Scriptable debugging, execution querying and other advanced debugging techniques

Maximilian Ignacio Willembrinck Santander

PharoDays 2022

Université de Lille

CRIStAL
Centre de Recherche en Informatique,
Signal et Automatique de Lille

Inria
INVENTEURS DU MONDE NUMÉRIQUE

RMod

Get the code here: https://github.com/maxwills/PharoDays2022

Presentation Format

A mix of hands-on, demonstration and explanations.

Follow the examples:

Code here: **https://github.com/maxwills/PharoDays2022**

Let's go!

Presentation Agenda

Context

- Introduction and what to expect — Next
- Motivation

Obtaining Execution Data

- Code insertion (+ Conditions)
- Instrumentation (+ Reflection)

Scriptable-Debuggers

- Sindarin
- Seeker

Get the code here: https://github.com/maxwills/PharoDays2022

What to expect (Spoilers)

We will explore a program execution, trying to answer a few debugging questions.

We will go from commonly-used tools and techniques to not so commonly-used:

Halts, Logging, Breakpoints, MetaLinks, MethodProxies

We will show usage of Scriptable Debuggers in Pharo.

Get the code here: https://github.com/maxwills/PharoDays2022

“The debugging activity does not take place only in the Debugger”.

Presentation Agenda

Context
- Introduction and what to expect
- Motivation

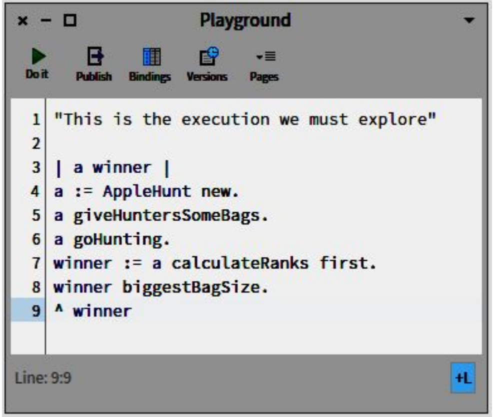
Next

Obtaining Execution Data
- Code insertion (+ Conditions)
- Instrumentation (+ Reflection)

Scriptable-Debuggers
- Sindarin
- Seeker

Motivation

We will explore a program
using several debugging tools and techniques



```
"This is the execution we must explore"

| a winner |
a := AppleHunt new.
a giveHuntersSomeBags.
a goHunting.
winner := a calculateRanks first.
winner biggestBagSize.
^ winner
```

Our program tells a story …

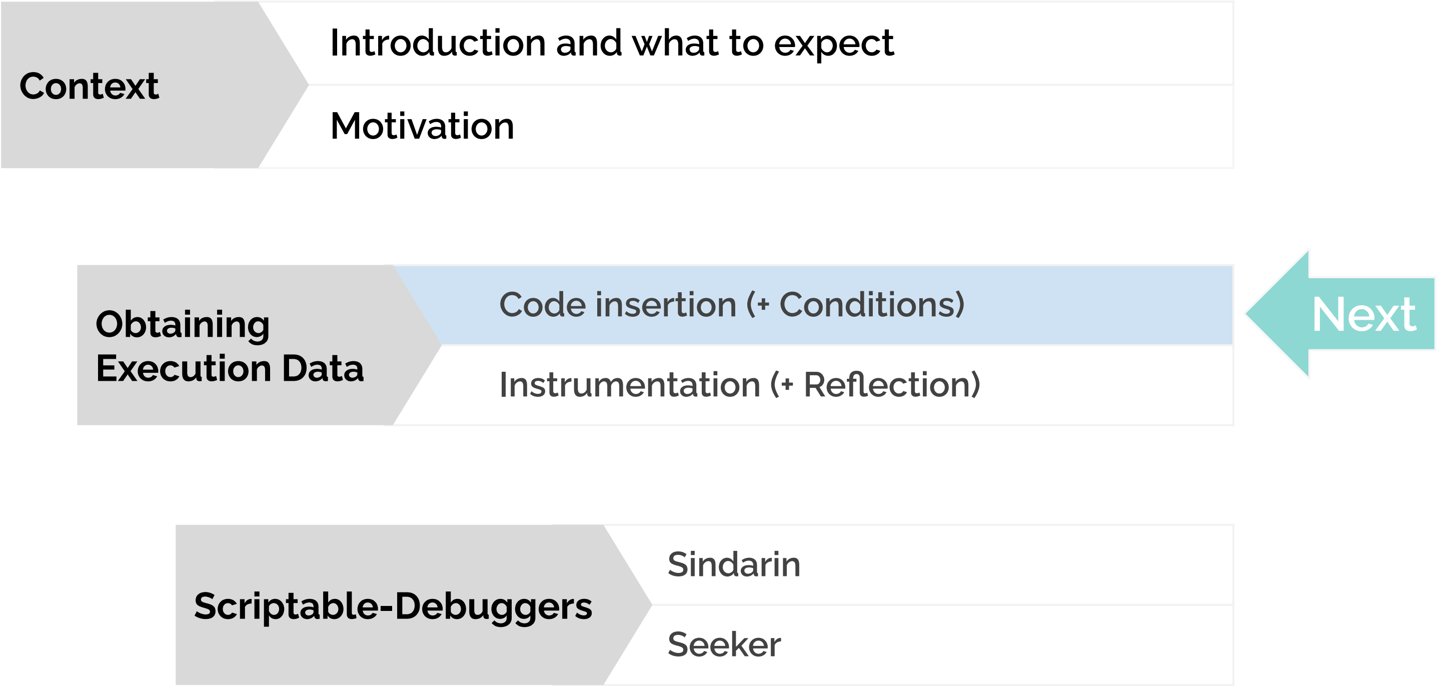
Motivation

Getting data from the execution:

Q1. How many times the method `OrderedCollection>>add:` is called? (and with an Apple as argument?)

Q2. How many times any method with selector `add:` is called? What is the actual method in every case?

Presentation Agenda



Context
- Introduction and what to expect
- Motivation

Obtaining Execution Data
- Code insertion (+ Conditions)
- Instrumentation (+ Reflection)

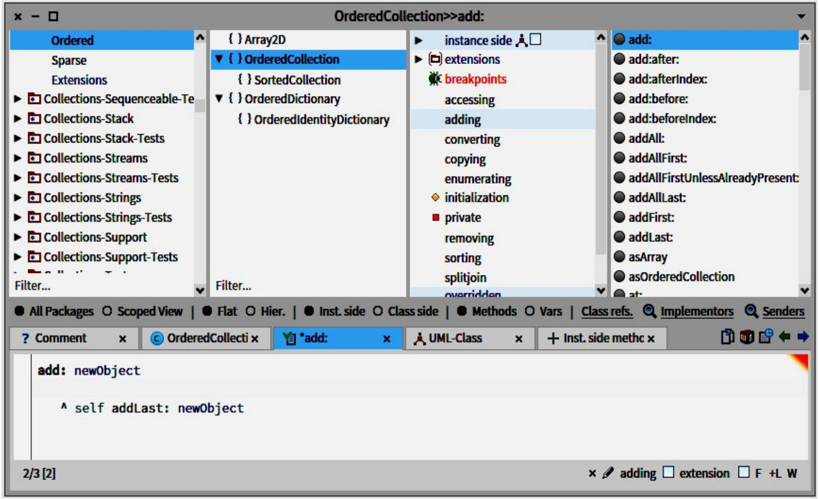
Scriptable-Debuggers
- Sindarin
- Seeker

(To the code!)

Obtaining Execution Data

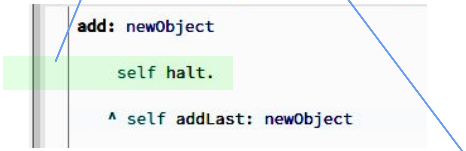
Code Insertion

We change the code of the execution to include our inquisitive actions.

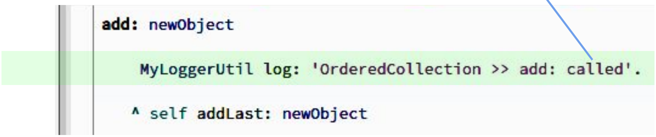


```
add: newObject

    ^ self addLast: newObject
```

Original OrderedCollection>>add: method
(comments removed)



```
add: newObject

    self halt.

    ^ self addLast: newObject
```



```
add: newObject

    MyLoggerUtil log: 'OrderedCollection >> add: called'.

    ^ self addLast: newObject
```

Note: Images for illustrative purpose only.
Don't put an Halt in OrderedCollection>>add:
:(

Obtaining Execution Data

Code Insertion

We change the code of the execution to include our inquisitive actions.

- **Logging (printing)**
- **Halt**

Getting data from the execution:

Q1. How many times the method `OrderedCollection>>add:` is called? (and with an Apple as argument?)

Obtaining Execution Data

Code Insertion

Getting data from the execution:

~~Q1. How many times the method **`OrderedCollection>>add:`** is called? (and with an Apple as argument?)~~

Q2. How many times any method with selector add: is called? What is the actual method in every case? Consider only cases when adding Apple, or Hunter objects.

Problem:

There are several possible methods with the #add: selector. What to do?

Presentation Agenda

Context

- Introduction and what to expect
- Motivation

Obtaining Execution Data

- Code insertion (+ Conditions)
- Instrumentation (+ Reflection)

Next

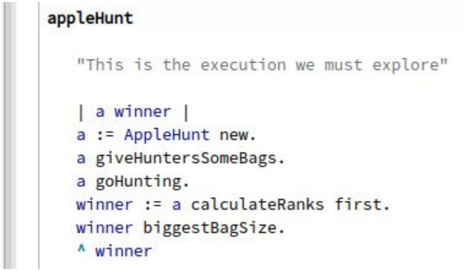
Scriptable-Debuggers

- Sindarin
- Seeker

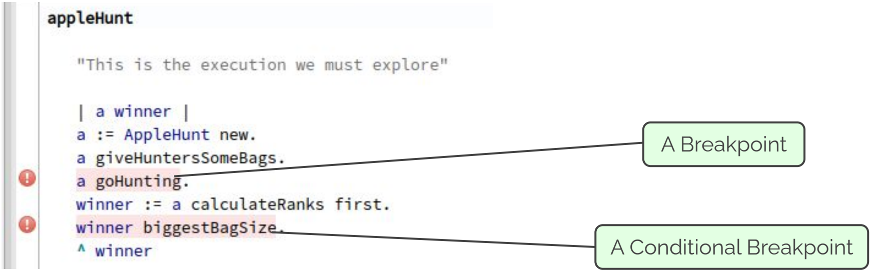
Obtaining Execution Data

Instrumentation

This time, to include our inquisitive actions, we change the execution without altering it's code.



```
appleHunt

   "This is the execution we must explore"

   | a winner |
   a := AppleHunt new.
   a giveHuntersSomeBags.
   a goHunting.
   winner := a calculateRanks first.
   winner biggestBagSize.
   ^ winner
```



```
appleHunt

   "This is the execution we must explore"

   | a winner |
   a := AppleHunt new.
   a giveHuntersSomeBags.
   a goHunting.
   winner := a calculateRanks first.
   winner biggestBagSize.
   ^ winner
```

A Breakpoint

A Conditional Breakpoint

+ MetaLinks MethodProxies

Obtaining Execution Data

Instrumentation

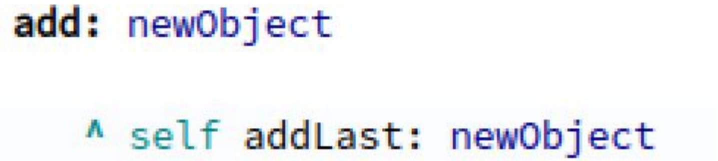
This time, to include our inquisitive actions, we change the execution without altering it's code.

- Breakpoints
- MetaLinks
- MethodProxies

Instrumentation with MetaLinks

Adds a extra instructions to our execution without modifying its code.

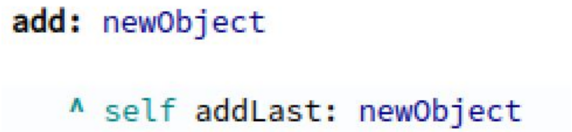
The code:



```
add: newObject

    ^ self addLast: newObject
```

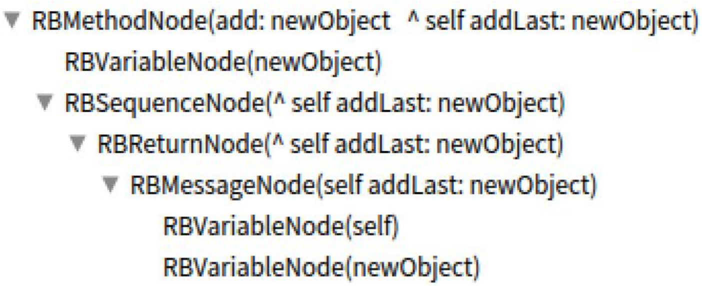
Instrumentation with MetaLinks

When a method is compiled:



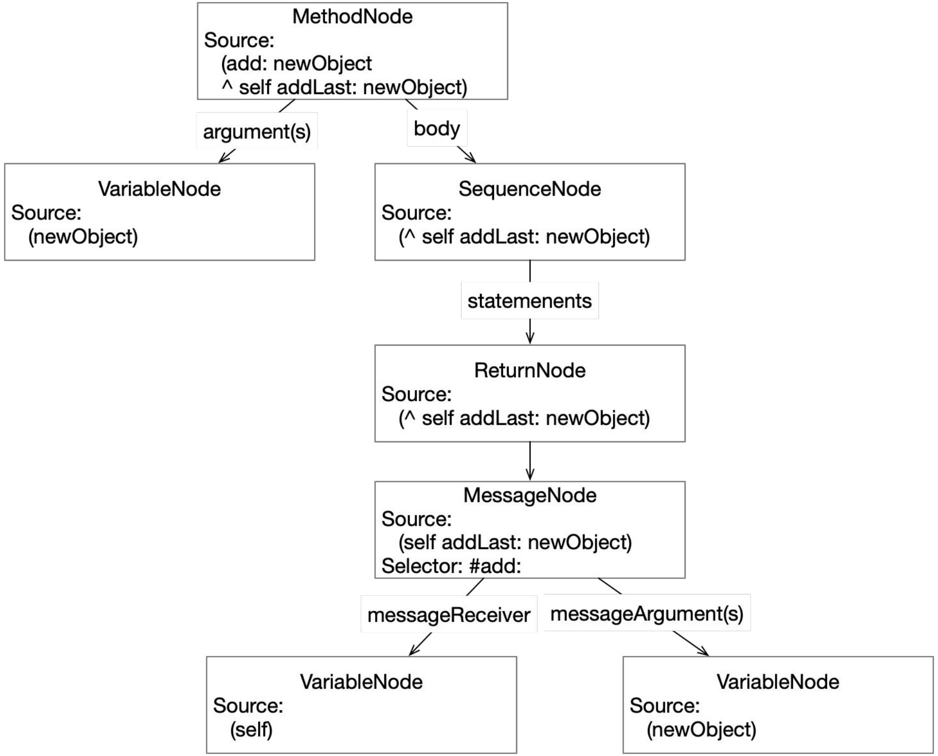
```
add: newObject

    ^ self addLast: newObject
```

An AST is produced:



```
▼ RBMethodNode(add: newObject  ^ self addLast: newObject)
    RBVariableNode(newObject)
  ▼ RBSequenceNode(^ self addLast: newObject)
    ▼ RBReturnNode(^ self addLast: newObject)
      ▼ RBMessageNode(self addLast: newObject)
          RBVariableNode(self)
          RBVariableNode(newObject)
```

Instrumentation with MetaLinks



MethodNode
Source:
(add: newObject
^ self addLast: newObject)

argument(s)

body

VariableNode
Source:
(newObject)

SequenceNode
Source:
(^ self addLast: newObject)

statemenents

ReturnNode
Source:
(^ self addLast: newObject)

MessageNode
Source:
(self addLast: newObject)
Selector: #add:

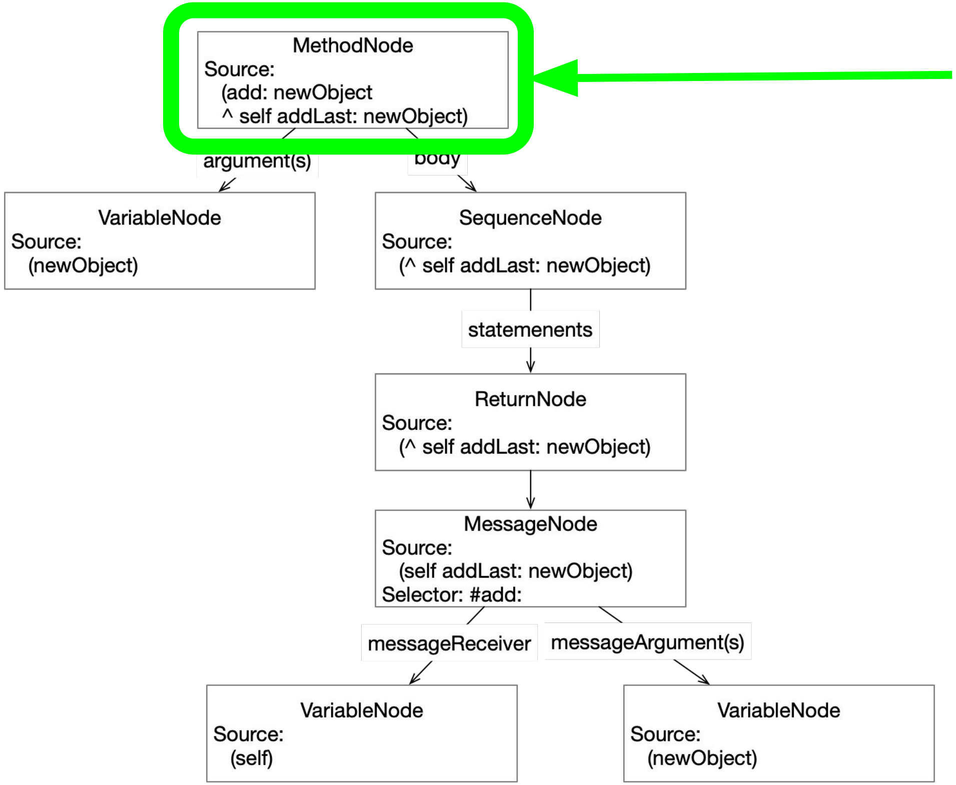
messageReceiver

messageArgument(s)

VariableNode
Source:
(self)

VariableNode
Source:
(newObject)

Instrumentation with MetaLinks



MethodNode
Source:
(add: newObject
^ self addLast: newObject)

argument(s)

body

VariableNode
Source:
(newObject)

SequenceNode
Source:
(^ self addLast: newObject)

statemenents

ReturnNode
Source:
(^ self addLast: newObject)

MessageNode
Source:
(self addLast: newObject)
Selector: #add:

messageReceiver

messageArgument(s)

VariableNode
Source:
(self)

VariableNode
Source:
(newObject)

“When **that** node is executed, execute **THIS** other code!”

```
Transcript show: ‘#add: called’
```

Showing the code in Pharo

Instrumentation through MetaLinks Examples
+ Reflection

MethodProxies

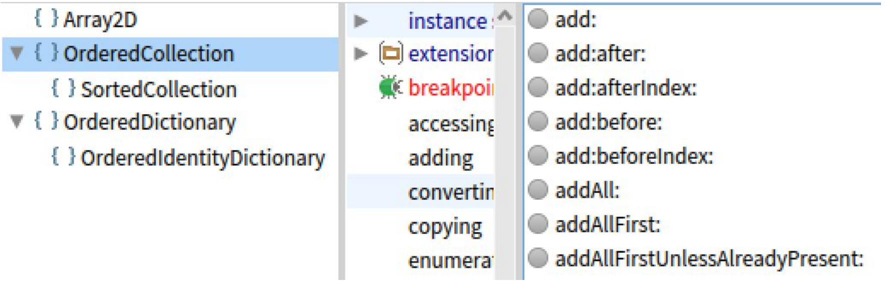
Some facts:

- Not included in Pharo.
- Get it here: https://github.com/pharo-contributions/MethodProxies

  The package is developed and maintained by S. Ducasse, G. Polito and P. Tesone, but feel free to give a hand.

Instrumentation with MethodProxies

Adds a extra instructions to our execution without modifying its code, by “proxying” its method(s).



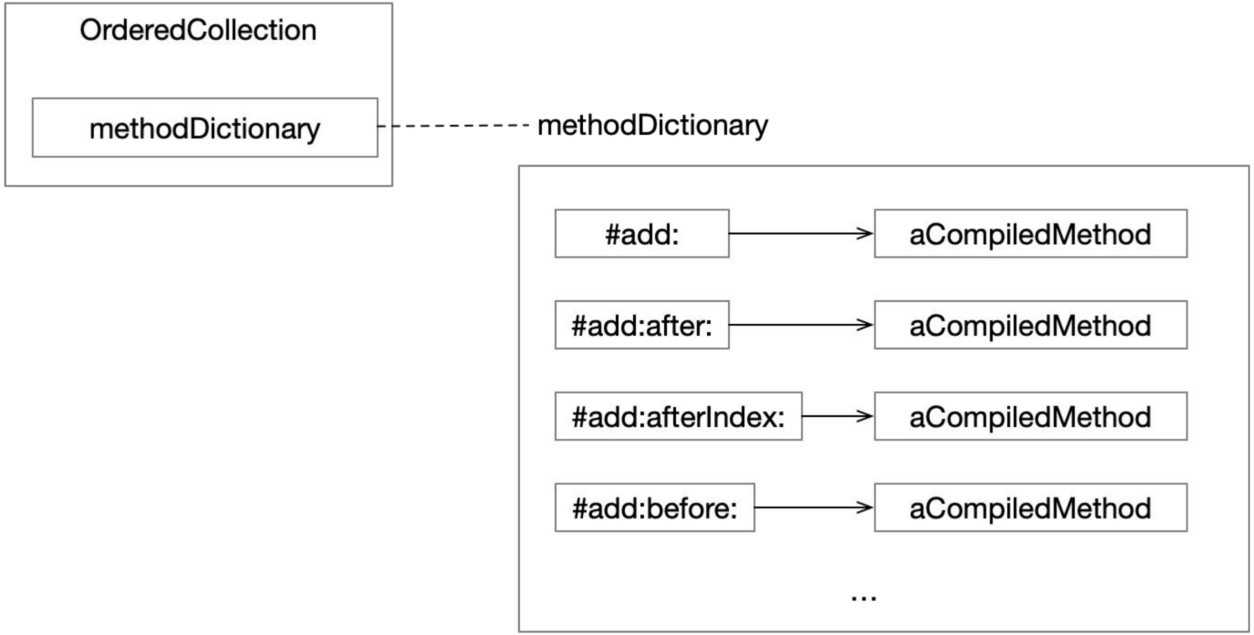
{ } Array2D
▼ { } OrderedCollection
{ } SortedCollection
▼ { } OrderedDictionary
{ } OrderedIdentityDictionary

▶ instance
▶ extensior
breakpoi
accessing
adding
convertir
copying
enumera

● add:
● add:after:
● add:afterIndex:
● add:before:
● add:beforeIndex:
● addAll:
● addAllFirst:
● addAllFirstUnlessAlreadyPresent:

Instrumentation with MethodProxies

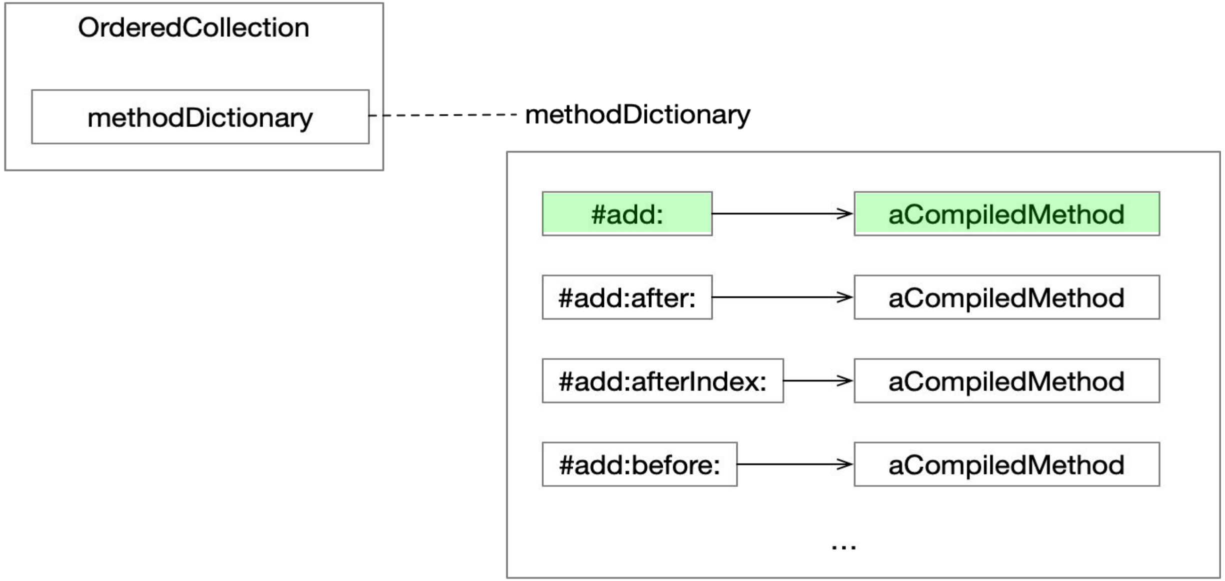
When a method is compiled, the created CompiledMethod object is stored in the methodDictionary of the class.



Instrumentation with MethodProxies

When sending a message to an object of our class, Pharo will get the CompiledMethod object of the dictionary, and will execute it.

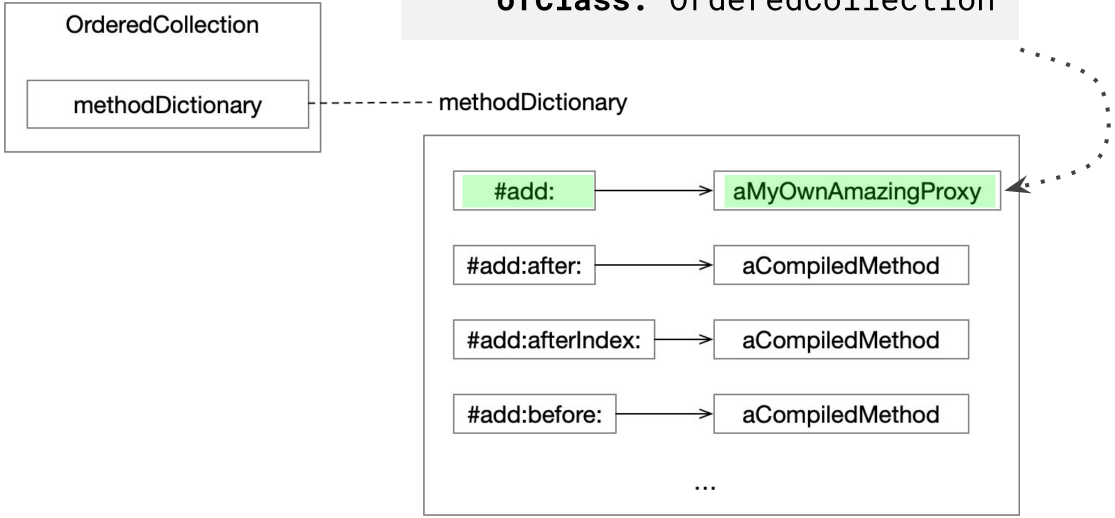
```
myCollection add: 1
```



Instrumentation with MethodProxies

To instrument our method, we can replace the CompiledMethod with a Proxy

```
MyOwnAmazingProxy
    installInMethod: #add:
    ofClass: OrderedCollection
```



(To the code!)

Obtaining Execution Data

Comparison of approaches

Code Insertion

- Halt
- Logging

Instrumentation

- Breakpoints
- MetaLinks
- MethodProxies

	Code Insertion	Instrumentation
“Simple” to understand.	🙂	🙁
“Simple” to debug.	🙂	🙁
Don’t modify the debugged program code.	🙁	🙂
Persistent (recompilation)	🙂	🙁
Good Scalability	🙁	🙂

By instrumenting a program, we alter its execution.
(Even if we don’t modify it’s code)

Can we extract execution data without altering the execution at all?

Presentation Agenda

Context

- Introduction and what to expect
- Motivation

Obtaining Execution Data

- Code insertion (+ Conditions)
- Instrumentation (+ Reflection)

Scriptable-Debuggers

- Sindarin
- Seeker

Next

Scriptable Debuggers

- Allow developers to automate debugging tasks.
- Expose an API to:
  - Manipulate the debugger and debugged execution.
  - Obtain information about the debugged execution.

Less tedious manual debugging work for the developer

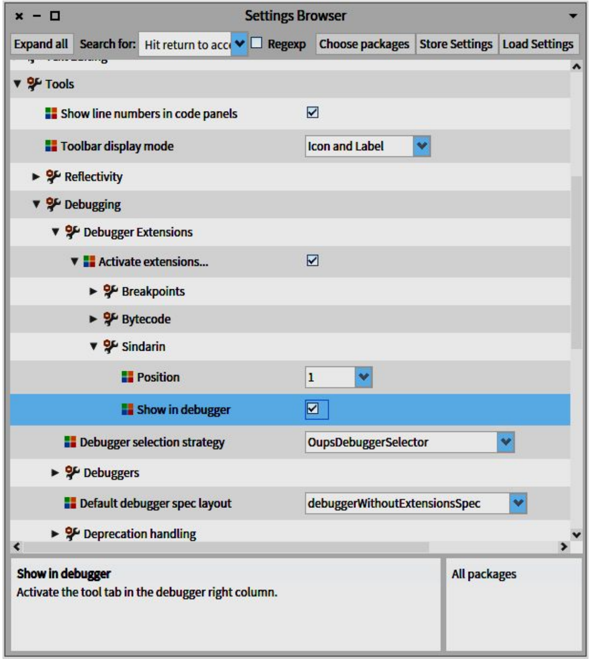
Sindarin

Included in Pharo

- Two flavors:
    - With UI: Extension of the StDebugger.
        - Has to activated from the Pharo settings.
    - Headless: The SindarinDebugger Object.
        - Already included in Pharo 9.0 and Pharo 10

Sindarin

Enabling the UI version

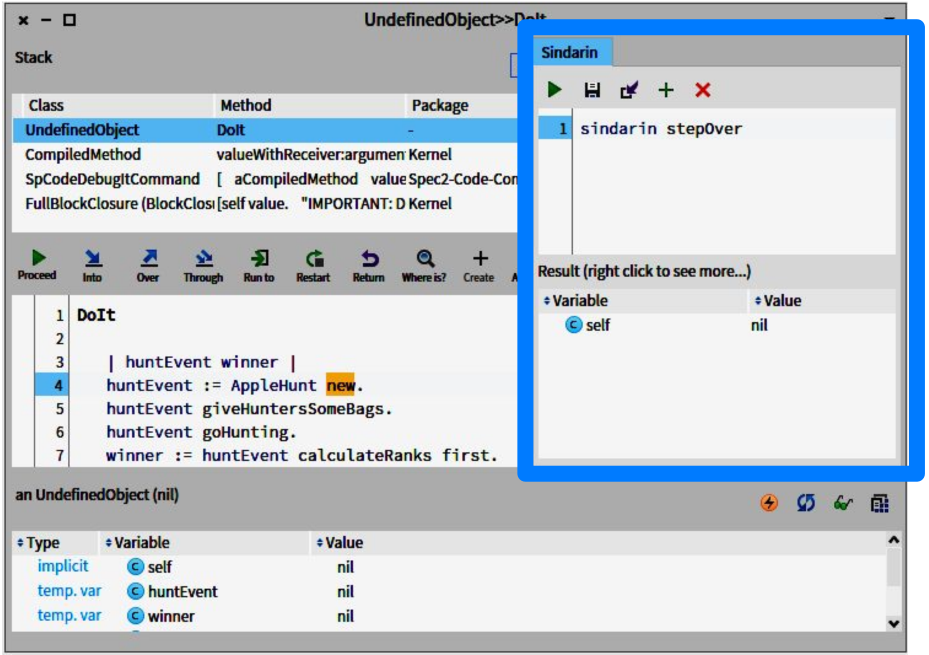


- ▼ Tools
  - Show line numbers in code panels ☑
  - Toolbar display mode: Icon and Label
  - ▶ Reflectivity
  - ▼ Debugging
    - ▼ Debugger Extensions
      - ▼ Activate extensions... ☑
        - ▶ Breakpoints
        - ▶ Bytecode
        - ▼ Sindarin
          - Position: 1
          - Show in debugger ☑
    - Debugger selection strategy: OupsDebuggerSelector
    - ▶ Debuggers
    - Default debugger spec layout: debuggerWithoutExtensionsSpec
    - ▶ Deprecation handling

Show in debugger
Activate the tool tab in the debugger right column.

All packages

Enable it in the Pharo Settings



Class	Method	Package
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:argumen	Kernel
SpCodeDebugItCommand	[aCompiledMethod value	Spec2-Code-Con
FullBlockClosure (BlockClos	[self value. "IMPORTANT: D	Kernel

```
DoIt

    | huntEvent winner |
    huntEvent := AppleHunt new.
    huntEvent giveHuntersSomeBags.
    huntEvent goHunting.
    winner := huntEvent calculateRanks first.
```

```
sindarin stepOver
```

Variable	Value
self	nil

Type	Variable	Value
implicit	self	nil
temp. var	huntEvent	nil
temp. var	winner	nil

StDebugger with Sindarin UI Extension

(To the code)

Presentation Agenda

Context

- Introduction and what to expect
- Motivation

Obtaining Execution Data

- Code insertion (+ Conditions)
- Instrumentation (+ Reflection)

Scriptable-Debuggers

- Sindarin
- Seeker

Next

Seeker

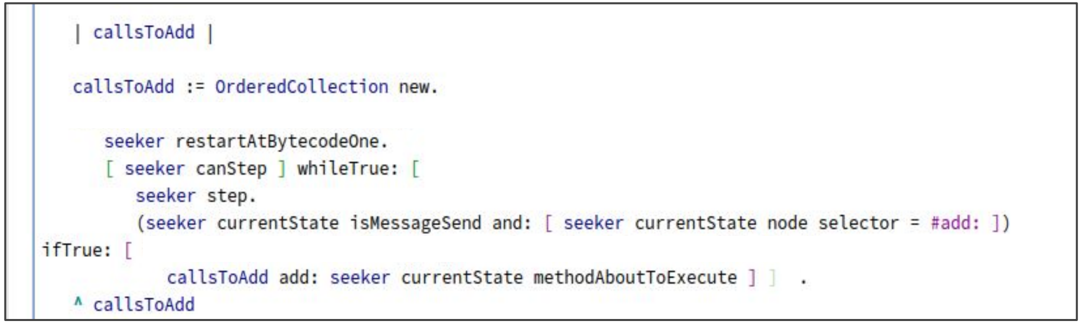
A prototype scriptable and queryable debugger

- UI version only (as an extension of the StDebugger).
- Not included in Pharo.
- Query-based debugging.
- Time-Traveling mechanics.

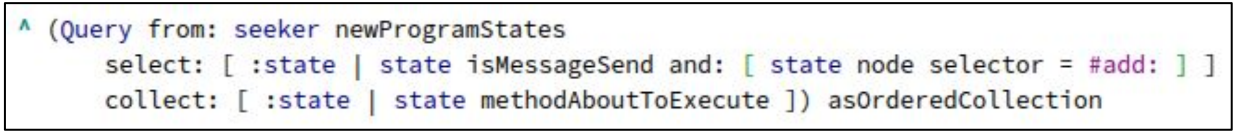
Get the code here: https://github.com/maxwills/PharoDays2022

(To the code)

The Query Notation
(From scripting to querying)



```
  | callsToAdd |

  callsToAdd := OrderedCollection new.

     seeker restartAtBytecodeOne.
     [ seeker canStep ] whileTrue: [
        seeker step.
        (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
           callsToAdd add: seeker currentState methodAboutToExecute ] ]  .
  ^ callsToAdd
```

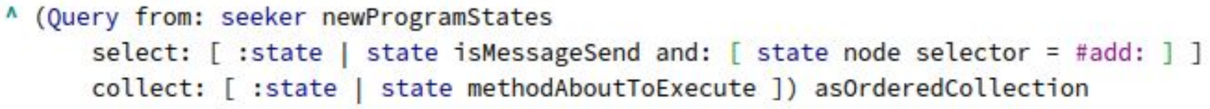


```
^ (Query from: seeker newProgramStates
      select: [ :state | state isMessageSend and: [ state node selector = #add: ] ]
      collect: [ :state | state methodAboutToExecute ]) asOrderedCollection
```

Standard Query Notation

Equivalent queries in other languages

Pharo (Prototype implementation for Pharo 9.0/Pharo10)



```
^ (Query from: seeker newProgramStates
    select: [ :state | state isMessageSend and: [ state node selector = #add: ] ]
    collect: [ :state | state methodAboutToExecute ]) asOrderedCollection
```

SQL Query

```
SELECT state.methodAboutToExecute
FROM ProgramStates
WHERE state.isMessageSend AND
      state.node.selector = 'add:'
```

C# (Linq)

```
var results=(
  FROM state in seeker.newProgramStates
  WHERE state.isMessageSend && state.node.selector == "add:"
  SELECT state.methodAboutToExecute
).ToList();
```

Python (List comprehension + properties)

```
results = [
   state.methodAboutToExecute
   for state in seeker.newProgramStates
   if state.isMessageSend and state.node.selector == "add:"
]
```

(To the code!)

Queries In Seeker

- Not any kind of queries, but Time-Traveling Queries (TTQs).

- A set of ready-to-use TTQs are provided.

- Developers can write their own Queries and TTQs.

Summary

- The debugger is not your only tool for debugging.
- Inserting extra behavior to study your program execution for your debugging sessions.
  - Code Insertion, Instrumentation, and dangers.
- Reflection is a powerful mechanism to obtain execution data.
- Scriptable debuggers: Sindarin.
- Scriptable Time-Traveling Queryable Debugger prototype: Seeker.
  - Query Notation and Time-Traveling Queries

Have a good day!

Get the code here: https://github.com/maxwills/PharoDays2022

Presentation is finished.

Extra slides next.

Context

What to expect

We will explore an execution, to find answer to debugging questions.

Everything goes!

> ***The standard debugger, breakpoints, logging, proxies, reflection, scripting, speculating, etc.***

I will break some Pharo Images, and show how you can avoid that (*).

*(*In some cases. There are no guarantees, so don't sue me)*

Context

What to expect

By the end of the session you will:

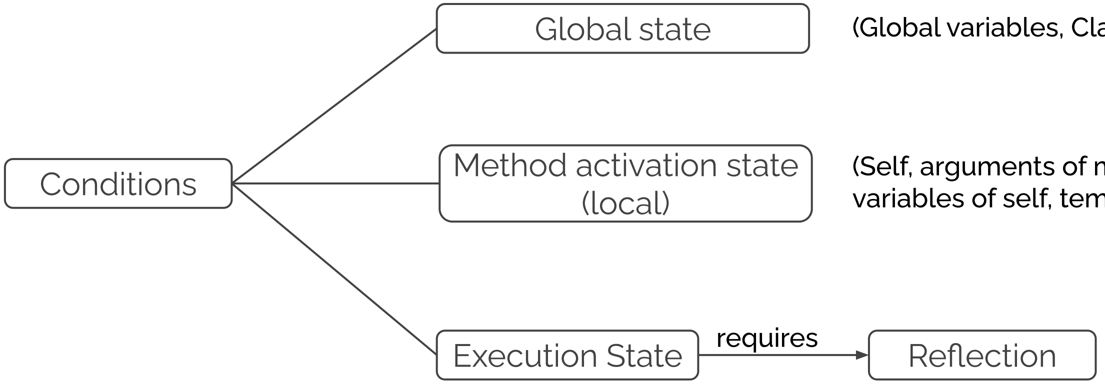
- Be aware of currently available yet not-so-commonly used tools in Pharo.
- Have the knowledge on what is to come in some aspects of debugging.

And (hopefully)

- Your debugging techniques repertory is expanded.
- You had a blast!

Get the code here: https://github.com/maxwills/PharoDays2022

More on Conditions ...



Conditions

Global state

(Global variables, Classes, SharedVariables, etc)

Method activation state (local)

(Self, arguments of method, instance variables of self, temporal variables, etc)

Execution State

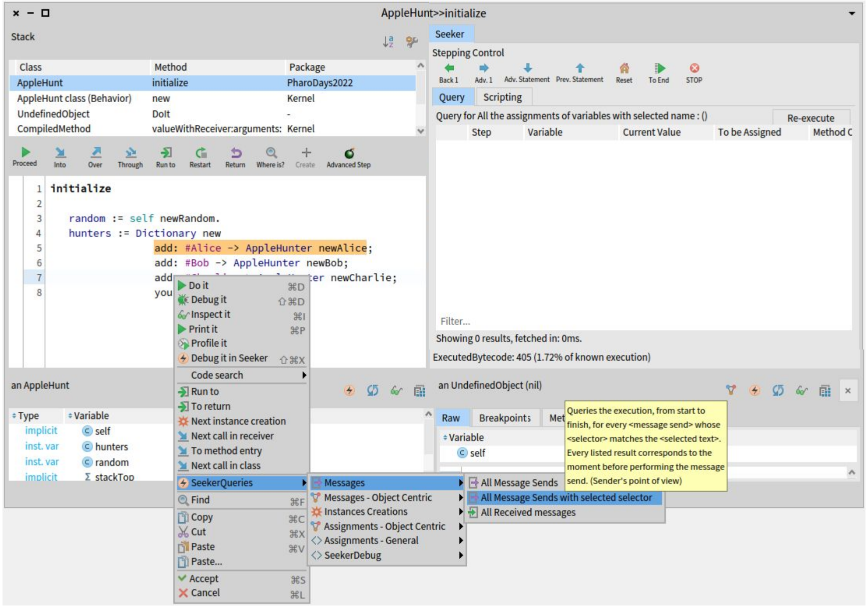
requires

Reflection

(thisContext, “execution stack”, activeProcess, etc)

Querying the execution

What is happening?



Class	Method	Package
AppleHunt	initialize	PharoDays2022
AppleHunt class (Behavior)	new	Kernel
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel

```
initialize

    random := self newRandom.
    hunters := Dictionary new
                   add: #Alice -> AppleHunter newAlice;
                   add: #Bob -> AppleHunter newBob;
                   add                       er newCharlie;
                   you
```

Query for All the assignments of variables with selected name : ()

Step	Variable	Current Value	To be Assigned	Method C

Filter...

Showing 0 results, fetched in: 0ms.

ExecutedBytecode: 405 (1.72% of known execution)

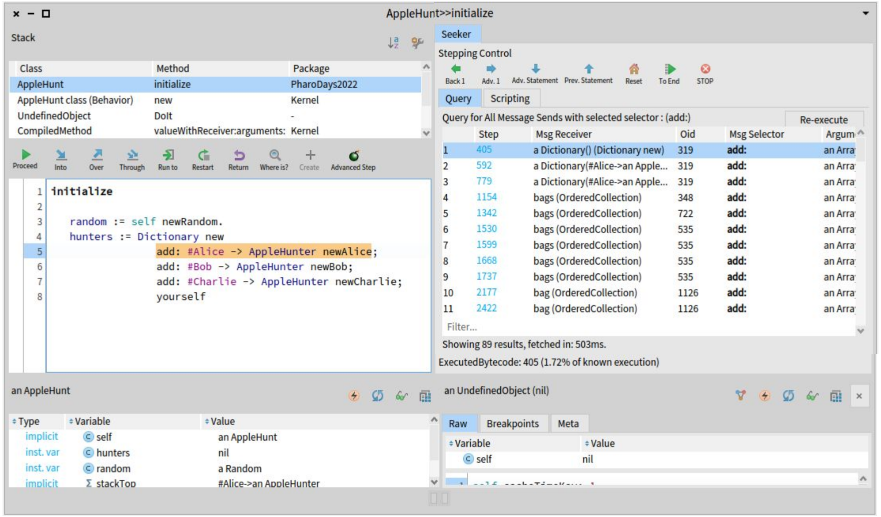
an AppleHunt

Type	Variable
implicit	self
inst. var	hunters
inst. var	random
implicit	stackTop

an UndefinedObject (nil)

Variable
self

Queries the execution, from start to finish, for every <message send> whose <selector> matches the <selected text>. Every listed result corresponds to the moment before performing the message send. (Sender's point of view)



Class	Method	Package
AppleHunt	initialize	PharoDays2022
AppleHunt class (Behavior)	new	Kernel
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel

```
initialize

    random := self newRandom.
    hunters := Dictionary new
                   add: #Alice -> AppleHunter newAlice;
                   add: #Bob -> AppleHunter newBob;
                   add: #Charlie -> AppleHunter newCharlie;
                   yourself
```

Query for All Message Sends with selected selector : (add:)

	Step	Msg Receiver	Oid	Msg Selector	Argum
1	405	a Dictionary() (Dictionary new)	319	add:	an Arra
2	592	a Dictionary(#Alice->an Apple...	319	add:	an Arra
3	779	a Dictionary(#Alice->an Apple...	319	add:	an Arra
4	1154	bags (OrderedCollection)	348	add:	an Arra
5	1342	bags (OrderedCollection)	722	add:	an Arra
6	1530	bags (OrderedCollection)	535	add:	an Arra
7	1599	bags (OrderedCollection)	535	add:	an Arra
8	1668	bags (OrderedCollection)	535	add:	an Arra
9	1737	bags (OrderedCollection)	535	add:	an Arra
10	2177	bag (OrderedCollection)	1126	add:	an Arra
11	2422	bag (OrderedCollection)	1126	add:	an Arra

Filter...

Showing 89 results, fetched in: 503ms.

ExecutedBytecode: 405 (1.72% of known execution)

an AppleHunt

Type	Variable	Value
implicit	self	an AppleHunt
inst. var	hunters	nil
inst. var	random	a Random
implicit	stackTop	#Alice->an AppleHunter

an UndefinedObject (nil)

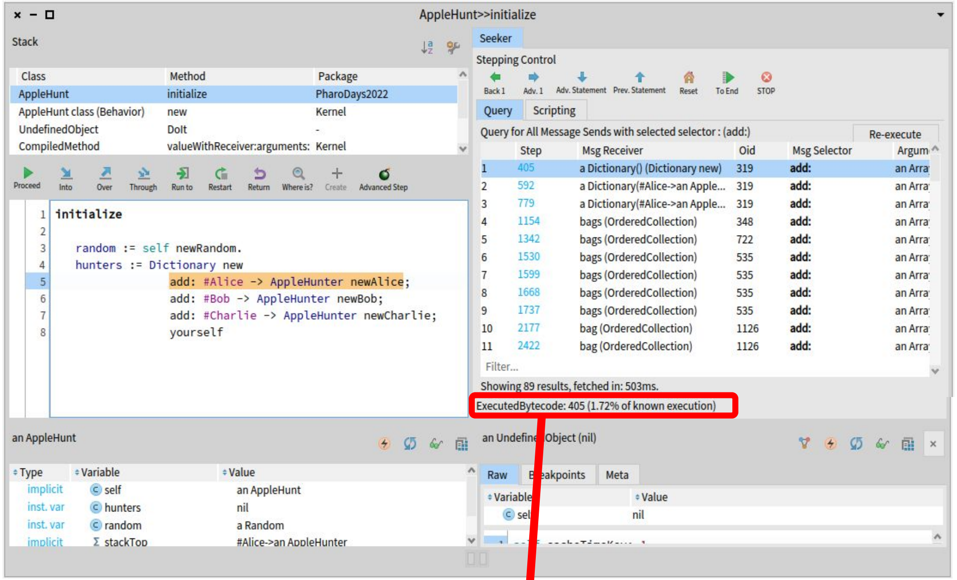
Variable	Value
self	nil

Querying during a debugging session

Results are display

Querying the execution

What is happening?



Class	Method	Package
AppleHunt	initialize	PharoDays2022
AppleHunt class (Behavior)	new	Kernel
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel

```
initialize

   random := self newRandom.
   hunters := Dictionary new
                  add: #Alice -> AppleHunter newAlice;
                  add: #Bob -> AppleHunter newBob;
                  add: #Charlie -> AppleHunter newCharlie;
                  yourself
```

Query for All Message Sends with selected selector : (add:)

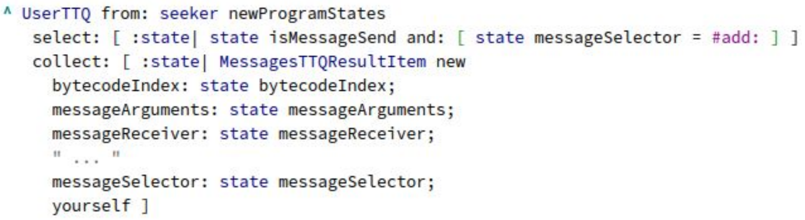
	Step	Msg Receiver	Oid	Msg Selector	Argum
1	405	a Dictionary() (Dictionary new)	319	add:	an Arra
2	592	a Dictionary(#Alice->an Apple...	319	add:	an Arra
3	779	a Dictionary(#Alice->an Apple...	319	add:	an Arra
4	1154	bags (OrderedCollection)	348	add:	an Arra
5	1342	bags (OrderedCollection)	722	add:	an Arra
6	1530	bags (OrderedCollection)	535	add:	an Arra
7	1599	bags (OrderedCollection)	535	add:	an Arra
8	1668	bags (OrderedCollection)	535	add:	an Arra
9	1737	bags (OrderedCollection)	535	add:	an Arra
10	2177	bag (OrderedCollection)	1126	add:	an Arra
11	2422	bag (OrderedCollection)	1126	add:	an Arra

Showing 89 results, fetched in: 503ms.

ExecutedBytecode: 405 (1.72% of known execution)

Type	Variable	Value
implicit	self	an AppleHunt
inst. var	hunters	nil
inst. var	random	a Random
implicit	Σ stackTop	#Alice->an AppleHunter

Variable	Value
sel	nil



```
^ UserTTQ from: seeker newProgramStates
  select: [ :state| state isMessageSend and: [ state messageSelector = #add: ] ]
  collect: [ :state| MessagesTTQResultItem new
    bytecodeIndex: state bytecodeIndex;
    messageArguments: state messageArguments;
    messageReceiver: state messageReceiver;
    " ... "
    messageSelector: state messageSelector;
    yourself ]
```

The developer is currently observing the [execution state 405]

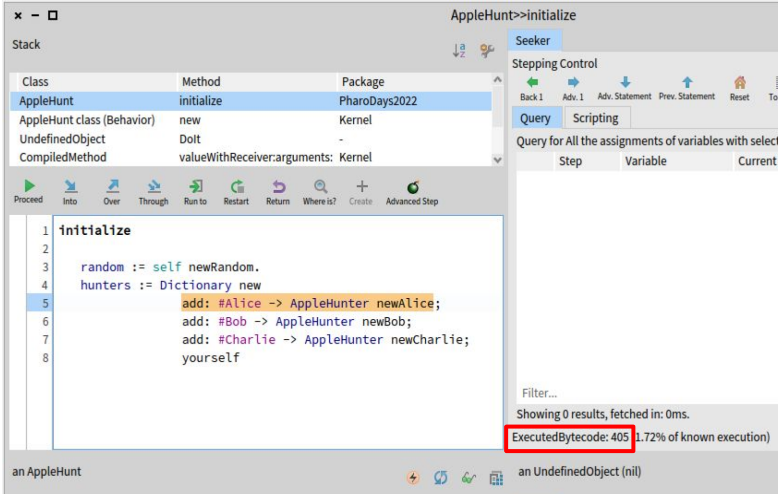
The TTQ to be executed looks like this

Querying the execution

What is happening?

On query activation:

1. The debugger traversing mechanism goes back to [execution state 1].



AppleHunt>>initialize

Class	Method	Package
AppleHunt	initialize	PharoDays2022
AppleHunt class (Behavior)	new	Kernel
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel

```
initialize

   random := self newRandom.
   hunters := Dictionary new
                  add: #Alice -> AppleHunter newAlice;
                  add: #Bob -> AppleHunter newBob;
                  add: #Charlie -> AppleHunter newCharlie;
                  yourself
```

Query for All the assignments of variables with select

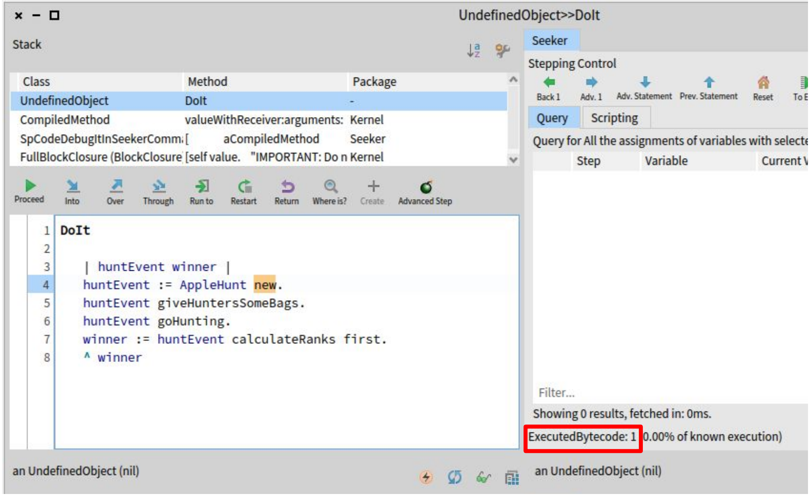
Step	Variable	Current

Showing 0 results, fetched in: 0ms.

ExecutedBytecode: 405 1.72% of known execution)

an AppleHunt

an UndefinedObject (nil)



UndefinedObject>>DoIt

Class	Method	Package
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItInSeekerComm:	[aCompiledMethod	Seeker
FullBlockClosure (BlockClosure	[self value. "IMPORTANT: Do n	Kernel

```
DoIt

   | huntEvent winner |
   huntEvent := AppleHunt new.
   huntEvent giveHuntersSomeBags.
   huntEvent goHunting.
   winner := huntEvent calculateRanks first.
   ^ winner
```

Query for All the assignments of variables with selecte

Step	Variable	Current V

Showing 0 results, fetched in: 0ms.

ExecutedBytecode: 1 0.00% of known execution)

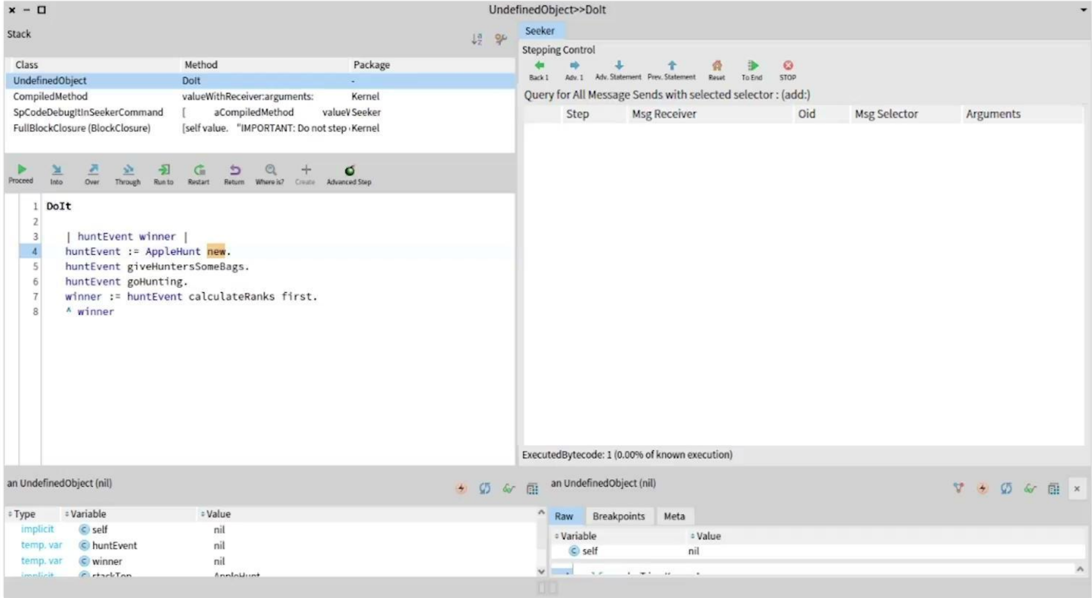
an UndefinedObject (nil)

an UndefinedObject (nil)

Querying the execution

What is happening?

2. The debugger traversing logic is executed, while selecting and collecting relevant data, until the end of the execution.



Class	Method	Package
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItInSeekerCommand	[aCompiledMethod valueV	Seeker
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step	Kernel

```
DoIt

   | huntEvent winner |
   huntEvent := AppleHunt new.
   huntEvent giveHuntersSomeBags.
   huntEvent goHunting.
   winner := huntEvent calculateRanks first.
   ^ winner
```

Query for All Message Sends with selected selector : (add:)

	Step	Msg Receiver	Oid	Msg Selector	Arguments

ExecutedBytecode: 1 (0.00% of known execution)

an UndefinedObject (nil)

Type	Variable	Value
implicit	self	nil
temp. var	huntEvent	nil
temp. var	winner	nil
implicit	stackTop	AppleHunt

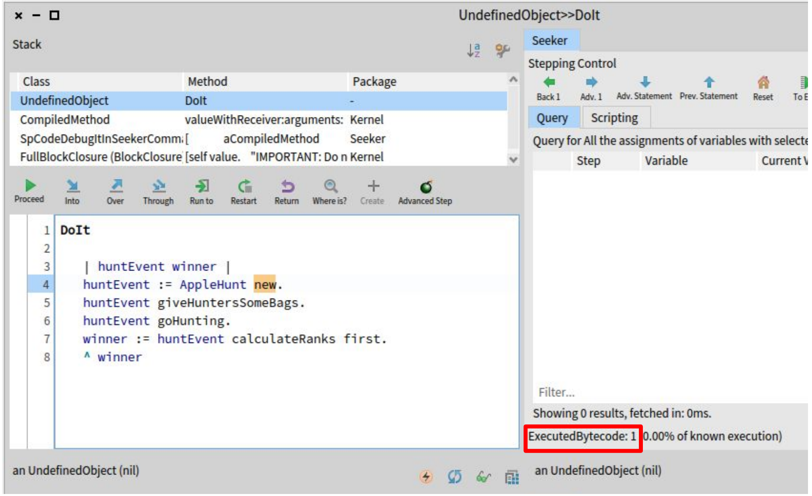
an UndefinedObject (nil)

Variable	Value
self	nil

Querying the execution

What is happening?

3. The debugger goes back to the state the developer was observing (Execution state 405).



UndefinedObject>>DoIt

Stack

Class	Method	Package
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItInSeekerComm	[aCompiledMethod	Seeker
FullBlockClosure (BlockClosure	[self value. "IMPORTANT: Do n	Kernel

```
DoIt

    | huntEvent winner |
    huntEvent := AppleHunt new.
    huntEvent giveHuntersSomeBags.
    huntEvent goHunting.
    winner := huntEvent calculateRanks first.
    ^ winner
```

an UndefinedObject (nil)

Seeker

Stepping Control

Query for All the assignments of variables with selecte

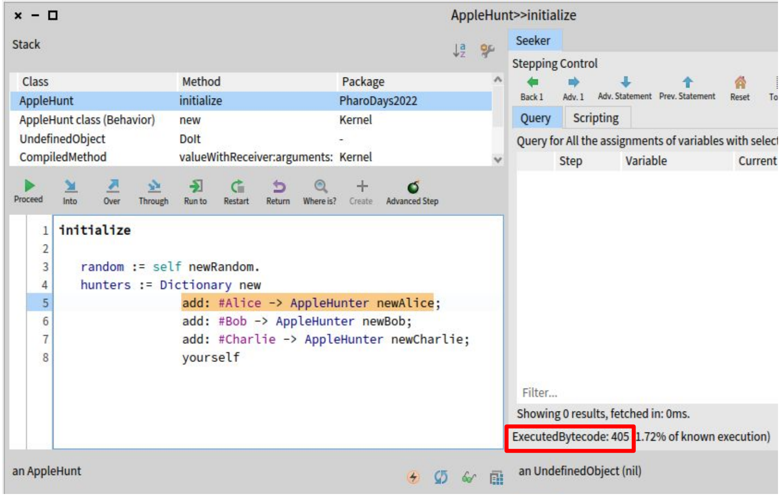
Step	Variable	Current V

Filter...

Showing 0 results, fetched in: 0ms.

ExecutedBytecode: 1 0.00% of known execution)

an UndefinedObject (nil)



AppleHunt>>initialize

Stack

Class	Method	Package
AppleHunt	initialize	PharoDays2022
AppleHunt class (Behavior)	new	Kernel
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel

```
initialize

    random := self newRandom.
    hunters := Dictionary new
                  add: #Alice -> AppleHunter newAlice;
                  add: #Bob -> AppleHunter newBob;
                  add: #Charlie -> AppleHunter newCharlie;
                  yourself
```

an AppleHunt

Seeker

Stepping Control

Query for All the assignments of variables with select

Step	Variable	Current

Filter...

Showing 0 results, fetched in: 0ms.

ExecutedBytecode: 405 1.72% of known execution)

an UndefinedObject (nil)

Querying the execution

What is happening?

(Remember: the developer was observing [execution state 405])

On query activation:

1. The debugger traversing mechanism goes back to [execution state 1].

2. The debugger traversing logic is executed, while selecting and collecting relevant data, until the end of the execution.

3. The debugger goes back to the state the developer was observing.

   All this, happens “*behind doors”.*

   The developer doesn’t see any stepping.

Installing the code used in the presentation

The code is here:
https://github.com/maxwills/PharoDays2022

In a Pharo10 image, run the following code:

Baseline in the repository.

But first

An anecdote…

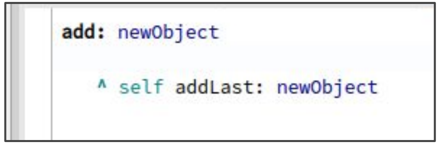
(To the code)

Get the code here: https://github.com/maxwills/PharoDays2022

Obtaining Execution Data

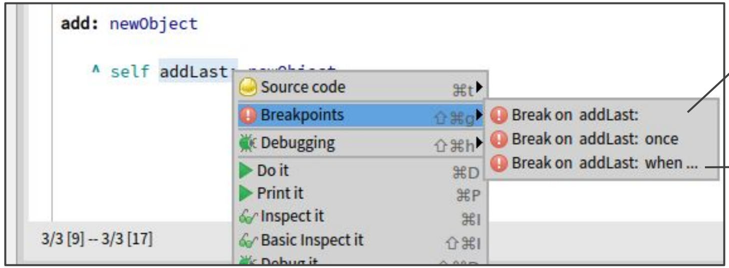
Instrumentation

Example: the previous “solutions” but with breakpoints.



```
add: newObject

    ^ self addLast: newObject
```

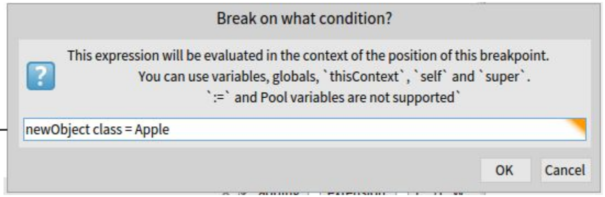
Original code



```
add: newObject

   ^ self addLast:
```

Equivalent to a halt. (Don’t put a breakpoint there)

Conditional



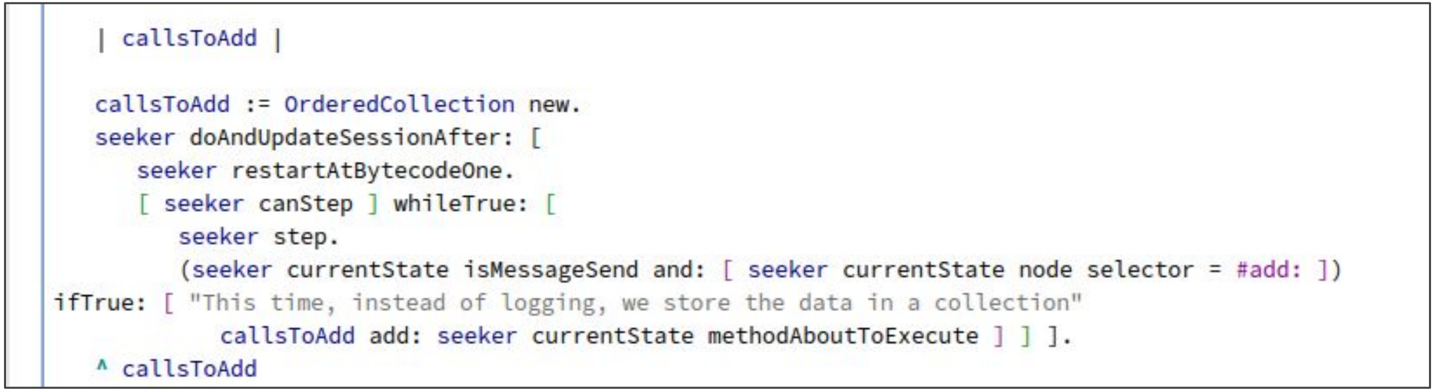
Break on what condition?

This expression will be evaluated in the context of the position of this breakpoint.
You can use variables, globals, `thisContext`, `self` and `super`.
`:=` and Pool variables are not supported`

newObject class = Apple

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

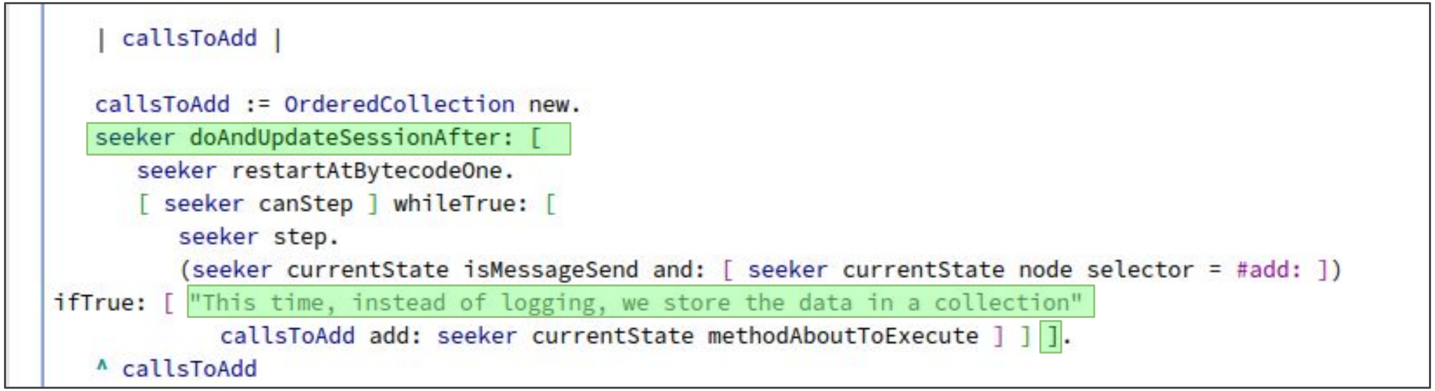


```
   | callsToAdd |

   callsToAdd := OrderedCollection new.
   seeker doAndUpdateSessionAfter: [
      seeker restartAtBytecodeOne.
      [ seeker canStep ] whileTrue: [
         seeker step.
         (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [ "This time, instead of logging, we store the data in a collection"
            callsToAdd add: seeker currentState methodAboutToExecute ] ] ].
   ^ callsToAdd
```

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

Cleaning up the code



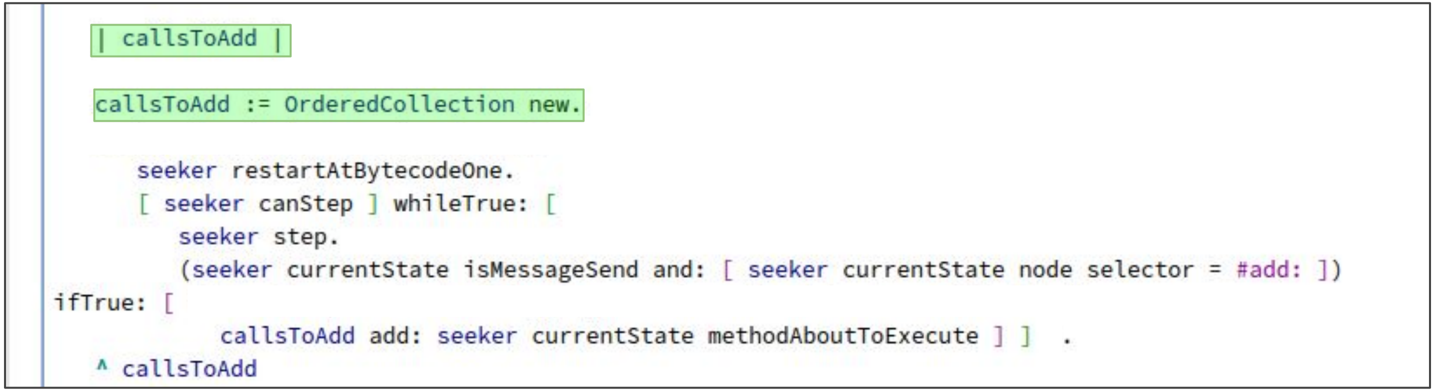
```
  | callsToAdd |

  callsToAdd := OrderedCollection new.
  seeker doAndUpdateSessionAfter: [
    seeker restartAtBytecodeOne.
    [ seeker canStep ] whileTrue: [
      seeker step.
      (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [ "This time, instead of logging, we store the data in a collection"
        callsToAdd add: seeker currentState methodAboutToExecute ] ] ].
  ^ callsToAdd
```

We will mask irrelevant code and comments

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

Prepare the storage of the collected results.

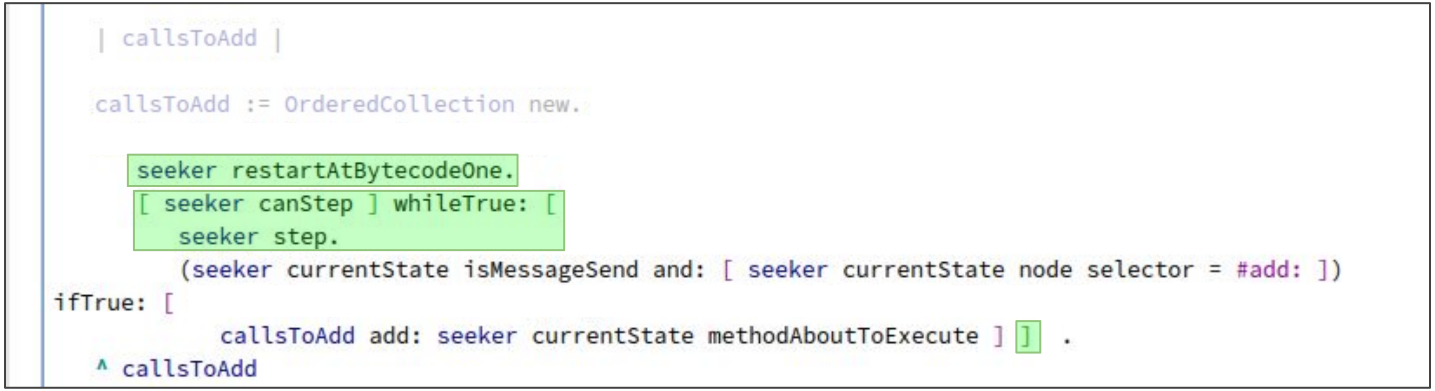


```
   | callsToAdd |

   callsToAdd := OrderedCollection new.

      seeker restartAtBytecodeOne.
      [ seeker canStep ] whileTrue: [
         seeker step.
         (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
            callsToAdd add: seeker currentState methodAboutToExecute ] ]  .
   ^ callsToAdd
```

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

“Execution Traversing” Logic.



```
  | callsToAdd |

  callsToAdd := OrderedCollection new.

    seeker restartAtBytecodeOne.
    [ seeker canStep ] whileTrue: [
      seeker step.
       (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
           callsToAdd add: seeker currentState methodAboutToExecute ] ]  .
  ^ callsToAdd
```

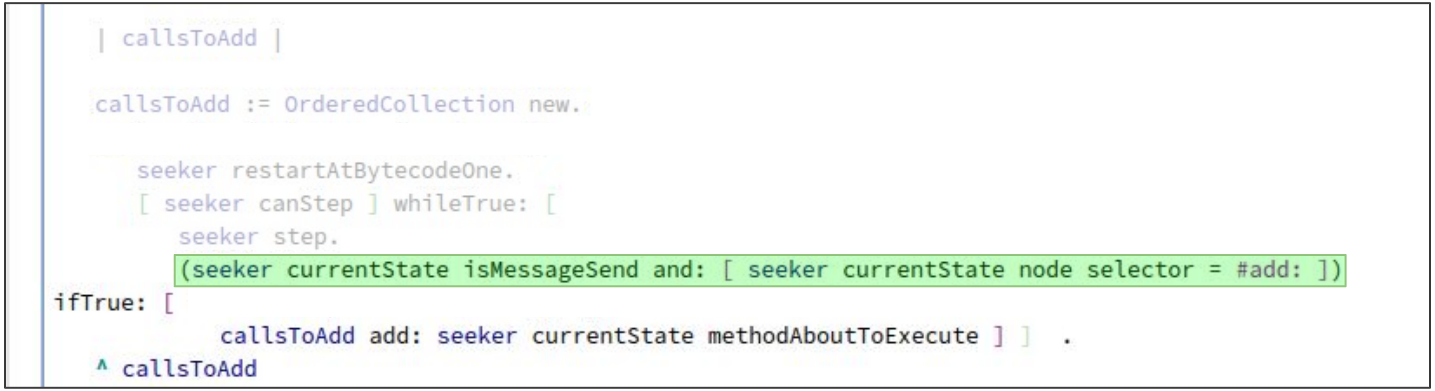
“Traversing the execution” logic:

1. Go to the beginning of the execution (restart).
2. Stepping the execution in a loop, until it finishes.

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

“Selecting the interesting states” Logic.



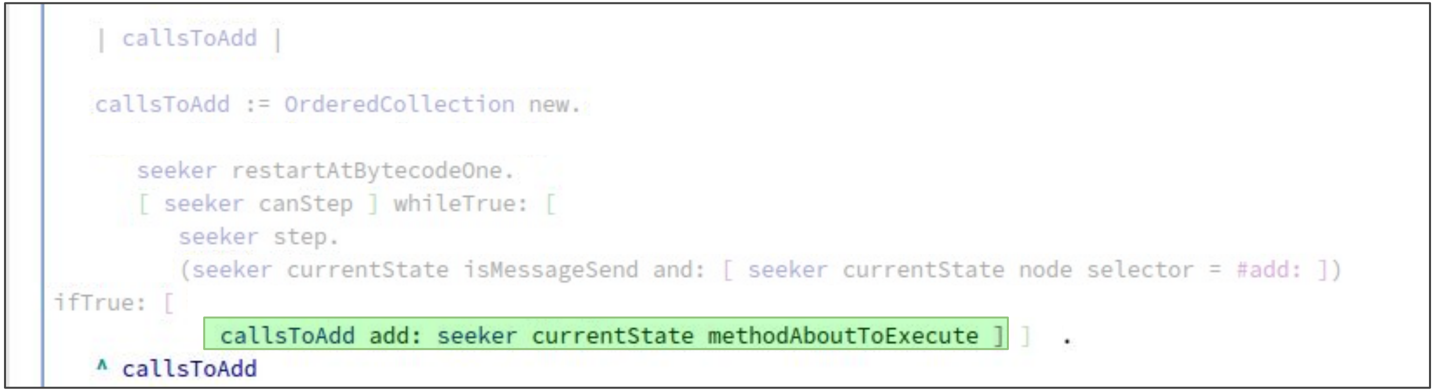
```
   | callsToAdd |

   callsToAdd := OrderedCollection new.

      seeker restartAtBytecodeOne.
      [ seeker canStep ] whileTrue: [
         seeker step.
         (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
            callsToAdd add: seeker currentState methodAboutToExecute ] ]  .
   ^ callsToAdd
```

The “selection condition” code evaluates to true or false on each execution state.

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

Addition of some execution data into the result.
The “Collecting” Logic.

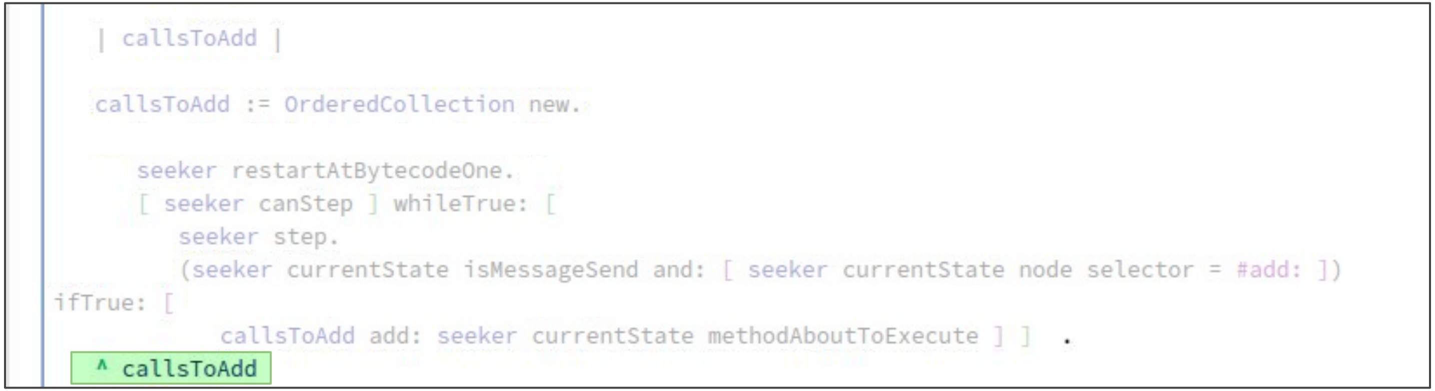


```
    | callsToAdd |

    callsToAdd := OrderedCollection new.

        seeker restartAtBytecodeOne.
        [ seeker canStep ] whileTrue: [
            seeker step.
            (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
                callsToAdd add: seeker currentState methodAboutToExecute ] ]   .
    ^ callsToAdd
```

Towards a Queryable Debugger

Dissecting The Collection of Execution Data

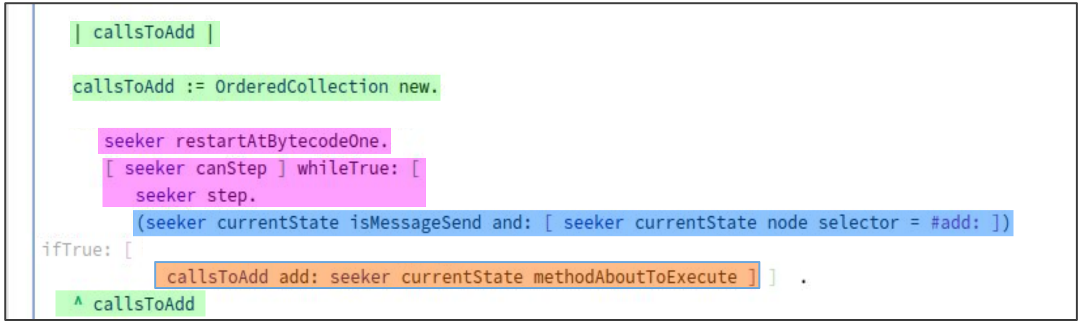
Return the collected results.



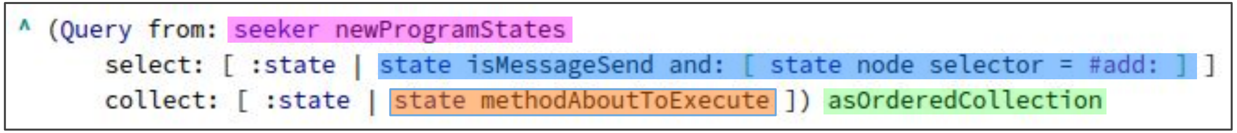
```
  | callsToAdd |

  callsToAdd := OrderedCollection new.

      seeker restartAtBytecodeOne.
      [ seeker canStep ] whileTrue: [
          seeker step.
          (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
              callsToAdd add: seeker currentState methodAboutToExecute ] ]  .
  ^ callsToAdd
```

From scripts to query notation

(Components mapping)



```
| callsToAdd |

callsToAdd := OrderedCollection new.

   seeker restartAtBytecodeOne.
   [ seeker canStep ] whileTrue: [
      seeker step.
      (seeker currentState isMessageSend and: [ seeker currentState node selector = #add: ])
ifTrue: [
         callsToAdd add: seeker currentState methodAboutToExecute ] ]  .
 ^ callsToAdd
```



```
^ (Query from: seeker newProgramStates
      select: [ :state | state isMessageSend and: [ state node selector = #add: ] ]
      collect: [ :state | state methodAboutToExecute ]) asOrderedCollection
```