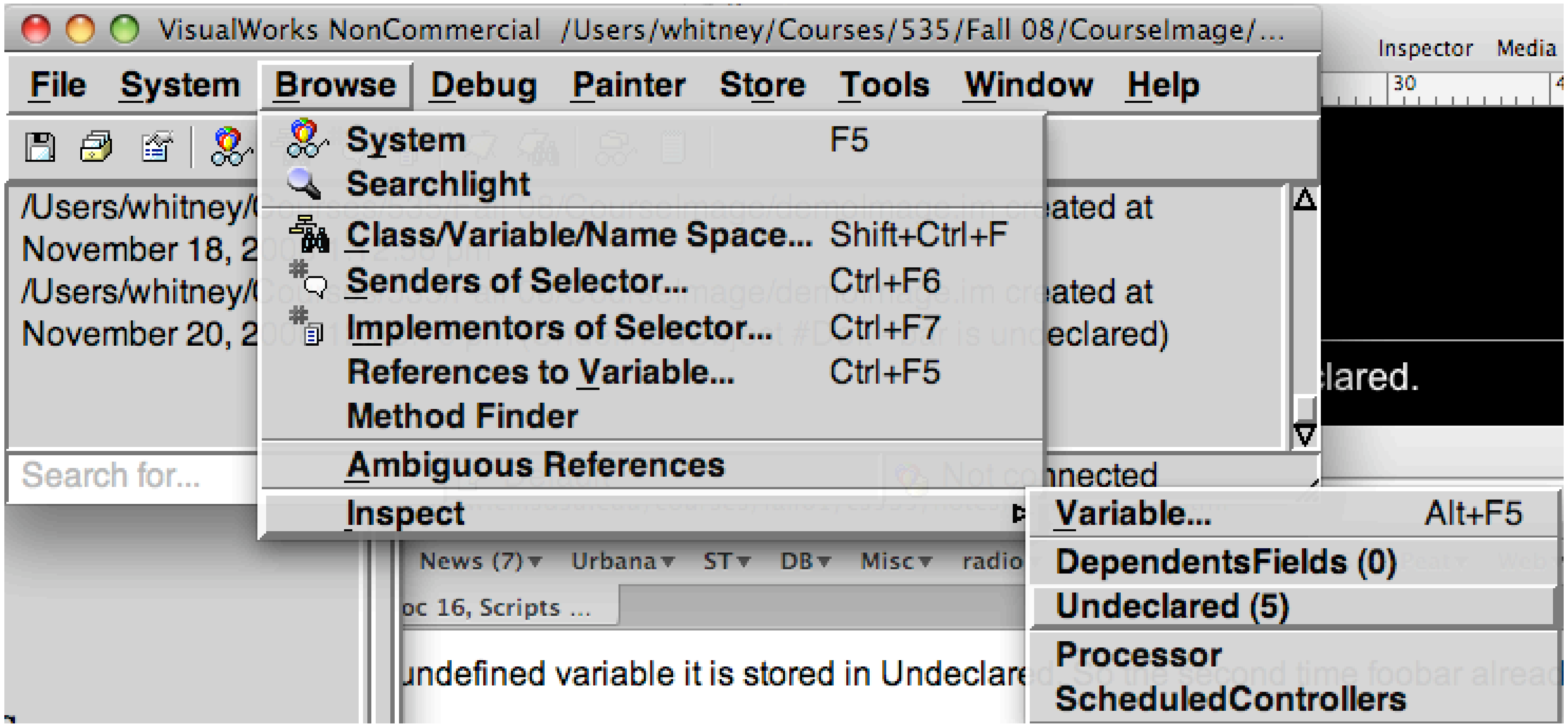
What is going on?

When running code has an undefined variable it is stored in Undeclared.

So the second time foobar already exists

- It exists in Undeclared.

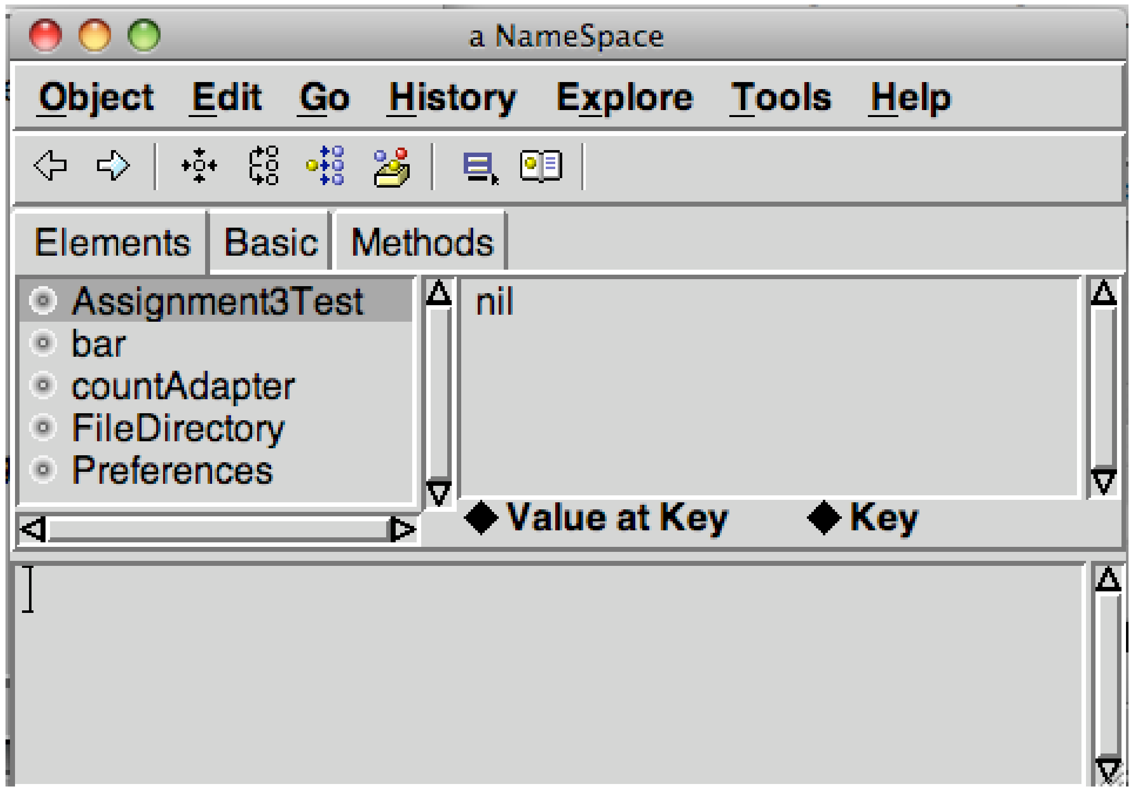
Viewing Undeclared

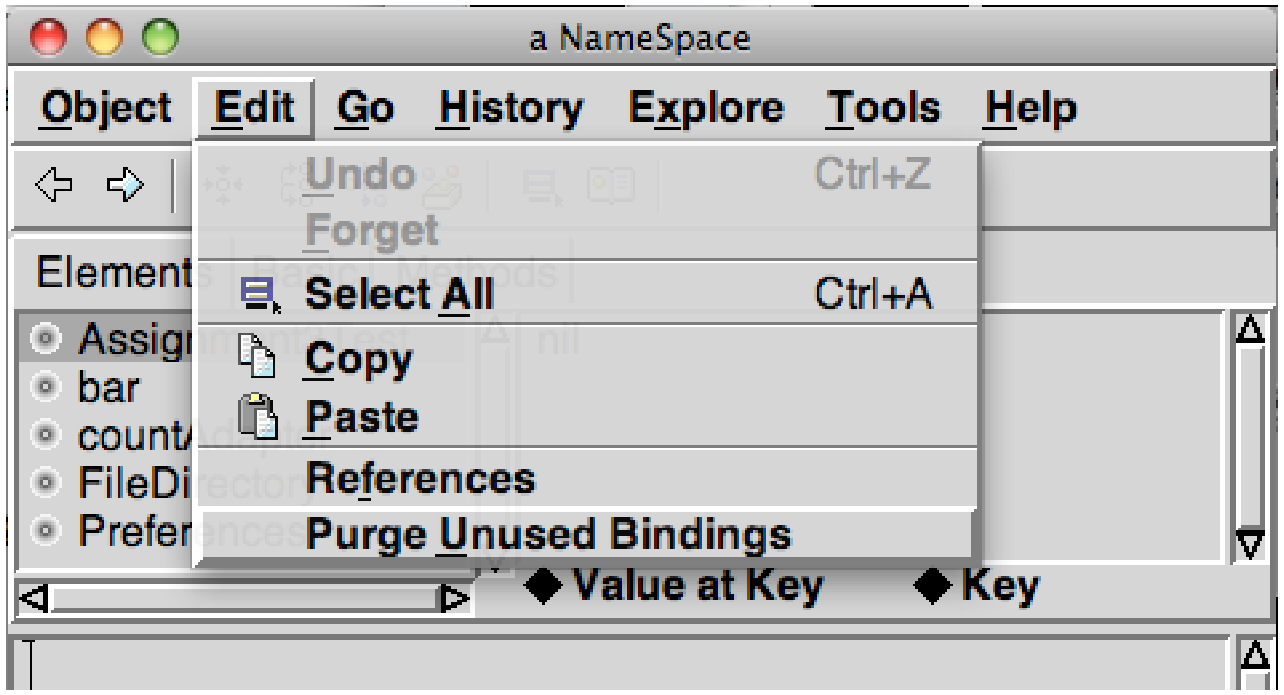


Or execute:

Undeclared inspect

Removing Undeclared Variables





Or execute:

```
Undeclared purgeUnusedBindings
```

Back to Turtle

```
Sample Program

penDown
move 5
turn 90 left
move 10
turn 90 left
move 5
turn 90 left
move 10
```

```
New Syntax

penDown
move: 5
turnLeft: 90
move: 10
turnLeft: 90
move: 5
turnLeft: 90
move: 10
```

```
| turtle |
turtle := Turtle new.
turtle
    penDown;
    move: 5;
    turnLeft: 90;
    move: 10;
    turnLeft: 90;
    move: 5;
    turnLeft: 90;
    move: 10
```

If we have control over syntax create so we can use compiler evaluate

Read the program, transform the string into complete Smalltalk code and use compiler evaluate:

Of course we could just require the user to enter the text on the right, which would make our job easier.

Domain-Specific language (DSL)

Language dedicated to a particular problem domain

Examples

UNIX shell scripts

ColdFusion Markup Language

FilterMeister

- For writing Photoshop plugins

Some Advantages

Program written in words from the domain

Domain experts can understand, validate, modify, and write programs

Self-documenting code

Enhance quality, productivity, reliability, maintainability, portability and reusability

Domain-specific languages allow validation at the domain level