

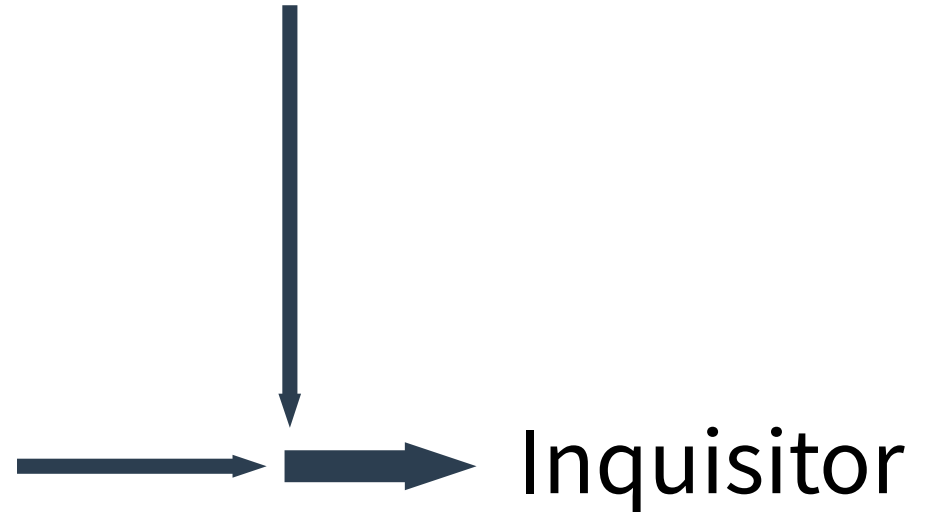
INQUISITOR

BLENDING DEBUGGER AND PROFILER

Thomas Dupriez – Stéphane Ducasse
Rmod – Inria Lille Nord Europe

IDEA

- **Debugger**
 - Interactive
 - Run execution step-by-step
 - Inspect specific execution points in detail
- **Profiler**
 - Uninteractive
 - Run entire execution
 - Global view

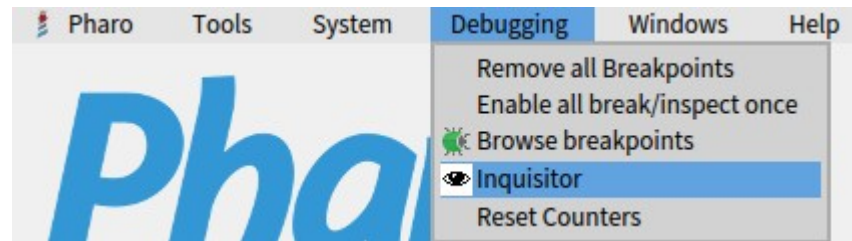


USAGE

- 1) Select an execution
 - Test method or custom code
- 2) Ask a question
 - Example: “Method, what do you return?”
- 3) Inquisitor runs the execution
- 4) Inspect the data
 - Example: All the values returned by the method
- 5) Open a debugger on any data point
- 6) *Combine/compare data from multiple questions*

DEMO: FACTORIAL

- 1) Open Inquisitor Workbench UI



WORKBENCH UI

The screenshot displays the Inquisitor Workbench interface, which is divided into three main panels:

- Inquisitor Panel:** Shows the current setup and execution steps. The setup step is `f := InqFactorial new.` and the execution step is `f computeFactorialOf: 6.`.
- Question Panel:** Shows the question being asked, which is `MethodReturn(7 returns: 1,1,2,6,24... in InqFactorial>>#compute`. It also displays the node, method, capture points, and termination status.
- Capture Points Panel:** Shows a table of capture points for the selected question. The table has four columns: TimeIndex, Value, MethodInvocation, and Receiver.

Below the panels, there are three icons: a green checkmark, a red bug, and a green checkmark. Below these icons is a text area with the label `capturePointsCollection`.

Manage Inquisitors in the image

Quick view of asked questions

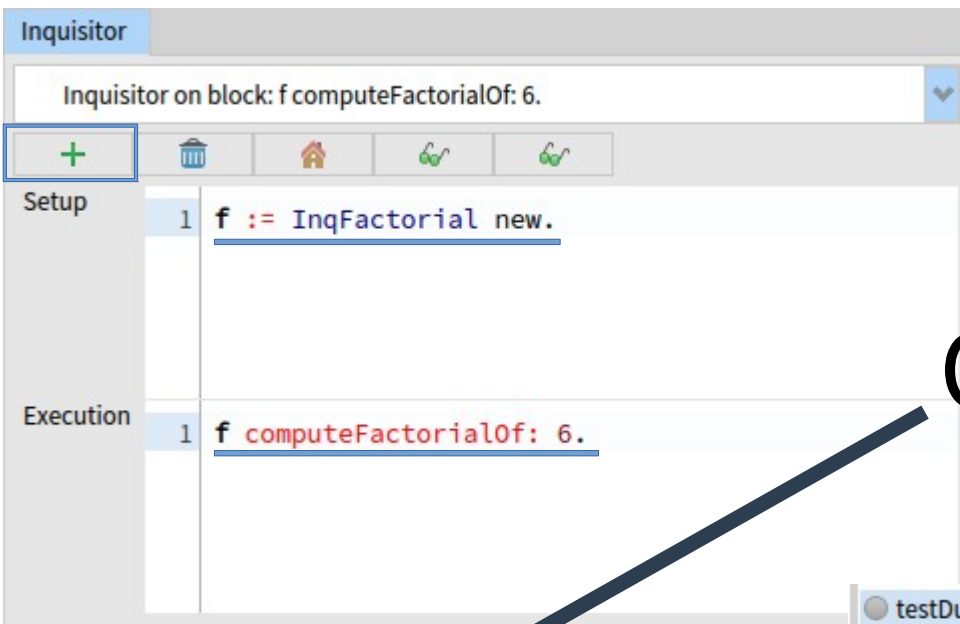
Inspect data of selected question

TimeIndex	Value	MethodInvocation	Receiver
1	1	197940736 InqFactorial>>#computeFactorialOf: an InqFactorial	
2	1	570068992 InqFactorial>>#computeFactorialOf: an InqFactorial	
3	2	589046272 InqFactorial>>#computeFactorialOf: an InqFactorial	
4	6	476171520 InqFactorial>>#computeFactorialOf: an InqFactorial	
5	24	963935488 InqFactorial>>#computeFactorialOf: an InqFactorial	

DEMO: FACTORIAL

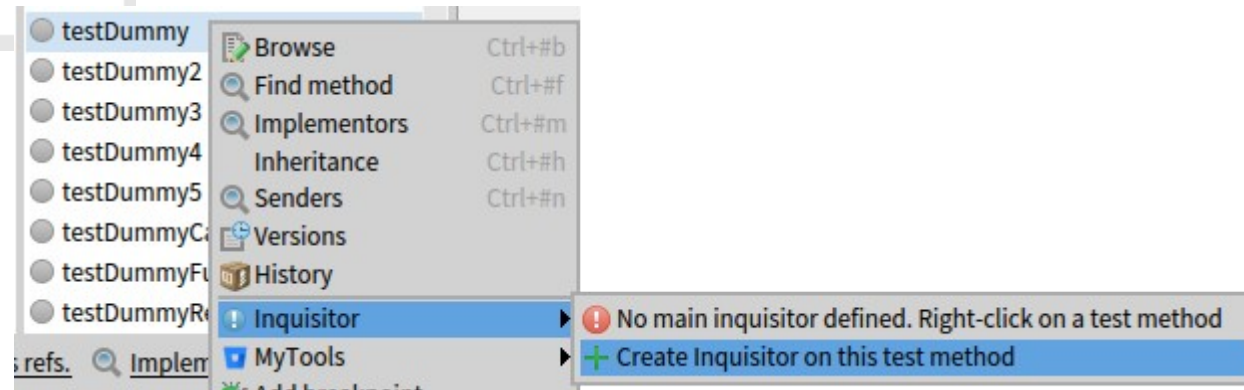
- 2) Create an Inquisitor

Via the UI (custom code)



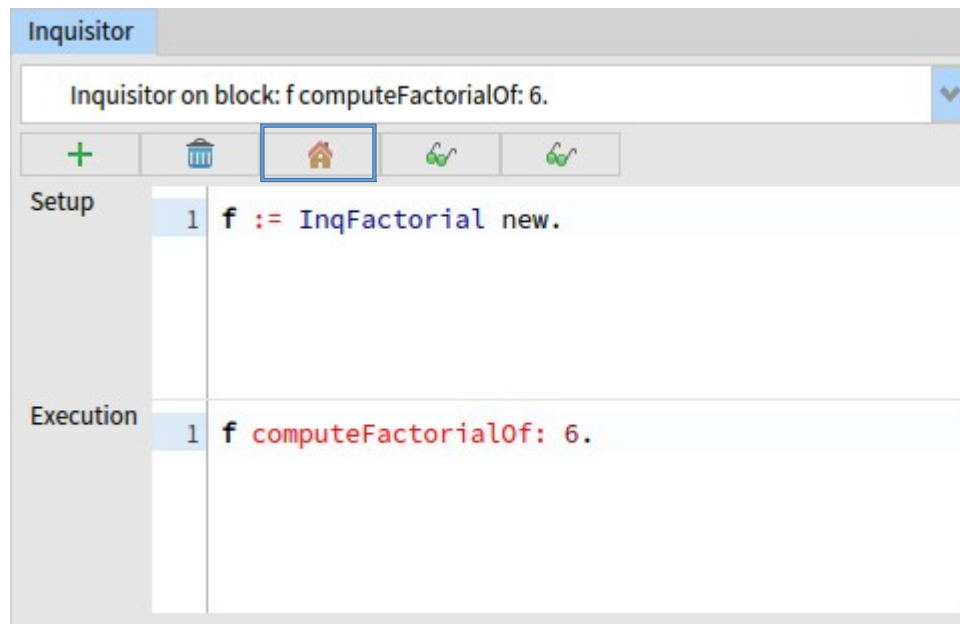
OR

Right-click a test method



DEMO: FACTORIAL

- 3) Set as main Inquisitor

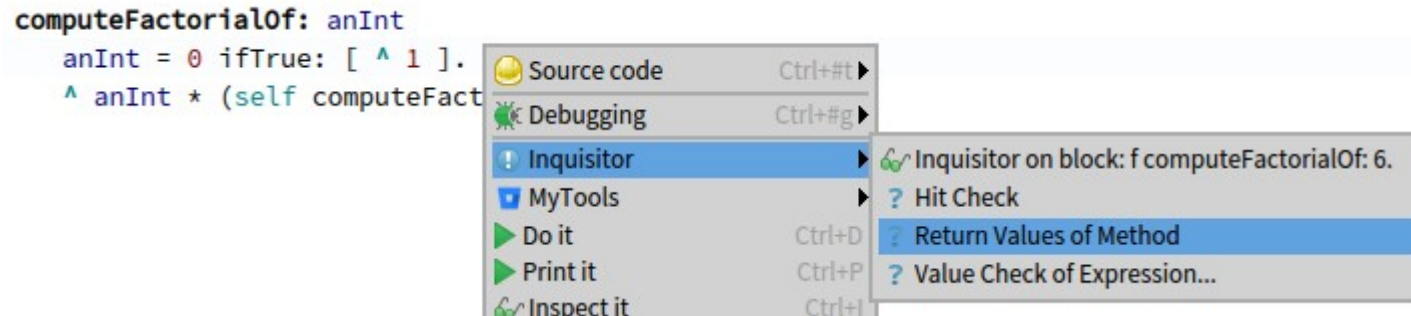


The main inquisitor will receive all the questions

DEMO: FACTORIAL

- 4) Ask the question

Right-click the method



DEMO: FACTORIAL

- 5) Inspect the data

The screenshot displays the Inquisitor Workbench interface, which is used for inspecting the execution of a program. The interface is divided into several sections:

- Inquisitor**: Shows the current block being inspected, "Inquisitor on block: f computeFactorialOf: 6.".
- Question**: Displays the method being executed, "MethodReturn(7 returns: 1,1,2,6,24... in InqFactorial>>#computeFactorialOf: 6)".
- Setup**: Shows the setup code, "1 f := InqFactorial new.".
- Execution**: Shows the execution code, "1 f computeFactorialOf: 6.".
- Capture Points**: A table showing the results of the execution, including the time index, value, method invocation, and receiver.

The **Capture Points** table is as follows:

TimeIndex	Value	MethodInvocation	Receiver
1	1	233219584 InqFactorial>>#computeFactorialOf:	an InqFactorial
2	1	218878976 InqFactorial>>#computeFactorialOf:	an InqFactorial
3	2	444575232 InqFactorial>>#computeFactorialOf:	an InqFactorial
4	6	48663552 InqFactorial>>#computeFactorialOf:	an InqFactorial
5	24	34332672 InqFactorial>>#computeFactorialOf:	an InqFactorial
6	120	1045416448 InqFactorial>>#computeFactorialOf:	an InqFactorial
7	720	521758720 InqFactorial>>#computeFactorialOf:	an InqFactorial

Below the table, the **capturePointsCollection** is shown as a list of 1 elements.

DEMO: FACTORIAL

- 6) Open a data point in a debugger (1/2)

The screenshot displays the Inquisitor Workbench interface, which is used for debugging and analyzing code execution. The interface is divided into several sections:

- Inquisitor**: Shows the current block being executed, "Inquisitor on block: f computeFactorialOf: 6.".
- Question**: Displays the method return value, "MethodReturn(7 returns: 1,1,2,6,24... in InqFactorial>>#computeFactorialOf: 6)".
- Setup**: Shows the initial state of the block, "1 f := InqFactorial new.".
- Execution**: Shows the current state of the block, "1 f computeFactorialOf: 6.".
- Capture Points**: A table showing the sequence of capture points during execution.
- Default**: A table showing the sequence of default capture points during execution.

The **Capture Points** table is currently selected, and it contains the following data:

TimeIndex	Value	MethodInvocation	Receiver
1	1	593302016 InqFactorial>>#computeFactorialOf: an InqFactorial	
2	1	951689728 InqFactorial>>#computeFactorialOf: an InqFactorial	
3	2	811932160 InqFactorial>>#computeFactorialOf: an InqFactorial	
4	6	700432384 InqFactorial>>#computeFactorialOf: an InqFactorial	
5	24	921117184 InqFactorial>>#computeFactorialOf: an InqFactorial	
6	120	434731008 InqFactorial>>#computeFactorialOf: an InqFactorial	
7	720	605787136 InqFactorial>>#computeFactorialOf: an InqFactorial	

The **Default** table is also visible, showing the sequence of default capture points during execution. The first row is highlighted, showing "1 capturePointsCollection".

DEMO: FACTORIAL

- 6) Open a data point in a debugger (2/2)

The screenshot shows a debugger window titled "Break". The "Stack" pane at the top lists four instances of the `InqFactorial` class, all with the `computeFactorialOf:` method from the `Inquisitor-SideStuff` package. Below the stack is a toolbar with buttons for `Proceed`, `Into`, `Over`, `Through`, `Run to`, `Restart`, `Return`, `Where is?`, `Create`, and `Advanced Step`.

The main code editor displays the following code:

```
1 computeFactorialOf: anInt
2   anInt = 0 ifTrue: [ ^ 1 ].
3   ^ anInt * (self computeFactorialOf: (anInt - 1))
```

The third line is highlighted, and a cursor is positioned at the opening parenthesis of the recursive call. Below the code editor, the "A StDebuggerContext (InqFac..." pane shows a table of variables:

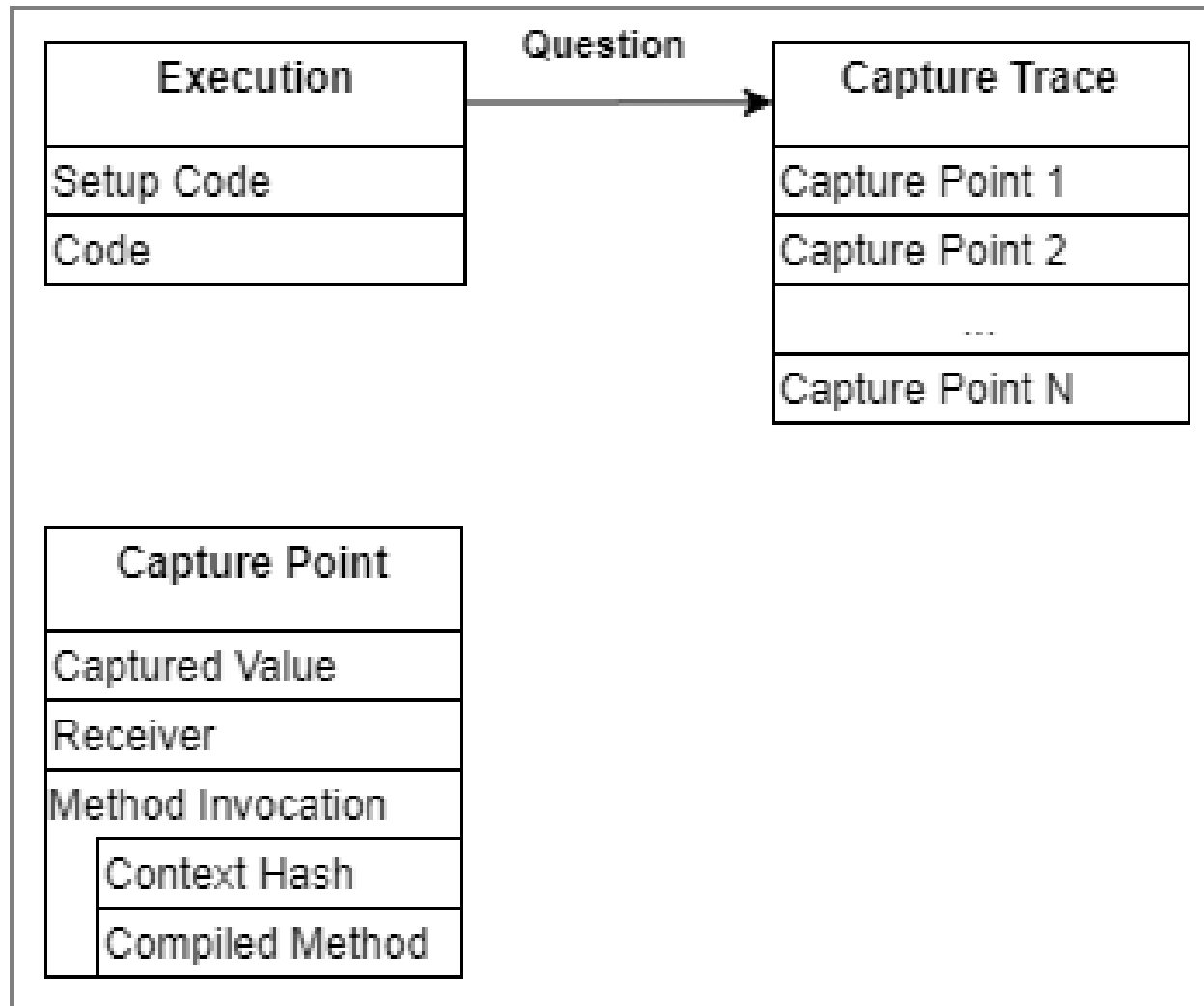
Type	Variable	Value
implicit	self	an InqFactorial
arg	Σ anInt	3
temp	Ⓒ RREifyValueVar	nil
implicit	Σ stackTop	6

The `stackTop` variable is highlighted with a blue box.

QUESTIONS

- **Hit Check**
 - AST node, when are you hit?
- **Value Check**
 - AST node, what's the value of <expression> when you're hit?
- **Method Return**
 - Method, what values do you return?
- **Variable History**
 - Variable, what values do you take?
- **Class Instantiation**
 - Class, when are you instantiated?

DEFINITIONS



IMPLEMENTATION

- Asking a question:
 - Install breakpoints at the right places
 - Run execution
 - Capture Break exceptions
 - Retrieve values from the signaler context
 - Create a Capture Point
 - Store in it the list of breakpoints the execution encountered until then
 - Remove breakpoints

IMPLEMENTATION

- **Opening a Capture Point**
 - Install stored breakpoints
 - Run execution
 - Skip every exception until all the stored breakpoints have been hit
 - Let the last exception go through → Debugger opens
 - Remove breakpoints

IMPLEMENTATION

- **Special case: Asking the Class Instanciation question**
 - Object creation is done via primitives
 - Primitives cannot be breapointed (image freeze)

IMPLEMENTATION

- Special case: Asking the Class Instanciation question
 - Object creation is done via primitives
 - Primitives cannot be breapointed (image freeze)

70: primitiveNew
Ex: Behavior>>#basicNew

148: primitiveClone
Ex: Object>>#shallowCopy

71: primitiveNew:
Ex: Behavior>>#basicNew:

79: primitiveNewMethod
Ex: CompiledCode class>>#newMethod:header:

160: primitiveAdoptInstance
Behavior>>#adoptInstance

160': primitiveAdoptInstance
MirrorPrimitive class>>#setClass:to:

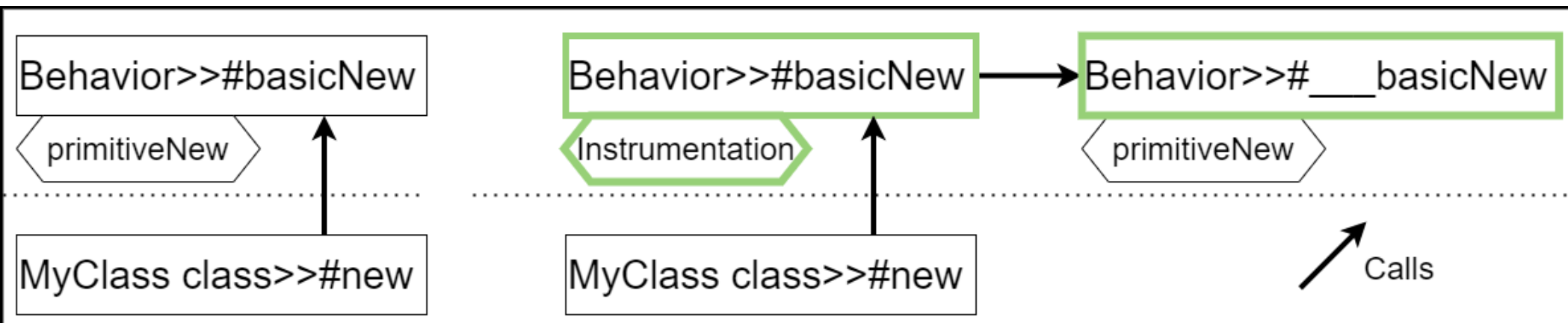
- Send a message to the class, and get an instance of it
- Send a message to an instance of a class, and get another instance of that class
- Change the class of an object:

IMPLEMENTATION



- Special case: Asking the Class Instanciacion question
 - Solution: proxy methods for primitives
 - Instrumentation code extracts the information required to create capture points
 - Breakpoints are created (not installed) on methods calling the instrumentation code (MyClass class>>#new)

Before

With proxy method



IMPLEMENTATION

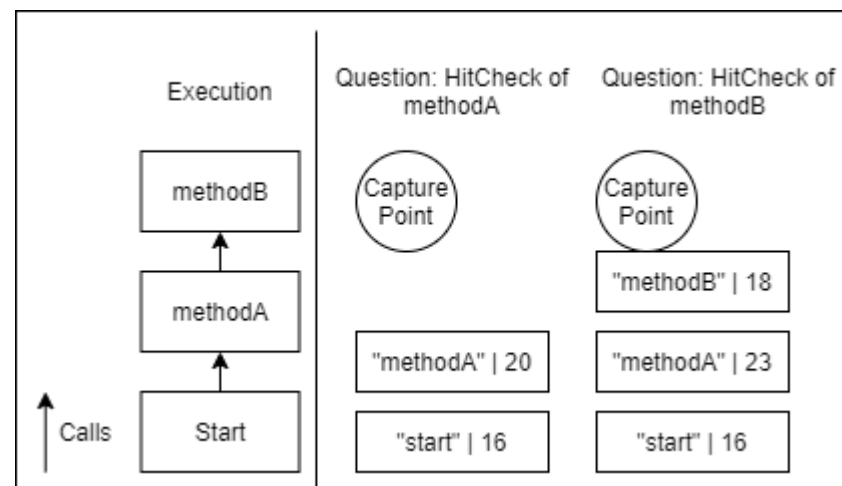
- Special case: Asking the Class Instanciation question
 - Solution: proxy methods for primitives
 - Results:
 - Answering the question: YES 
 - Opening a debugger on a capture point: NO, image freeze 
 - Probably because it usually means placing a breakpoint in `Behavior>>#new`
 - Adding conditions to these breakpoint (so that they only trigger if the class about to be instanciated is the target class) does not solve the image freeze

META OPERATIONS ON CAPTURE TRACES (NOT IMPLEMENTED)

- **Combine two capture traces (from the same execution code)**
 - Each trace has an order on its capture points
 - Even though it's the same code, the executions were not the same, so the context objects aren't ==
- **Compare two capture traces (same question, on different execution code)**

IDEA TO MERGE CAPTURE TRACES

- Synopsis:
 - Ask question 1 → capture trace 1
 - Ask question 2 → capture trace 2
 - Objective: merge capture trace 1 and 2
- Requires: time-ordering capture points from 2 traces
- Idea: upon creation, capture points store a view of the current stack (method name + pc for each context)



CONCLUSION

- Ask questions to your execution
- Open interesting points in a debugger
- Available on github

```
Metacello new  
  baseline: 'Inquisitor';  
  repository: 'github://dupriez/inquisitor';  
  load.
```



- Hit Check: AST node, when are you hit?
- Value Check: AST node, what's the value of <expression> when you're hit?
- Method Return: Method, what values do you return?
- Variable History: Variable, what values do you take?
- Class Instanciation: Class, when are you instanciated?