

C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

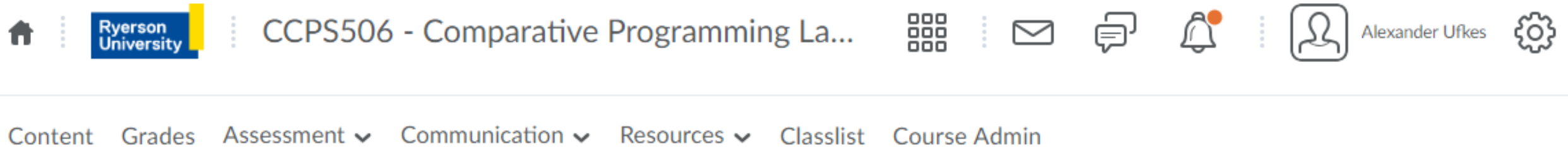
Topic 2: Control flow & collections in Smalltalk

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

The copyright to this original work is held by Alex Ufkes. Students registered in course C/CPS 506 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration (CCPS)



The screenshot shows a course administration interface. At the top, there is a navigation bar with a home icon, the Ryerson University logo, the course title "CCPS506 - Comparative Programming La...", a grid icon, an envelope icon, a speech bubble icon, a bell icon with a red notification dot, a user profile icon for "Alexander Ufkes", and a gear icon. Below the navigation bar is a menu with the following items: "Content", "Grades", "Assessment" (with a dropdown arrow), "Communication" (with a dropdown arrow), "Resources" (with a dropdown arrow), "Classlist", and "Course Admin".

- Labs #1 & #2 posted
- Assignment description posted
- See D2L or outline for due dates

Let's Get Started!

Smalltalk: OOP cranked up to 11



Objects in Smalltalk

Everything is an object. *Everything* is an instance of a corresponding class

A Smalltalk object can do exactly three things:

1. Hold state (assignment)
2. Receive a message (from itself or another object)
3. Send message (to itself or another object)

Message passing is **central** in Smalltalk.

Message Summary

Unary Messages:

sqrt, squared
asInteger
class, cr
floor, ceiling
sin, cos, tan

Any message without
argument(s)

Binary Messages:

+, -, *, /
//, \\
=, ==,
<, <=, >, >=

Arithmetic,
comparison, etc.

Keyword Messages:

raisedTo:
bitAnd:, bitOr:
show:
ifTrue:ifFalse:

Message with one or
more arguments,
ending in colon:

http://squeak.org/documentation/terse_guide/

Summary: Literals

b x	
b := true.	"true constant"
b := false.	"false constant"
x := nil.	"nil object constant"
x := 1.	"integer constants"
x := 3.14.	"float constants"
x := 2e-2.	"fractional constants"
x := 16r0F.	"hex constant"
x := -1.	"negative constants"
x := 'Hello'.	"string constant"
x := 'I'm here'.	"single quote escape"
x := \$A.	"character constant"
x := \$.	"character constant (space)"
x := #aSymbol.	"symbol constants"
x := #(3 2 1).	"array constants"
x := #('abc' 2 \$a).	"mixing of types allowed"

Example: What is the Result?

Which messages are unary? Binary? Keywords?

3 factorial + 4 factorial between: 10 and: 100

1. factorial gets sent to 3, then 4.

2. + is sent to 6 with 24 as argument

3. between:and: sent to 30 with 10 and 100 as arguments

6 + 24 between: 10 and: 100

30 between: 10 and: 100

30 between: 10 and: 100

true

Continuing on...

Today

Continuing study of Smalltalk:

- Blocks, control flow
- Some Smalltalk collections

Blocks



Blocks

Defined with square brackets []

Within the [] is Smalltalk code.

```
[ Transcript show: 27; cr ].
```

This block contains familiar code – we show the Integer 27 and do a carriage return.

Blocks

Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo...)

Pharo Tools System Debugging Windows Help

Playground

Page

```
Transcript clear.  
[ Transcript show: 27; cr ] value.
```

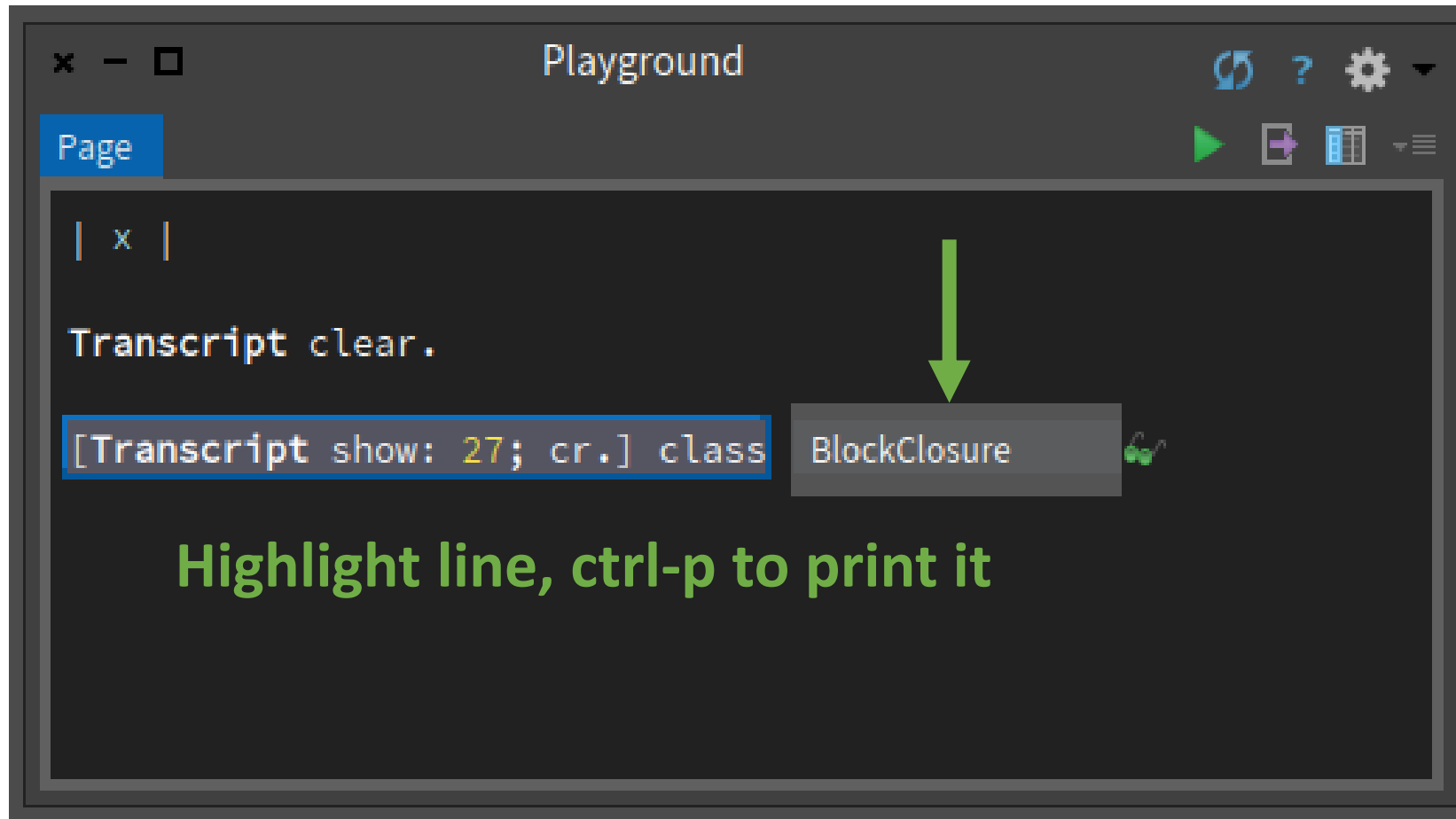
Transcript

27

To execute a block, pass it the "value" message

Playground Transcript

Blocks are Objects!



The screenshot shows a window titled "Playground" with a dark theme. The code editor contains the following text:

```
| x |  
Transcript clear.  
[Transcript show: 27; cr.] class
```

The line `[Transcript show: 27; cr.] class` is highlighted in blue. A green arrow points from this line to a tooltip box labeled "BlockClosure".

Below the code editor, the text "Highlight line, ctrl-p to print it" is displayed in green.

Blocks

Blocks are **objects**! They may be assigned to a variable:

```
Transcript clear.  
fun := [ Transcript show: 27; cr ].  
fun value.
```

Now we can execute the block using the variable

27 ←

- We are passing the **value** message to a block object.
- **BlockClosure** has a method called **value**.

Blocks as *Anonymous Functions*

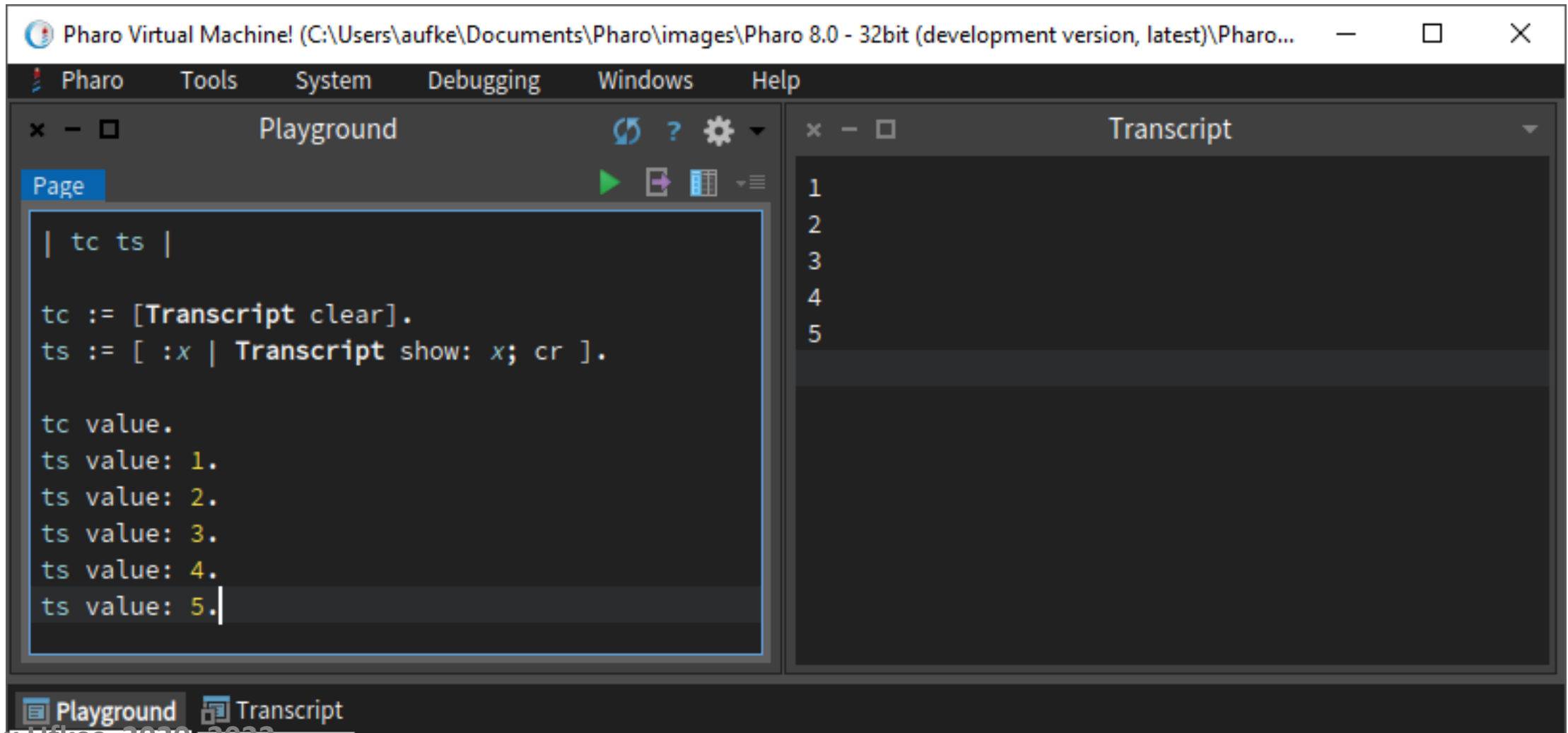
```
func := [ :x | Transcript show: x; cr. ].
```

- This is a parameter. Pharo allows up to **four**.
- *Think about why this limit of four might exist in practice.*

Argument(s) can be passed in when we send the keyword **value:** message (as opposed to the unary **value** message)

```
func value: 27.
```

Blocks with Arguments



The screenshot shows the Pharo IDE interface. The title bar reads "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help".

The "Playground" window contains the following code:

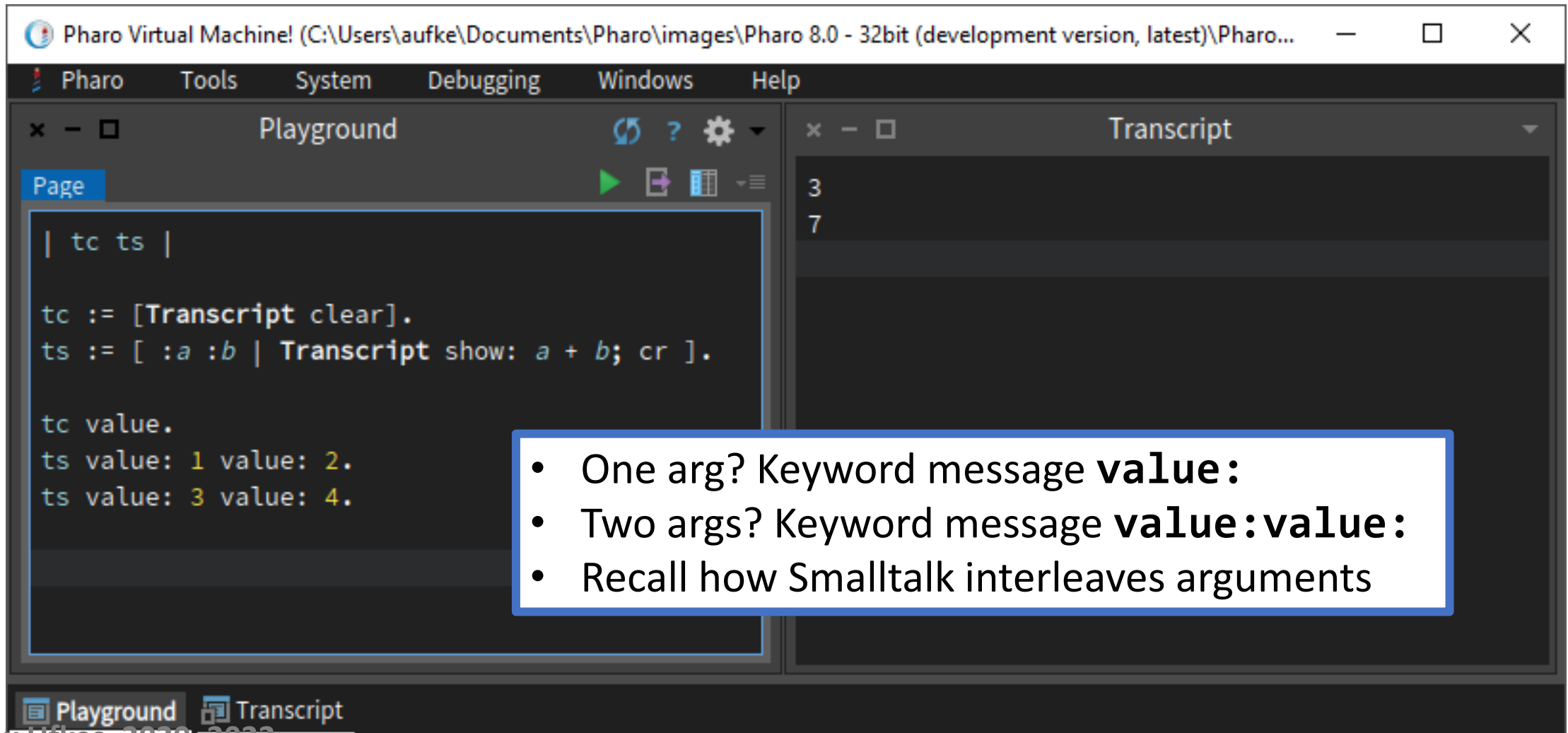
```
| tc ts |  
  
tc := [Transcript clear].  
ts := [ :x | Transcript show: x; cr ].  
  
tc value.  
ts value: 1.  
ts value: 2.  
ts value: 3.  
ts value: 4.  
ts value: 5.
```

The "Transcript" window displays the output of the code, which is a list of numbers from 1 to 5:

```
1  
2  
3  
4  
5
```

At the bottom of the IDE, there are tabs for "Playground" and "Transcript".

Blocks with Multiple Arguments



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The "Playground" window contains the following Smalltalk code:

```
| tc ts |  
  
tc := [Transcript clear].  
ts := [ :a :b | Transcript show: a + b; cr ].  
  
tc value.  
ts value: 1 value: 2.  
ts value: 3 value: 4.
```

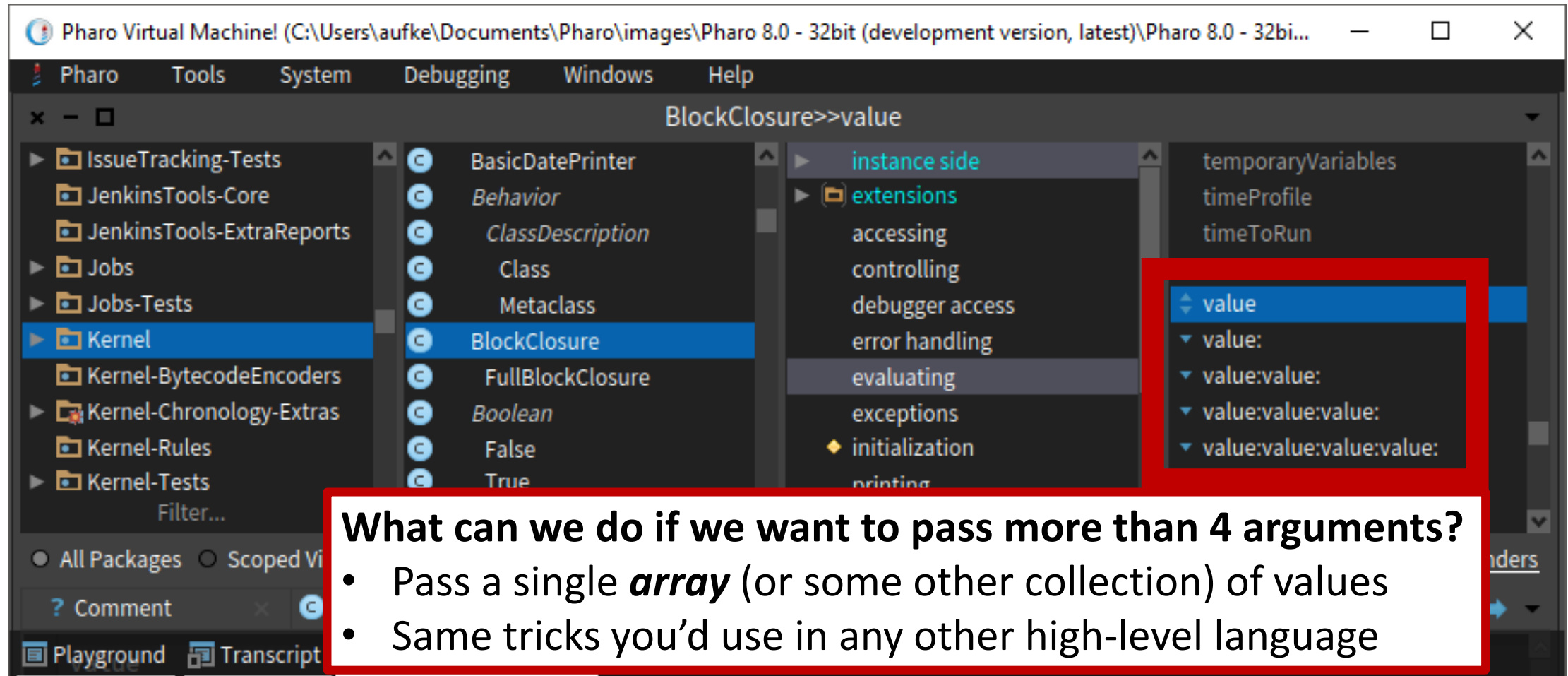
The "Transcript" window shows the output of the code:

```
3  
7
```

A callout box highlights the following points:

- One arg? Keyword message **value:**
- Two args? Keyword message **value:value:**
- Recall how Smalltalk interleaves arguments

value:value:value:value:value:value:value...



The screenshot shows the Pharo IDE interface with the BlockClosure class selected. The instance variables list on the right includes 'value', 'value:', 'value:value:', 'value:value:value:', and 'value:value:value:value:'. A red box highlights the 'value' variable and its nested structure.

What can we do if we want to pass more than 4 arguments?

- Pass a single **array** (or some other collection) of values
- Same tricks you'd use in any other high-level language

Reminder: Terminator VS Separator

`func := [:x | Transcript show: x; cr.].`

`func := [:x | Transcript show: x; cr].`

These both work. What's the difference, if any?

Reminder: Terminator VS Separator

In Java, semi-colon (;) is the statement ***terminator***.
In Smalltalk, period (.) is the statement ***separator***.

- Thus, we do not need a period after the last statement we're executing.
- This applies to blocks as well. Period not required after the last statement in a block
- Of course, it's not *illegal* to have it there, just redundant

Blocks: Multiple Statements

```
Playground
Page
| tc ts |
tc := [Transcript clear].
ts := [ :a :b | Transcript show: a + b; cr.
      Transcript show: a - b; cr.
      Transcript show: a * b; cr.
      Transcript show: a / b; cr ].
tc value.
ts value: 3 value: 4.
```

```
Transcript
7
-1
12
(3/4)
```

Nice blocks, what can you build with them?





Control Structures

Boolean Expressions

Control Structures: Branching/Selection

In Java we have syntax (reserved words) for selection:

```
if (x > y)
    System.out.println("True");
else
    System.out.println("False");
```

Control structures in Smalltalk do not have special syntax.

They are realized using blocks and message passing!

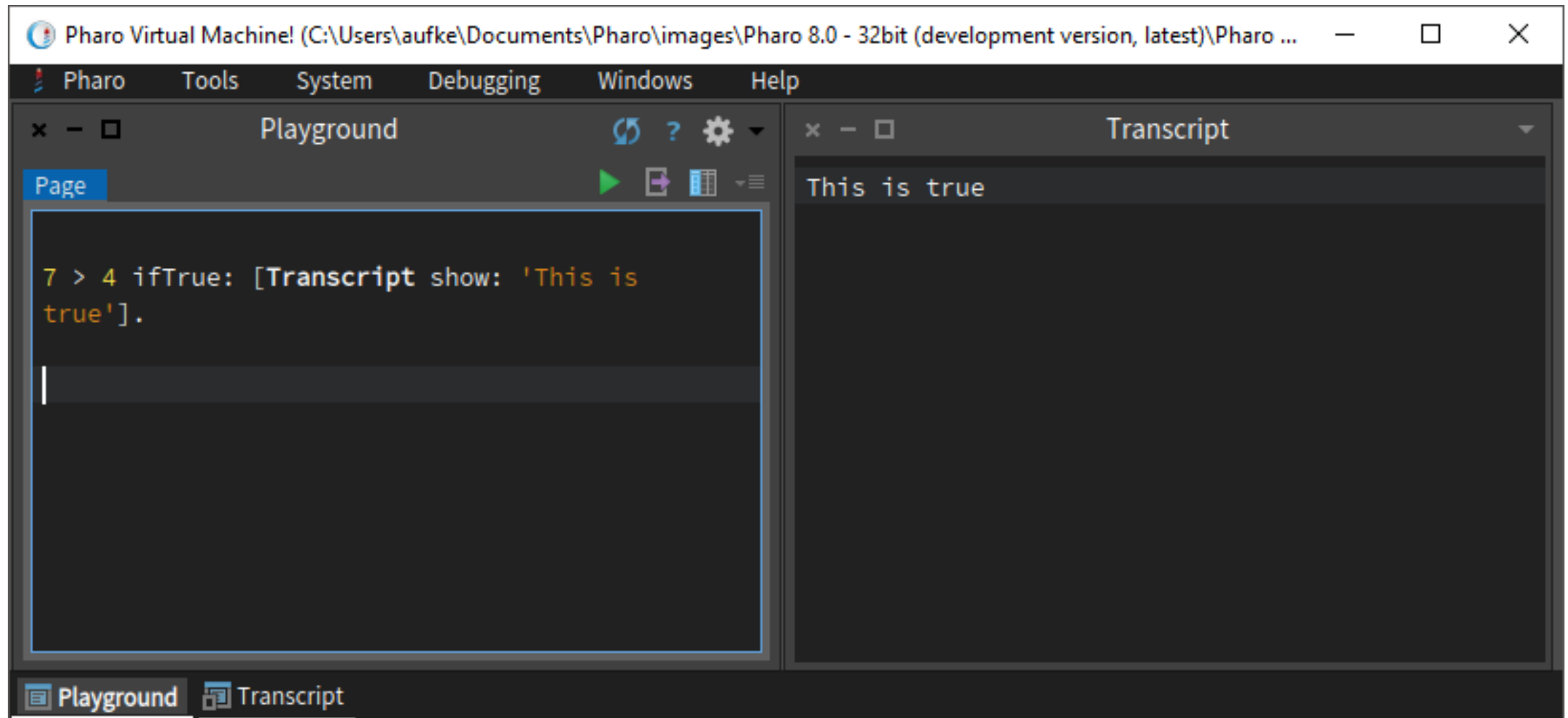
Control Structures: Branching/Selection

They are realized using blocks and message passing!

7 > 4 ifTrue: [Transcript show: 'This is true'].

- 1) **7 > 4** evaluated first (*why?*), results in Boolean object
- 2) **ifTrue:** message sent to Boolean object with BlockClosure argument [Transcript show: 'This is true'].
 - Suggests that a Boolean object (true, false) knows how to handle the **ifTrue:** message.
 - Who can guess what the **ifTrue:** method is going to do with its argument?

7 > 4 `ifTrue: [Transcript show: 'This is true']`.



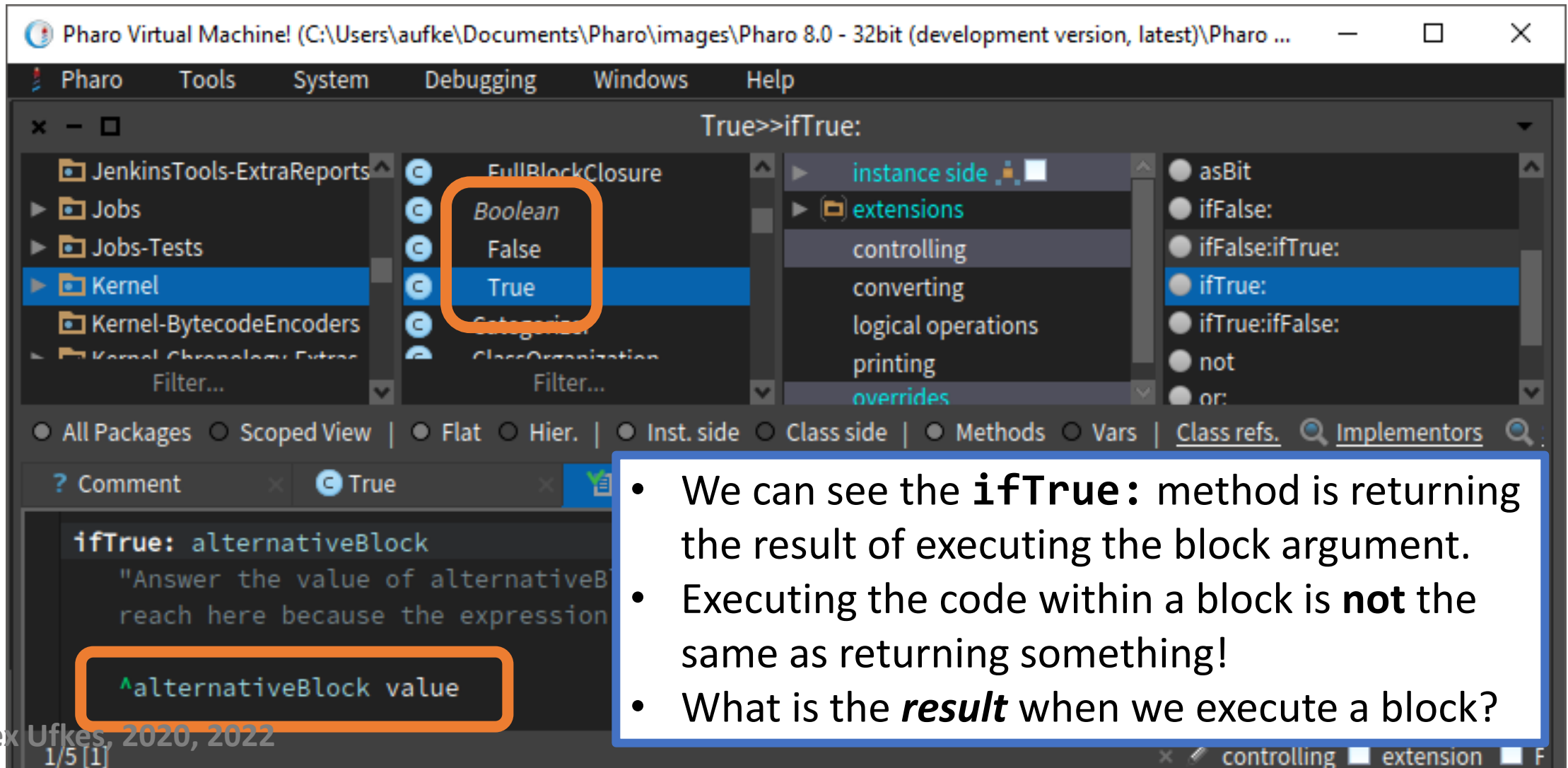
ifTrue: must be very complicated...

The screenshot shows the Pharo IDE interface. The top window title is "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The main workspace displays the class hierarchy for the `True` object, with `Boolean` and `True` highlighted. The `ifTrue:` message is selected in the right-hand pane. The bottom pane shows the implementation of `ifTrue:` as an `alternativeBlock` that sends the `value` message to the argument. The `^alternativeBlock value` line is highlighted with an orange box.

```
ifTrue: alternativeBlock
  "Answer the value of alternativeBlock
  reach here because the expression is"
  ^alternativeBlock value
```

- A **true** object receives **ifTrue:** message.
- It sends the **value** message to the argument.
- The **value** message executes a block.
- It really is that simple!

ifTrue: must be very complicated...



Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...)

Pharo Tools System Debugging Windows Help

True>>ifTrue:

- instance side
- extensions
- controlling
- converting
- logical operations
- printing
- overrides

- asBit
- ifFalse:
- ifFalse;ifTrue:
- ifTrue:**
- ifTrue;ifFalse:
- not
- or:

All Packages Scoped View | Flat Hier. | Inst. side Class side | Methods Vars | Class refs. Implementors

Comment True

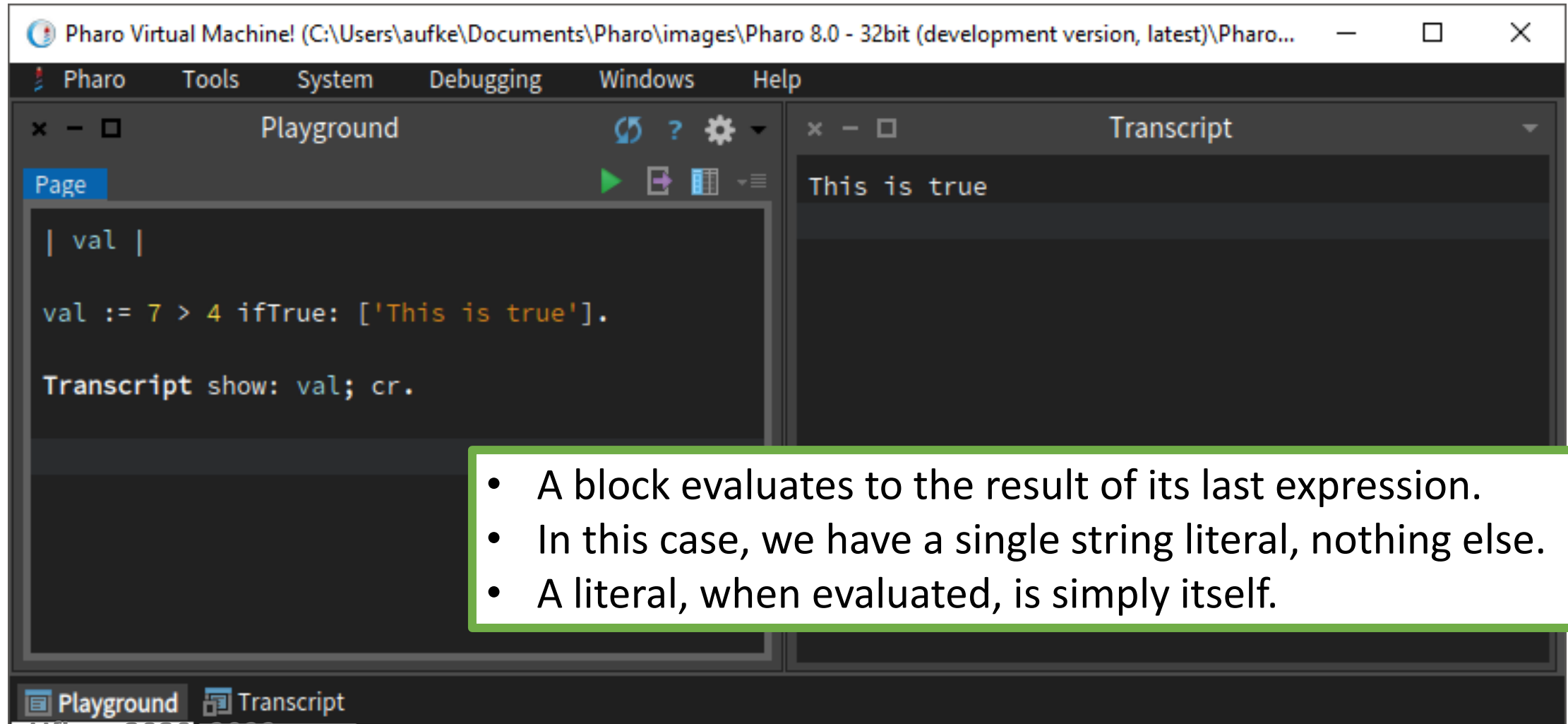
```
ifTrue: alternativeBlock
  "Answer the value of alternativeBlock
  reach here because the expression"
  ^alternativeBlock value
```

- We can see the **ifTrue:** method is returning the result of executing the block argument.
- Executing the code within a block is **not** the same as returning something!
- What is the **result** when we execute a block?

© Alex Ufkes, 2020, 2022
1/5 [1]

controlling extension F

Evaluating Blocks



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The "Playground" window contains the following code:

```
| val |  
val := 7 > 4 ifTrue: ['This is true'].  
Transcript show: val; cr.
```

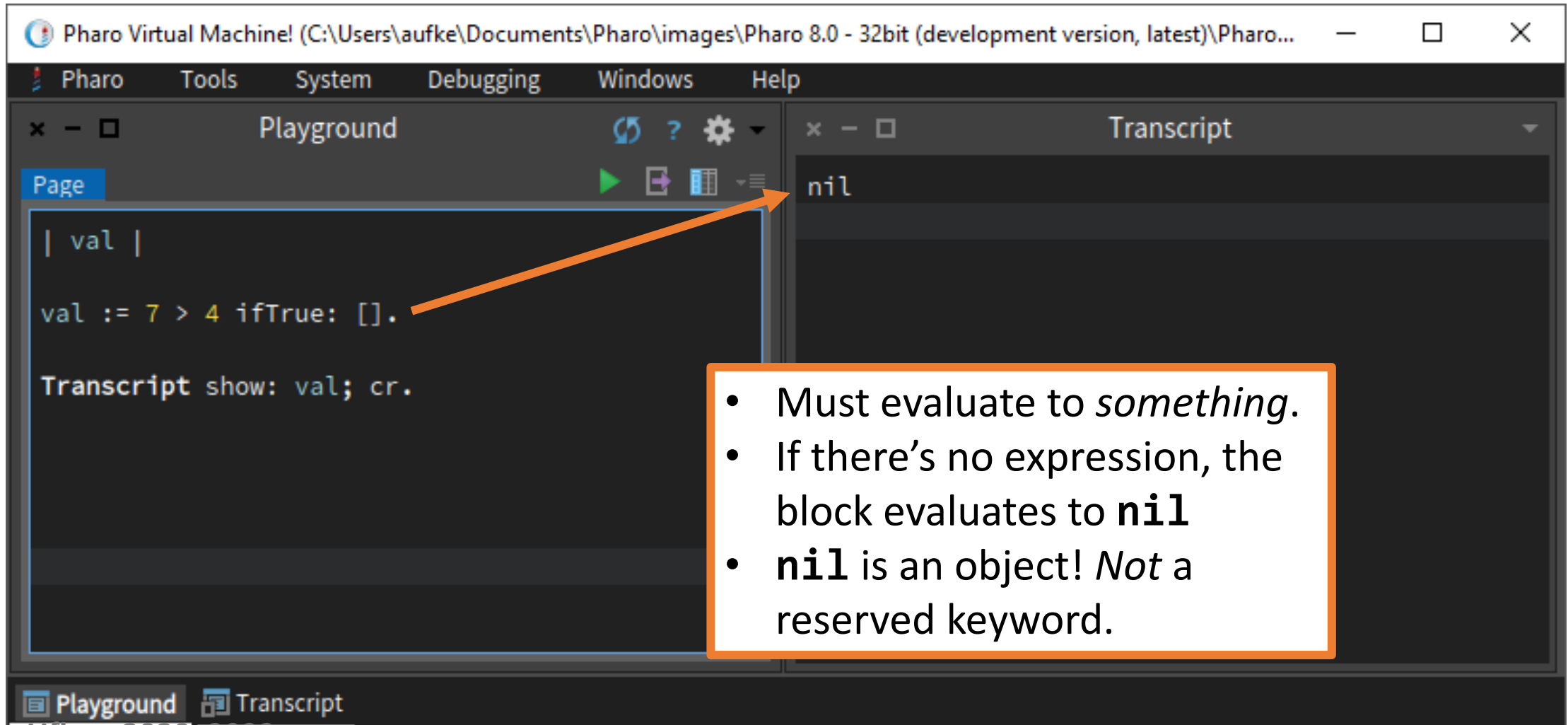
The "Transcript" window shows the output of the code execution:

```
This is true
```

A green-bordered box highlights the following list of bullet points:

- A block evaluates to the result of its last expression.
- In this case, we have a single string literal, nothing else.
- A literal, when evaluated, is simply itself.

Evaluating Blocks



The screenshot shows the Pharo IDE interface. The title bar reads "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main window is split into two panes: "Playground" on the left and "Transcript" on the right. In the "Playground" pane, the code is:

```
| val |  
val := 7 > 4 ifTrue: [].  
Transcript show: val; cr.
```

An orange arrow points from the code line `val := 7 > 4 ifTrue: [].` to the "Transcript" pane, which displays the result `nil`. A callout box with an orange border contains the following text:

- Must evaluate to *something*.
- If there's no expression, the block evaluates to **nil**
- **nil** is an object! *Not* a reserved keyword.

nil

Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo...)

Pharo Tools System Debugging Windows Help

Playground

```
| val |  
val := 4 > 7 ifTrue: ['This is true'].  
Transcript show: val; cr.  
Transcript show: val class; cr.
```

Transcript

```
nil  
UndefinedObject
```

Turns out...

- If we send **ifTrue:** to a **False** object, we also get **nil**.
- **nil** is an instance of the **UndefinedObject** class.
- Unassigned variables also reference the **nil** object.

nil

The screenshot shows the Pharo IDE interface. The title bar reads "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo 8.0 - 32bi...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main workspace is titled "False>>ifTrue:". On the left, a package browser shows "Kernel" selected. The middle pane shows the "Boolean" class with "False" highlighted. The right pane shows the "instance side" of the "False" object with the "ifTrue:" method highlighted. Below the panes, a toolbar shows "All Packages", "Scoped View", "Flat", "Hier.", "Inst. side", "Class side", "Methods", "Vars", "Class refs.", "Implementors", and "Senders". The bottom pane shows the source code for the "ifTrue:" method:

```
ifTrue: alternativeBlock
    "Since the condition is false, answer the value of the false alternative,
    which is nil. Execution does not actually reach here because the
    expression is compiled in-line."
```

The result of the execution, `^nil`, is shown at the bottom of the pane and is circled in red. The status bar at the bottom indicates "1/6 [1]" and "controlling extension F +L W".

ifTrue:ifFalse:

- By now, we know this is a keyword message that takes two arguments. Both are blocks.
- The block that gets executed depends on whether the message is sent to a **true** object or a **false** object.

ifTrue:ifFalse:

The screenshot shows the Pharo IDE interface. The title bar reads "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo 8.0 - 32bi...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main window title is "True>>ifTrue:ifFalse:". The interface is divided into several panes:

- Left Pane:** A package browser showing a tree structure with "Kernel" selected.
- Middle Pane:** A class browser for the "True" class, listing "FullBlockClosure", "Boolean", "False", "True", and "Categorizer".
- Right Pane:** A method browser for the "ifTrue:ifFalse:" method, listing "instance side", "extensions", "controlling", "converting", "logical operations", and "printing".
- Bottom Pane:** A code editor showing the method definition:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock  
"Answer with the value of trueAlternativeBlock. Execution does not  
actually reach here because the expression is compiled in-line."  
  
^trueAlternativeBlock value
```

At the bottom of the IDE, there is a status bar with "1/5 [1]" on the left and "controlling extension F +L W" on the right. The bottom-most bar shows "© Alex Ufkes, 2020, 2022" on the left and "36" on the right.

ifTrue:ifFalse:

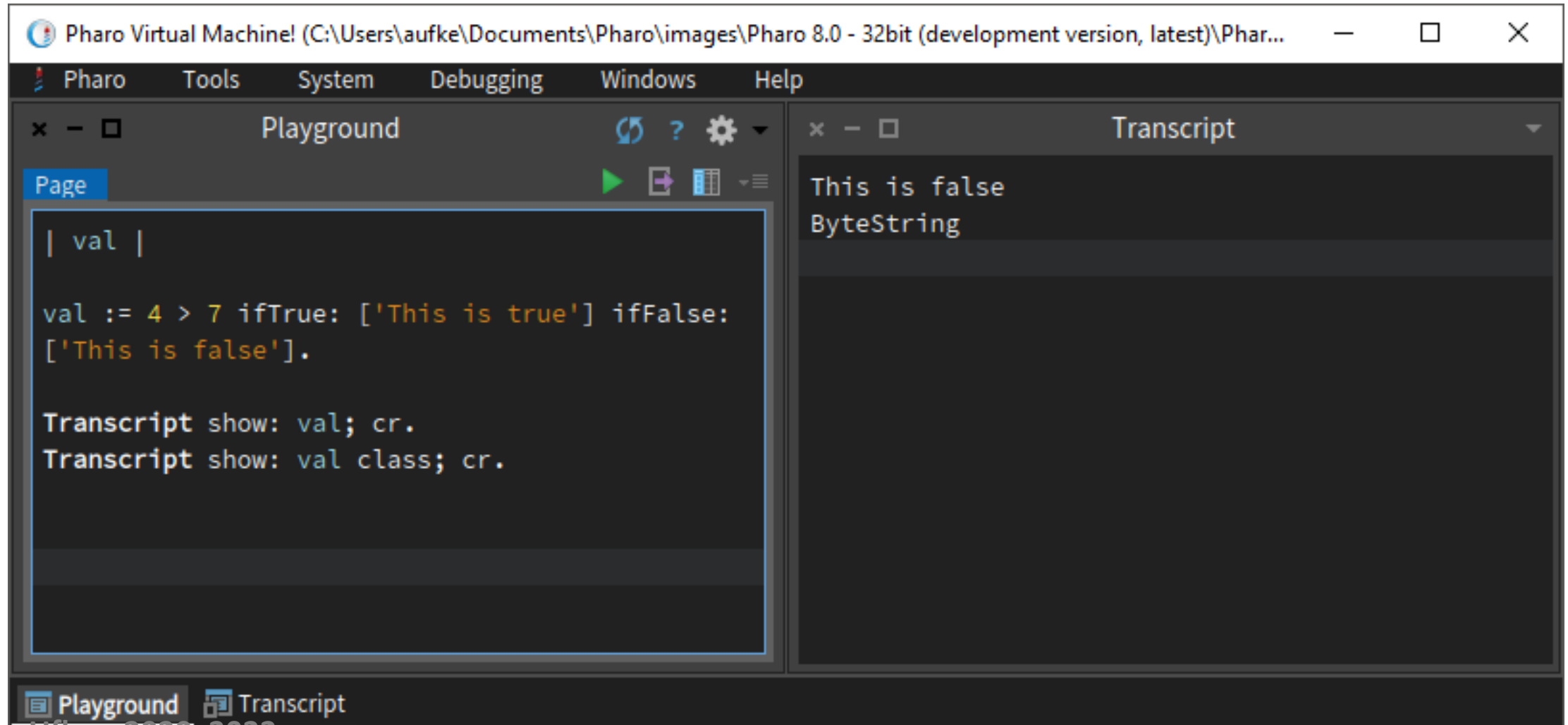
The screenshot shows the Pharo IDE interface. The title bar reads "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo 8.0 - 32bi...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main window title is "False>>ifTrue:ifFalse:". The left sidebar shows a package browser with "Kernel" selected. The middle pane shows a list of classes, with "False" selected. The right pane shows a list of methods, with "ifTrue:ifFalse:" selected. Below the panes, there are navigation options: "All Packages", "Scoped View", "Flat", "Hier.", "Inst. side", "Class side", "Methods", "Vars", "Class refs.", "Implementors", and "Senders". The bottom pane shows the source code for the "ifTrue:ifFalse:" method:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  "Answer the value of falseAlternativeBlock. Execution does not
  actually reach here because the expression is compiled in-line."

  ^falseAlternativeBlock value
```

At the bottom of the IDE, there is a status bar with "1/5 [1]" and "controlling extension F +L W".

ifTrue:ifFalse:



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Phar...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main workspace is split into two panes: "Playground" and "Transcript".

In the "Playground" pane, the following code is entered:

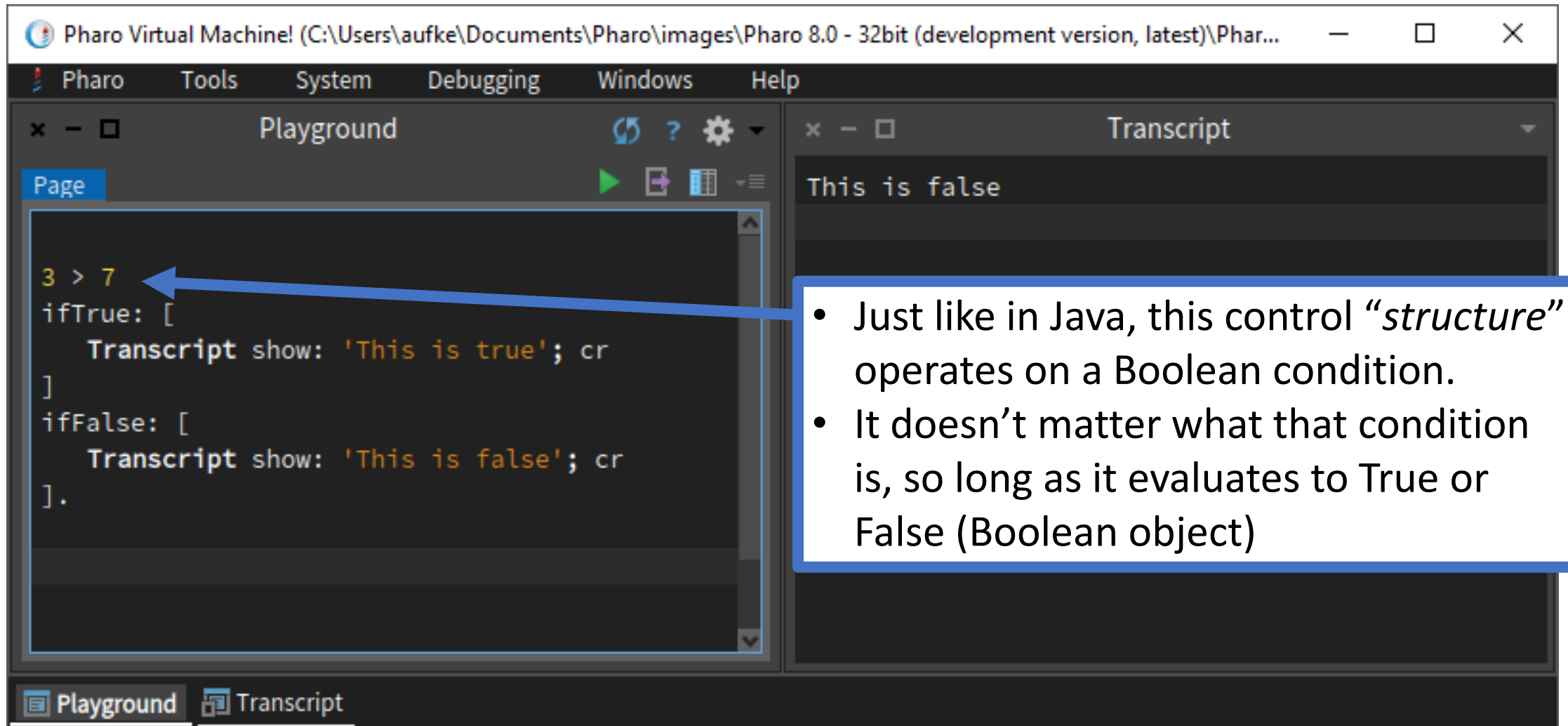
```
| val |  
  
val := 4 > 7 ifTrue: ['This is true'] ifFalse:  
['This is false'].  
  
Transcript show: val; cr.  
Transcript show: val class; cr.
```

The "Transcript" pane shows the output of the code execution:

```
This is false  
ByteString
```

The bottom of the window shows two tabs: "Playground" and "Transcript".

We can arrange the code to make the structure look more familiar:



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Phar...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main workspace is split into two panes: "Playground" on the left and "Transcript" on the right. The "Playground" pane shows the following code:

```
3 > 7  
ifTrue: [  
  Transcript show: 'This is true'; cr  
]  
ifFalse: [  
  Transcript show: 'This is false'; cr  
].
```

The "Transcript" pane shows the output: "This is false". A blue arrow points from the "3 > 7" condition in the code to a callout box. The callout box contains the following text:

- Just like in Java, this control *“structure”* operates on a Boolean condition.
- It doesn't matter what that condition is, so long as it evaluates to True or False (Boolean object)

Nesting? Blocks can be nested!

```
| temp |
temp := 103.
Transcript clear.

temp <= 0
ifTrue: [
  Transcript show: 'Solid'; cr.
]
ifFalse: [
  temp >= 100
  ifTrue: [
    Transcript show: 'Gas'; cr.
  ]
  ifFalse: [
    Transcript show: 'Liquid'; cr.
  ]
]
```

If temp <= 0, execute this block

If not, execute this block

Gas

We can write this a slightly different way – the entire structure can be an input argument to show:

```
| temp |  
temp := -88.  
Transcript clear.
```

```
Transcript show:  
(  
  temp <= 0  
  ifTrue: ['Solid']  
  ifFalse: [  
    temp >= 100  
    ifTrue: ['Gas']  
    ifFalse: ['Liquid']  
  ]  
); cr.
```

Now, instead of including *Transcript show:* statements in the blocks, we can just use the blocks to pick a string literal object to be shown.

Different Syntax, Same Semantics

```
Playground
Page
| temp |
temp := 103.
Transcript clear.

temp <= 0
ifTrue: [
  Transcript show: 'Solid'; cr.
]
ifFalse: [
  temp >= 100
  ifTrue: [
    Transcript show: 'Gas'; cr.
  ]
  ifFalse: [
    Transcript show: 'Liquid'; cr.
  ]
]
```

```
Playground
Page
| temp |
temp := -88.
Transcript clear.

Transcript show:
(
  temp <= 0
  ifTrue: ['Solid']
  ifFalse: [
    temp >= 100
    ifTrue: ['Gas']
    ifFalse: ['Liquid']
  ]
); cr.
```

Different Syntax, Same Semantics

```
Playground
Page
| temp |
temp := -88.
Transcript clear.

Transcript show:
(
  temp <= 0
  ifTrue: ['Solid']
  ifFalse: [
    temp >= 100
    ifTrue: ['Gas']
    ifFalse: ['Liquid']
  ]
); cr.
```

```
Playground
Page
| temp result |
temp := -88.

result := temp <= 0 ifTrue: ['Solid']
         ifFalse: [ temp >= 100 ifTrue: ['Gas']
                  ifFalse: ['Liquid'] ].

Transcript clear; show: result; cr.
```

This is great practice – rearrange code to be as small/efficient as possible. It will help you truly understand how the syntax works.

temp := 88.

Does not evaluate yet!

temp <= 0 ifTrue: ['Solid'] ifFalse: [temp >= 100 ifTrue: ['Gas'] ifFalse: ['Liquid']].

This goes first!

false ifTrue: ['Solid'] ifFalse: [temp >= 100 ifTrue: ['Gas'] ifFalse: ['Liquid']].

Passed to false object

temp >= 100 ifTrue: ['Gas'] ifFalse: ['Liquid']

This goes first!

false ifTrue: ['Gas'] ifFalse: ['Liquid']

Passed to false object

```
false ifTrue: ['Gas'] ifFalse: ['Liquid'].
```



Passed to `false` object

'Liquid'

Finally, when all the message passes have been evaluated, we're left with 'Liquid'

A large Ferris wheel is the central focus, set against a twilight sky. A thick yellow arrow starts at the top of the wheel and curves clockwise around its perimeter, ending with a downward-pointing arrowhead. At the bottom center, a white box with a black border contains the word "Condition". A thick green arrow originates from the left side of the image, passes through the "Condition" box, and extends to the right edge of the frame.

Repetition

Condition

Repetition Using Messages & Blocks

The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Phar...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The "Playground" window contains the following code:

```
| x y |  
x := 8. y := 2.  
  
Transcript clear.  
  
[ x > 0 ] whileTrue: [  
  x := x - 1.  
  y := y + 2.  
  Transcript show: x; tab; show: y; cr.  
]
```

The "Transcript" window shows the output of the code execution:

```
7 4  
6 6  
5 8  
4 10  
3 12  
2 14  
1 16  
0 18
```

A green box highlights the `whileTrue:` block in the code, and another green box highlights the question and answer text below.

What messages are sent to what objects?

- `whileTrue:` message sent to block object with another block as an argument

Repetition Using Messages & Blocks

```
[x > 0] whileTrue: [ x := x - 1. y := y + 2. ]
```

whileTrue: message sent to block containing Boolean expression $x > 0$

The BlockClosure class understands the **whileTrue:** message.

The argument that accompanies the **whileTrue:** message is a block containing the code to be repeated.

BlockClosure>>#whileTrue:

Scoped Variables History Navigator

Type: Pkg1|^Pkg2|Pk.*Core\$

- IssueTracking
- IssueTracking-Tests
- Jobs
- JobsTests
- Kernel**
- Kernel-Rules
- Kernel-Tests
- Kernel-Tests-Rules
- Keymapping-Core
- Keymapping-KeyCombir

- ClassDescription
- Class
- Metaclass
- BenchmarkResult
- BlockClosure**
- FullBlockClosure
- Boolean
- False
- True

Hier. Class Com.

- all --
- accessing
- controlling**
- debugger access
- error handling
- evaluating
- exceptions
- initialize-release
- printing
- private
- scanning

- valueWithPossibleArgs:
- valueWithPossibleArgument:
- valueWithin:onTimeout:
- valueWithoutNotifications
- whileFalse
- whileFalse:
- whileNil:
- whileNotNil:
- whileTrue
- whileTrue:**

`whileTrue: aBlock`

"Ordinarily compiled in-line, and therefore not overridable.
 This is in case the message is sent to other than a literal block.
 Evaluate the argument, aBlock, as long as the value of the receiver is true."

`self value ifTrue: [aBlock value. self whileTrue: aBlock]` **Recursive!**

- Send `ifTrue:` message to Boolean
- Argument is a Block object

Finally, send **`whileTrue:`** once more to the same block, with the same argument.

```
self value ifTrue: [ aBlock value. self whileTrue: aBlock ]
```

- Send `value` message to **`self`**, which executes it
- **`self`** evaluates to a Bool

- **`aBlock`** is the input argument first passed with the **`whileTrue:`** message
- We execute it once

Repetition Using Messages & Blocks

The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Phar...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help".

The "Playground" window contains the following code:

```
| x y |  
x := 8. y := 2.  
  
Transcript clear.  
  
[ x <= 0 ] whileFalse: [  
  x := x - 1.  
  y := y + 2.  
  Transcript show: x; tab; show: y; cr.  
]
```

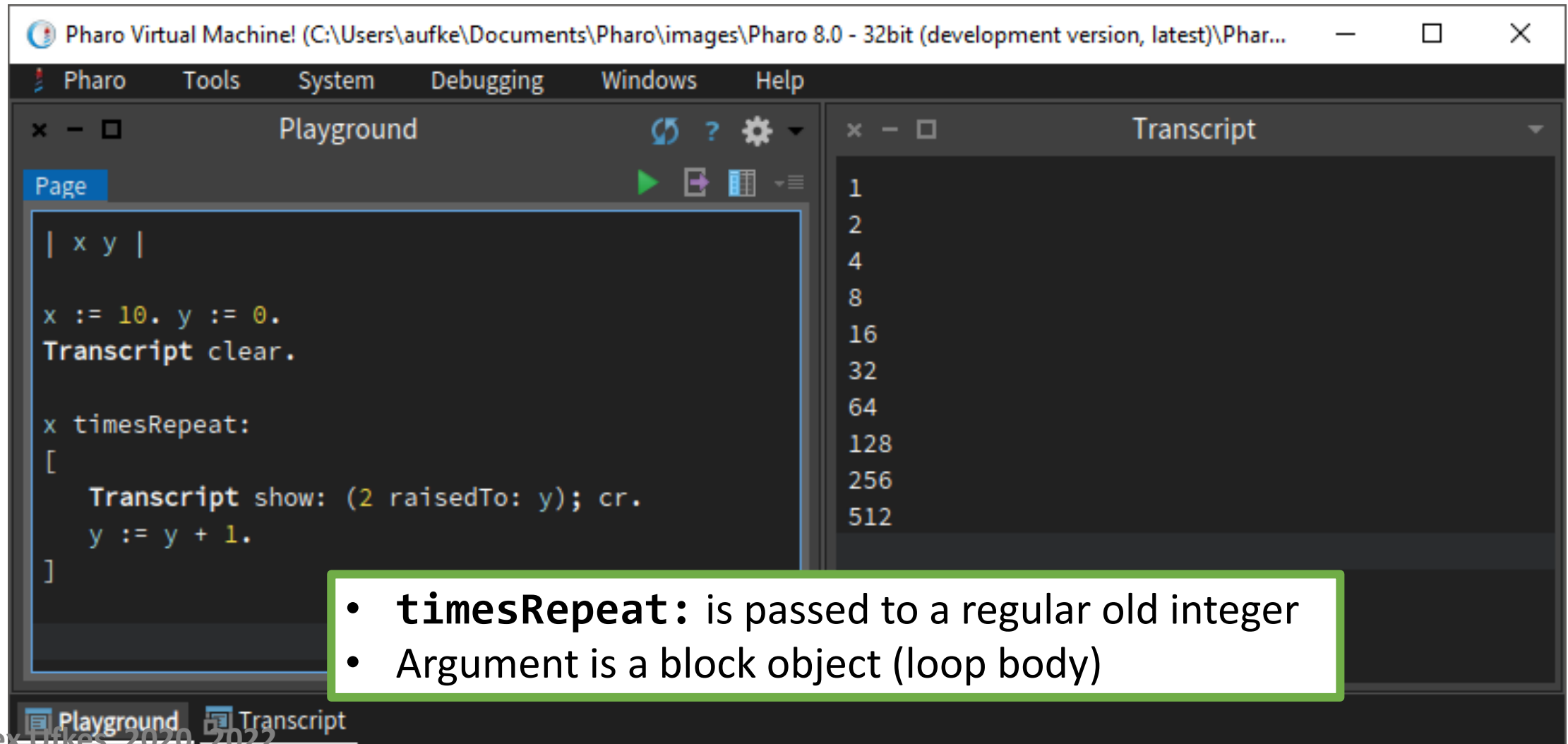
An orange box highlights the line `[x <= 0] whileFalse: [`. A text box above the code says "Much like selection, there are many options".

The "Transcript" window shows the output of the code:

```
7 4  
6 6  
5 8  
4 10  
3 12  
2 14  
1 16  
0 18
```

The bottom of the window shows tabs for "Playground" and "Transcript".

timesRepeat:



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Phar...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main area is split into two panes: "Playground" on the left and "Transcript" on the right.

In the "Playground" pane, the following code is visible:

```
| x y |  
  
x := 10. y := 0.  
Transcript clear.  
  
x timesRepeat:  
[  
  Transcript show: (2 raisedTo: y); cr.  
  y := y + 1.  
]
```

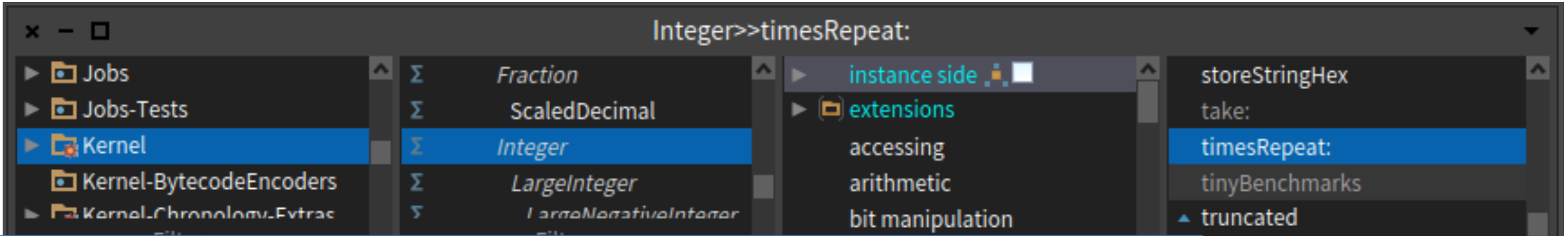
In the "Transcript" pane, the output of the code is shown as a list of powers of 2:

```
1  
2  
4  
8  
16  
32  
64  
128  
256  
512
```

A green-bordered box highlights the following bullet points:

- **timesRepeat:** is passed to a regular old integer
- Argument is a block object (loop body)

At the bottom of the IDE, there are tabs for "Playground" and "Transcript".



timesRepeat: aBlock

"Evaluate the argument, aBlock, the number of times to evaluate the receiver."

```
| count |  
count := 1.  
[count <= self]  
  whileTrue:  
    [aBlock value.  
     count := count + 1]
```

timesRepeat: is just a wrapper for **whileTrue:**



For-loop equivalent - to:do:

Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Phar... — □ ×

Pharo Tools System Debugging Windows Help

Playground

```
| x |  
x := 10.  
Transcript clear.  
  
1 to: x do:  
[ :a |  
  Transcript show: (2 raisedTo: a); cr.  
]
```

Transcript

```
2  
4  
8  
16  
32  
64  
128  
256  
512  
1024
```

- **a** is our loop index
- Can be used in the “body” of the loop
- I.e., the block

- **to:do:** message passed to an Integer object
- Two arguments – ending index and block representing loop body.

© Alex Ofkes, 2020, 2022

Number>>to:do:

- Jobs
- Jobs-Tests
- Kernel**
- Kernel-BytecodeEncoders
- Kernel-Chronology-Extras

- DateAndTime
- Duration
- Number**
- Float
- RopedFloat64

- instance side
- extensions
- should be implemented
- arithmetic
- converting
- mathematical functions

- to:by:
- to:by:do:
- to:do:**
- truncateTo:
- truncated
- week

All Packages | Scoped View | Flat | Hier. | Inst. side | Class side | Methods | Vars | Class refs. | Implementors | Senders

Comment | Number | **to:do:** | + Inst. side method

```
to: stop do: aBlock
  "Normally compiled in-line, and therefore not overridable.
  Evaluate aBlock for each element of the interval (self to: stop by: 1)."
  | nextValue |
  nextValue := self.
  [nextValue <= stop]
  whileTrue:
    [aBlock value: nextValue.
     nextValue := nextValue + 1]
```

- Same as **timesRepeat:**, just a wrapper for **whileTrue:**
- *Why is this implemented under **Number**, and not **Integer**?*

1/9 [1] Collections-Sequenceable extension F +L W

For-loop equivalent - to:do:

*Why is this implemented under **Number**, and not **Integer**?*

In the case of timesRepeat:

- We cannot execute a block 4.7 times.
- This makes no sense.

x to: y do: [...]

- Will count from x to y by 1.
- We *can* count from 2.1 to 4.7.
- **2.1, 3.1, 4.1, 5.1**

Iterate Over Arrays

- In Java we have a different version of for loop for safely iterating over arrays.
- Prevents us from accidentally going out of bounds.

```
public static void main(String[] args)
{
    int[] nums = {-11, 68, 4, 0, 99};
    int sum = 0;
    for (int e : nums)
        sum += e;
    System.out.println(sum);
}
```

- **e** will take the value of each element in the array **nums**.
- Written this way, the loop will automatically go through each element in **nums**.
- Don't need to keep track of index or conditions, it's done for us.

Iterate Over Arrays

The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The "Playground" window contains the following code:

```
| x sum |  
x := #(1 2 3 4 5 6 7 8 9 10).  
sum := 0.  
x do: [ :a | sum := sum + a ].  
Transcript clear; show: sum; cr.
```

The "Transcript" window shows the output: 55.

- Send **do:** message to an array object with block as argument.
- Block argument is each array element
- Also works with other collections
- Speaking of collections...

Smalltalk Collections

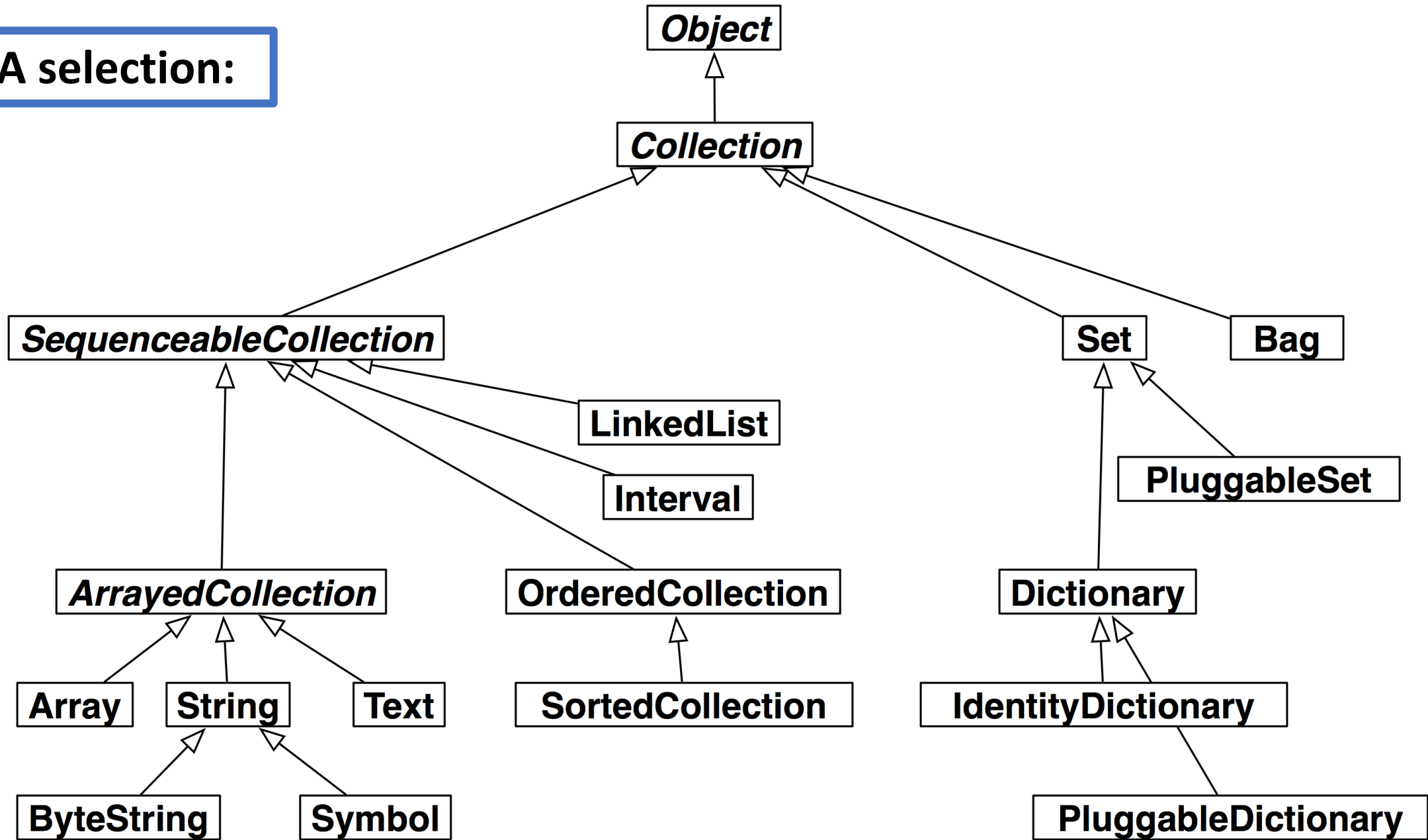


We've seen arrays.

In Java, we have things like ArrayLists, Vectors, LinkedLists, PriorityQueues, and so on.

What about Smalltalk?

A selection:



Ordered Collection

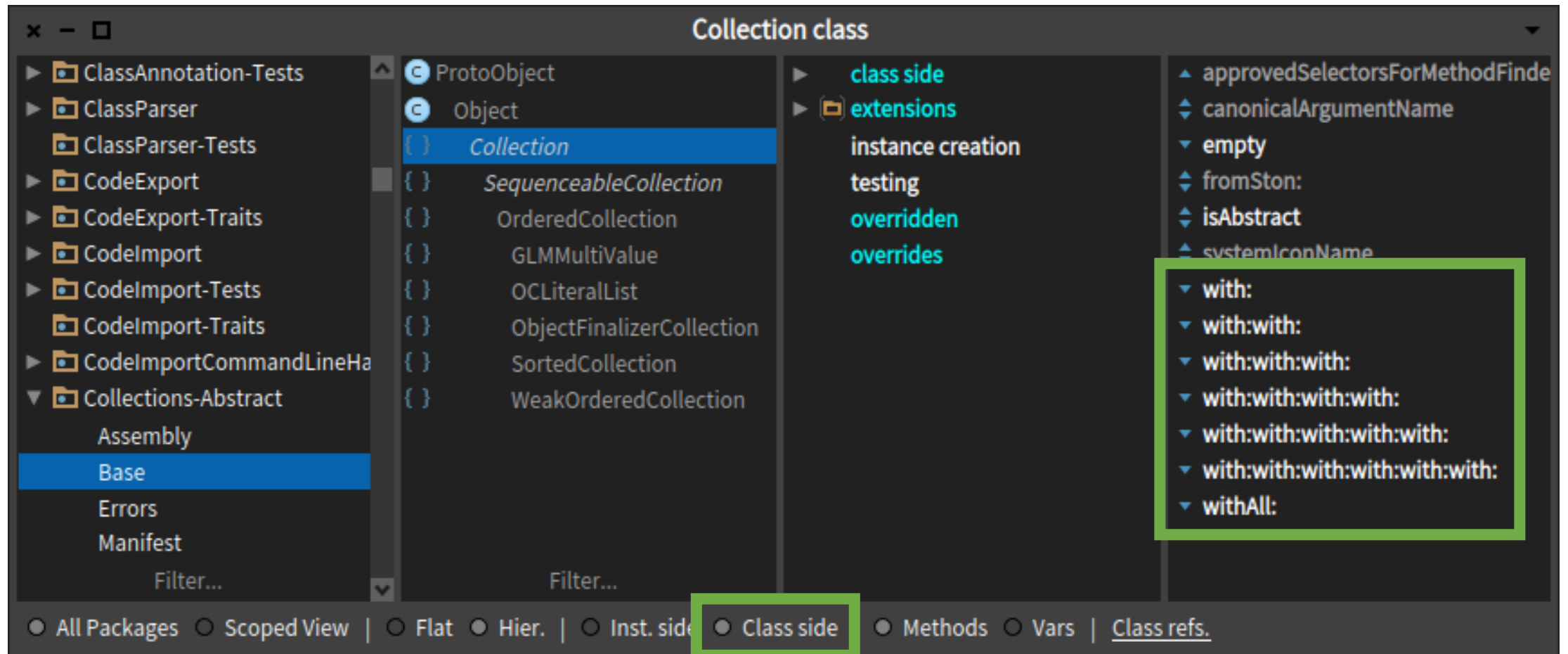
Similar to Java ArrayList. Acts as an expandable Array.

```
| oc1 oc2 oc3 |  
  
oc1 := OrderedCollection new.  
oc2 := OrderedCollection with: 1 with: 2 with: 3.  
oc3 := OrderedCollection with: 1 with: $A.  
  
Transcript clear.  
Transcript show: oc1; cr.  
Transcript show: oc2; cr.  
Transcript show: oc3; cr.
```

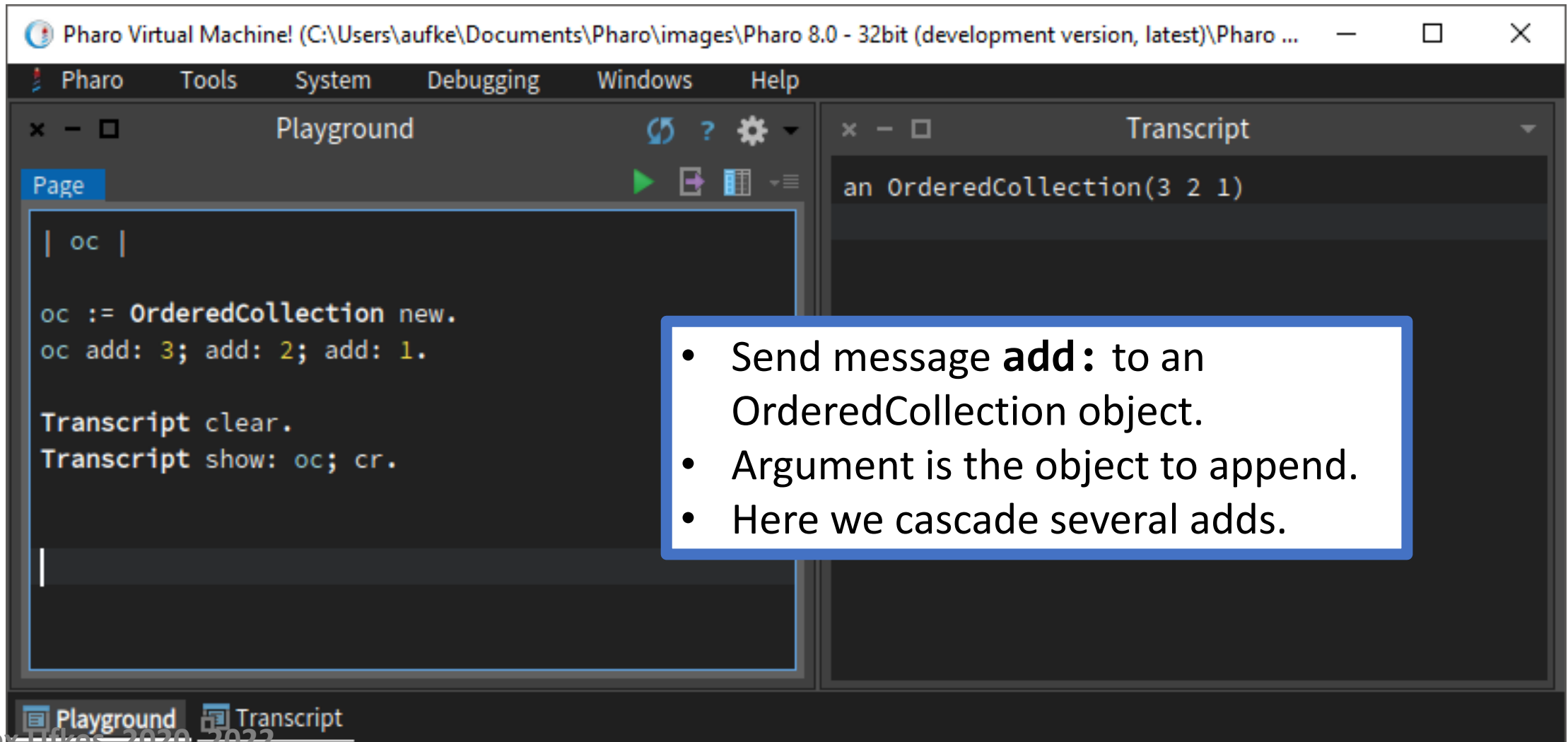
```
an OrderedCollection()  
an OrderedCollection(1 2 3)  
an OrderedCollection(1 $A)
```

- Can initialize as empty
- Or with some initial elements
- Can be heterogeneous!
- Transcript can print them

with:with:with:with:with:with:with:with:with:...



Ordered Collection: Add Elements



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The "Playground" window contains the following code:

```
| oc |  
  
oc := OrderedCollection new.  
oc add: 3; add: 2; add: 1.  
  
Transcript clear.  
Transcript show: oc; cr.
```

The "Transcript" window shows the output: "an OrderedCollection(3 2 1)".

- Send message **add:** to an OrderedCollection object.
- Argument is the object to append.
- Here we cascade several adds.

Ordered Collection: Add Elements

The screenshot shows the Pharo IDE interface. The title bar reads "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help".

The "Playground" window contains the following code:

```
| oc |  
oc := OrderedCollection new.  
oc add: 3; add: 2; add: 1.  
oc addFirst: 'Hello'.  
oc addFirst: #($a $b).  
  
Transcript clear.  
Transcript show: oc; cr.
```

The "Transcript" window shows the output: "an OrderedCollection(#(\$a \$b) 'Hello' 3 2 1)".

A callout box with a green border contains the following text:

- Notice we can add a whole array!
- Arrays are objects, just like integers.

Green arrows point from the callout box to the code line `oc addFirst: #($a $b).` and the output line `an OrderedCollection(#($a $b) 'Hello' 3 2 1)`.

Ordered Collection: at:put:, addAll:

The screenshot shows the Pharo IDE interface. The 'Playground' window contains the following code:

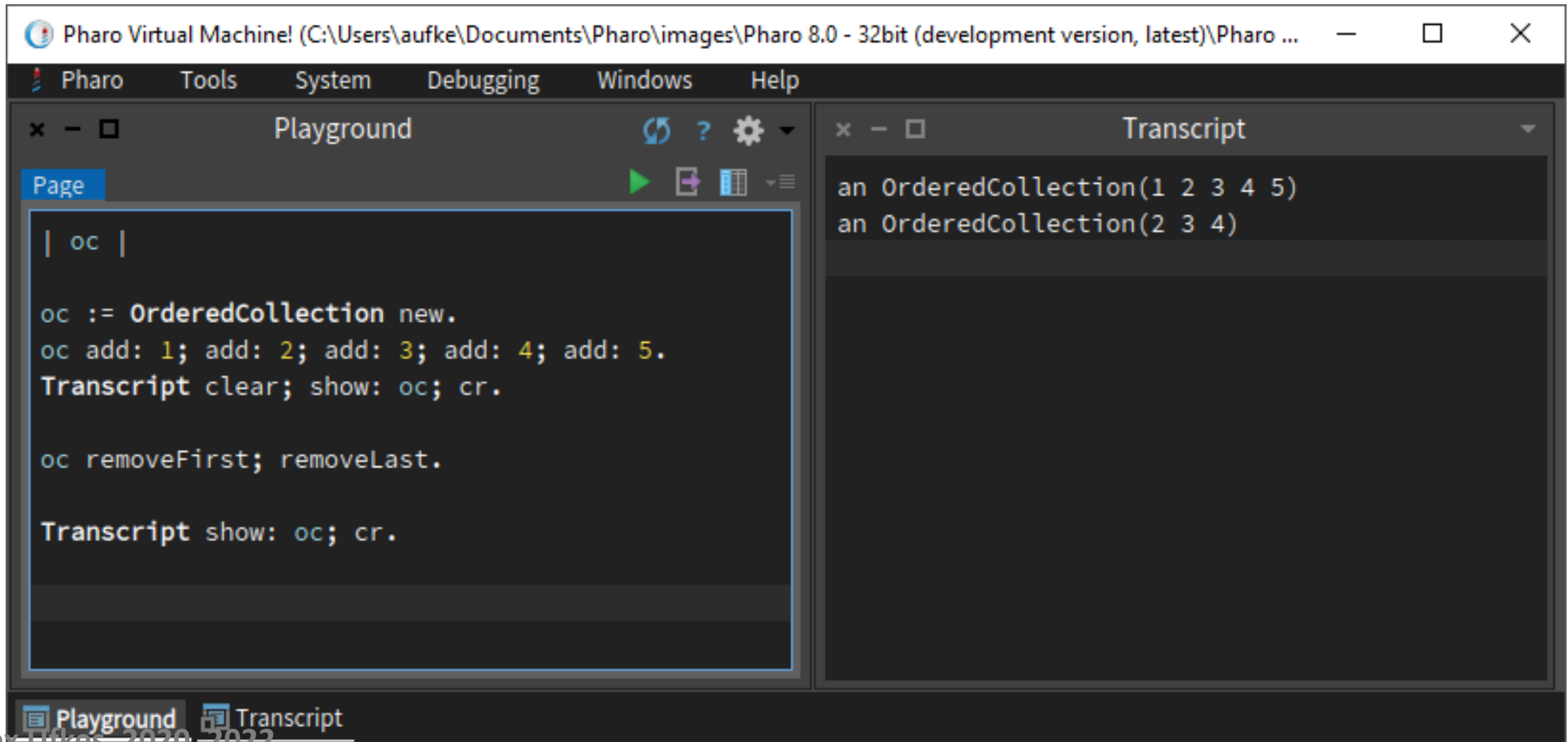
```
| oc |  
  
oc := OrderedCollection new.  
oc add: 1; add: 2; add: 3; add: 4; add: 5.  
Transcript clear; show: oc; cr.  
  
oc at: 2 put: 9.  
Transcript show: oc; cr.  
  
oc addAll: #(9 8 7).  
Transcript show: oc; cr.
```

The 'Transcript' window shows the output of the code:

```
an OrderedCollection(1 2 3 4 5)  
an OrderedCollection(1 9 3 4 5)  
an OrderedCollection(1 9 3 4 5 9 8 7)
```

An orange bracket highlights the third line of the transcript, and an orange callout box contains the text: "Notice! **addAll:** adds the *elements* of an array, **not** the array itself."

Ordered Collection: Removing



The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". There are two panes: "Playground" and "Transcript".

The "Playground" pane contains the following code:

```
| oc |  
  
oc := OrderedCollection new.  
oc add: 1; add: 2; add: 3; add: 4; add: 5.  
Transcript clear; show: oc; cr.  
  
oc removeFirst; removeLast.  
  
Transcript show: oc; cr.
```

The "Transcript" pane shows the output of the code execution:

```
an OrderedCollection(1 2 3 4 5)  
an OrderedCollection(2 3 4)
```

At the bottom of the IDE, there are tabs for "Playground" and "Transcript".

Ordered Collection: Removing

The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help". The main workspace is split into two panes: "Playground" on the left and "Transcript" on the right.

In the "Playground" pane, the following code is entered:

```
| oc |  
  
oc := OrderedCollection new.  
oc add: 0; add: 2; add: 7; add: 9; add: 4.  
Transcript clear; show: oc; cr.  
  
oc removeAll: #(2 7 4).  
Transcript show: oc; cr.
```

The "Transcript" pane shows the output of the code:

```
an OrderedCollection(0 2 7 9 4)  
an OrderedCollection(0 9)
```

A callout box with a blue border contains the following text:

- Remove values. Not indexes!
- Runtime error if value doesn't exist

At the bottom of the IDE, there are tabs for "Playground" and "Transcript".

Ordered Collection: Removing

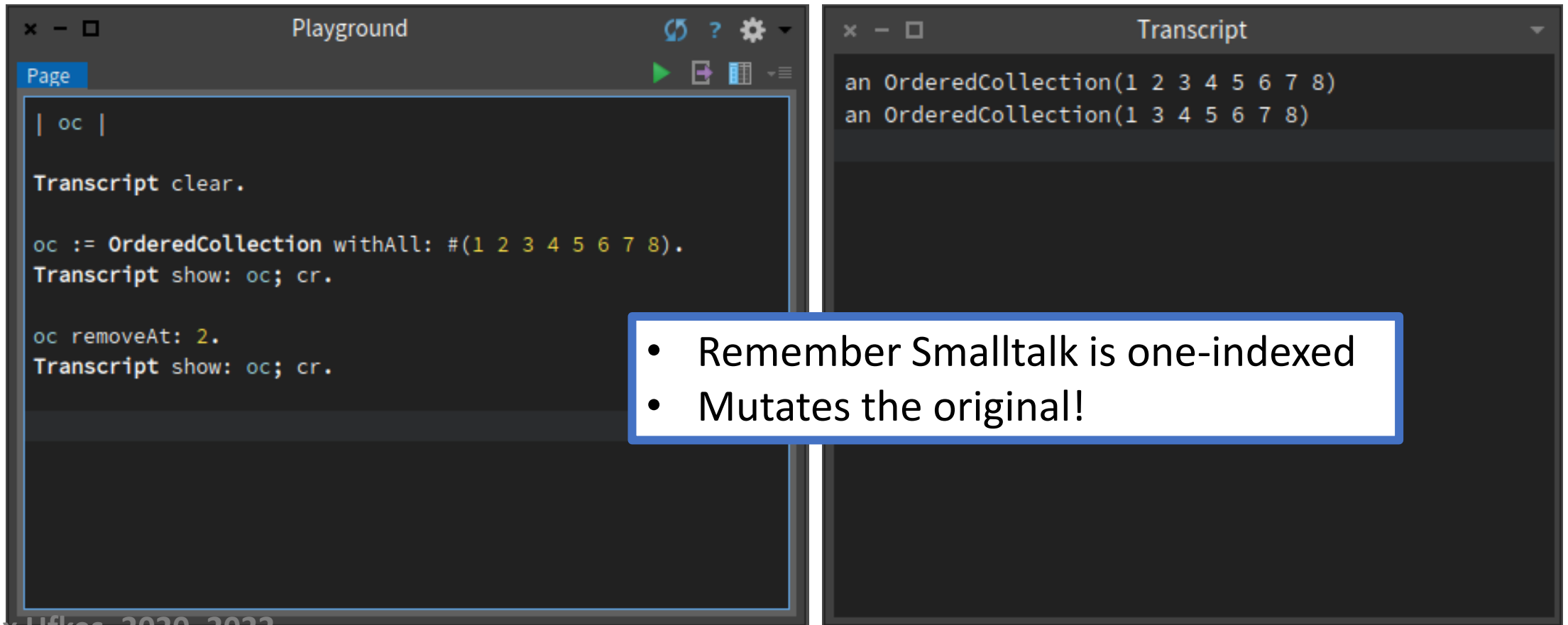
The screenshot displays the Pharo IDE interface. On the left, the 'Playground' window contains the following Smalltalk code:

```
| oc |  
oc := OrderedCollection new.  
oc add: 0; add: 2; add: 7; add: 99.  
Transcript clear; show: oc; cr.  
  
oc removeAll: #(2 99).  
Transcript show: oc; cr.
```

The number **99** in the second line of the second block is circled in red. A red arrow points from this circle to the error message in the 'Transcript' window. The error message reads: **NotFound: 99 not found in OrderedCollection**. Below the error message, the 'Stack' window shows the call stack, with the top frame being `OrderedCollection(Collection) errorNotFound:`. The 'Source' window shows the definition of `errorNotFound:` as `anObject` and the message `signalFor: anObject`. The 'Variables' window at the bottom shows the current state of the environment.

Remove by Index

Can use `removeAt:`



The image shows two side-by-side windows from a Smalltalk environment. The left window, titled "Playground", contains the following code:

```
| oc |  
  
Transcript clear.  
  
oc := OrderedCollection withAll: #(1 2 3 4 5 6 7 8).  
Transcript show: oc; cr.  
  
oc removeAt: 2.  
Transcript show: oc; cr.
```

The right window, titled "Transcript", shows the output of the code:

```
an OrderedCollection(1 2 3 4 5 6 7 8)  
an OrderedCollection(1 3 4 5 6 7 8)
```

A callout box with a blue border and white background is overlaid on the bottom right of the transcript window, containing the following text:

- Remember Smalltalk is one-indexed
- Mutates the original!

With Arrays?

No! Arrays cannot change size once created:

The screenshot shows a Smalltalk playground interface. The main editor contains the following code:

```
| arr |  
  
Transcript clear.  
  
arr := #(1 2 3 4 5 6 7 8).  
Transcript show: arr; cr.  
  
arr removeAt: 2.  
Transcript show: arr; cr.
```

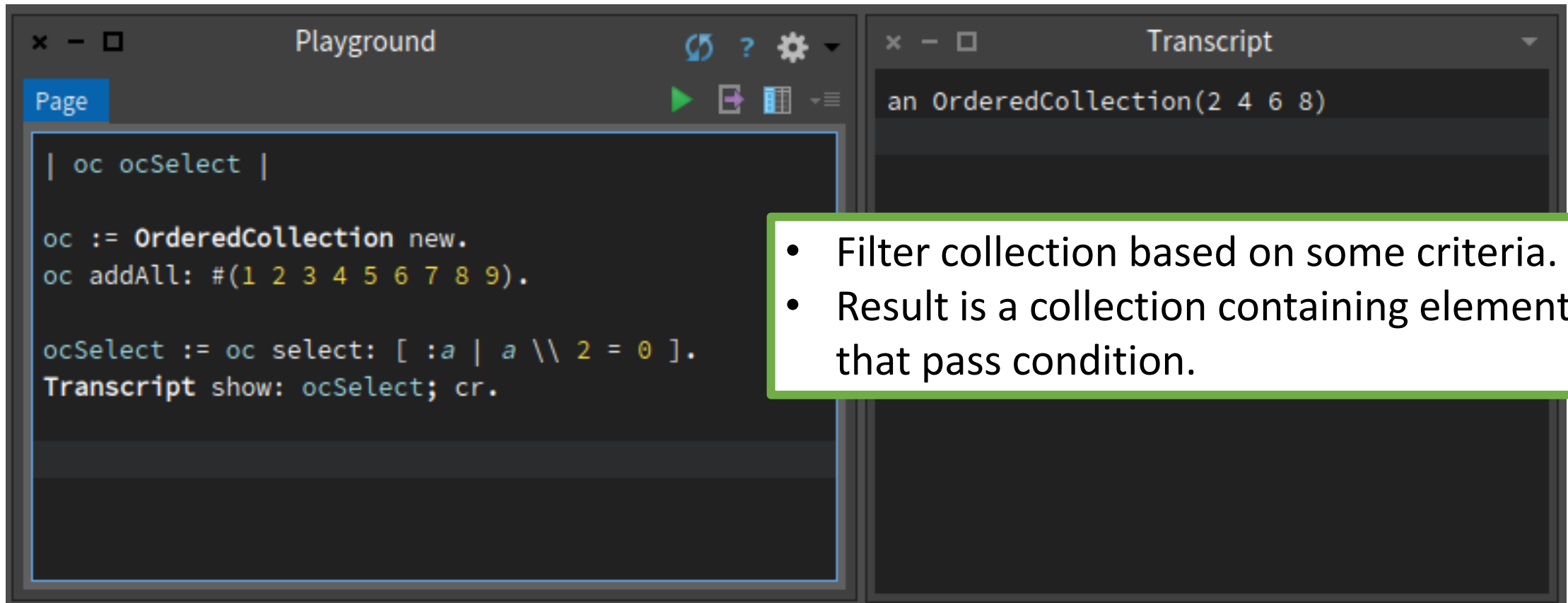
The error message, highlighted with a red box, reads: "Instance of Array did not understand #removeAt:". Below the error message is a stack trace:

Stack	Message
UndefinedObject	DoIt
OpalCompiler	evaluate
RubSmalltalkEditor	evaluate:andDo:
RubSmalltalkEditor	highlightEvaluateAndDo:
GLMMorphicPharoScriptRenderer(GLMMorphicPharoCc	actOnHighlightAndEvaluate: [textMor
RubEditingArea(RubAbstractTextArea)	handleEdit:

The source code in the bottom panel shows the first two lines of the code from the main editor:

```
DoIt  
| arr |  
Transcript clear.  
arr := #(1 2 3 4 5 6 7 8).
```


Ordered Collection: select:



The image shows a screenshot of a Scala Playground interface. On the left, the 'Playground' window contains the following code:

```
| oc ocSelect |  
  
oc := OrderedCollection new.  
oc addAll: #(1 2 3 4 5 6 7 8 9).  
  
ocSelect := oc select: [ :a | a \% 2 = 0 ].  
Transcript show: ocSelect; cr.
```

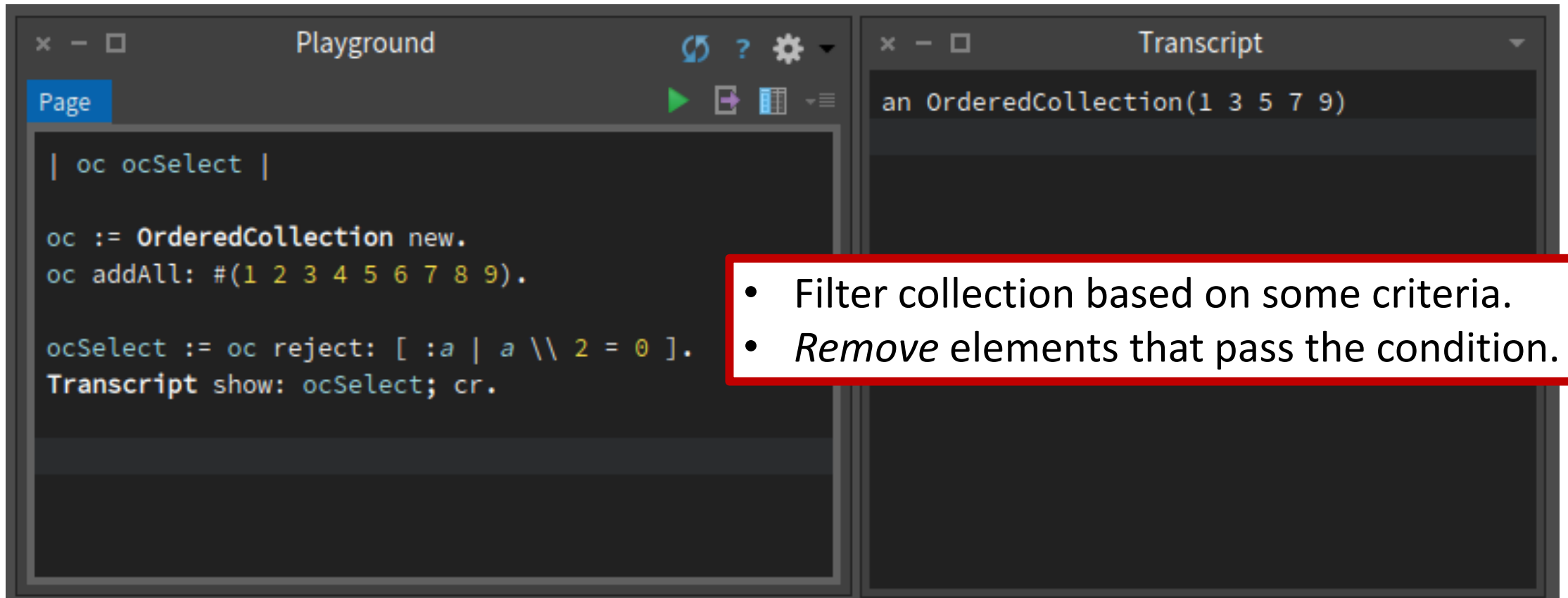
On the right, the 'Transcript' window shows the output:

```
an OrderedCollection(2 4 6 8)
```

A callout box with a green border contains the following text:

- Filter collection based on some criteria.
- Result is a collection containing elements that pass condition.

Ordered Collection: reject:



The screenshot shows a Scala Playground interface with two panes. The left pane, titled "Playground", contains the following code:

```
| oc ocSelect |  
  
oc := OrderedCollection new.  
oc addAll: #(1 2 3 4 5 6 7 8 9).  
  
ocSelect := oc reject: [ :a | a \% 2 = 0 ].  
Transcript show: ocSelect; cr.
```

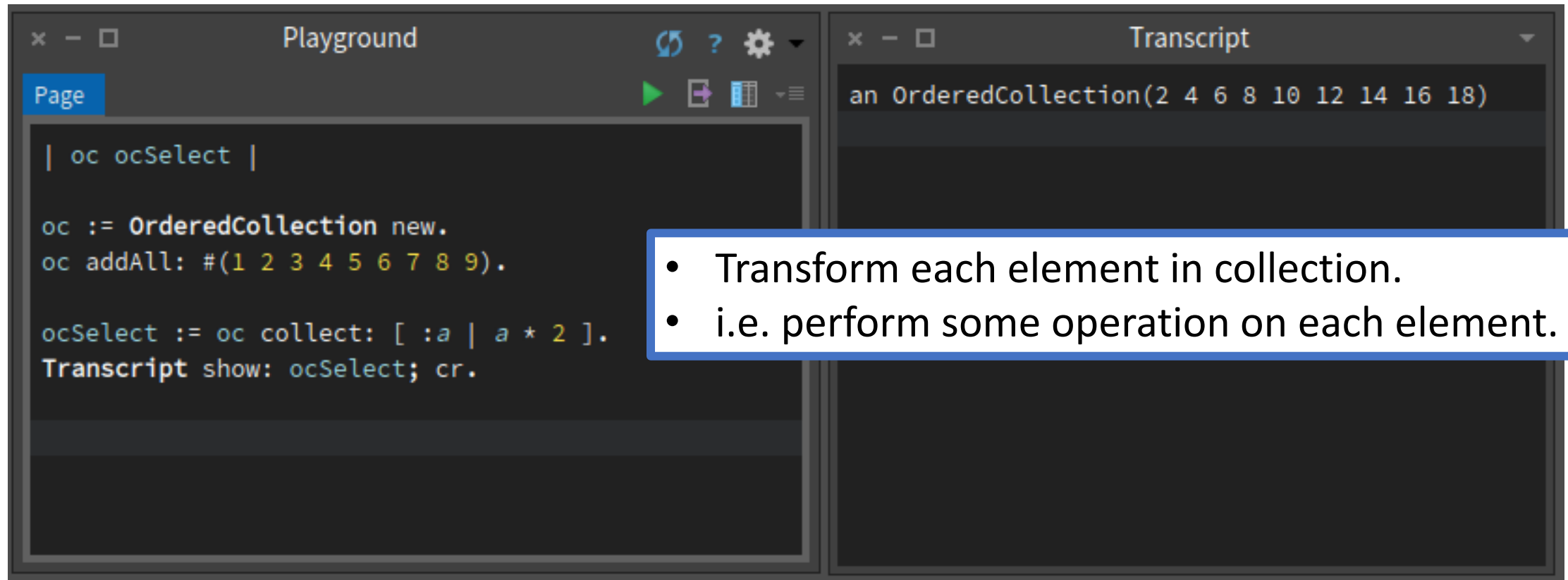
The right pane, titled "Transcript", shows the output of the code:

```
an OrderedCollection(1 3 5 7 9)
```

A red-bordered box highlights the following list of bullet points:

- Filter collection based on some criteria.
- *Remove* elements that pass the condition.

Ordered Collection: collect:



The image shows a code playground interface with two panes. The left pane, titled "Playground", contains the following code:

```
| oc ocSelect |  
  
oc := OrderedCollection new.  
oc addAll: #(1 2 3 4 5 6 7 8 9).  
  
ocSelect := oc collect: [ :a | a * 2 ].  
Transcript show: ocSelect; cr.
```

The right pane, titled "Transcript", shows the output of the code execution:

```
an OrderedCollection(2 4 6 8 10 12 14 16 18)
```

A callout box with a blue border is overlaid on the right side of the playground, containing the following text:

- Transform each element in collection.
- i.e. perform some operation on each element.

Are we mutating the original?

The screenshot shows the Pharo IDE interface. The main window is titled "Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo ...". The menu bar includes "Pharo", "Tools", "System", "Debugging", "Windows", and "Help".

The "Playground" window contains the following code:

```
| oc |  
  
oc := OrderedCollection new.  
oc addAll: #(1 2 3 4 5 6 7 8 9).  
Transcript clear; show: oc; cr.  
  
oc collect: [ :a | a * 2 ].  
Transcript show: oc; cr.  
  
oc := oc collect: [ :a | a * 2 ].  
Transcript show: oc; cr.
```

The "Transcript" window shows the output of the code:

```
an OrderedCollection(1 2 3 4 5 6 7 8 9)  
an OrderedCollection(1 2 3 4 5 6 7 8 9)  
an OrderedCollection(2 4 6 8 10 12 14 16 18)
```

At the bottom of the transcript window, the word "Nope." is written in large blue text.

OrderedCollection

Acts like an expandable array

And More!

```
| b x y sum max |
x := OrderedCollection with: 4 with: 3 with: 2 with: 1.
x := OrderedCollection new.
x add: 3; add: 2; add: 1; add: 4; yourself.
y := x addFirst: 5.
y := x removeFirst.
y := x addLast: 6.
y := x removeLast.
y := x addAll: #(7 8 9).
y := x removeAll: #(7 8 9).
x at: 2 put: 3.
y := x remove: 5 ifAbsent: [].
b := x isEmpty.
y := x size.
y := x at: 2.
y := x first.
y := x last.
b := x includes: 5.
y := x copyFrom: 2 to: 3.
y := x indexOf: 3 ifAbsent: [0].
y := x occurrencesOf: 3.
x do: [:a | Transcript show: a printString; cr].
b := x conform: [:a | (a >= 1) & (a <= 4)].
y := x select: [:a | a > 2].
y := x reject: [:a | a < 2].
y := x collect: [:a | a + a].
y := x detect: [:a | a > 3] ifNone: [].
sum := 0. x do: [:a | sum := sum + a]. sum.
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)].
sum := x inject: 0 into: [:a :c | a + c].
max := x inject: 0 into: [:a :c | (a > c)
    ifTrue: [a]
    ifFalse: [c]].
y := x shuffled.
y := x asArray
```

"create collection with up to 4 elements"
"allocate collection"
"add element to collection"
"add element at beginning of collection"
"remove first element in collection"
"add element at end of collection"
"remove last element in collection"
"add multiple elements to collection"
"remove multiple elements from collection"
"set element at index"
"remove element from collection"
"test if empty"
"number of elements"
"retrieve element at index"
"retrieve first element in collection"
"retrieve last element in collection"
"test if element is in collection"
"subcollection"
"first position of element within collection"
"number of times object in collection"
"iterate over the collection"
"test if all elements meet condition"
"return collection of elements that pass test"
"return collection of elements that fail test"
"transform each element for new collection"
"find position of first element that passes test"
"sum elements"
"sum elements"
"sum elements"
"find max element in collection"

"randomly shuffle collection"
"convert to array"

Sorted Collection

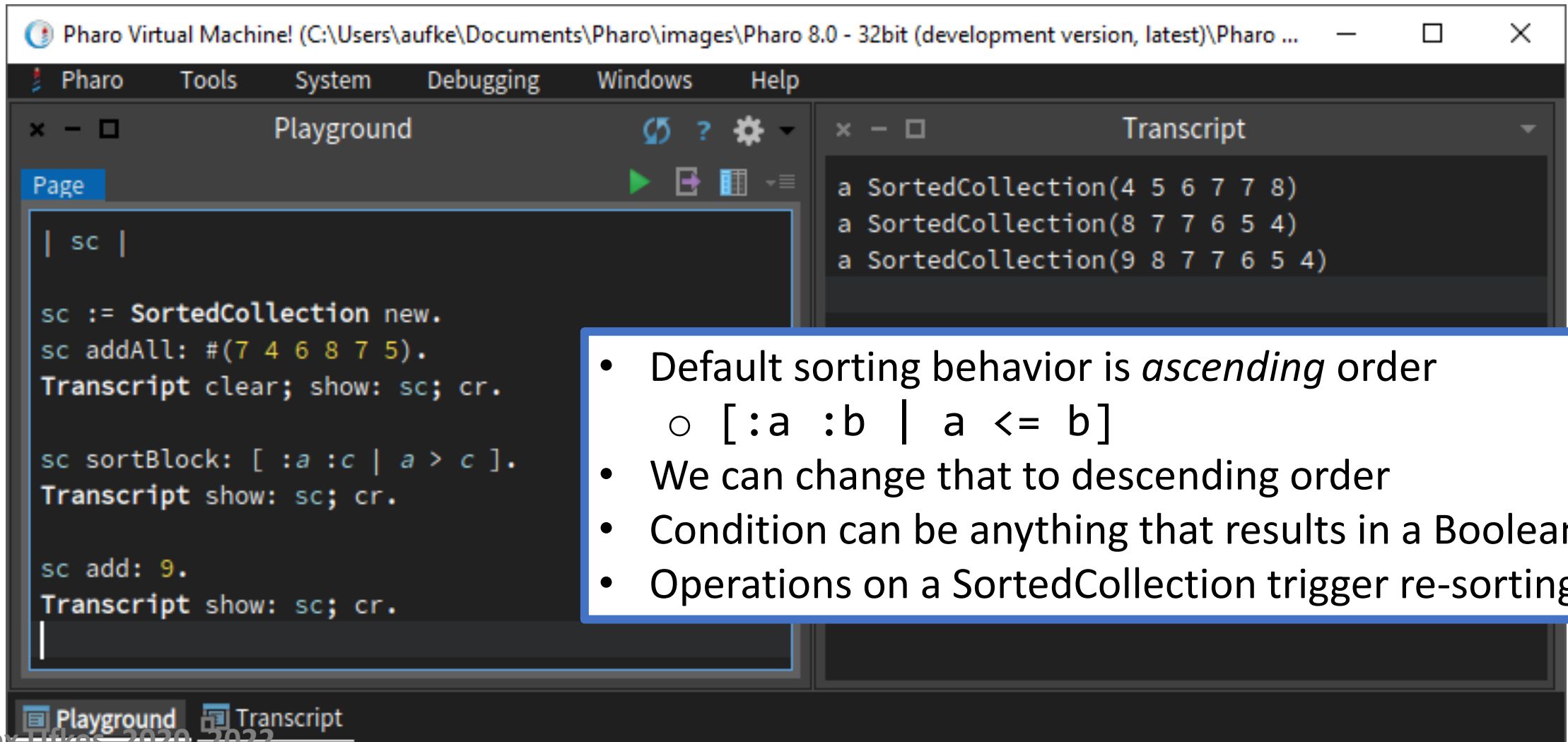
Similar to an `OrderedCollection`, but, you know, sorted.

Operations are all very similar to `OrderedCollection`, but here we can specify a *sorting criteria*:

[:a :c | a <= c]

- A block with two inputs, that implement a Boolean condition.

Sorted Collection



The screenshot shows a Pharo Virtual Machine window with a 'Playground' pane on the left and a 'Transcript' pane on the right. The playground contains the following code:

```
| sc |  
  
sc := SortedCollection new.  
sc addAll: #(7 4 6 8 7 5).  
Transcript clear; show: sc; cr.  
  
sc sortBlock: [ :a :c | a > c ].  
Transcript show: sc; cr.  
  
sc add: 9.  
Transcript show: sc; cr.
```

The transcript pane shows the output of the code:

```
a SortedCollection(4 5 6 7 7 8)  
a SortedCollection(8 7 7 6 5 4)  
a SortedCollection(9 8 7 7 6 5 4)
```

A blue-bordered box highlights the following list of bullet points:

- Default sorting behavior is *ascending* order
 - [:a :b | a <= b]
- We can change that to descending order
- Condition can be anything that results in a Boolean
- Operations on a SortedCollection trigger re-sorting.

Sorted Collection

- Default sorting behavior is *ascending* order
 - `[:a :c | a <= c]`

- This is a block that evaluates to Boolean when executed
- We're defining the condition for **a** appearing before **c** in the sequence
- Ascending order: **a** comes before **c** if **a** is less than or equal to **c**
- If block is true, **a** comes first.

This is just like implementing a **compareTo()** method in Java

Sorted Collection

Silly sorting criteria?

`[:a :b | a \% 2 = 0]`

The image shows a screenshot of a Scala Playground and its Transcript window. The Playground window on the left contains the following code:

```
| sc |
sc := SortedCollection new.
sc := SortedCollection sortBlock: [ :a :c | a \% 2 = 0 ].
sc addAll: #( 1 2 3 4 5 6 7 8 ).

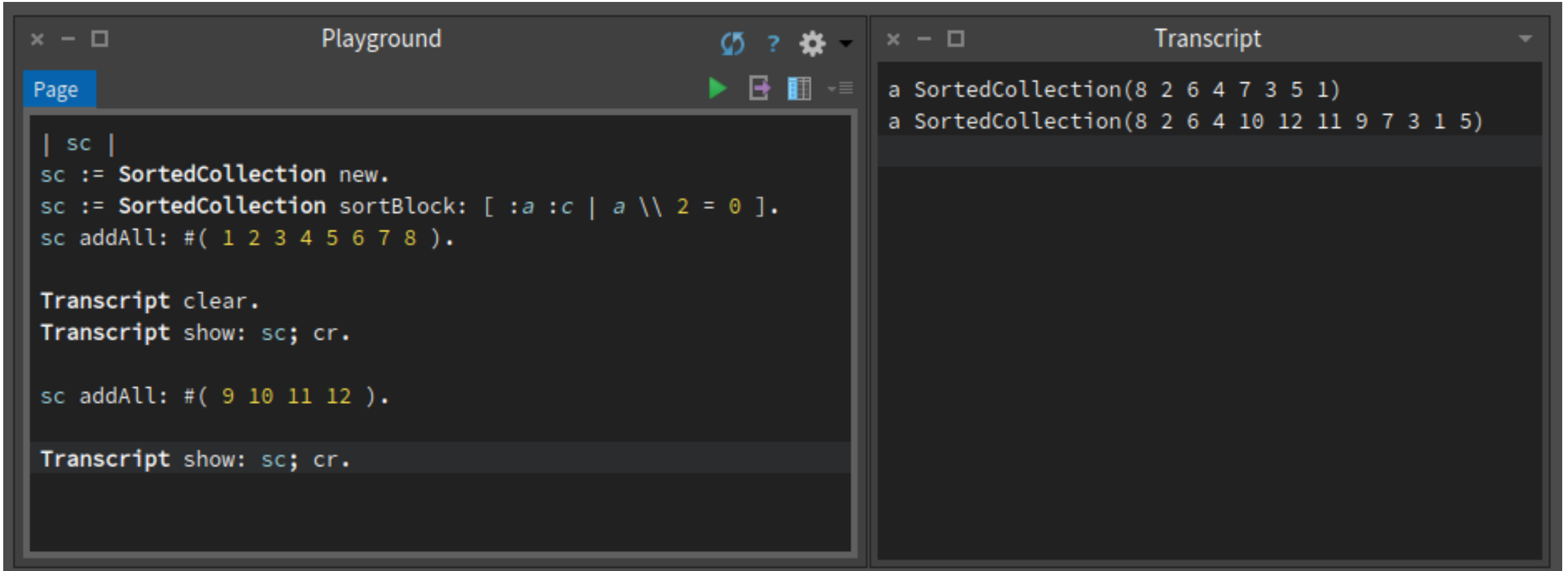
Transcript clear.
Transcript show: sc; cr.
```

The Transcript window on the right shows the output of the code:

```
a SortedCollection(8 2 6 4 7 3 5 1)
```

An orange-bordered box highlights the text "a comes before c if a is even" in the transcript, explaining the sorting criteria used in the code.

Modifying collection triggers re-sorting



The image shows a Scala Playground window on the left and its Transcript window on the right. The Playground window contains the following code:

```
| sc |
sc := SortedCollection new.
sc := SortedCollection sortBlock: [ :a :c | a \\ 2 = 0 ].
sc addAll: #( 1 2 3 4 5 6 7 8 ).

Transcript clear.
Transcript show: sc; cr.

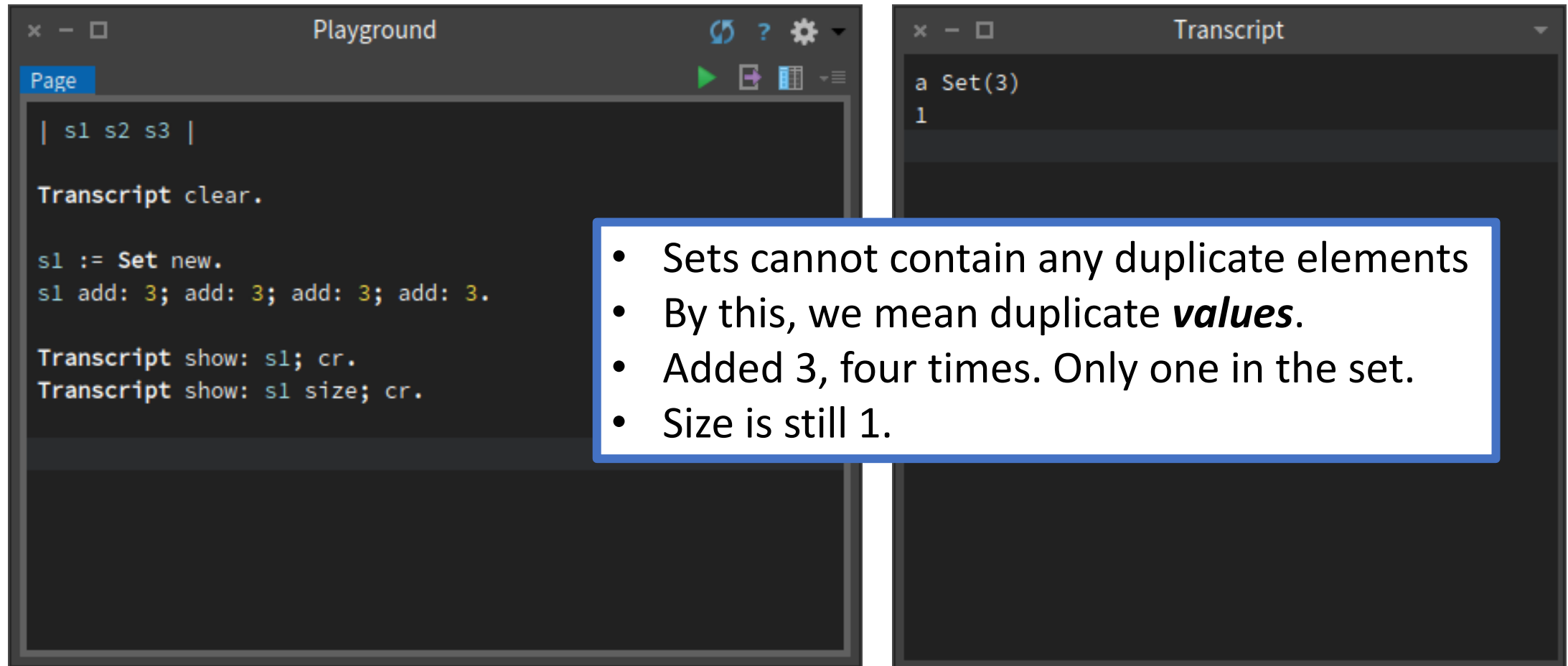
sc addAll: #( 9 10 11 12 ).

Transcript show: sc; cr.
```

The Transcript window shows the output of the code:

```
a SortedCollection(8 2 6 4 7 3 5 1)
a SortedCollection(8 2 6 4 10 12 11 9 7 3 1 5)
```

Collections: Sets



The image shows two side-by-side windows from a Scala Playground. The left window, titled 'Playground', contains the following code:

```
| s1 s2 s3 |  
  
Transcript clear.  
  
s1 := Set new.  
s1 add: 3; add: 3; add: 3; add: 3.  
  
Transcript show: s1; cr.  
Transcript show: s1 size; cr.
```

The right window, titled 'Transcript', shows the output of the code:

```
a Set(3)  
1
```

A callout box with a blue border contains the following list of bullet points:

- Sets cannot contain any duplicate elements
- By this, we mean duplicate *values*.
- Added 3, four times. Only one in the set.
- Size is still 1.

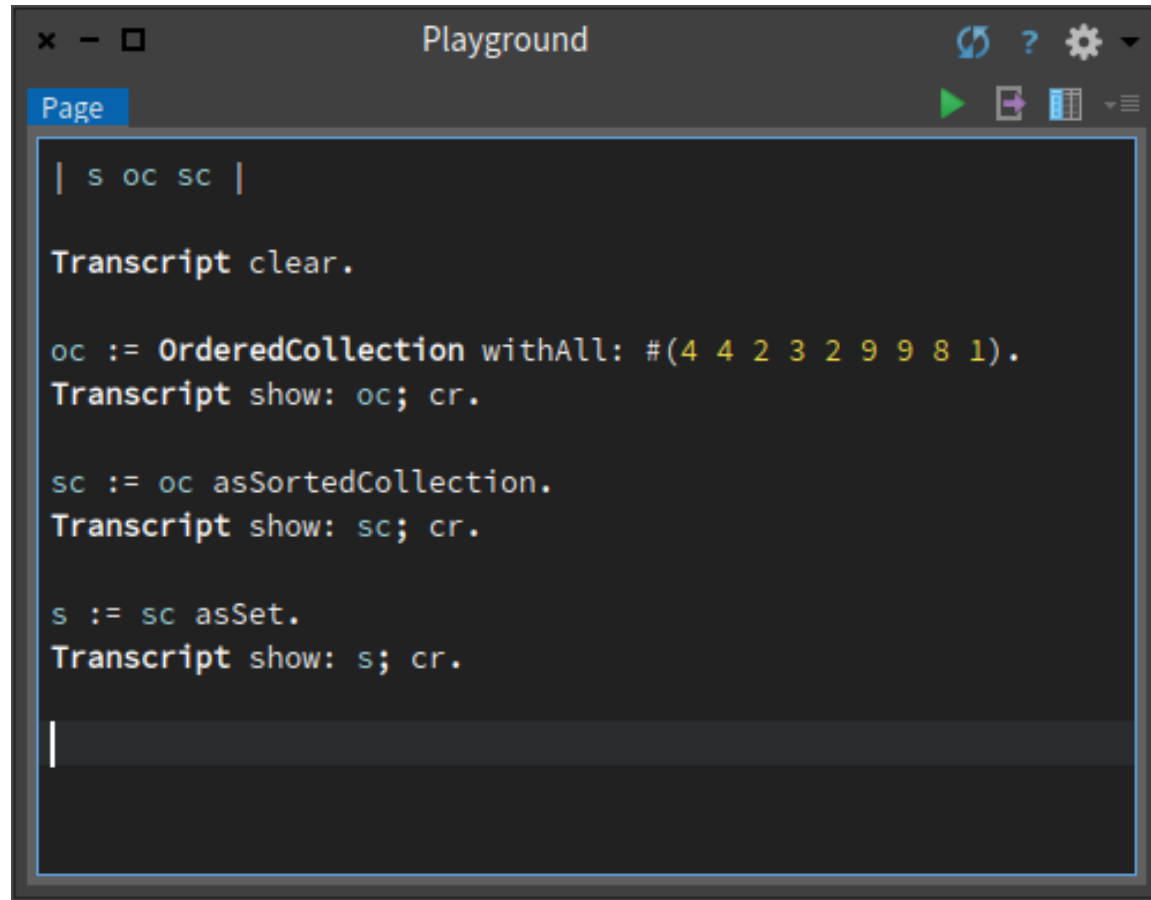
Collections: Sets

```
Playground
Page
| s1 s2 s3 |
Transcript clear.
s1 := Set new.
s1 add: 'Hello'; add: ('Hel', 'lo').
Transcript show: s1; cr.
Transcript show: s1 size; cr.
```

- Different ByteString *objects*
- Same *value*.

```
Transcript
a Set('Hello')
1
```

Collections: Sets



```
Playground
Page
| s oc sc |
Transcript clear.
oc := OrderedCollection withAll: #(4 4 2 3 2 9 9 8 1).
Transcript show: oc; cr.

sc := oc asSortedCollection.
Transcript show: sc; cr.

s := sc asSet.
Transcript show: s; cr.
```

- We can convert back and forth between collection types.
- `asSet`, `asSortedCollection`, etc
- `asInteger`, `asArray`, lots of coercion messages.

Collections: Sets

```
Playground  
Page  
| s oc sc |  
  
Transcript clear.  
  
oc := OrderedCollection withAll: #(4 4 2 3 2 9 9 8 1).  
Transcript show: oc; cr.  
  
sc := oc asSortedCollection.  
Transcript show: sc; cr.  
  
s := sc asSet.  
Transcript show: s; cr.  
  
|
```

```
Transcript  
an OrderedCollection(4 4 2 3 2 9 9 8 1)  
a SortedCollection(1 2 2 3 4 4 8 9 9)  
a Set(1 2 3 4 8 9)
```

Remove by Index?

No! Sets do not have *order*. They cannot be accessed by index:

The screenshot shows a Smalltalk playground interface. The main editor contains the following code:

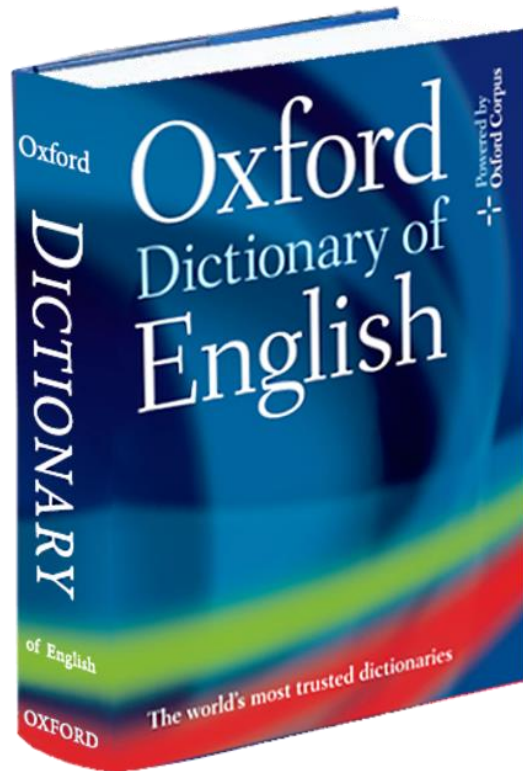
```
| s |  
Transcript clear.  
s := Set withAll: #(1 2 3 4 5 6 7 8).  
Transcript show: oc; cr.  
  
s removeAt: 2.  
Transcript show: oc; cr.
```

The error message, highlighted with a red box, reads: "Instance of Set did not understand #removeAt:". The stack trace shows the following frames:

Class	Method
UndefinedObject	DoIt
OpalCompiler	evaluate
RubSmalltalkEditor	evaluate:andDo:
RubSmalltalkEditor	highlightEvaluateAndDo:
GLMMorphicPharoScriptRenderer(GLMMorphicPharoCc	actOnHighlightAndEvaluate: [textM
RubEditingArea(RubAbstractTextArea)	handleEdit:

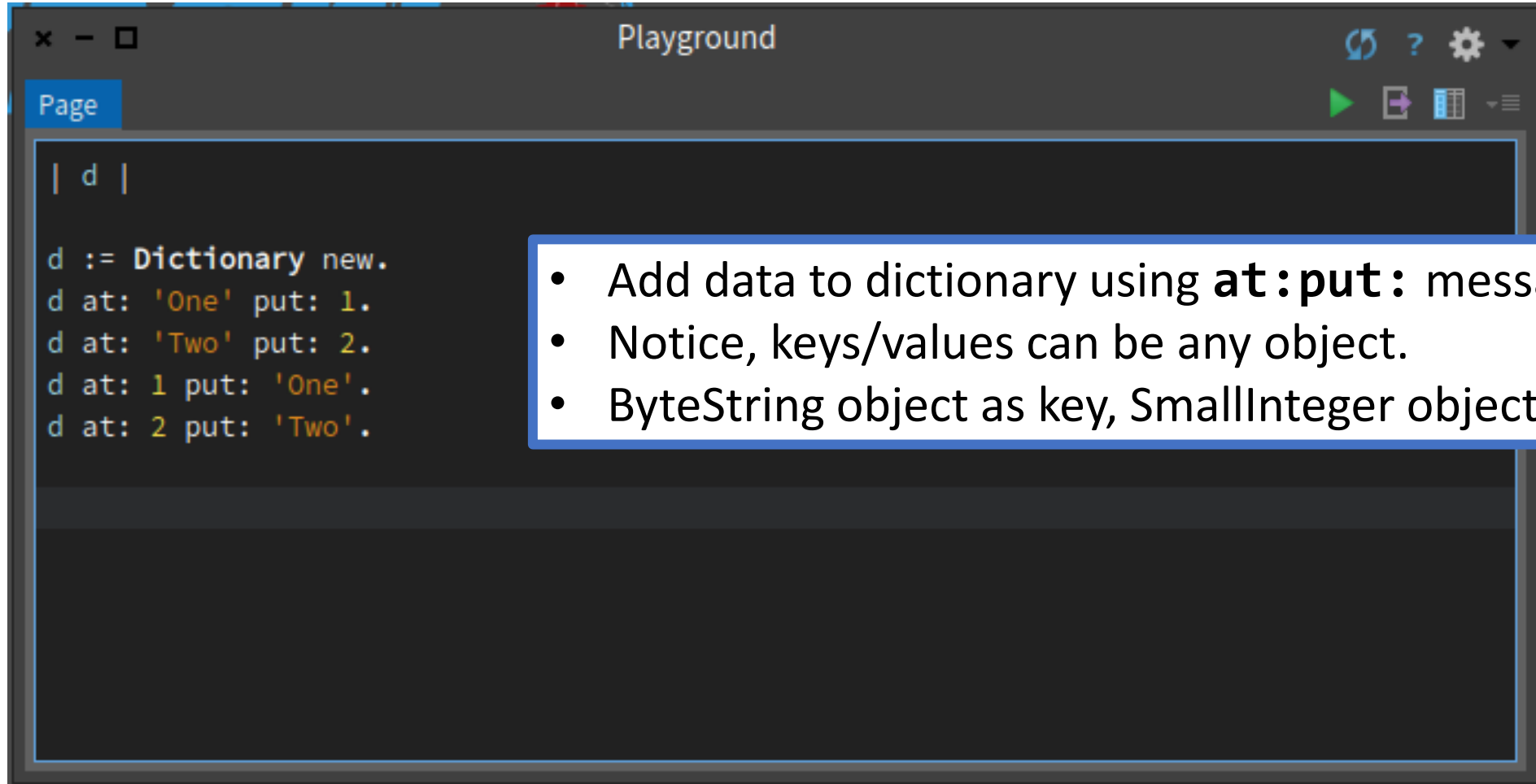
The source code in the bottom panel is identical to the code in the main editor.

Dictionary



- Arrays are indexed with integers; Dictionaries are indexed with *any object at all*.
- You'll know exactly how this works if you learned Python in C/CPS 109
- Store key/value pairs, key can be anything.
- Very powerful, but we give up ordering (hash table implementation)

Dictionary: Adding Entries

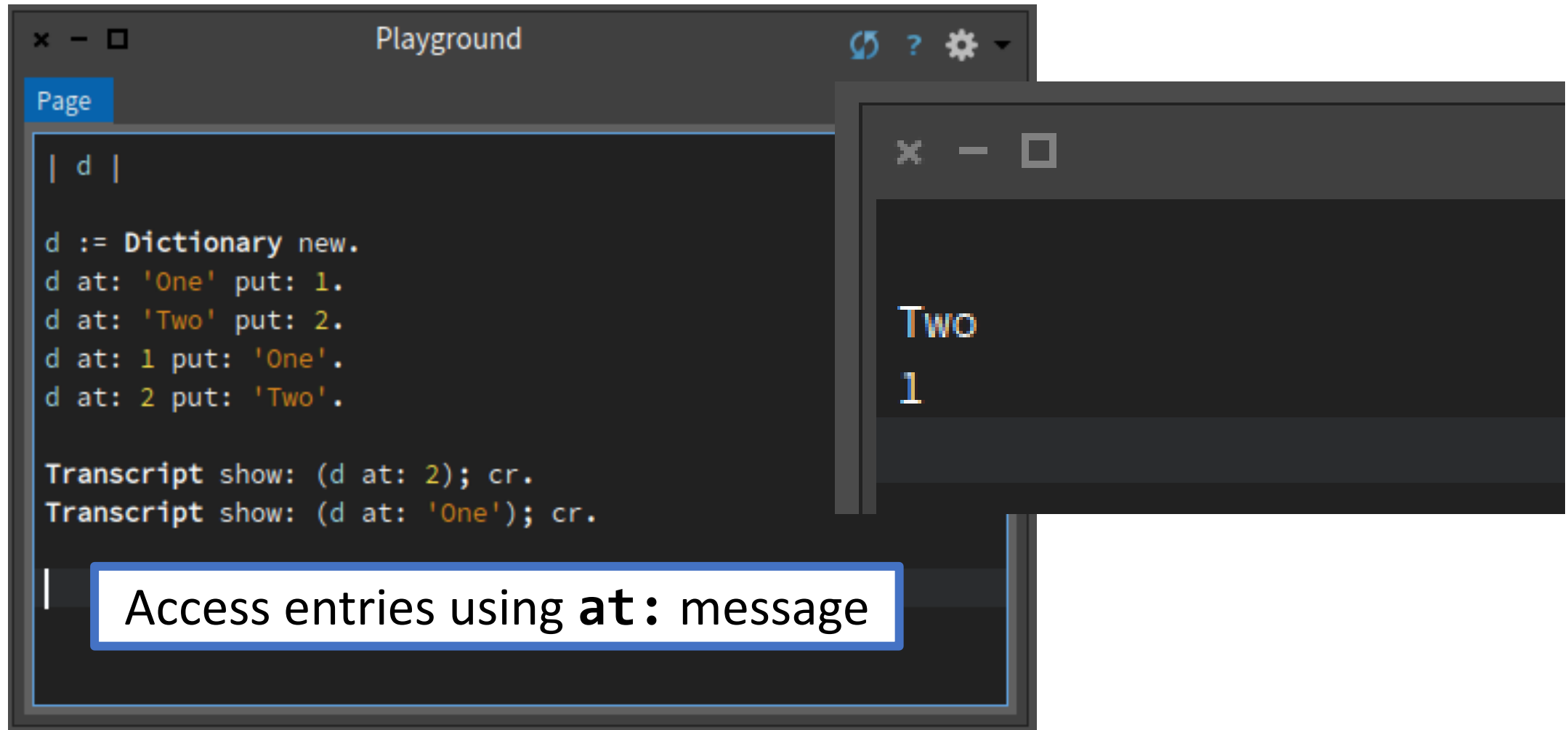


The screenshot shows a Scala Playground window with the following code:

```
| d |  
  
d := Dictionary new.  
d at: 'One' put: 1.  
d at: 'Two' put: 2.  
d at: 1 put: 'One'.  
d at: 2 put: 'Two'.
```

- Add data to dictionary using **at:put:** message
- Notice, keys/values can be any object.
- ByteString object as key, SmallInteger object as key.

Dictionary: Getting Value with Key



The image shows a 'Playground' window with a code editor and a console. The code editor contains the following code:

```
| d |  
  
d := Dictionary new.  
d at: 'One' put: 1.  
d at: 'Two' put: 2.  
d at: 1 put: 'One'.  
d at: 2 put: 'Two'.  
  
Transcript show: (d at: 2); cr.  
Transcript show: (d at: 'One'); cr.
```

The console shows the output of the last two lines of code:

```
TWO  
1
```

A callout box at the bottom of the code editor contains the text: "Access entries using **at:** message".

Dictionary: Key not Found?

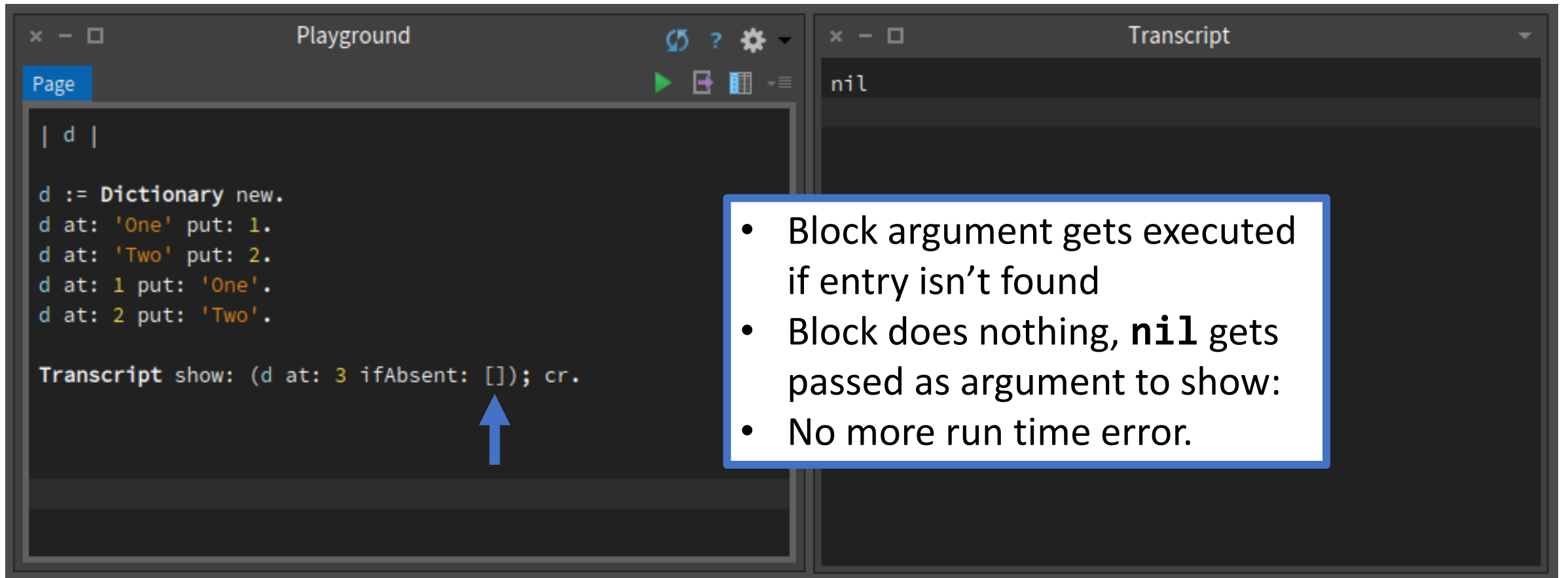
The image shows a 'Playground' window with a code editor and an error dialog box. The code editor contains the following Ruby code:

```
| d |  
  
d := Dictionary new.  
d at: 'One' put: 1.  
d at: 'Two' put: 2.  
d at: 1 put: 'One'.  
d at: 2 put: 'Two'.  
  
Transcript show: (d at: 3); cr.  
Transcript show: (d at: 'Three'); cr.
```

The error dialog box is titled 'KeyNotFound: key 3 not found in Dictionary' and has three buttons: 'Proceed', 'Abandon', and 'Debug'. A red arrow points from the 'Abandon' button to the error message. The error message lists several objects, including 'Dictionary', 'UndefinedObject', 'OpalCompiler', 'RubSmalltalkEditor', and 'GLMMorphicPharoScriptRenderer'. The error message also includes the text 'errorKeyNotFound:' and 'at:'. A red box highlights the two lines of code in the playground that caused the error.

Not very graceful...

Try `at:ifAbsent:` instead

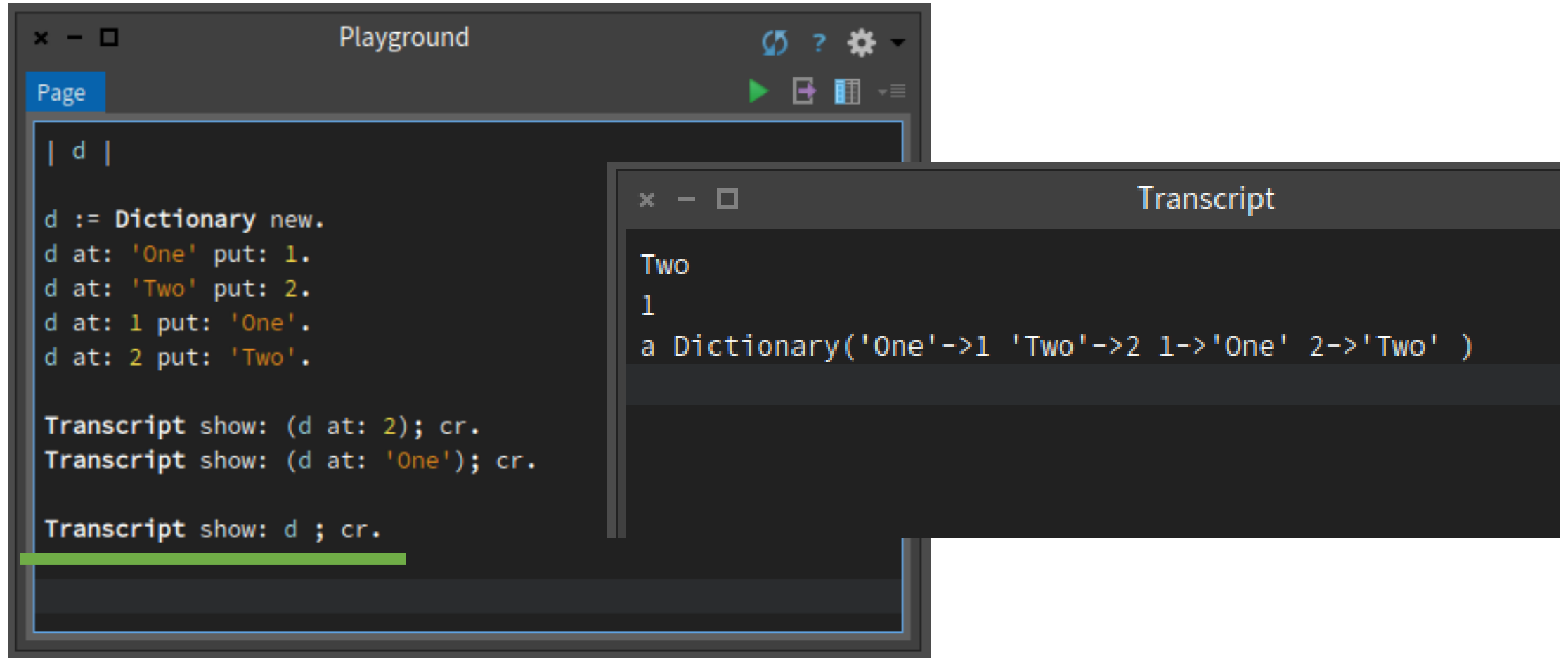


```
| d |  
  
d := Dictionary new.  
d at: 'One' put: 1.  
d at: 'Two' put: 2.  
d at: 1 put: 'One'.  
d at: 2 put: 'Two'.  
  
Transcript show: (d at: 3 ifAbsent: []); cr.
```

```
nil
```

- Block argument gets executed if entry isn't found
- Block does nothing, **nil** gets passed as argument to show:
- No more run time error.

Dictionary: Printing



The image shows a REPL window titled "Playground" with a "Page" tab. The code in the editor is as follows:

```
| d |  
  
d := Dictionary new.  
d at: 'One' put: 1.  
d at: 'Two' put: 2.  
d at: 1 put: 'One'.  
d at: 2 put: 'Two'.  
  
Transcript show: (d at: 2); cr.  
Transcript show: (d at: 'One'); cr.  
  
Transcript show: d ; cr.
```

The output window titled "Transcript" shows the following output:

```
Two  
1  
a Dictionary('One' -> 1 'Two' -> 2 1 -> 'One' 2 -> 'Two' )
```

Identity Dictionary

- When searching for a key in a regular dictionary, the result of the `=` and **hash** messages are used.
- I.e., hash to index into table, compare key to resolve collision
 - *Remember hash tables from C/CPS 305*
- An *Identity Dictionary* uses `==` message, which checks if the key is the same **object**
- Other than that, methods are the same
- Identity dictionary works great with symbols, less so with strings. Why?
 - Identical strings are *not necessarily* the same object!

Symbol Reminder

Symbols:

- # followed by string literal
 - #`'aSymbol'` same as `#aSymbol` (no whitespace, quotes implied)
 - #`'symbol one'` #`'symbol two'`
- Symbols are *globally unique*. Strings are *not*.

Meaning:

- Two *identical* (value) strings can exist as two *different* objects
- For every *unique* symbol value, there can be only *one* object.

Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\...

Pharo Tools System Debugging Windows Help

Playground

```
| a b x y |
Transcript clear.
a := 'Hello'.
b := 'Hel','lo'. "String concatenation"
Transcript show: a = b; cr.
Transcript show: a == b; cr.
x := #Hello
y := (#Hel,#lo) asSymbol.
Transcript show: x = y; cr.
Transcript show: x == y; cr.
```

Transcript

```
true
false
true
true
```

Same value, same object!

- Symbol concatenation returns a string
- Pass the `asSymbol` message to a string to convert it to a symbol.

Playground Transcript

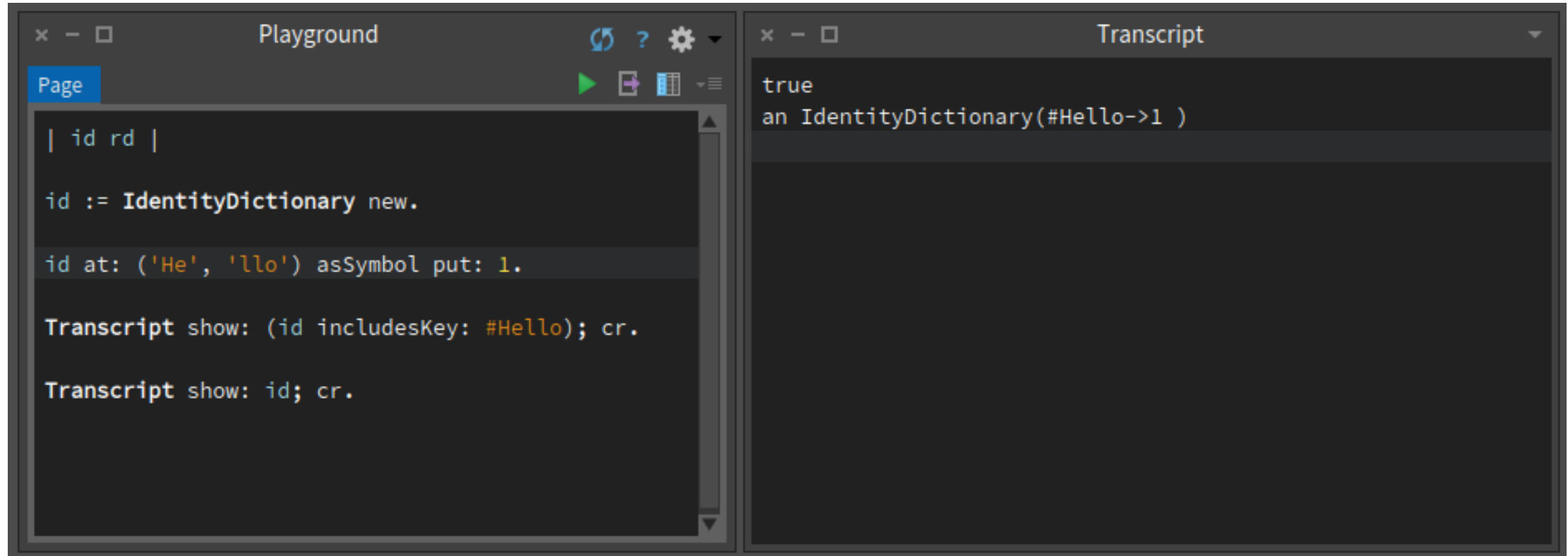
Identity Dictionary

```
Playground  
Page  
| id rd |  
  
id := IdentityDictionary new.  
rd := Dictionary new.  
  
id at: ('Hel', 'lo') put: 1.  
rd at: ('Hel', 'lo') put: 2.  
  
Transcript show: (id includesKey: 'Hello'); cr.  
Transcript show: (rd includesKey: 'Hello'); cr.  
Transcript show: id; cr; show: rd; cr.
```

```
Transcript  
false  
true  
an IdentityDictionary('Hello'->1 )  
a Dictionary('Hello'->2 )
```

Same key *value*,
different *object*

With Symbols?



The image shows a Swift Playground window on the left and its corresponding Transcript window on the right. The Playground window contains the following code:

```
| id rd |  
  
id := IdentityDictionary new.  
  
id at: ('He', 'llo') asSymbol put: 1.  
  
Transcript show: (id includesKey: #Hello); cr.  
  
Transcript show: id; cr.
```

The Transcript window shows the output of the code:

```
true  
an IdentityDictionary(#Hello->1 )
```

Dictionaries

And much more!

```
| b x y sum max |
x := Dictionary new.
x add: #a->4; add: #b->3; add: #c->1; add: #d->2; yourself.
x at: #e put: 3.
b := x isEmpty.
y := x size.
y := x at: #a ifAbsent: [].
y := x keyAtValue: 3 ifAbsent: [].
y := x removeKey: #e ifAbsent: [].
b := x includes: 3.
b := x includesKey: #a.
y := x occurrencesOf: 3.
y := x keys.
y := x values.
x do: [:a | Transcript show: a printString; cr].
x keysDo: [:a | Transcript show: a printString; cr].
x associationsDo: [:a | Transcript show: a printString; cr].
x keysAndValuesDo: [:aKey :aValue | Transcript
  show: aKey printString; space;
  show: aValue printString; cr].
b := x conform: [:a | (a >= 1) & (a <= 4)].
y := x select: [:a | a > 2].
y := x reject: [:a | a < 2].
y := x collect: [:a | a + a].
y := x detect: [:a | a > 3] ifNone: [].
sum := 0. x do: [:a | sum := sum + a]. sum.
sum := x inject: 0 into: [:a :c | a + c].
max := x inject: 0 into: [:a :c | (a > c)
  ifTrue: [a]
  ifFalse: [c]].
y := x asArray.
y := x asOrderedCollection.
y := x asSortedCollection.
y := x asSet.
```

```
"allocate collection"
"add element to collection"
"set element at index"
"test if empty"
"number of elements"
"retrieve element at index"
"retrieve key for given value with error block"
"remove element from collection"
"test if element is in values collection"
"test if element is in keys collection"
"number of times object in collection"
"set of keys"
"bag of values"
"iterate over the values collection"
"iterate over the keys collection"
"iterate over the associations"
"iterate over keys and values"

"test if all elements meet condition"
"return collection of elements that pass test"
"return collection of elements that fail test"
"transform each element for new collection"
"find position of first element that passes test"
"sum elements"
"sum elements"
"find max element in collection"

"convert to array"
"convert to ordered collection"
"convert to sorted collection"
"convert to bag collection"
"convert to set collection"
```

Date

```
| b x y |
x := Date today.
x := Date dateAndTimeNow.
x := Date readFromString: '01/01/2022'.
x := Date newDay: 12 month: #January.
x := Date fromDays: 36000.
y := Date dayOfWeek: #Monday.
y := Date indexOfMonth: #January.
y := Date daysInMonth: 2 forYear: 2022.
y := Date daysInYear: 1996.
y := Date nameOfDay: 1.
y := Date nameOfMonth: 1.
y := Date leapYear: 1996.
y := x weekday.
y := x previous: #Monday.
y := x dayOfMonth.
y := x day.
y := x firstDayOfMonth.
y := x monthName.
y := x monthIndex.
y := x daysInMonth.
y := x year.
y := x daysInYear.
y := x daysLeftInYear.
y := x asSeconds.
y := x addDays: 10.
y := x subtractDays: 10.
y := x subtractDate: (Date today).
y := x printFormat: #(2 1 3 $/).
b := (x <= Date today).
```

Time

```
| b x y |
x := Time now.
x := Time dateAndTimeNow.
x := Time readFromString: '3:4'.
x := Time fromSeconds: (60 * 60).
y := Time millisecondClockValue.
y := Time totalSeconds.
y := x seconds.
y := x minutes.
```

```
Smalltalk removeKey: #CMRDictionary ifAbsent:
```

Internal Stream

```
| b x ios |
ios := ReadStream on: 'Hello read stream'.
ios := ReadStream on: 'Hello read stream' from
[(x := ios nextLine) notNil]
whileTrue: [Transcript show: x; cr].
ios position.
ios position.
x := ios next.
x := ios peek.
x := ios contents.
b := ios atEnd.

ios position.
ios nextPutAll: 'Chris'.
x := ios next.
x := ios peek.
x := ios contents.
b := ios atEnd.
```

There are many other classes for you to explore:

FileStream

```
| b x ios |
ios := FileStream newFileName: 'ios.txt'.
ios nextPut: $H; cr.
ios nextPutAll: 'Hello File'; cr.
'Hello File' printOn: ios.
'Hello File' storeOn: ios.

"total seconds since
"seconds past minute
"minutes past hour"
```

```
y := x asBag.
y := x asSet.
```

Interval

```
| b x y sum max |
x := Interval from: 5 to: 10.
x := 5 to: 10.
x := Interval from: 5 to: 10 by: 2.
x := 5 to: 10 by: 2.
b := x isEmpty.
y := x size.
x includes: 9.
x do: [:k | Transcript show: k printString; cr].
b := x conform: [:a | (a >= 1) & (a <= 4)].
y := x select: [:a | a > 7].
y := x select: [:a | a < 2].
y := x select: [:a | a > 3] ifNone: [].
sum := sum + a]. sum.
) do: [:a | sum := sum + (x at: a)].
[:a :c | a + c].
[:a :c | (a > c)

"create
"create
"test if
"number
"test if
"iterate
"test if
"return
"transfo
"find po
"sum ele
"sum ele
"sum ele
"find ma
"convert
"convert
"convert
"convert
"convert"
```

Associations

```
| x y |
x := #myVar->'hello'.
y := x key.
y := x value.
```

Dictionaries

- Dictionary
- IdentityDictionary: uses identity test (==) rather than 100

Topic 3: Summary

Continuing study of Smalltalk:

- More advanced syntax/semantics:
 - Blocks
 - Control “Structures”
 - Several Smalltalk collections

