C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 2: Control flow & collections in Smalltalk

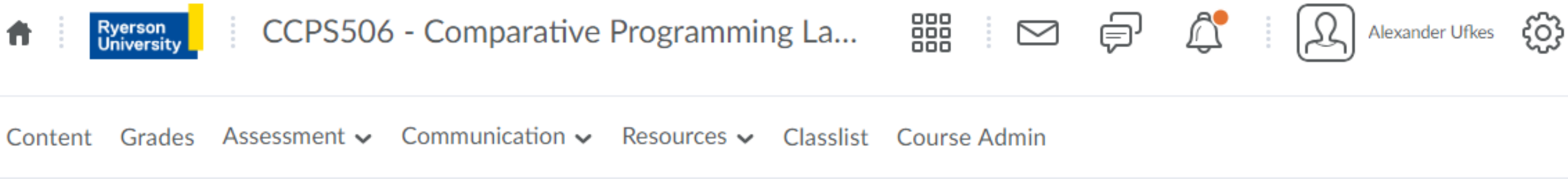
Ryerson University

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.

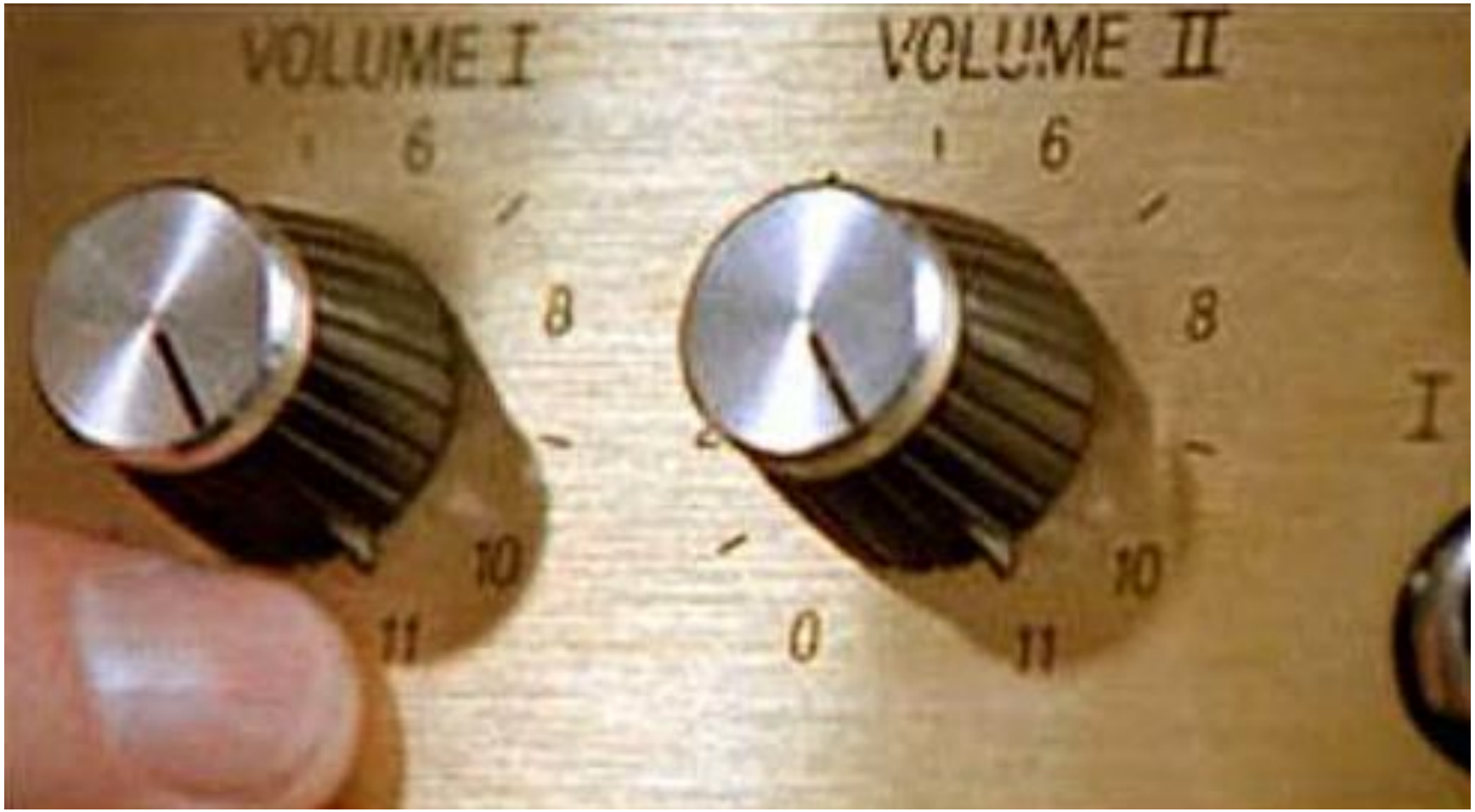
Course Administration (CCPS)



- 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

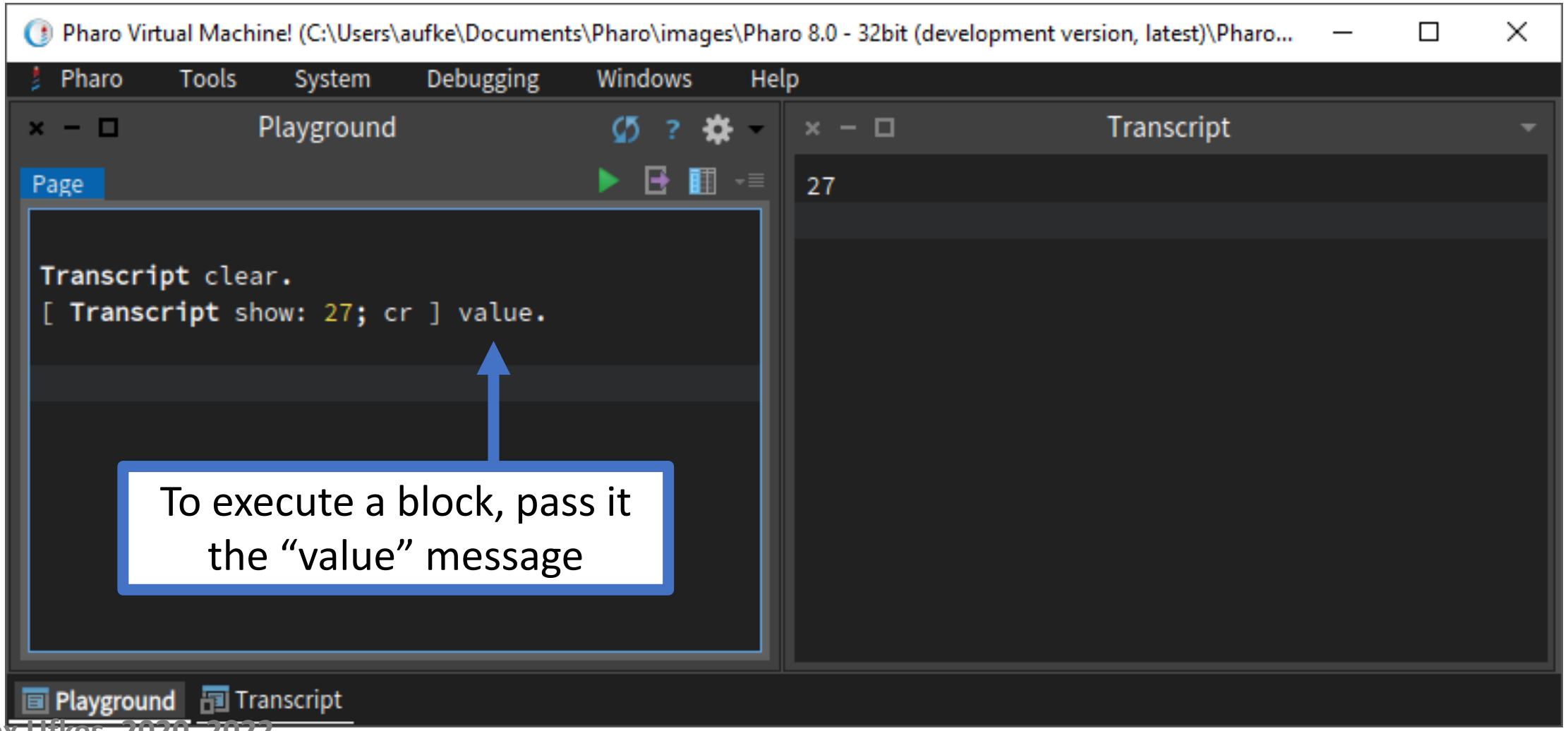
Defined with square brackets [ ]

Within the [ ] is Smalltalk code.

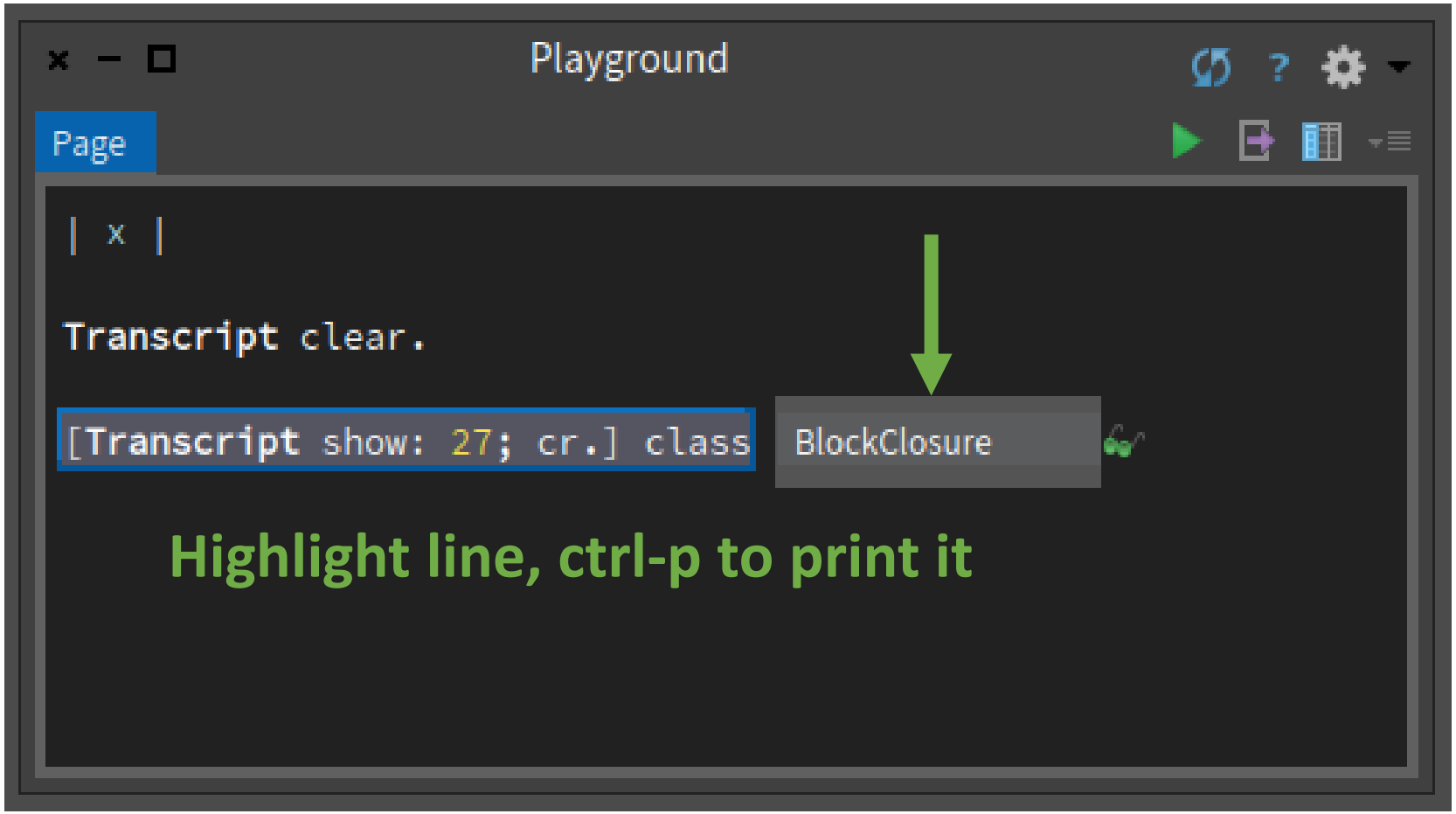
```
[ Transcript show: 27; cr ].
```

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

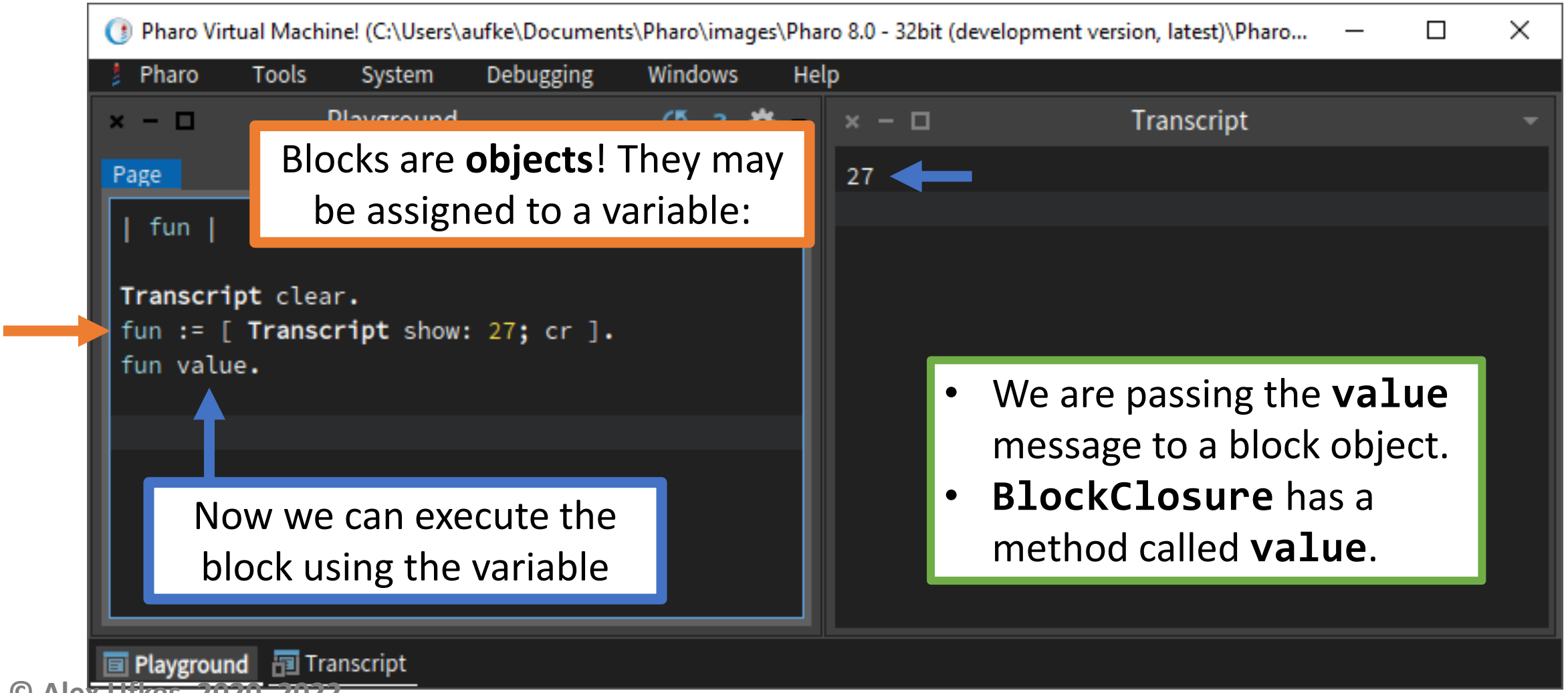
Blocks



Blocks are ***Objects***!



Blocks



Blocks as *Anonymous Functions*

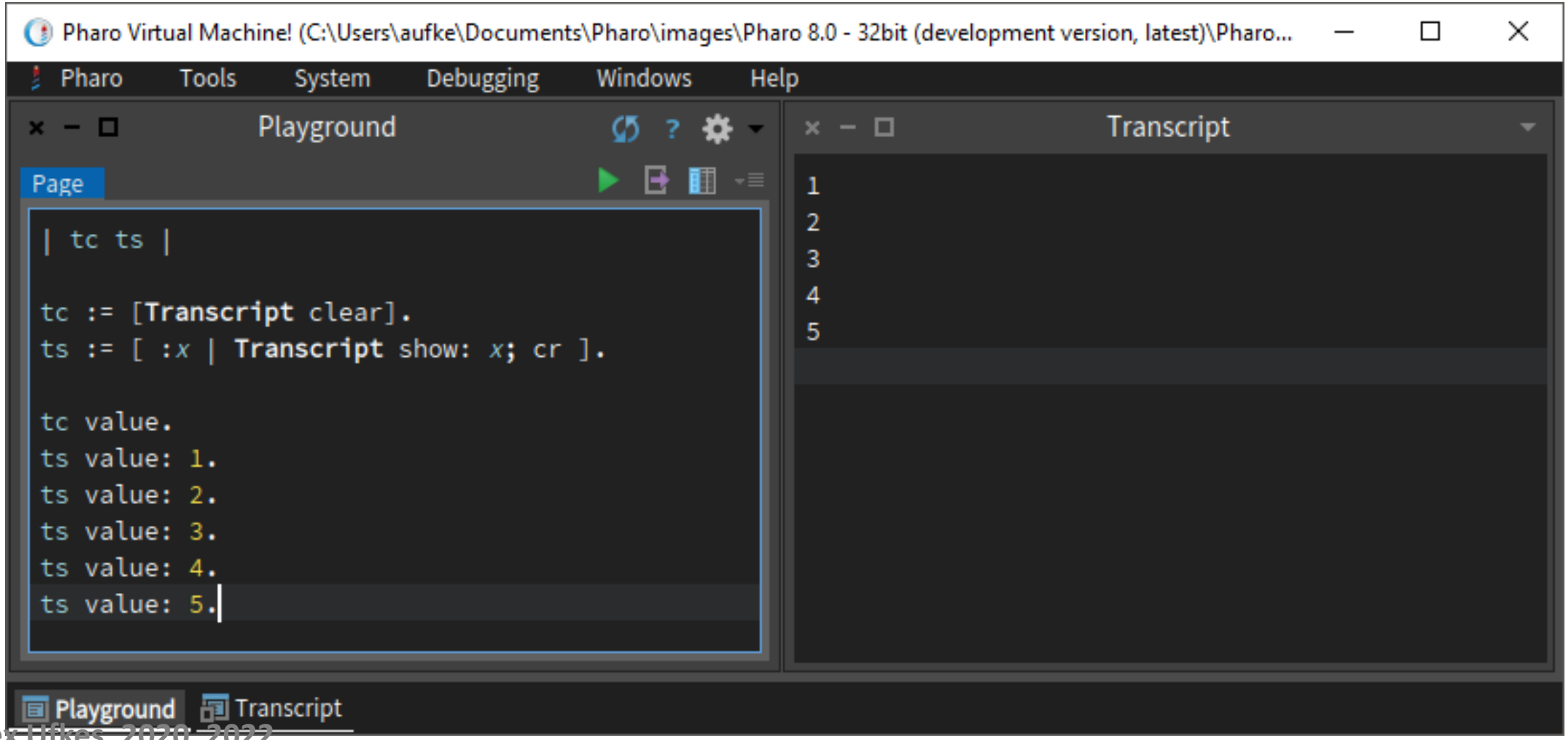
```
func := [ :x | Transcript show: x; cr. ].
```

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

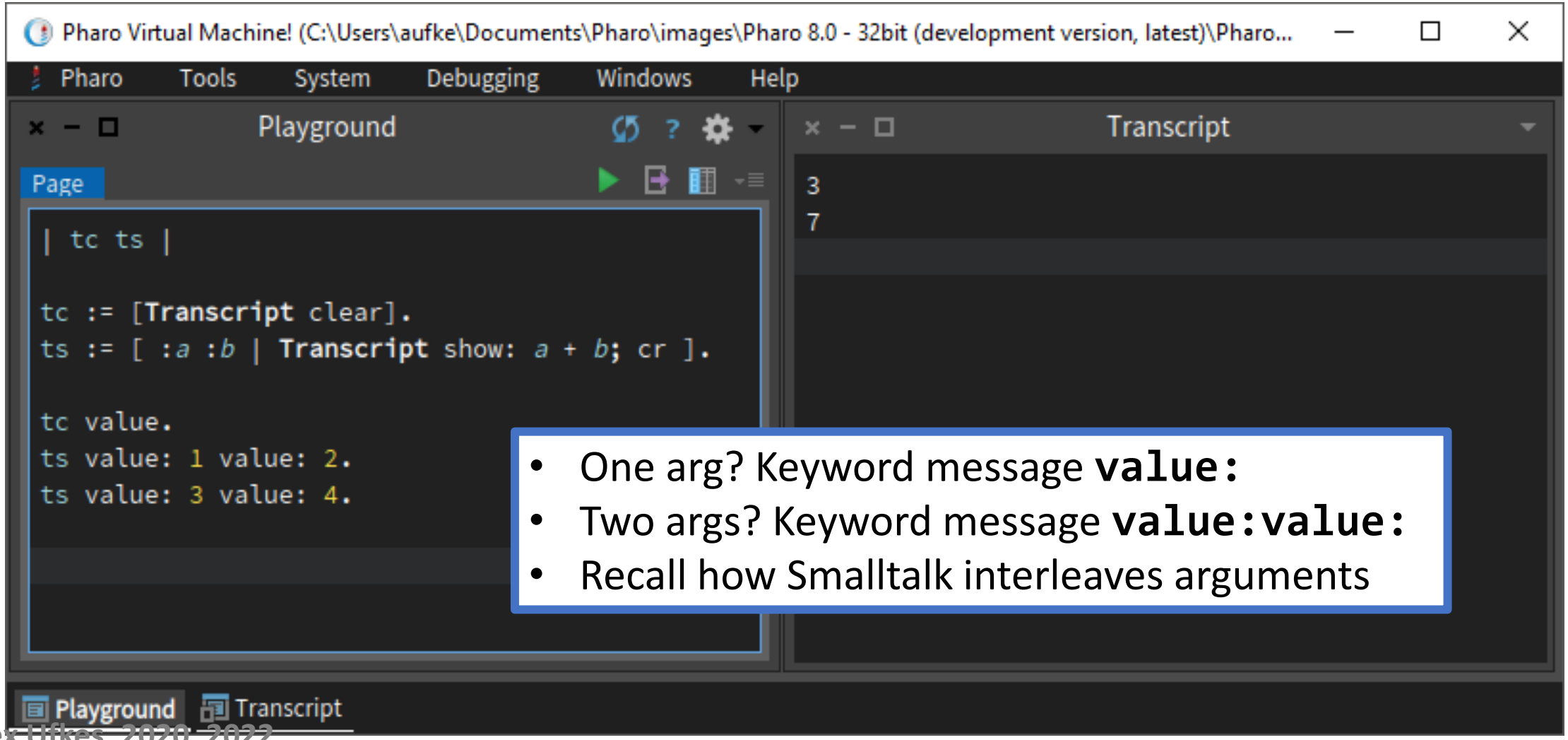
Argument(s) can be passed in when we send the keyword **`value:`** message (as opposed to the unary **`value`** message)

```
func value: 27.
```

Blocks with Arguments

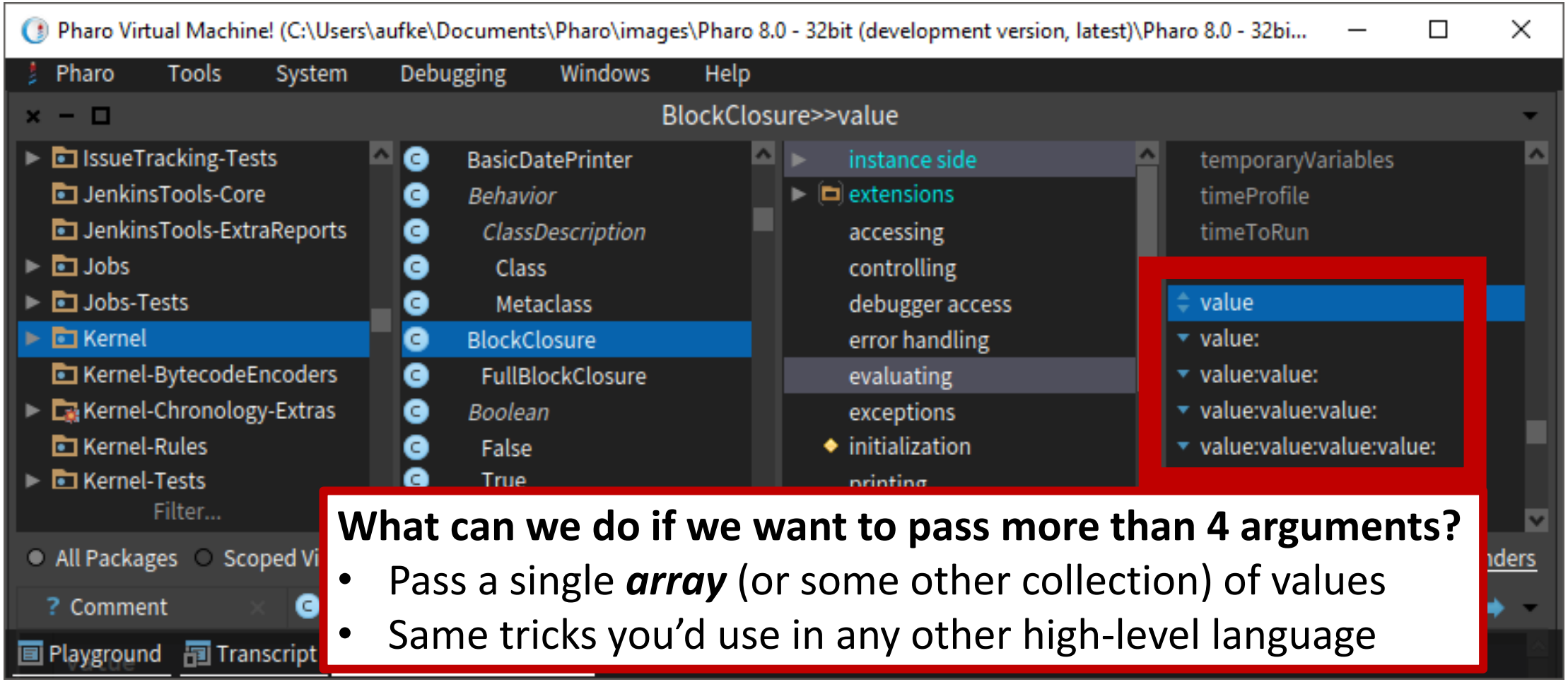


Blocks with Multiple Arguments



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

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



Reminder: Terminator VS Separator

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

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

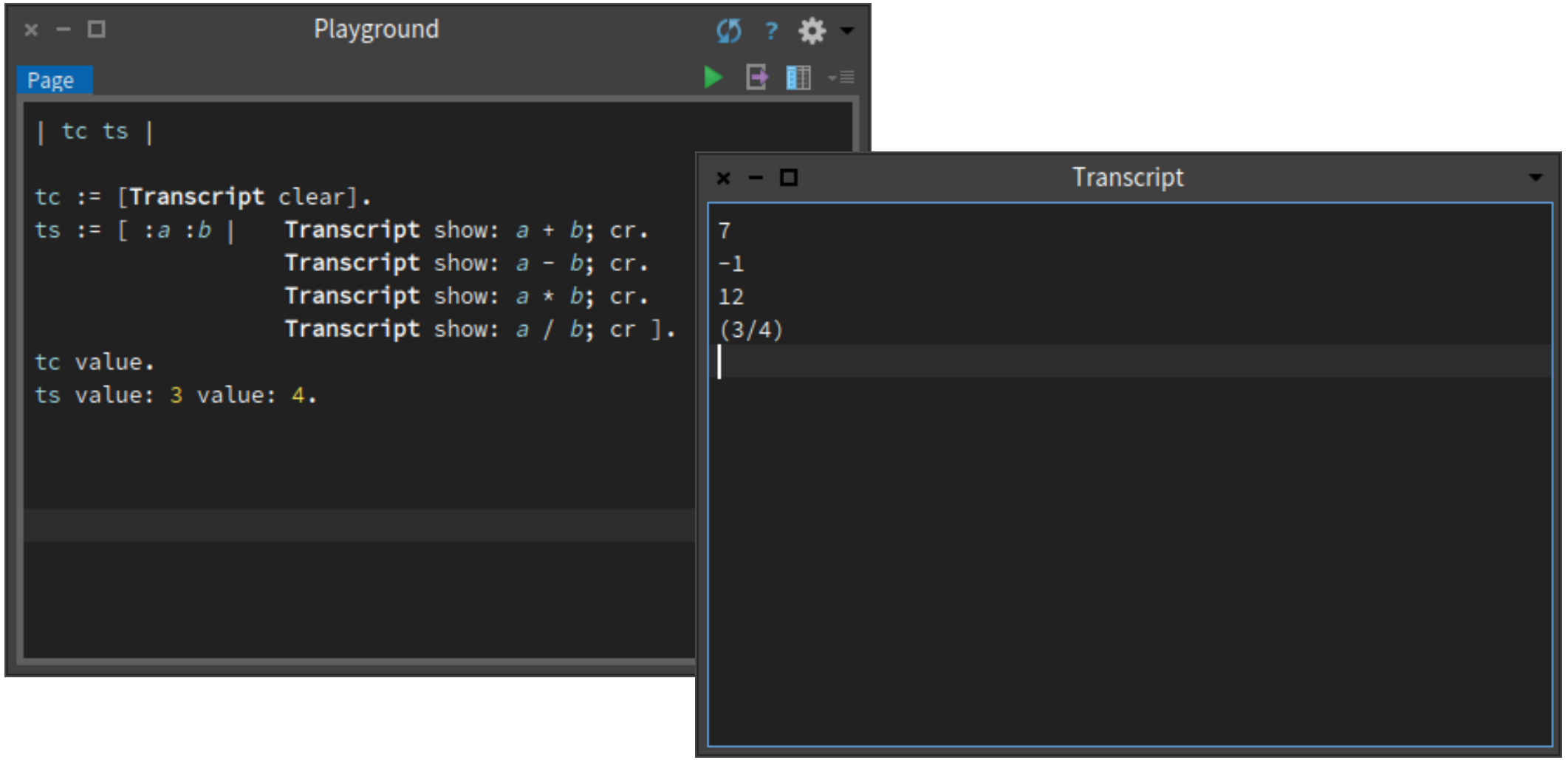
These both work. What's the difference, if any?

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



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!

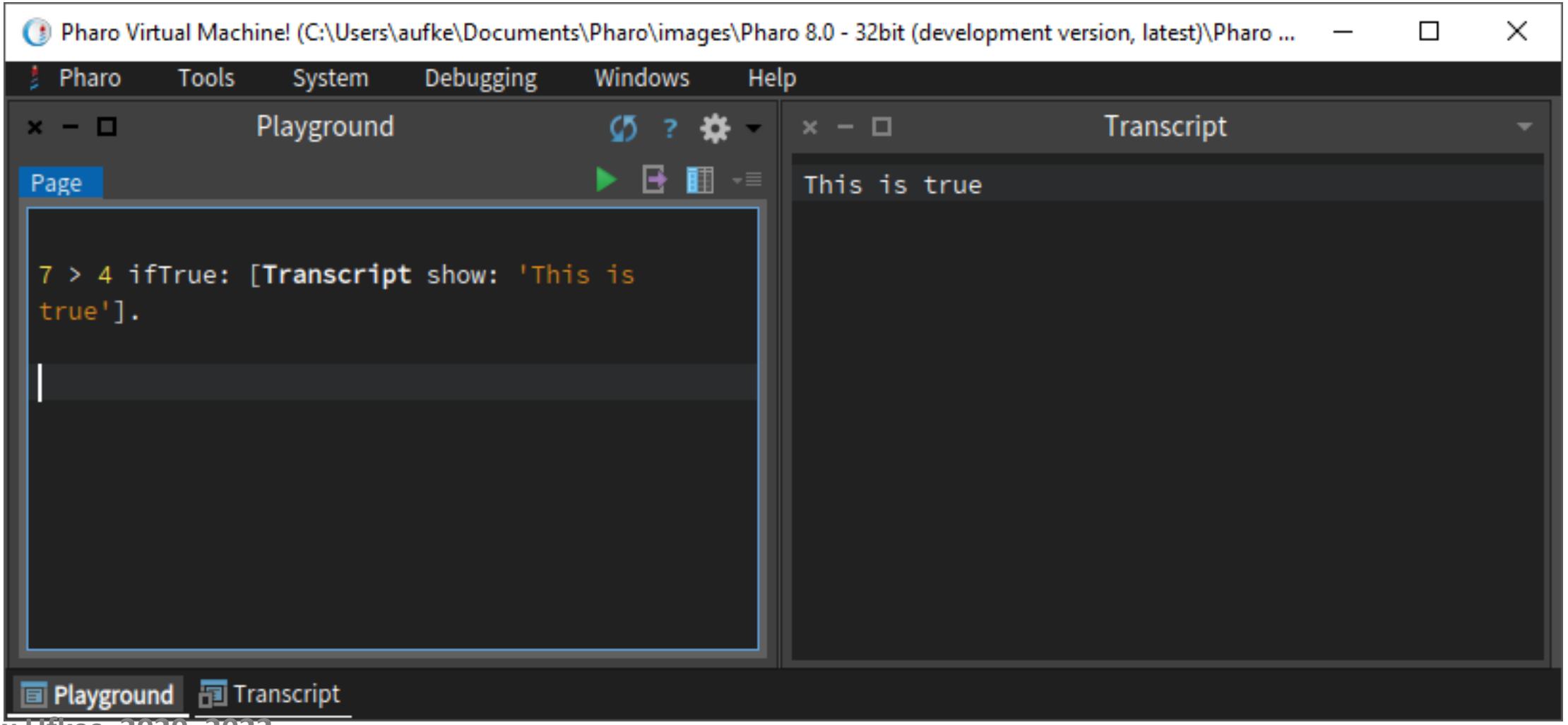
Control Structures: Branching/Selection

They are realized using **blocks** and **message passing**!

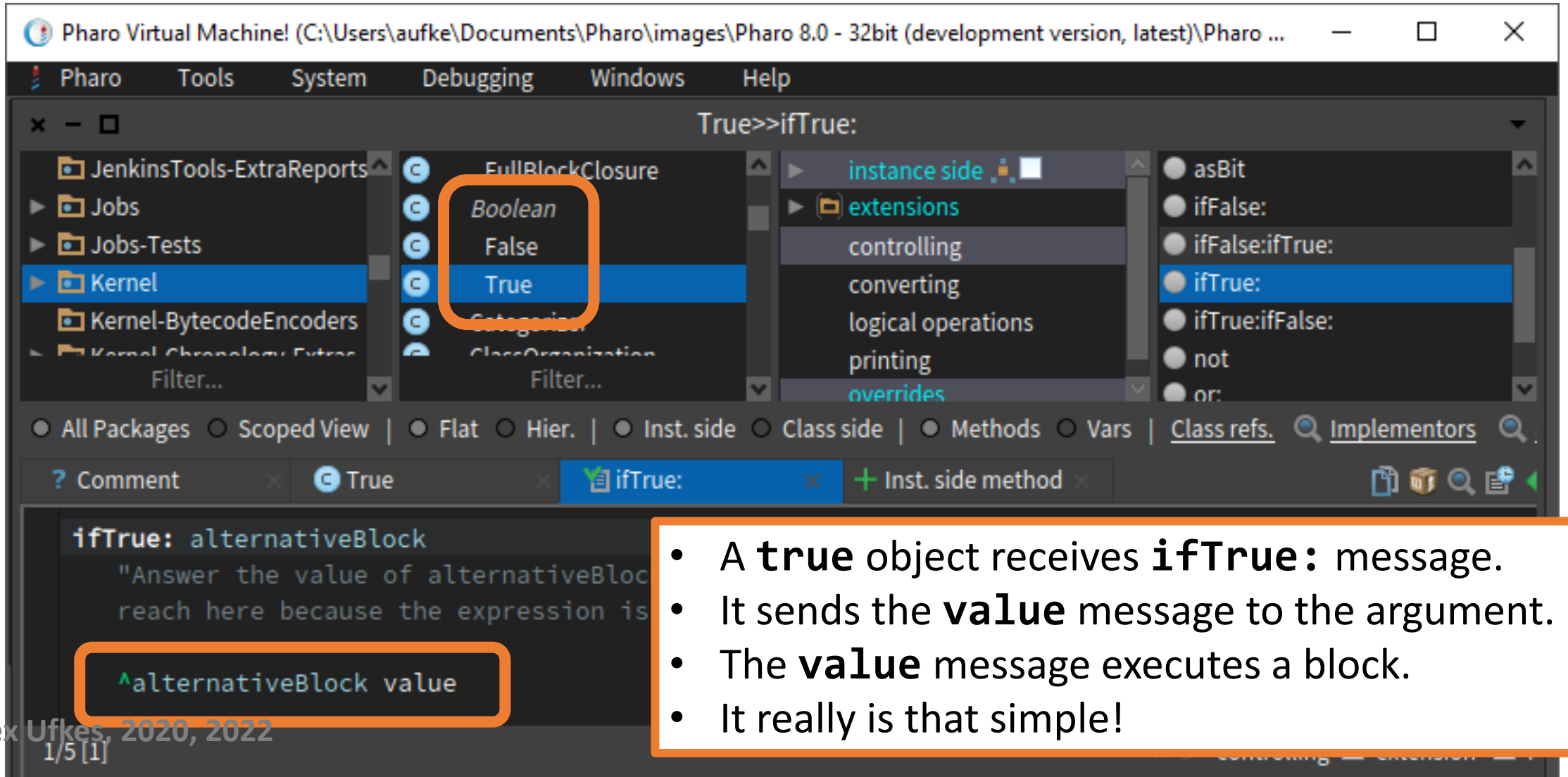
```
7 > 4 ifTrue: [Transcript show: 'This is true'].
```

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'].
```

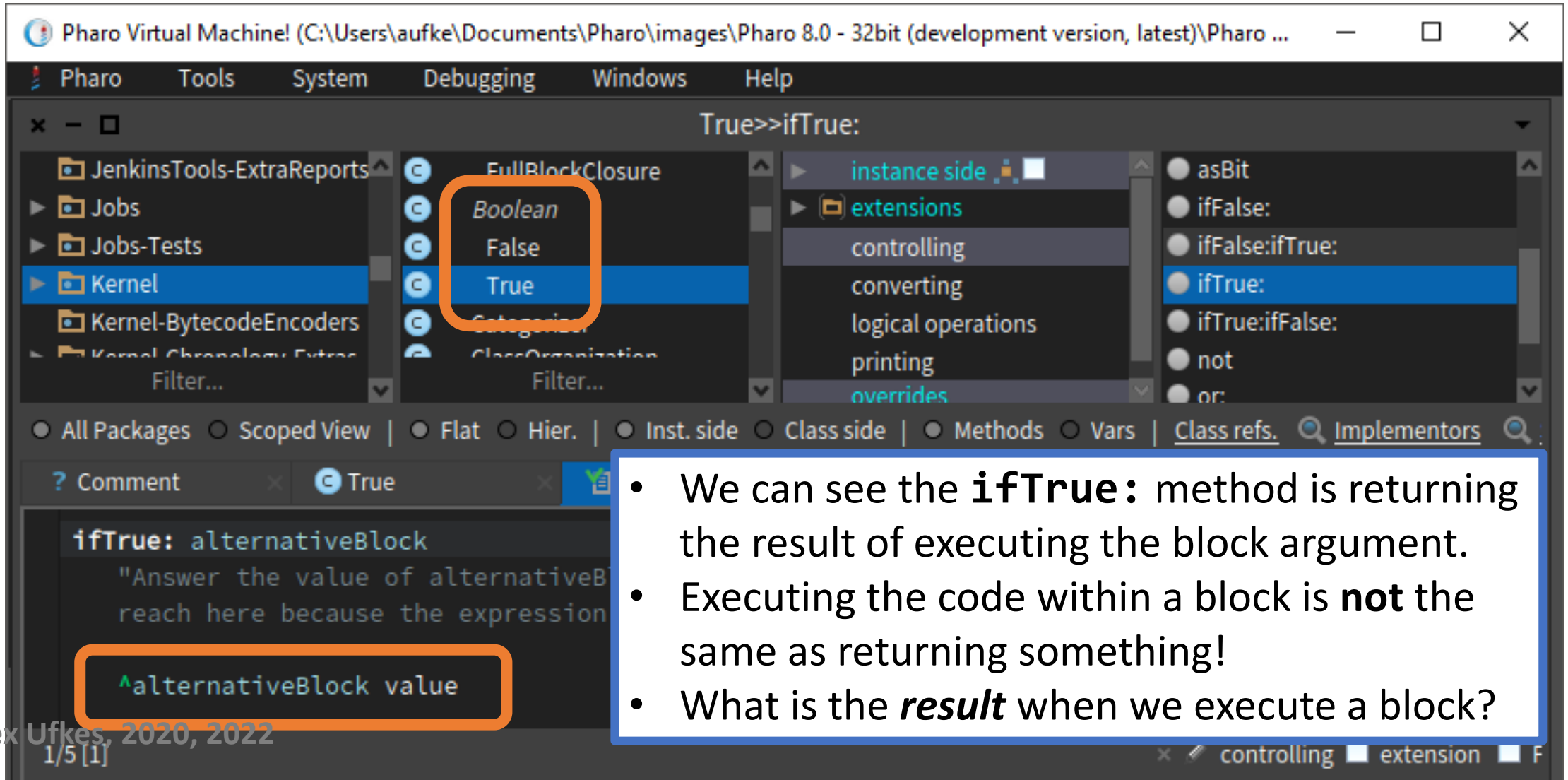


`ifTrue:` must be very complicated…



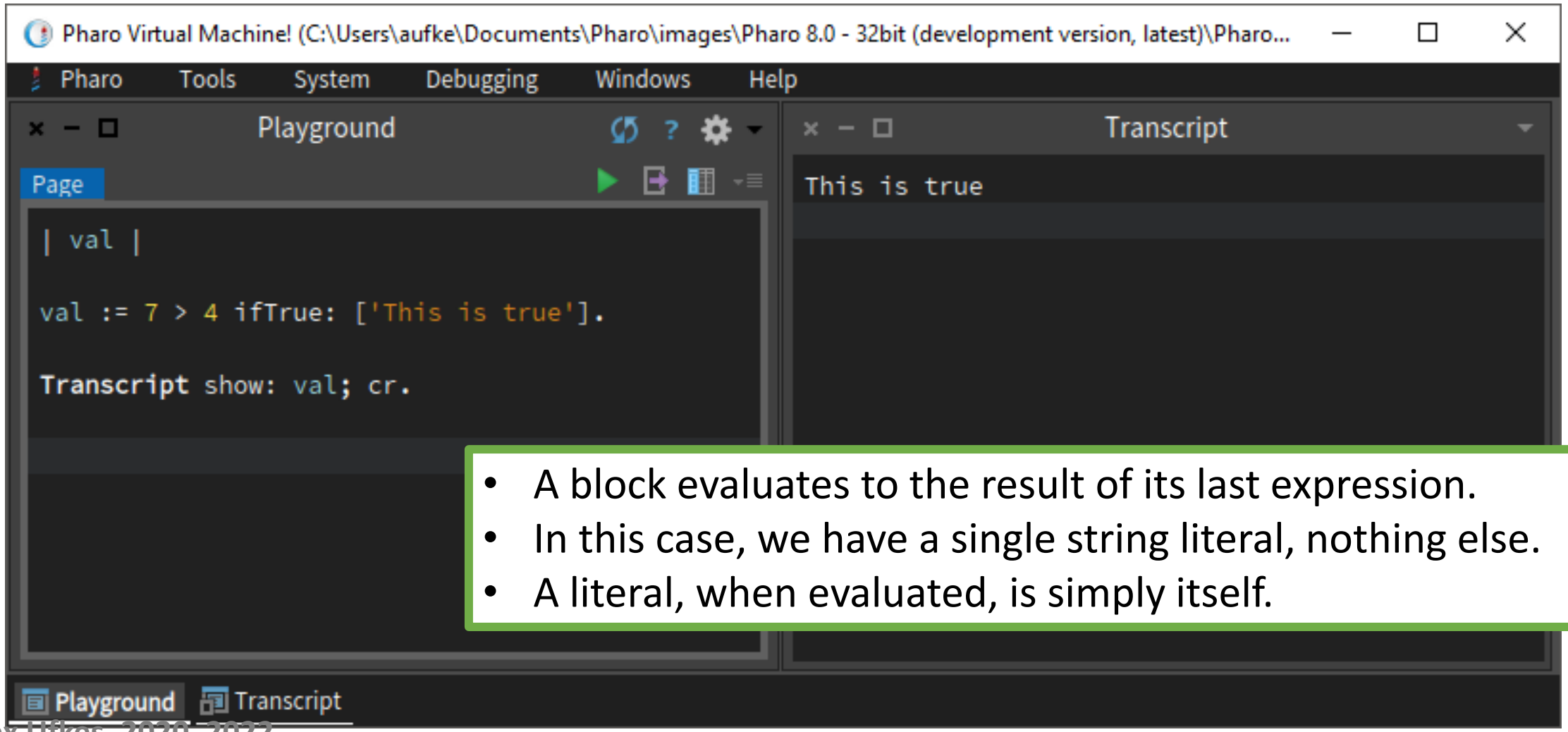
- A **`true`** object receives **`ifTrue:`** message.
- It sends the **`value`** message to the argument.
- The **`value`** message executes a block.
- It really is that simple!

`ifTrue:` must be very complicated…



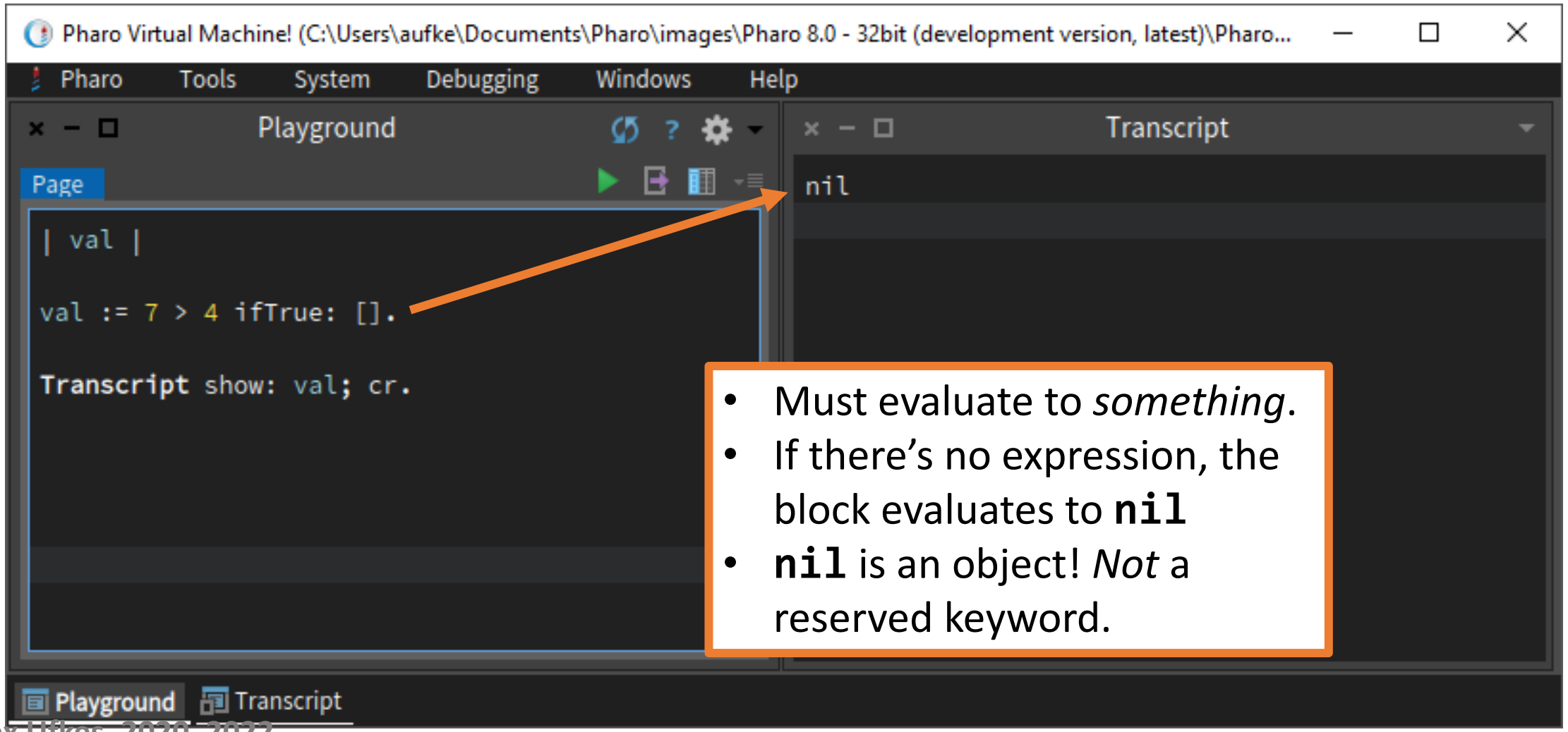
- We can see the **`ifTrue:`** method is returning the result of executing the block argument.
- Executing the code within a block is **not** the same as returning something!
- What is the ***result*** when we execute a block?

Evaluating Blocks



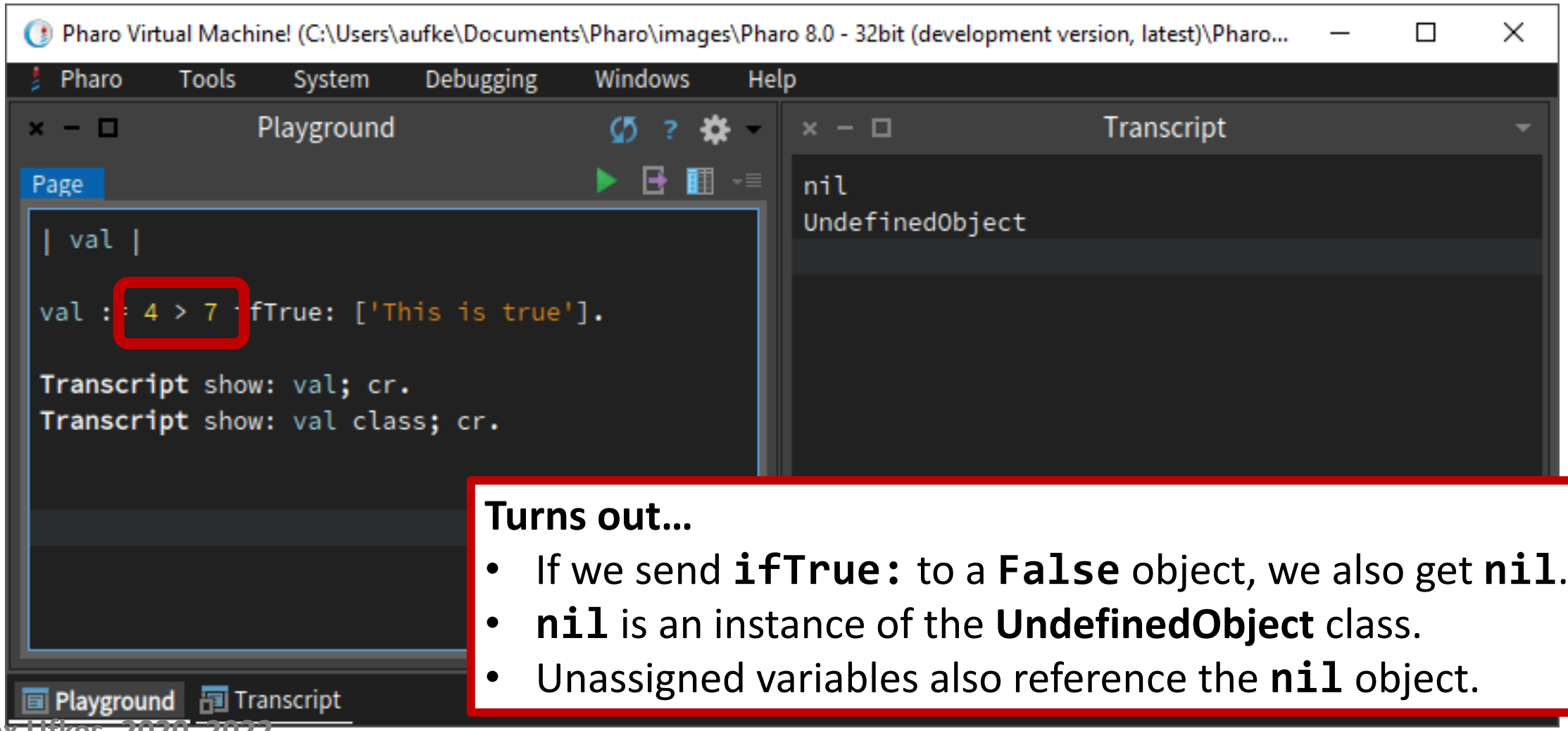
- A block evaluates to the result of its last expression.
- In this case, we have a single string literal, nothing else.
- A literal, when evaluated, is simply itself.

Evaluating Blocks



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

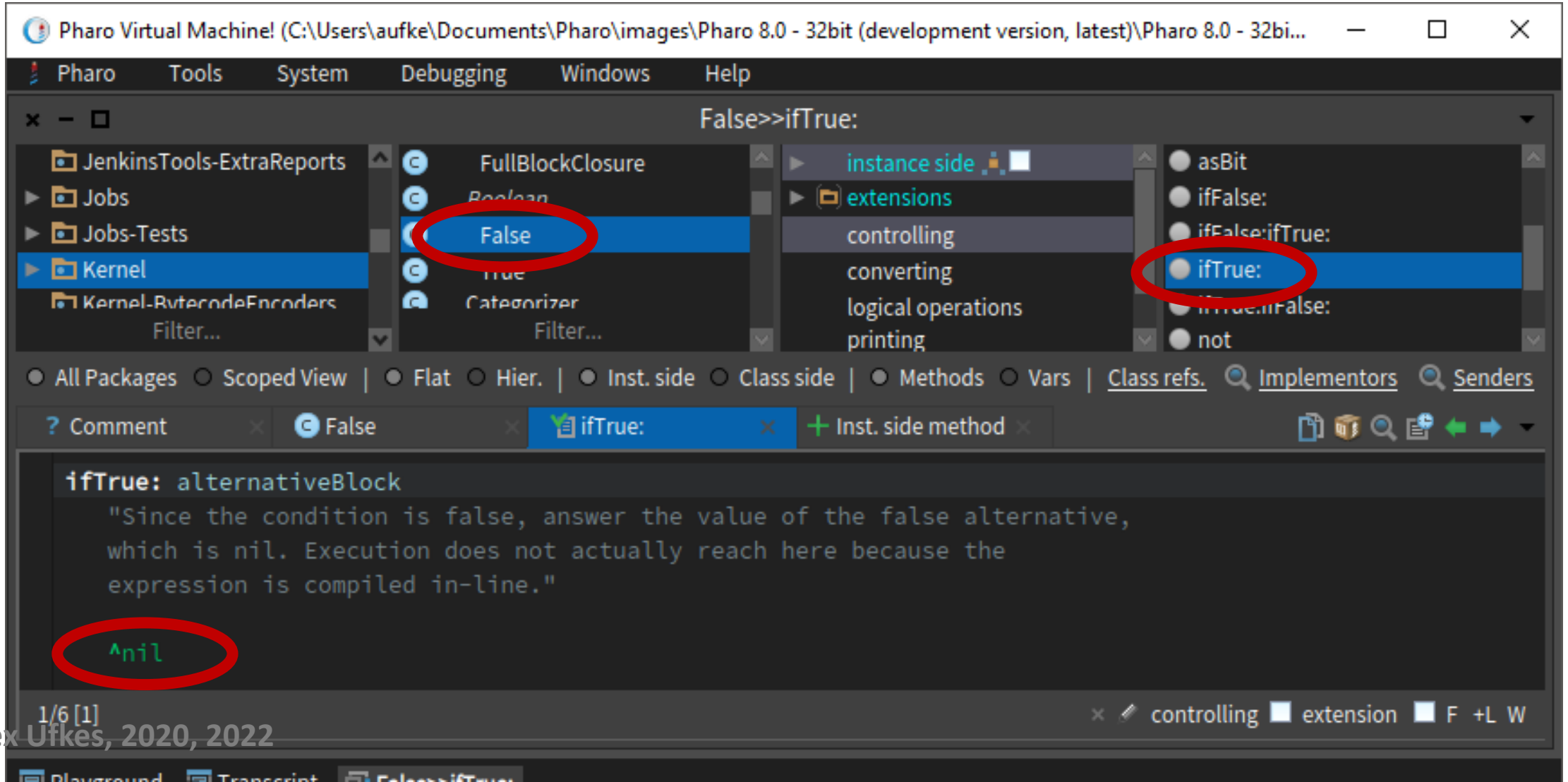
nil



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.

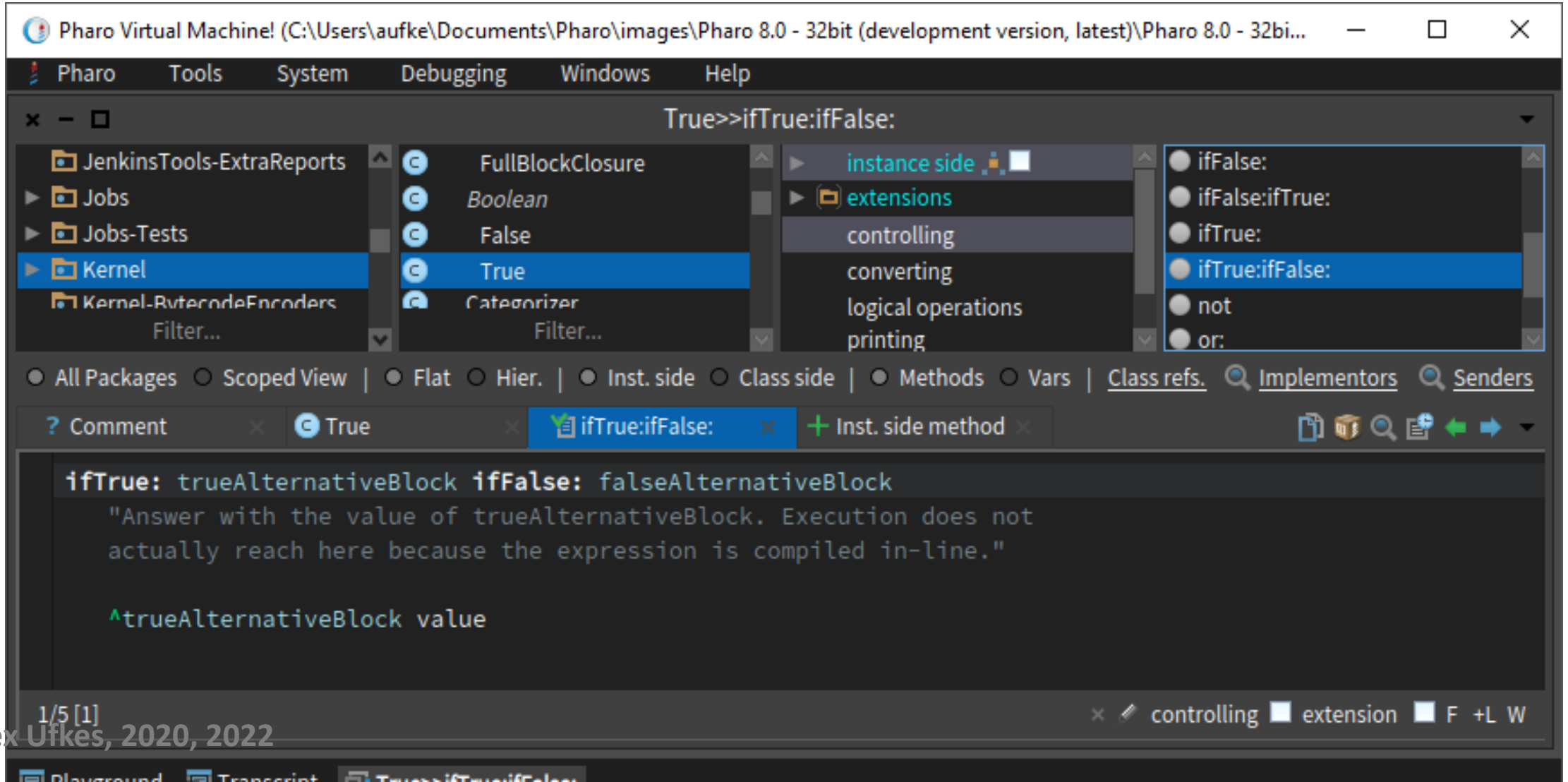
nil



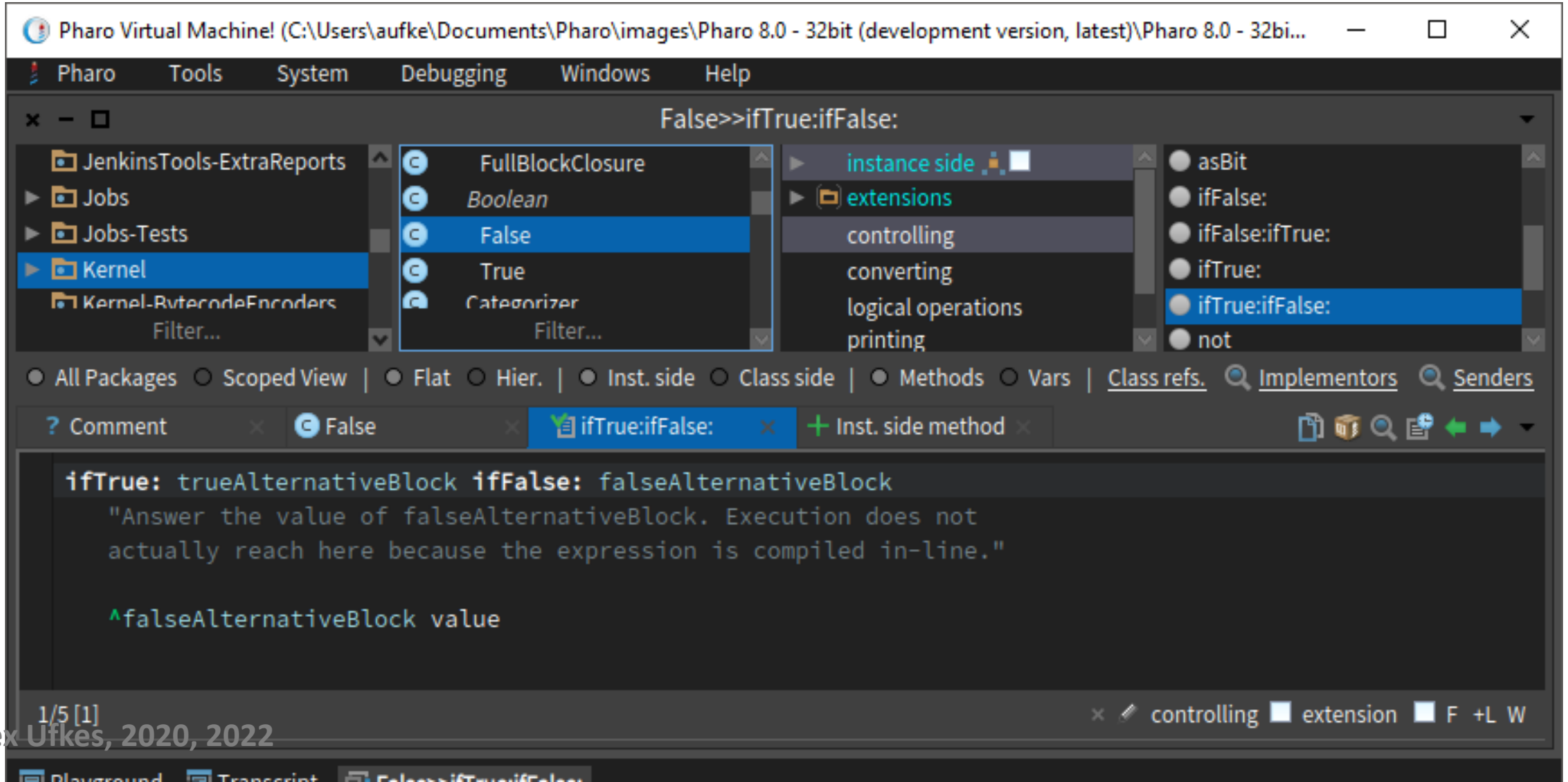
`ifTrue:ifFalse:`

- 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.

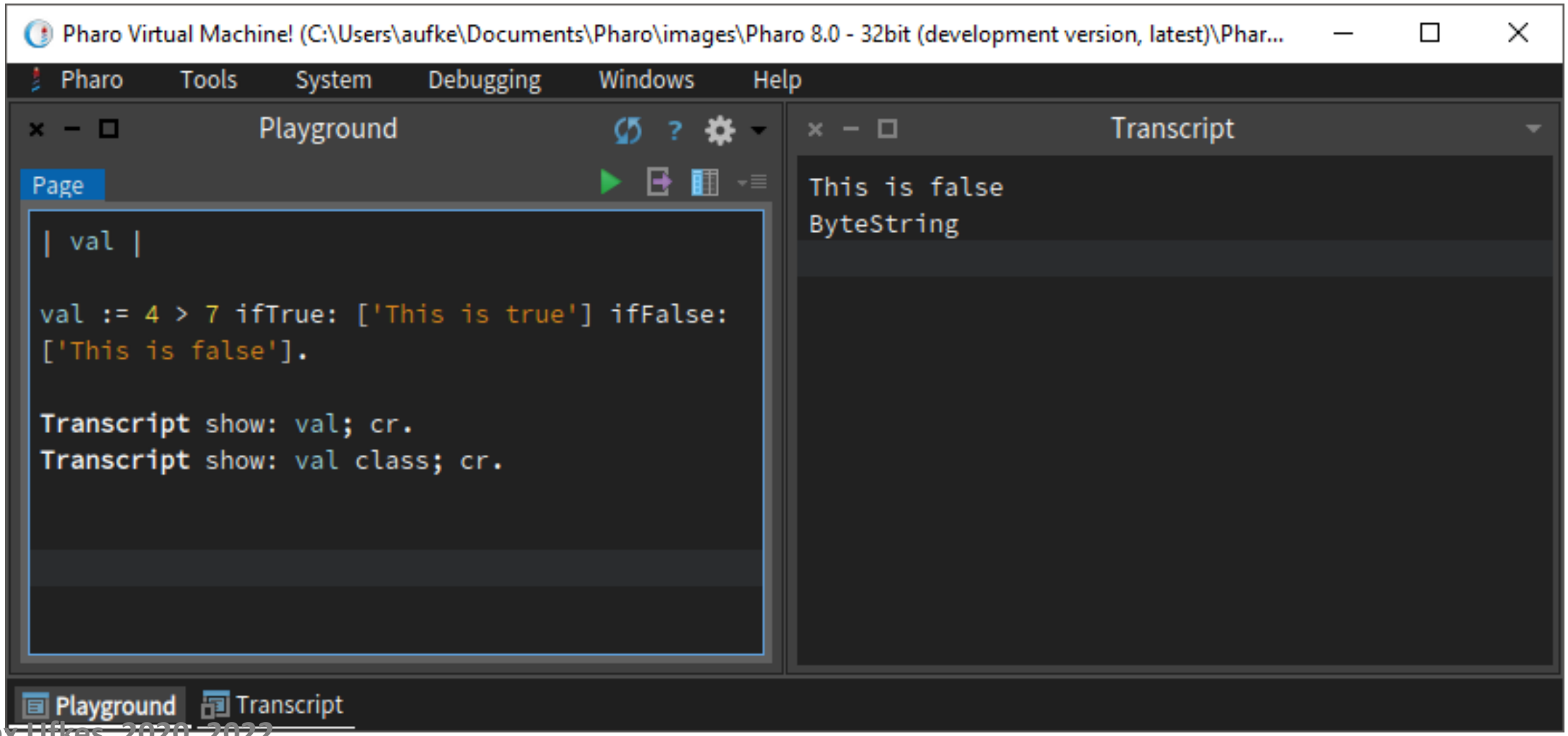
ifTrue:ifFalse:



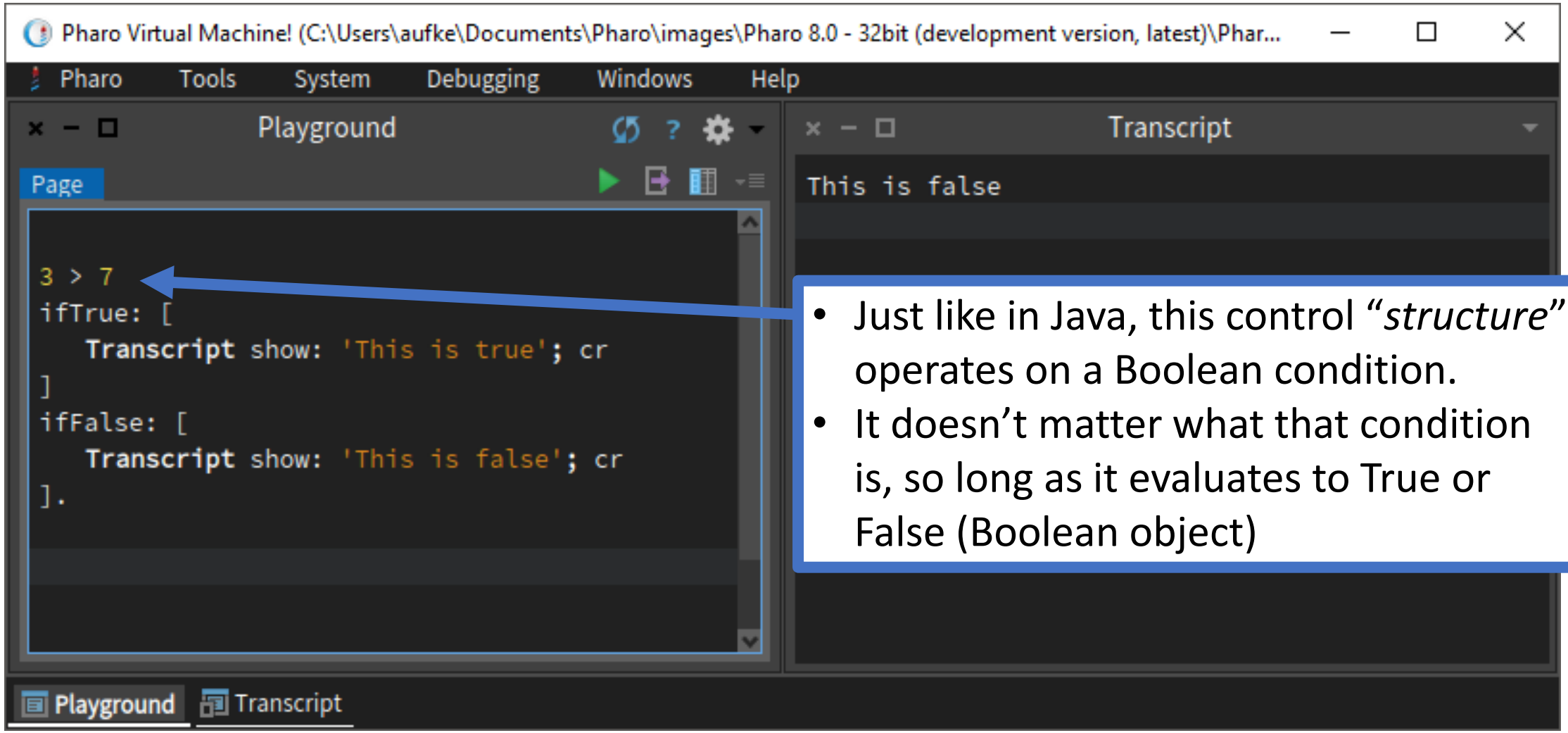
ifTrue:ifFalse:



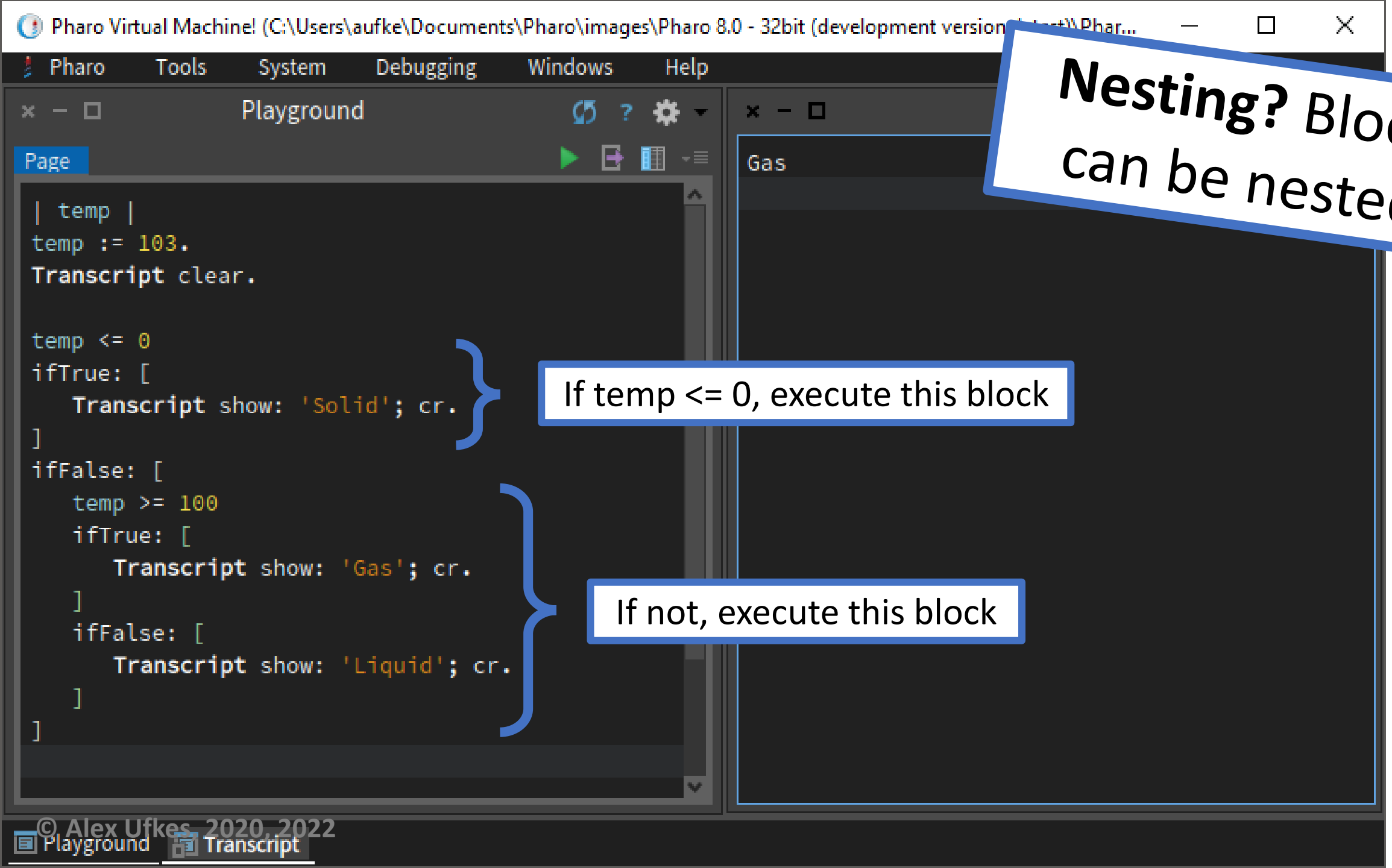
ifTrue:ifFalse:

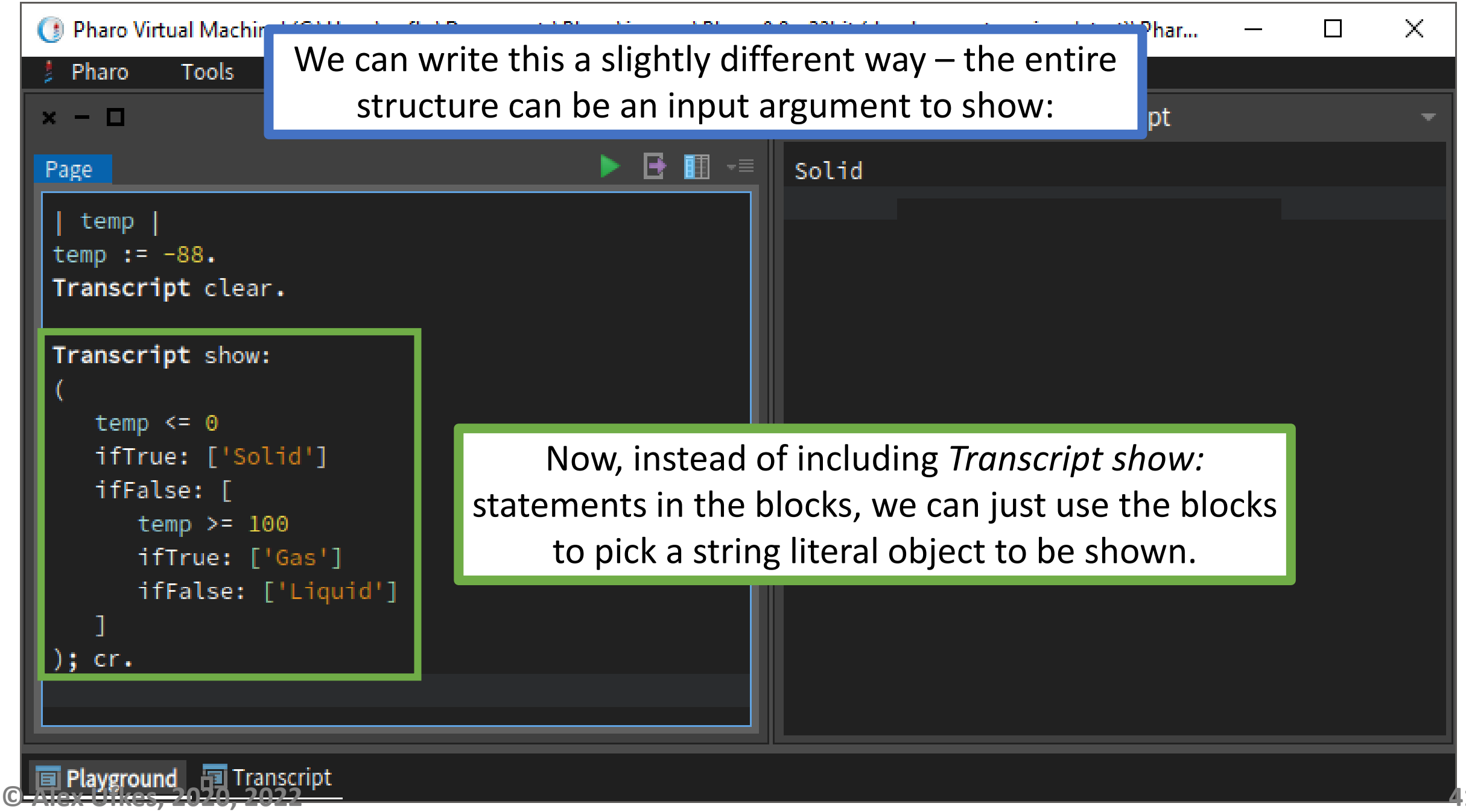


We can arrange the code to make the structure look more familiar:

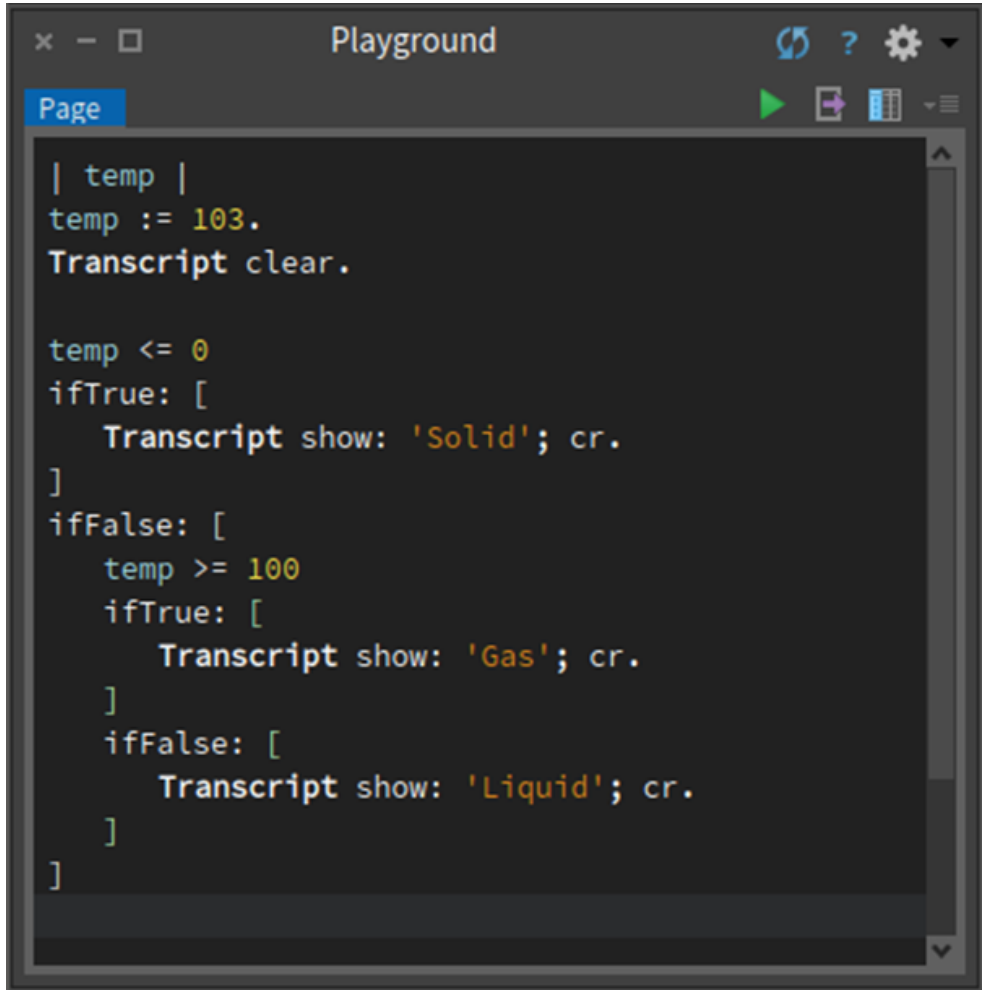


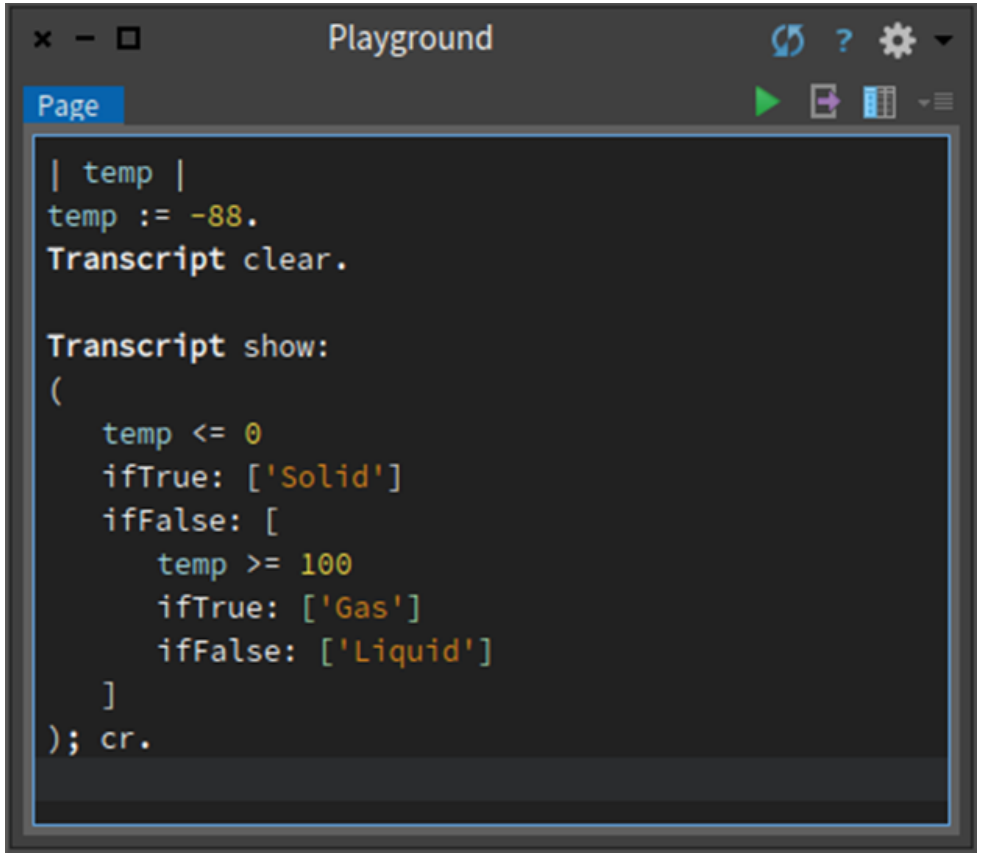
- Just like in Java, this control "*structure*" operates on a Boolean condition.
- It doesn't matter what that condition is, so long as it evaluates to True or False (Boolean object)





Different Syntax, Same Semantics





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.

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

Does not evaluate yet!

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']
```

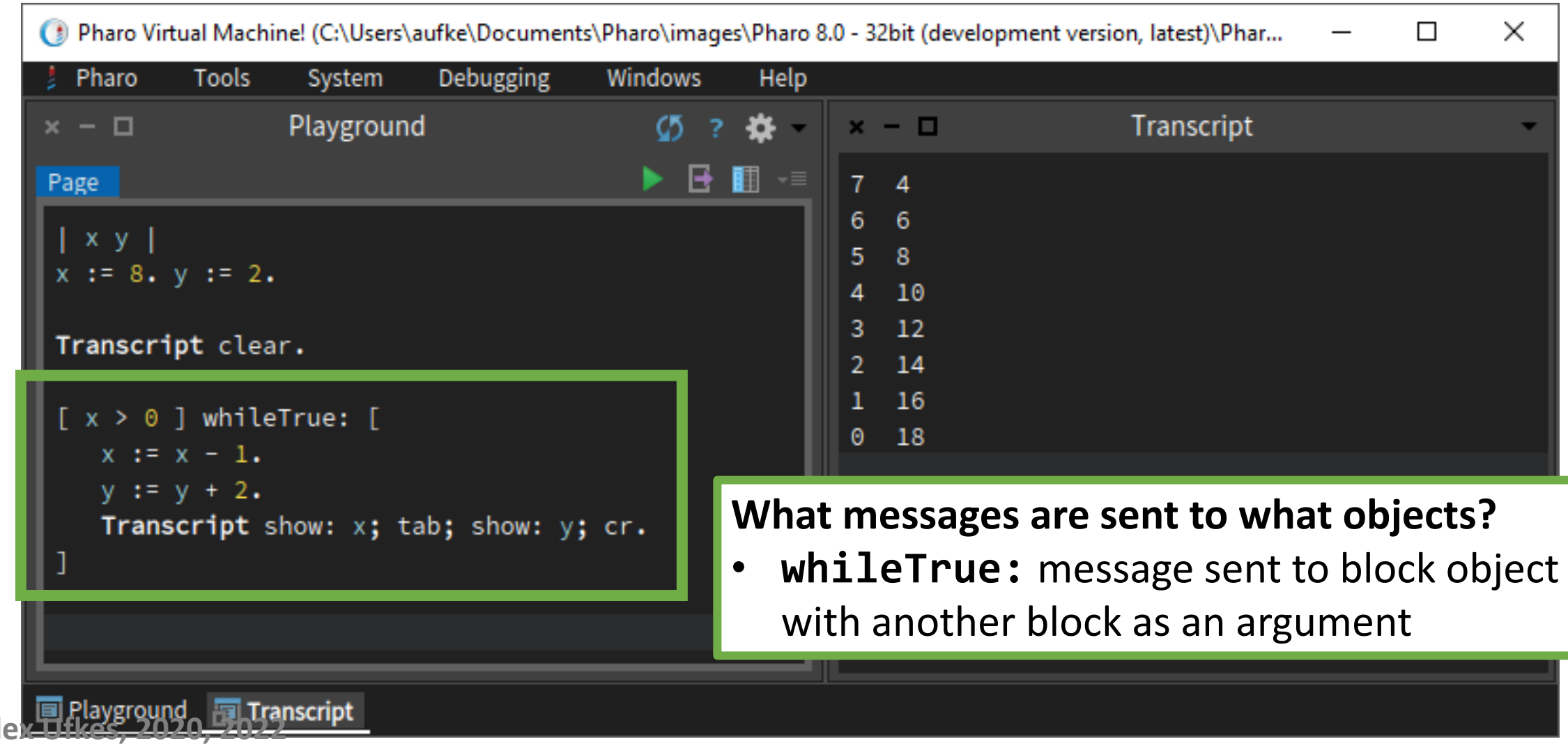
Passed to false object

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


```
'Liquid'
```

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

Repetition Using Messages & Blocks



What messages are sent to what objects?

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

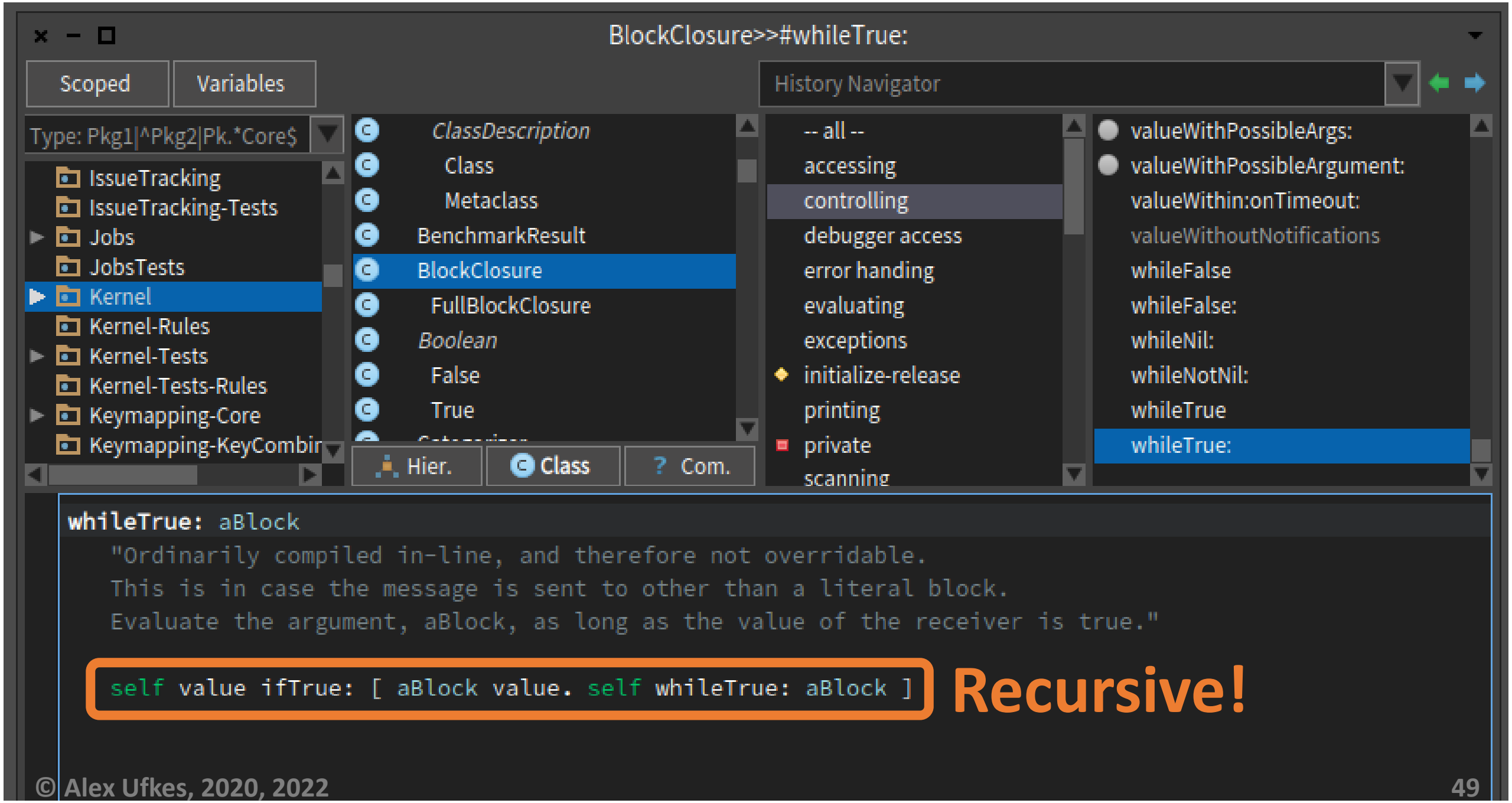
Repetition Using Messages & Blocks

```
[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.



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

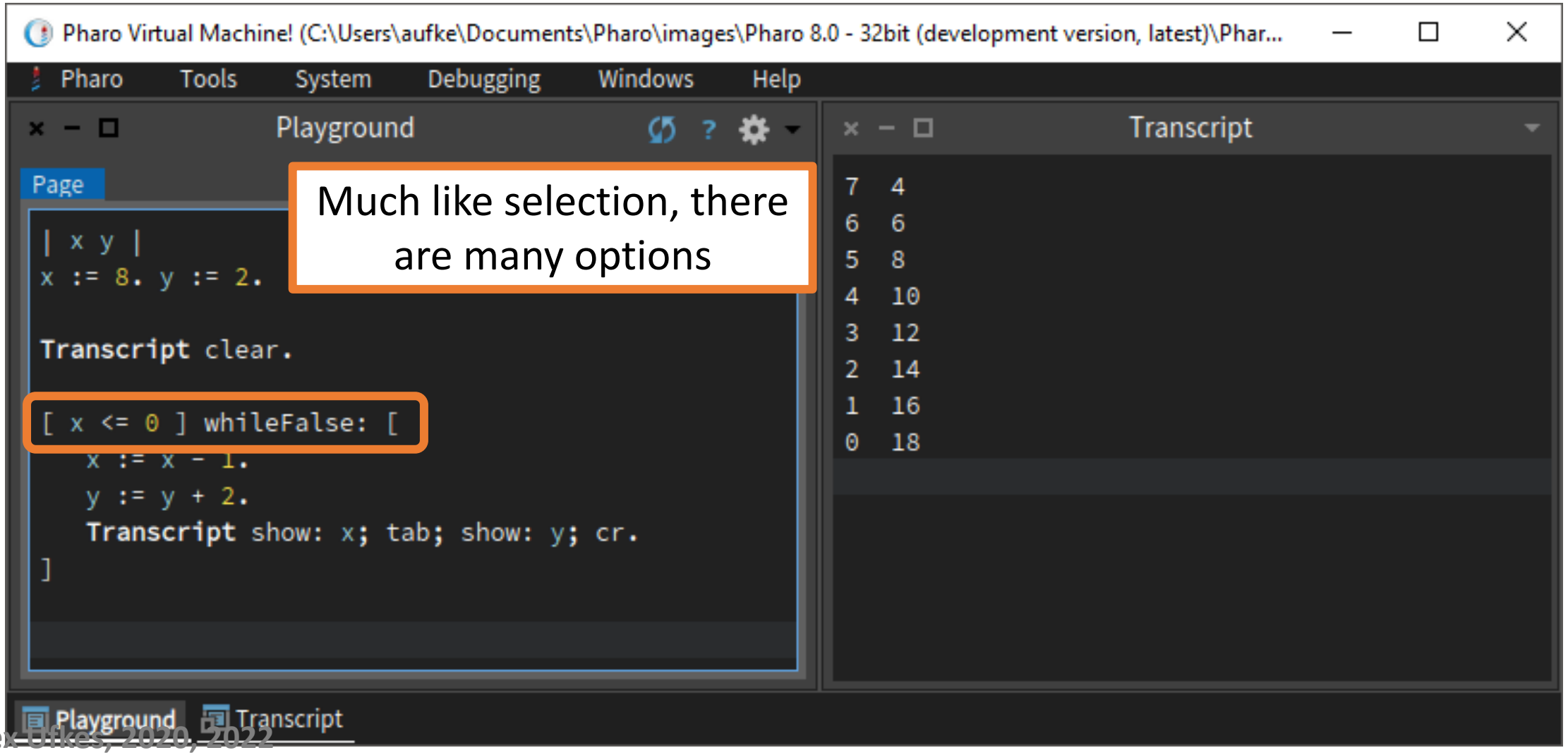
Finally, send **`whileTrue:`** once more to the same block, with the same argument.

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

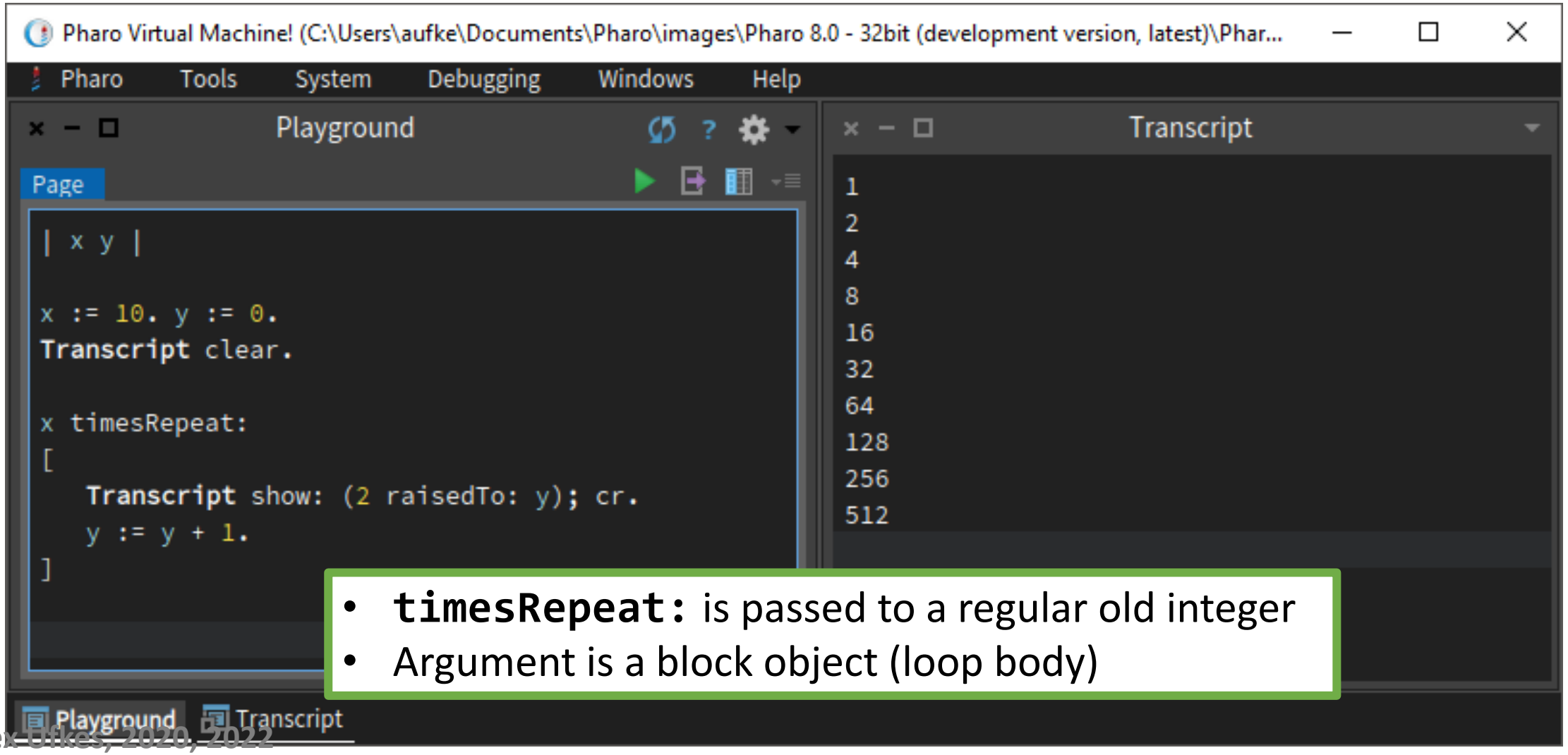
- Send value message to **`self`**, which executes it
- **`self`** evaluates to a Bool

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

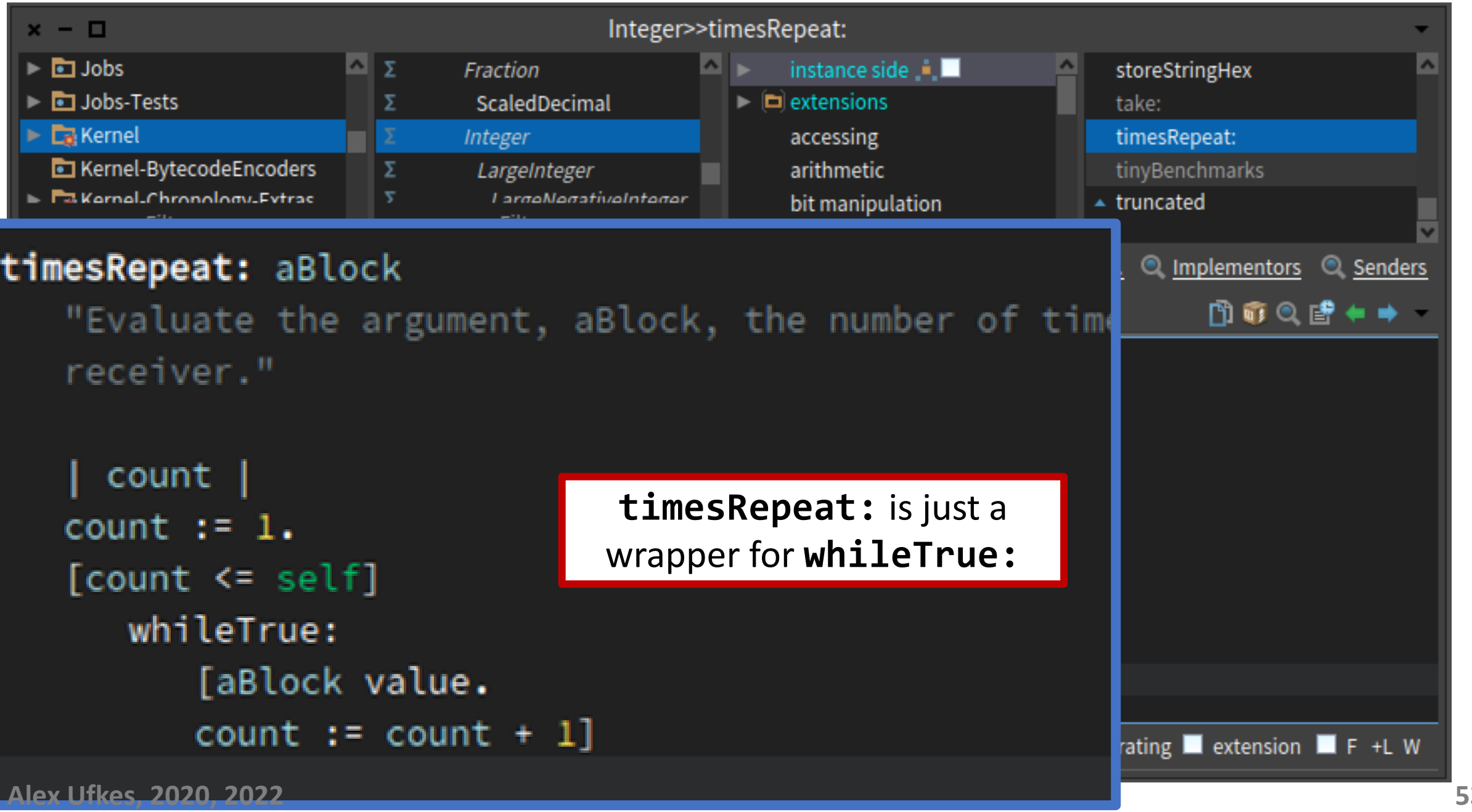
Repetition Using Messages & Blocks



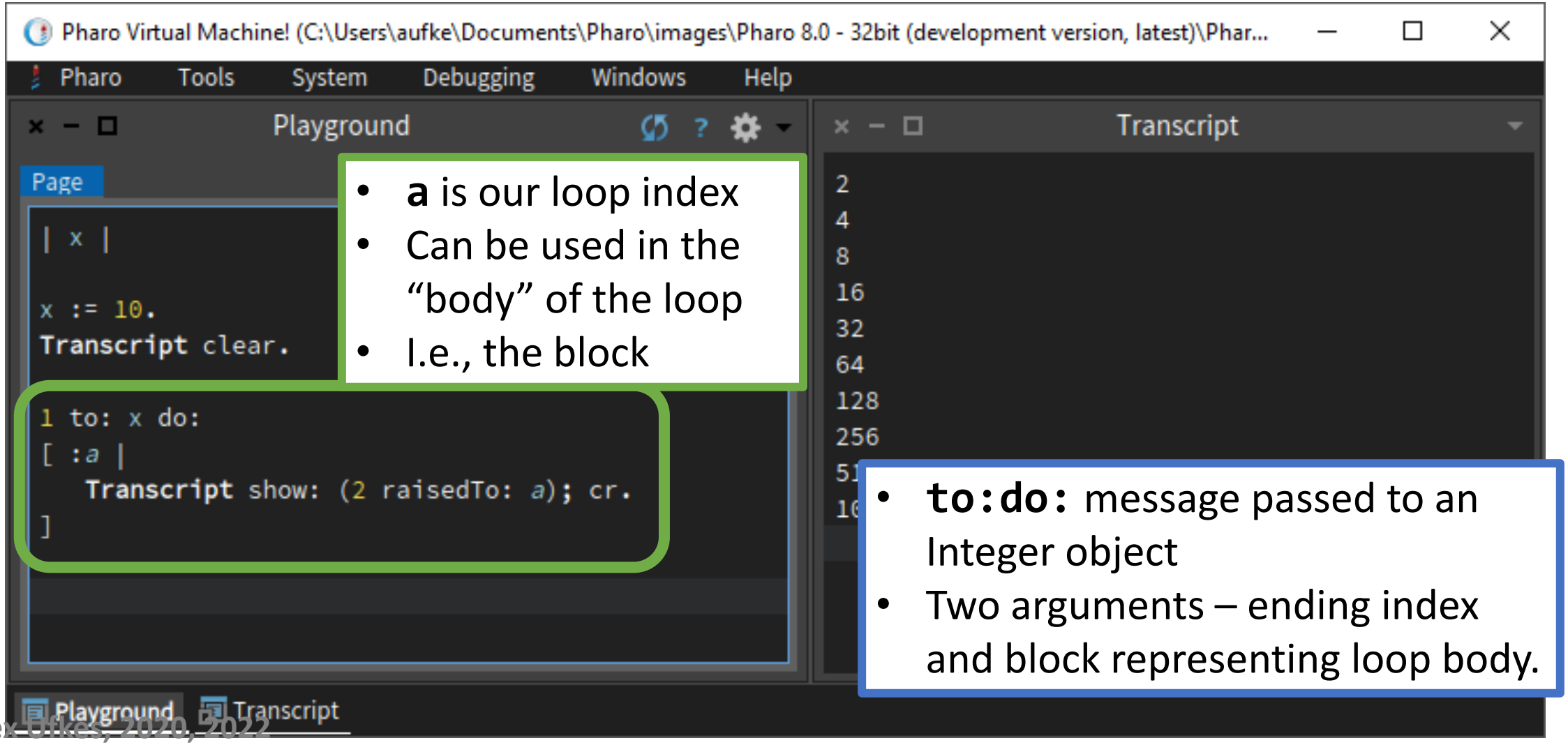
`timesRepeat:`



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

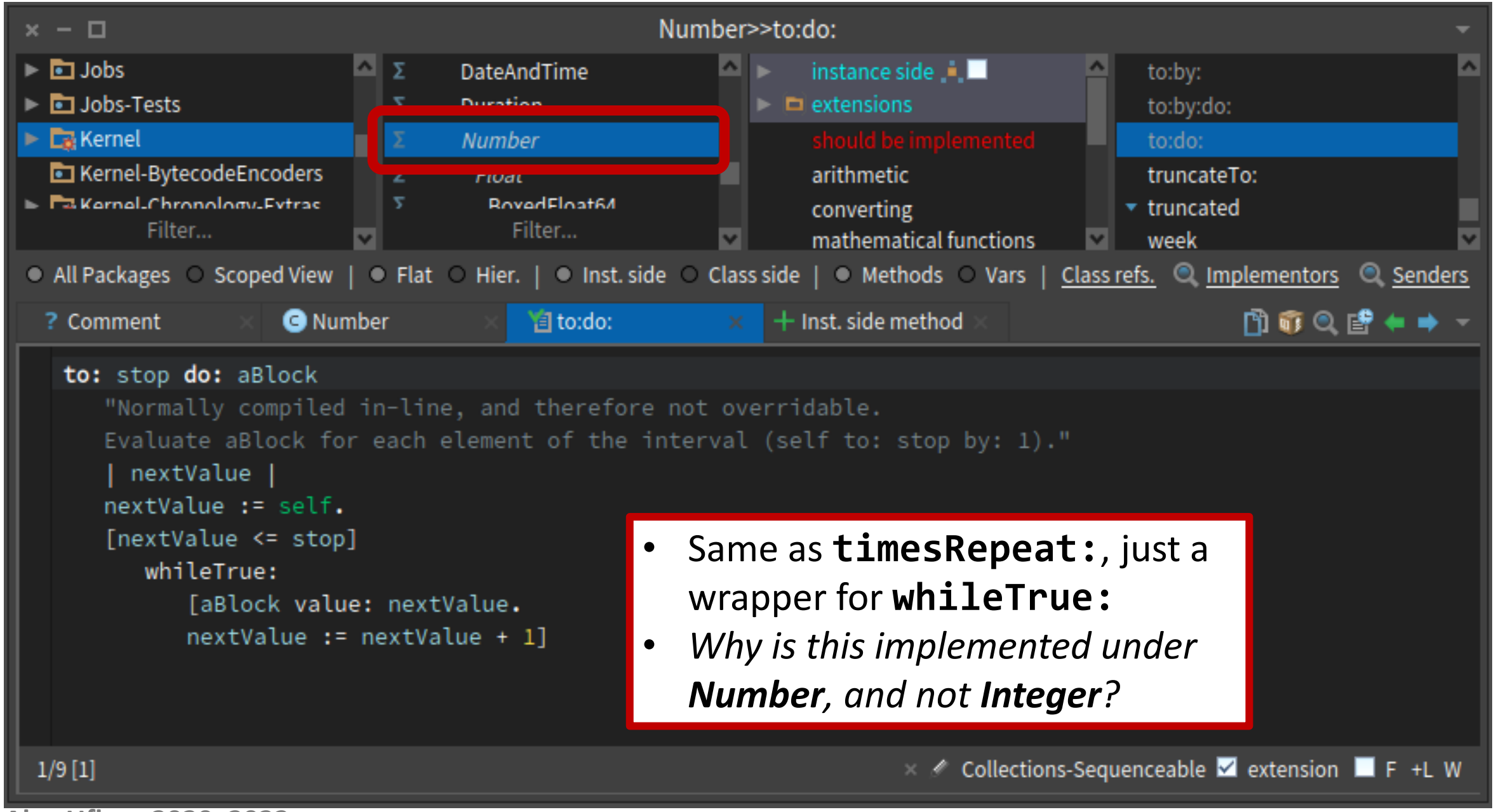


For-loop equivalent - `to:do:`



- **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.



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

For-loop equivalent - `to:do:`

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

In the case of timesReapeat:

- 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.

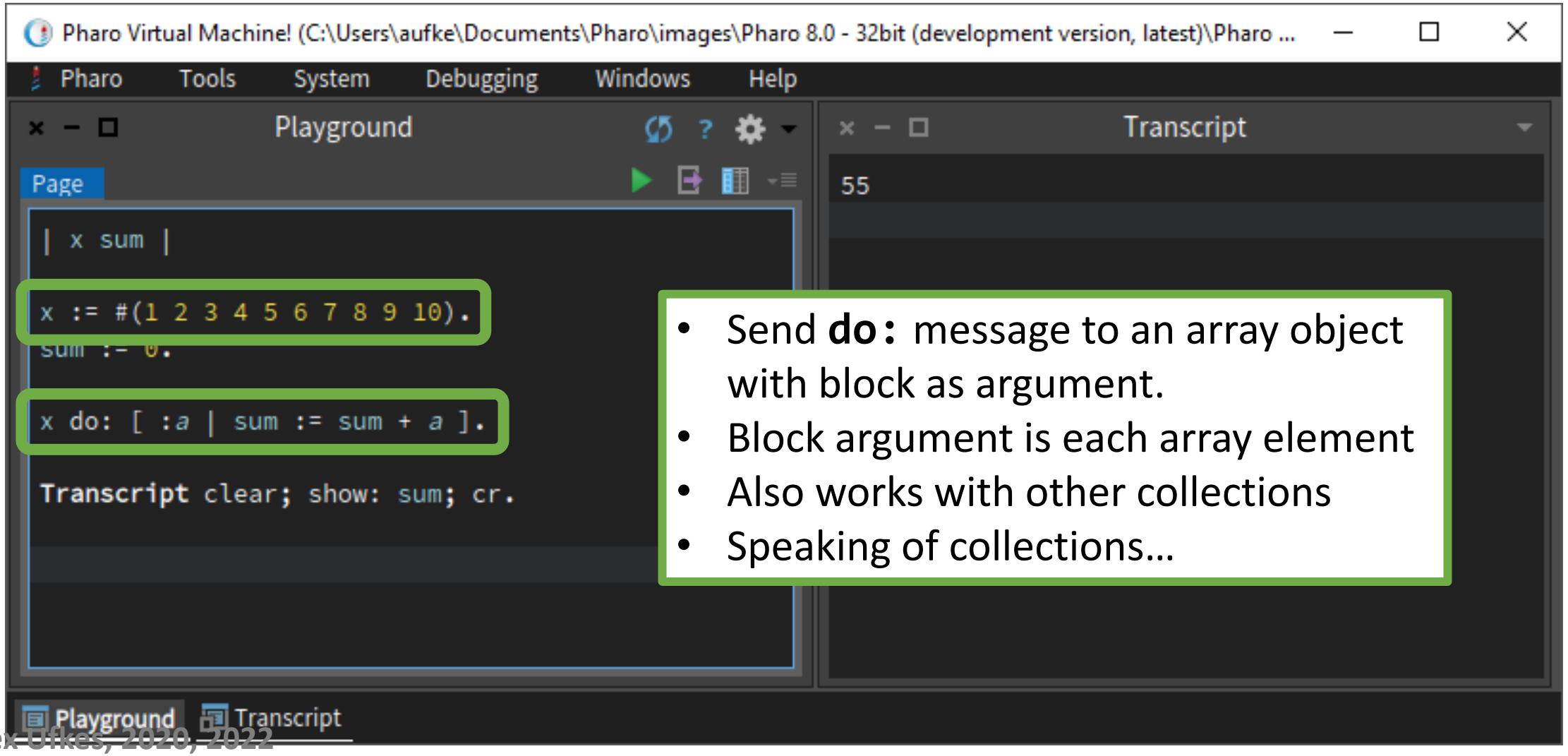
```
public static void main(String[] args)
{
    int[] nums = {-11, 68, 4, 0, 99};
    int sum = 0;

    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



- 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…

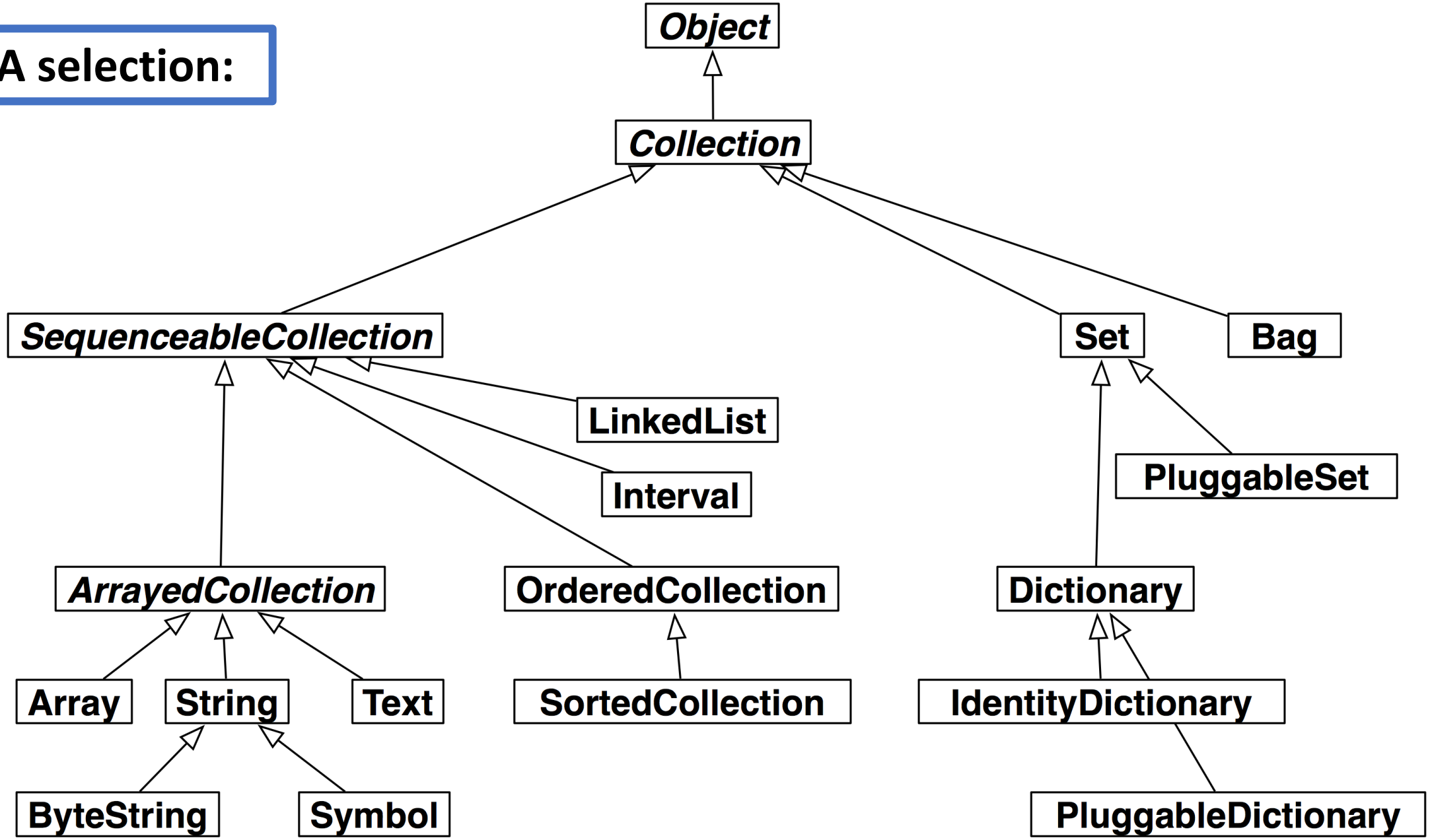
Smalltalk Collections

We've seen arrays.

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

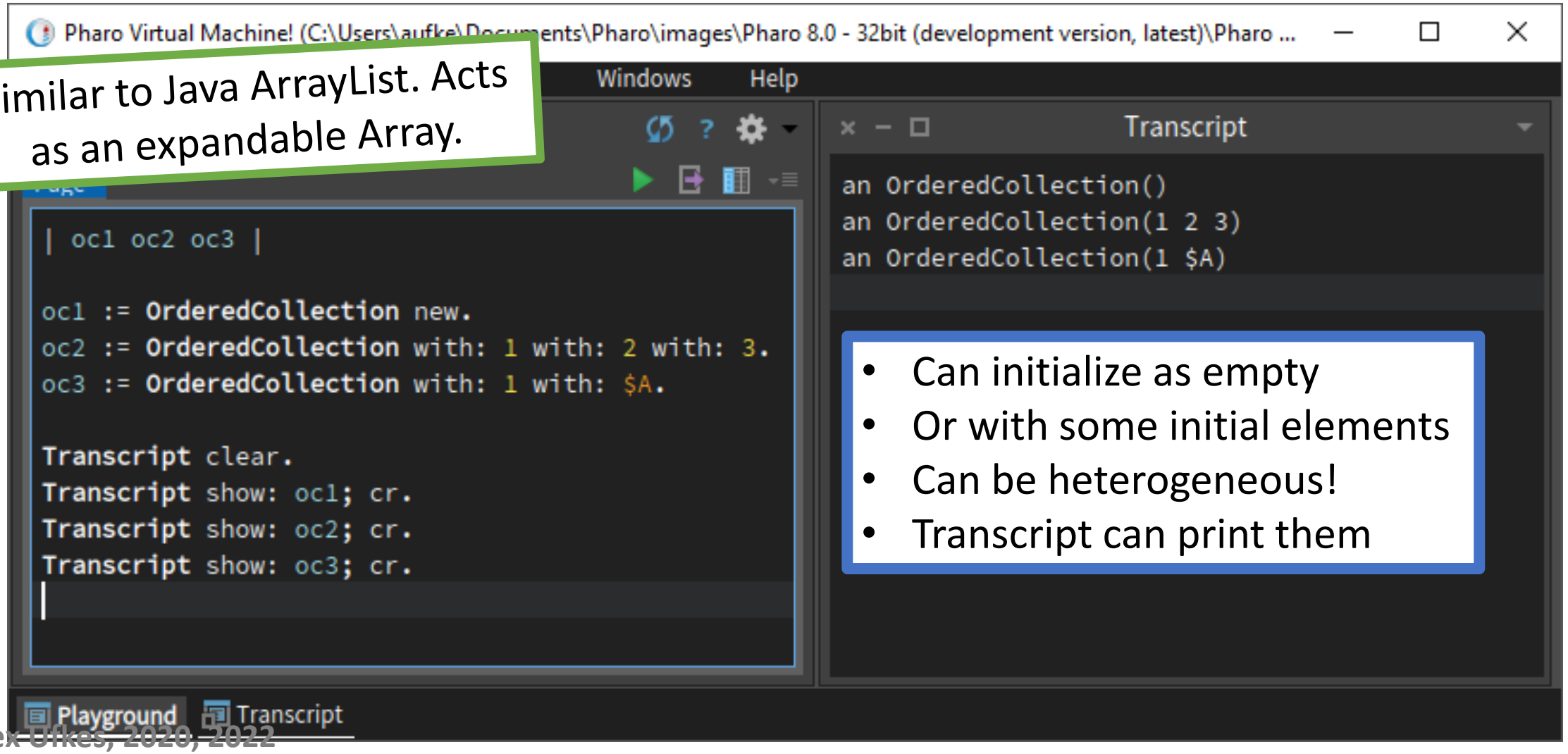
What about Smalltalk?

A selection:



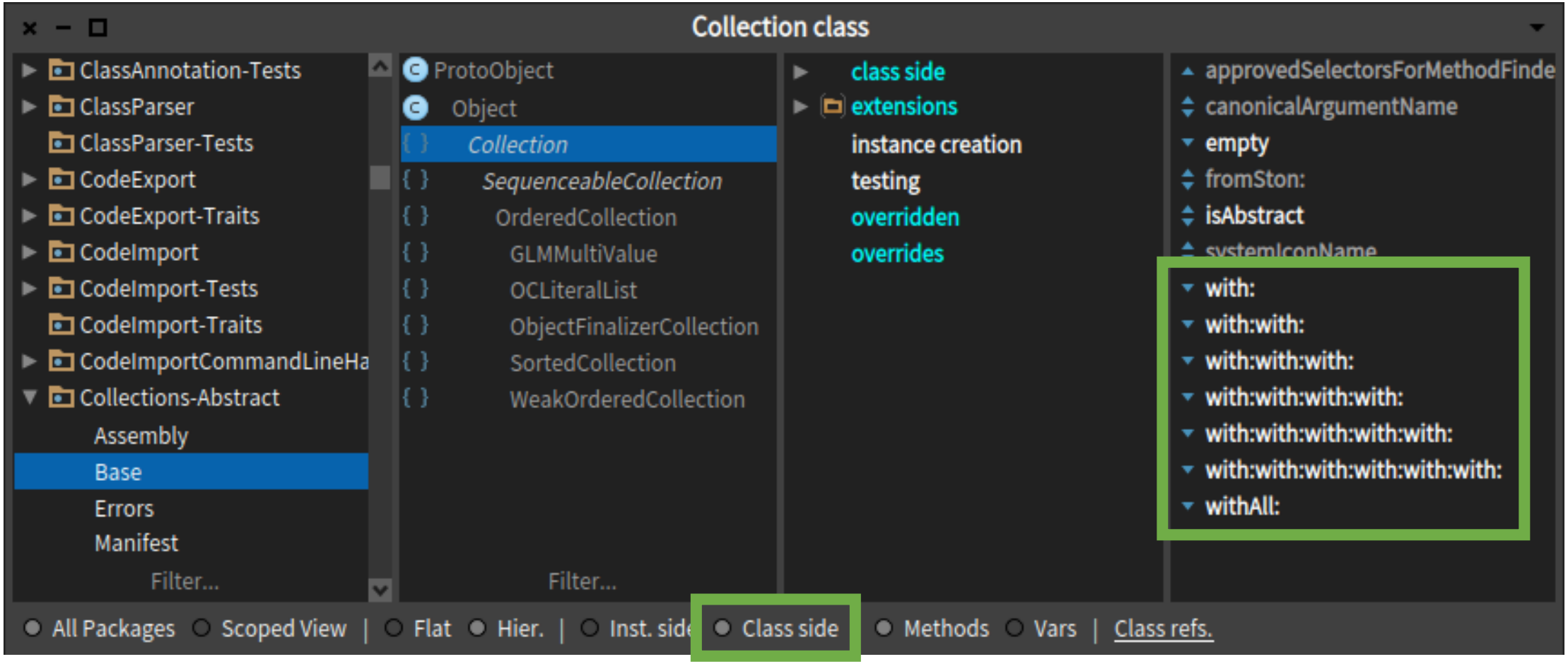
Ordered Collection

Similar to Java ArrayList. Acts as an expandable Array.

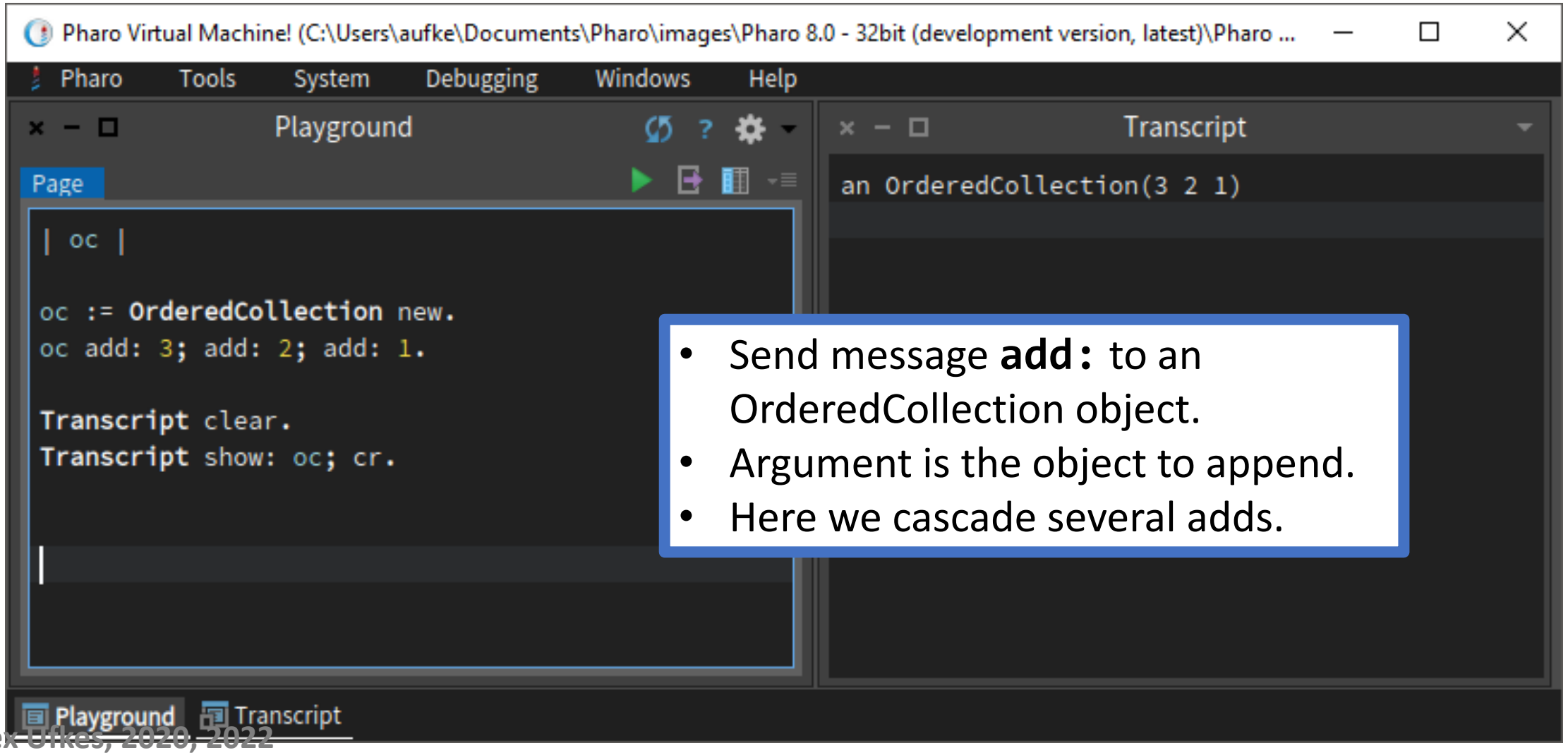


- Can initialize as empty
- Or with some initial elements
- Can be heterogeneous!
- Transcript can print them

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

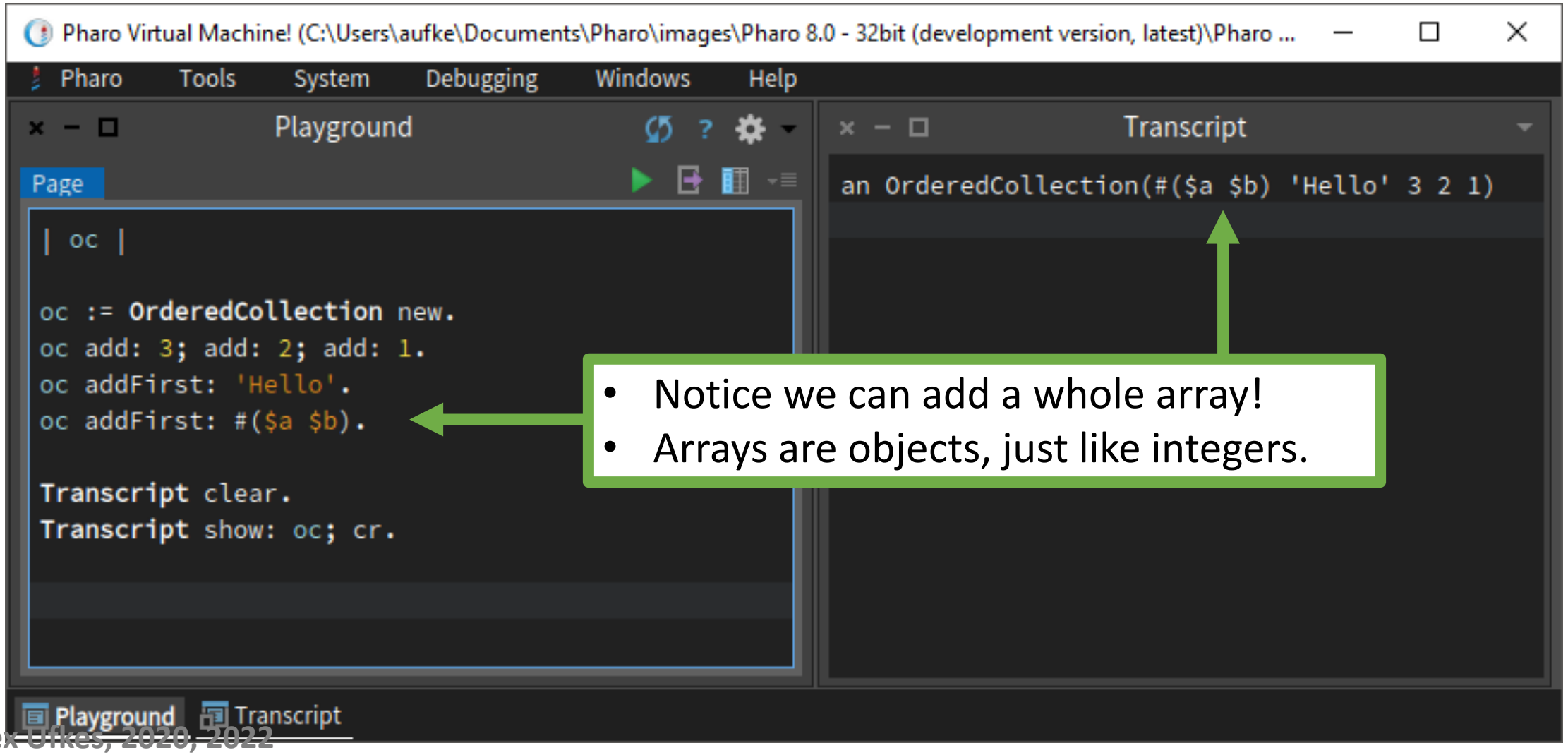


Ordered Collection: Add Elements

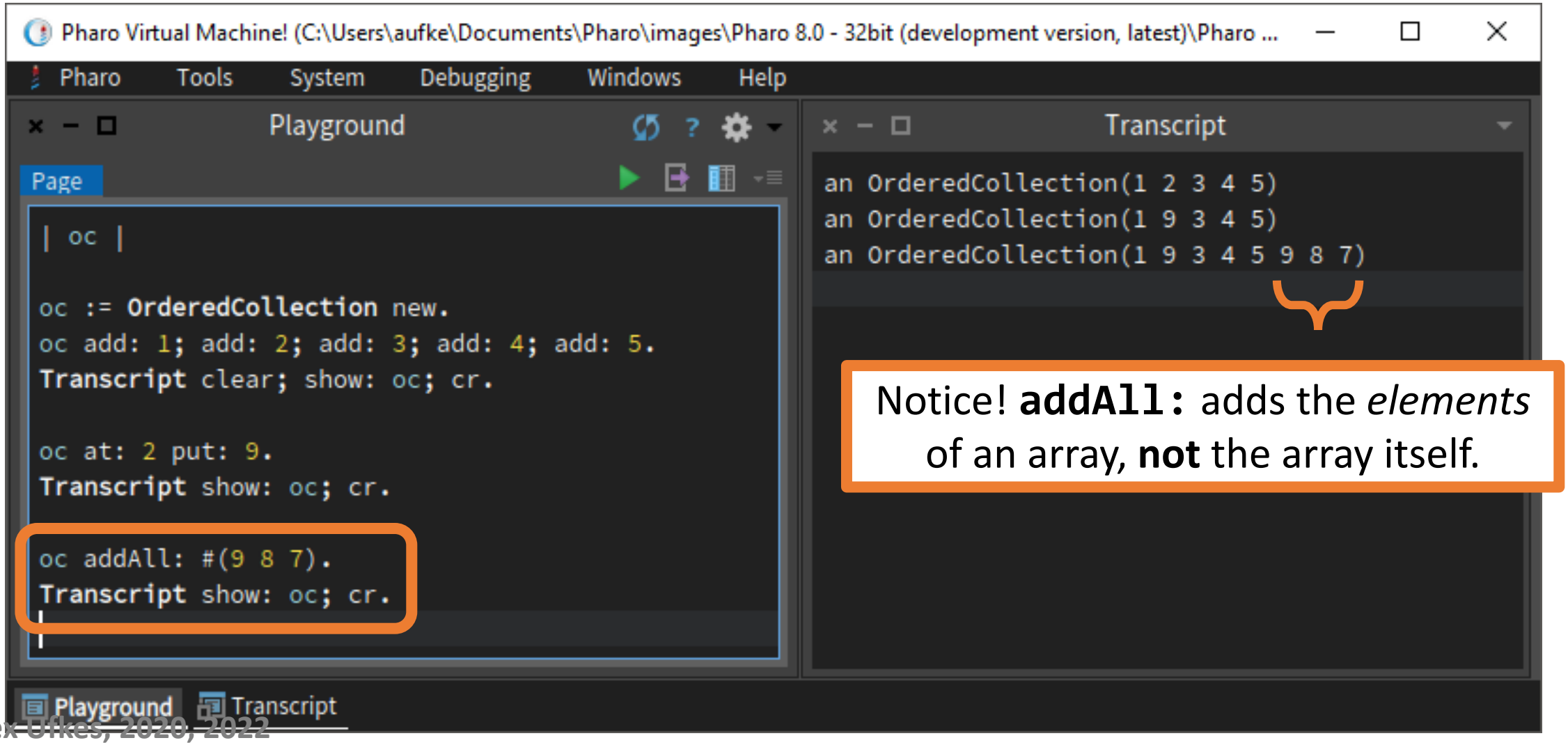


- Send message **add:** to an OrderedCollection object.
- Argument is the object to append.
- Here we cascade several adds.

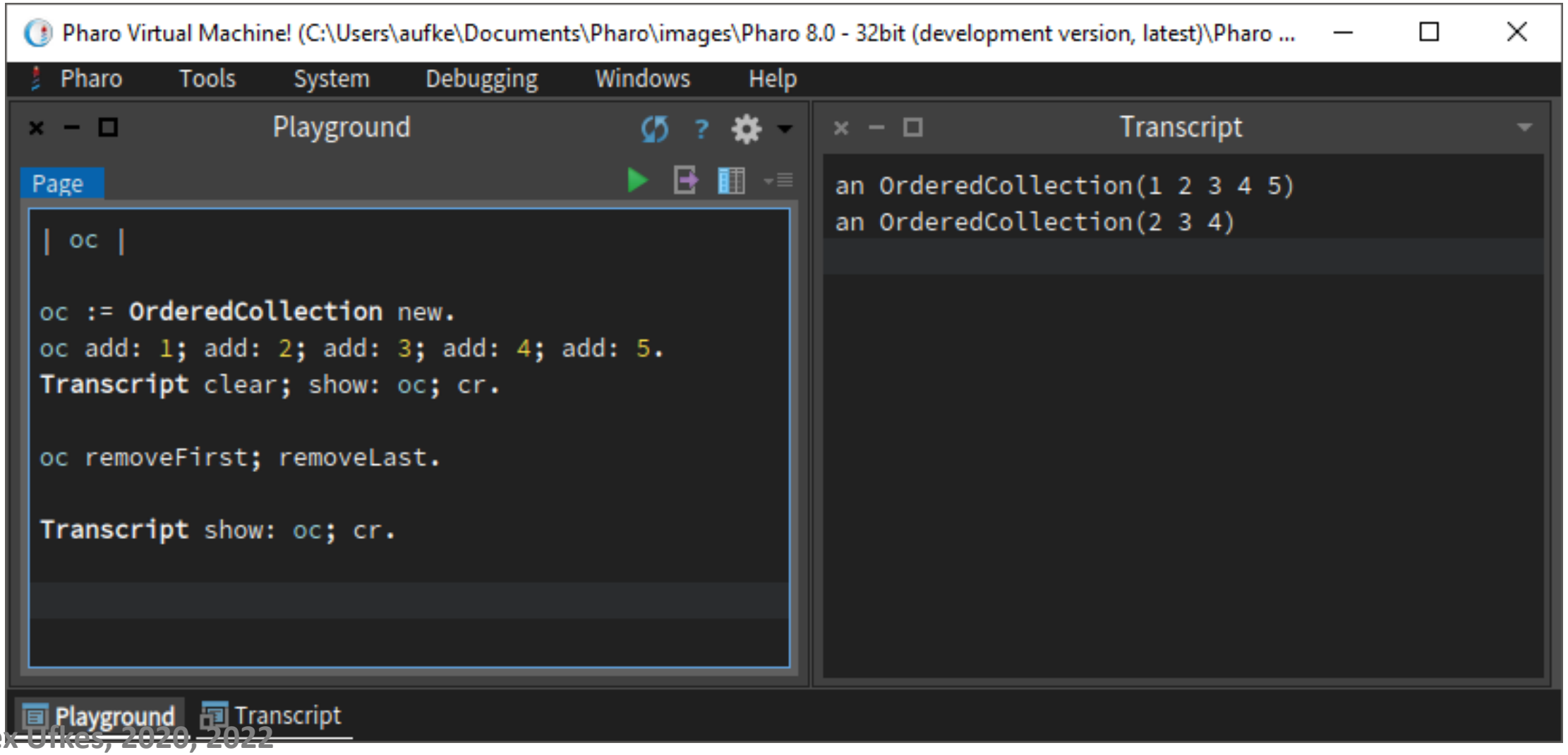
Ordered Collection: Add Elements



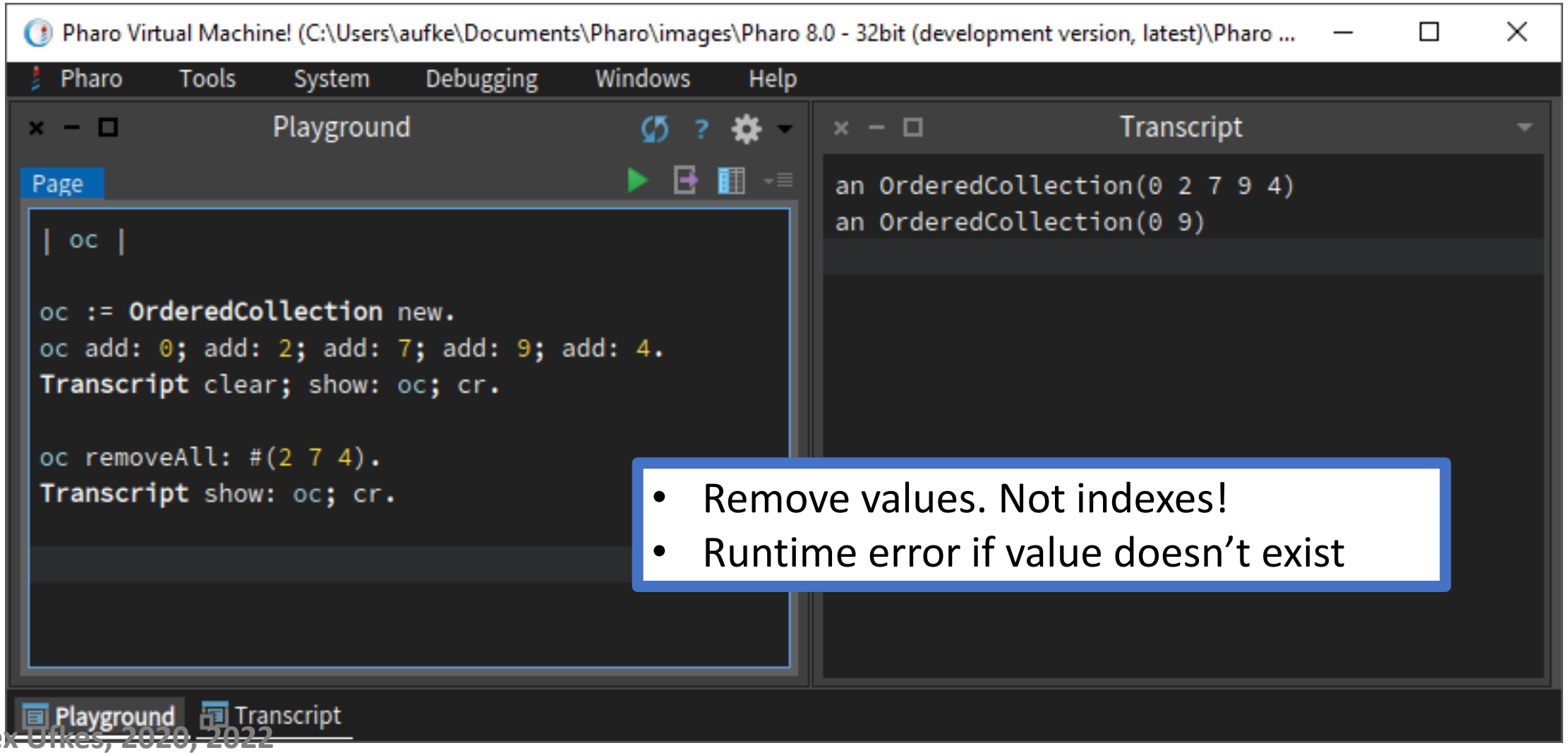
Ordered Collection: `at:put:`, `addAll:`



Ordered Collection: Removing

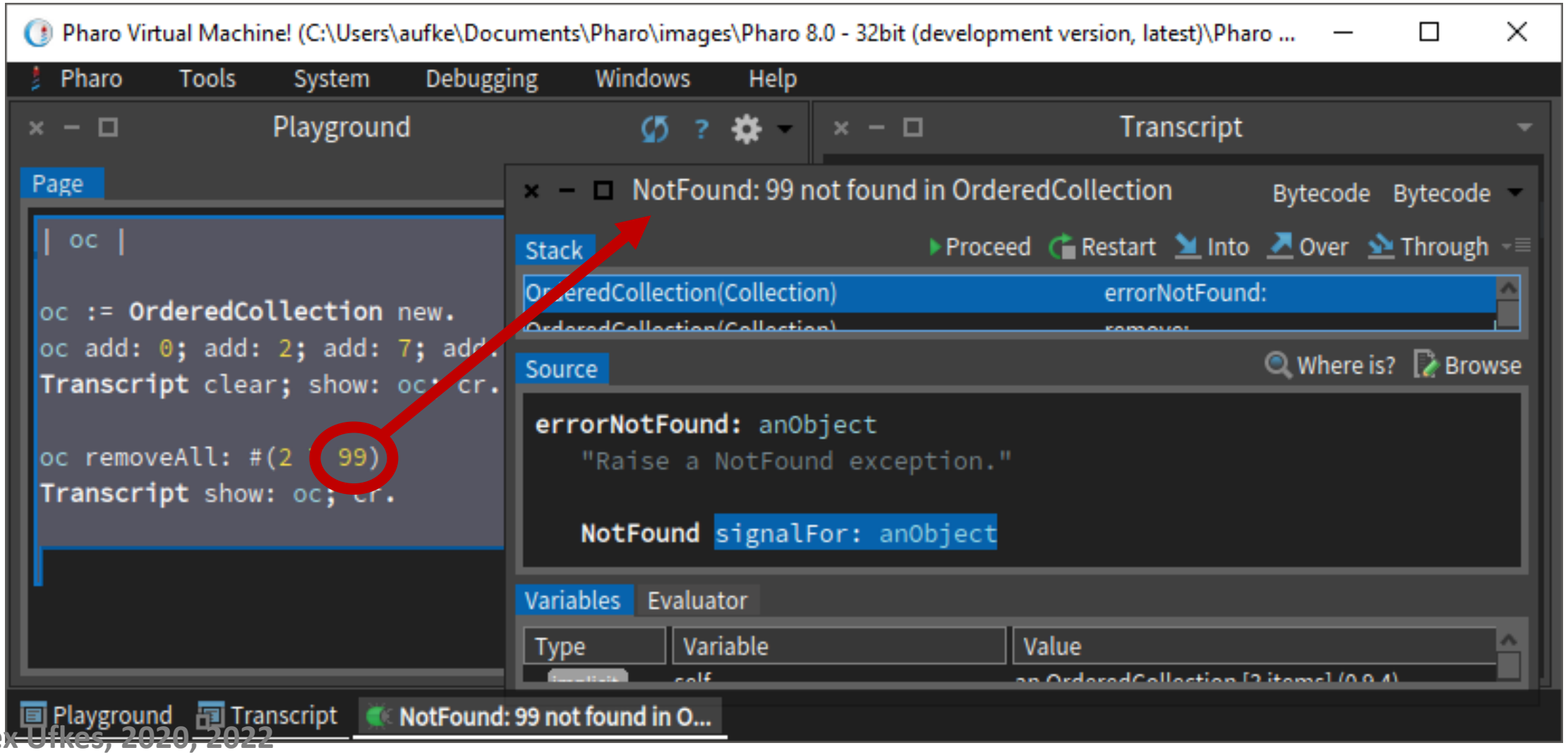


Ordered Collection: Removing



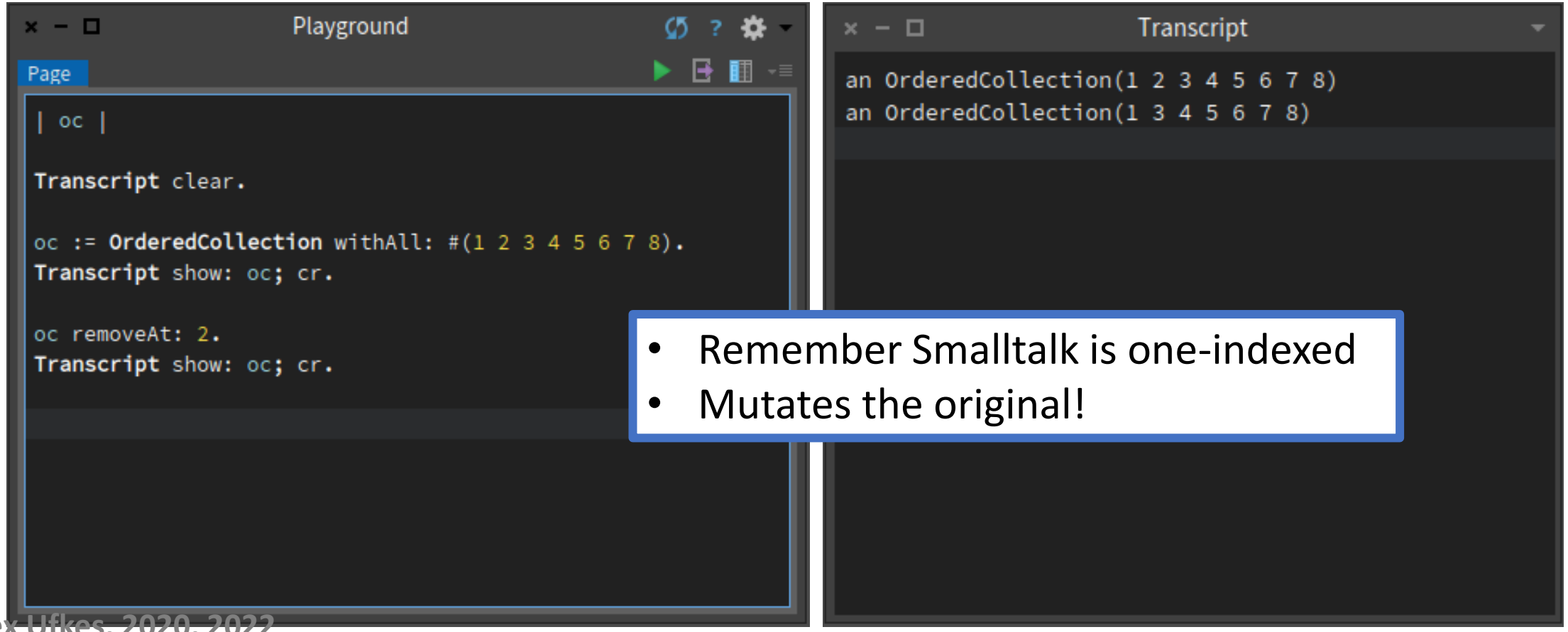
- Remove values. Not indexes!
- Runtime error if value doesn't exist

Ordered Collection: Removing



Remove by Index

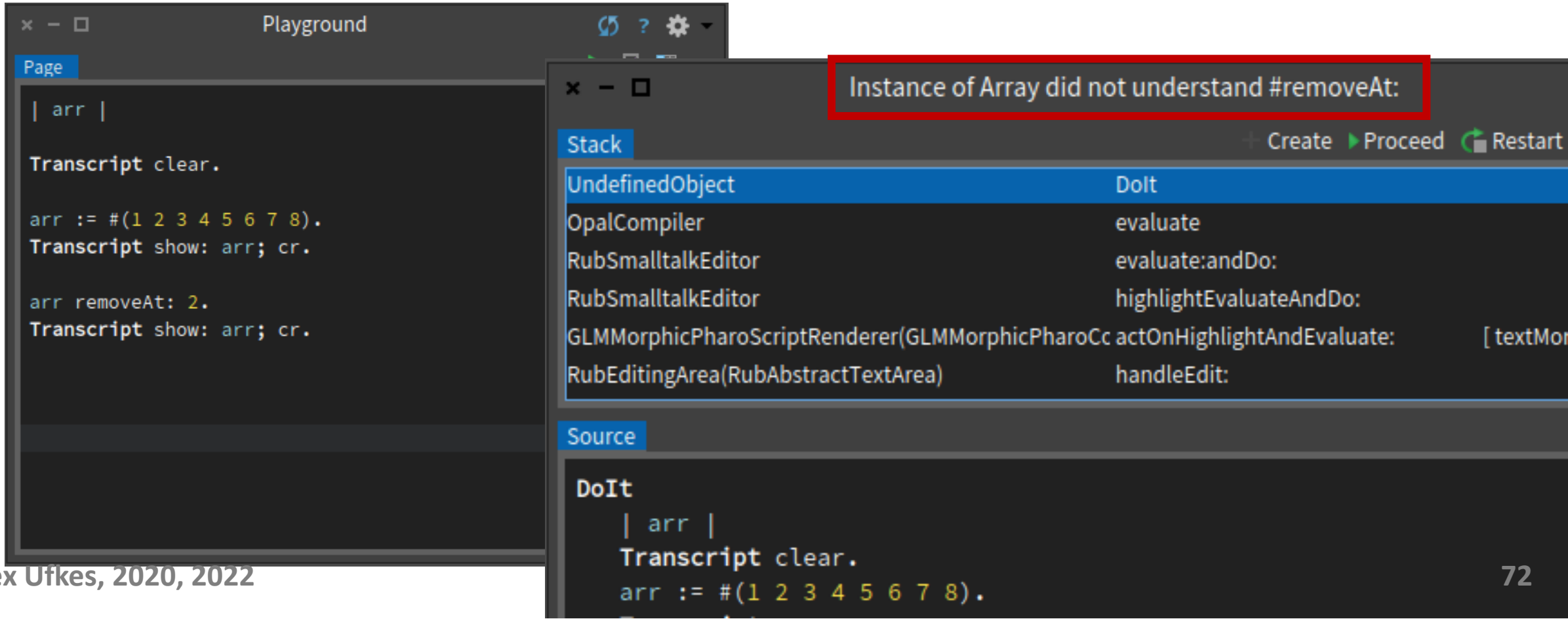
Can use **`removeAt:`**



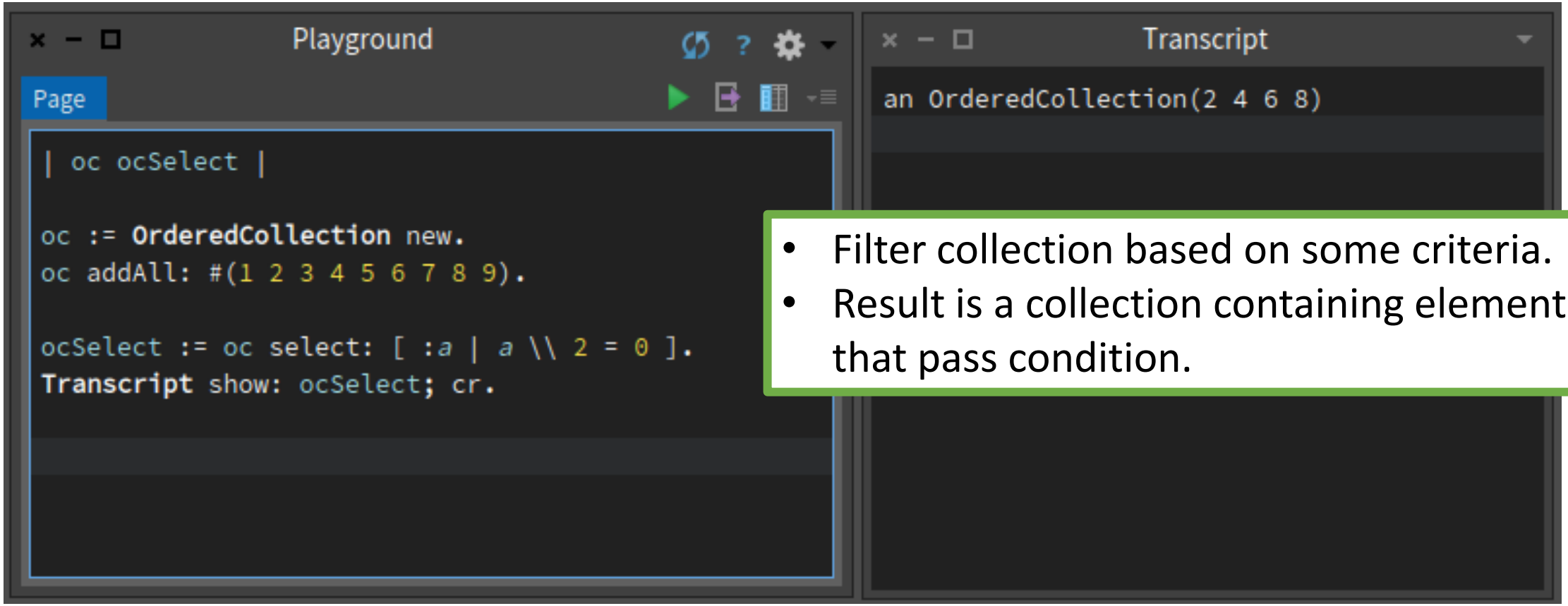
- Remember Smalltalk is one-indexed
- Mutates the original!

With Arrays?

No! Arrays cannot change size once created:

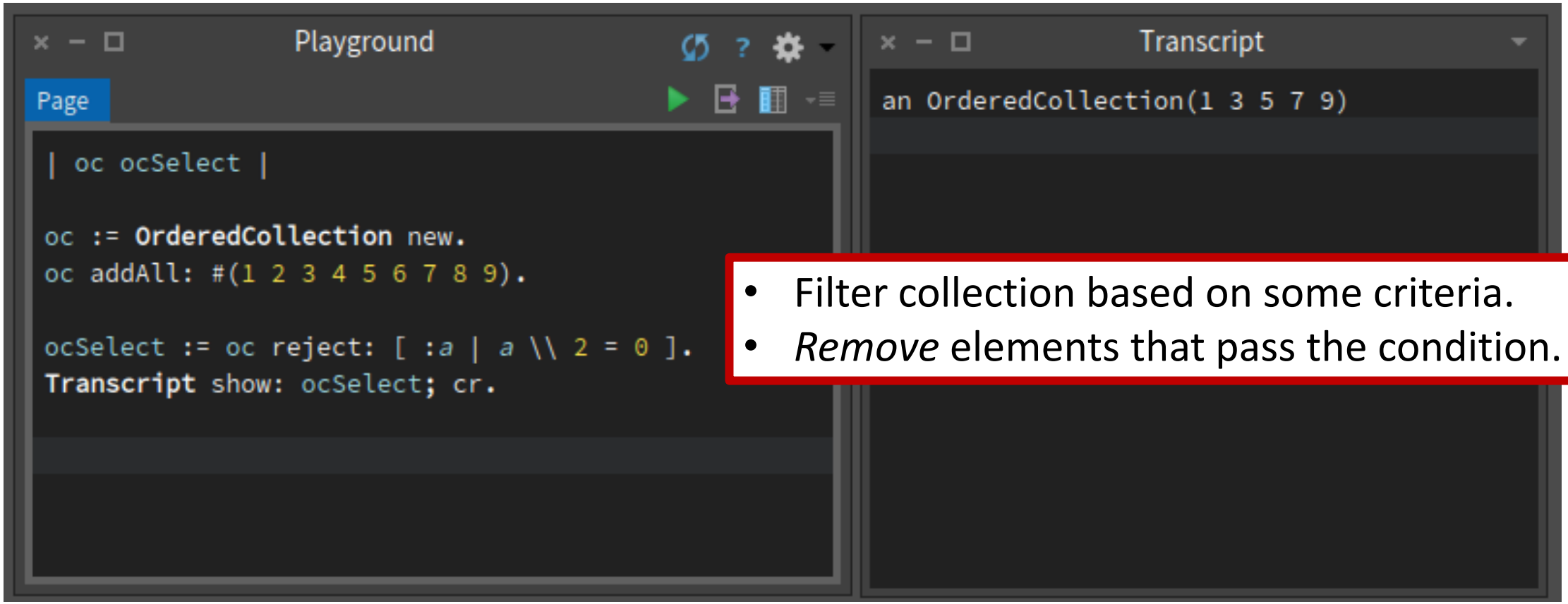


Ordered Collection: `select:`



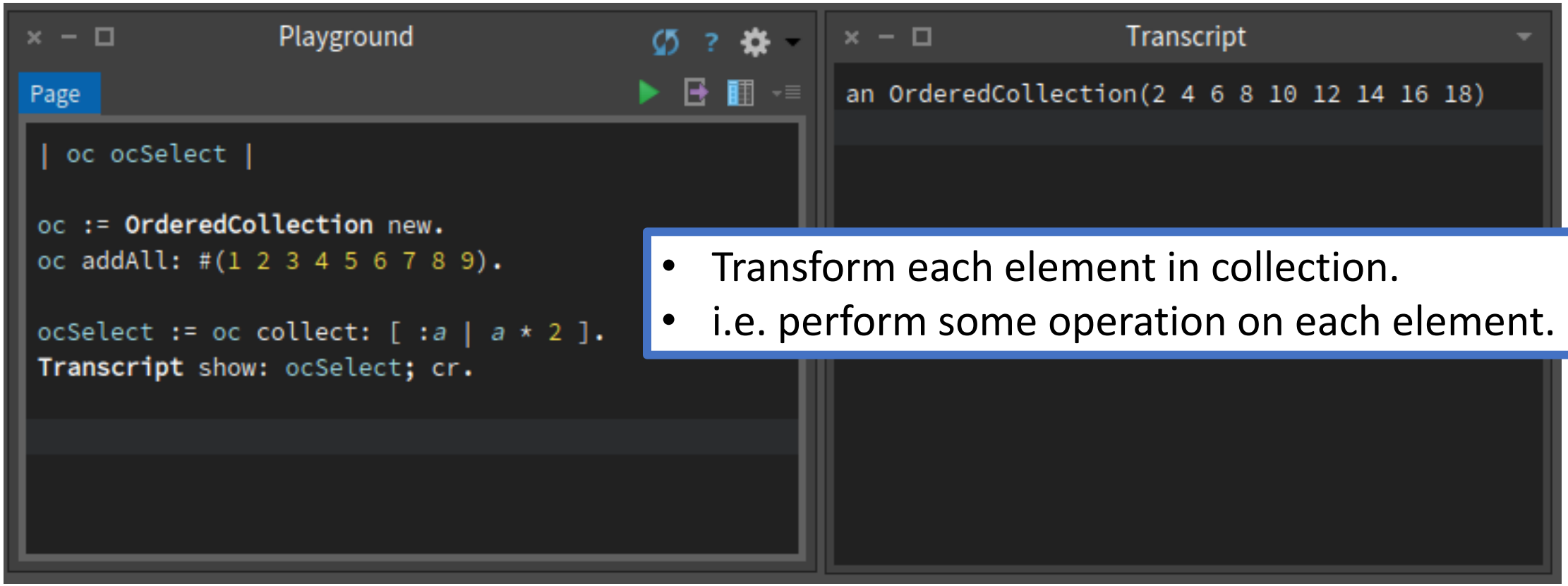
- Filter collection based on some criteria.
- Result is a collection containing elements that pass condition.

Ordered Collection: `reject:`



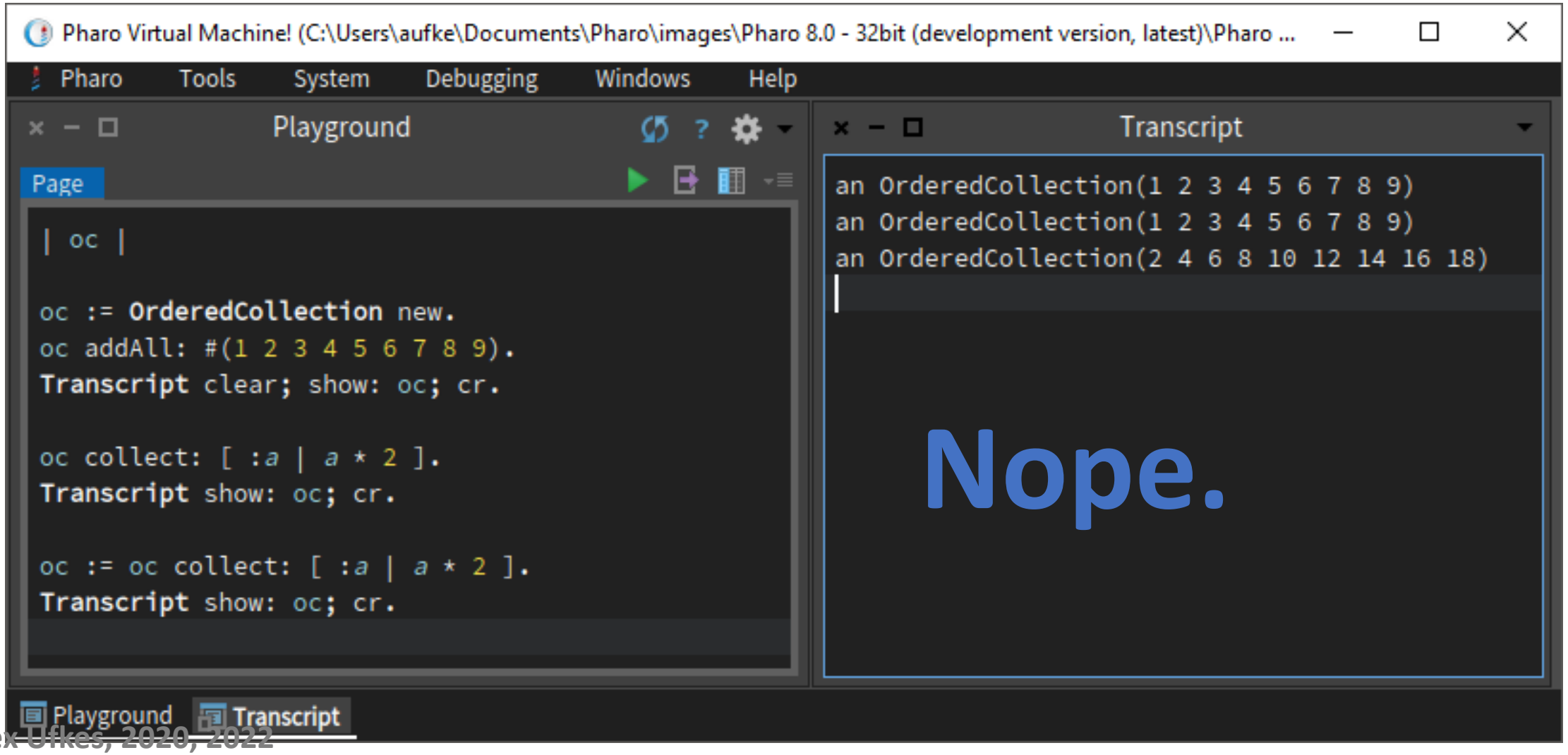
- Filter collection based on some criteria.
- *Remove* elements that pass the condition.

Ordered Collection: `collect:`



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

Are we mutating the original?



And More!

OrderedCollection

Acts like an expandable array

```
| b x y sum max |
x := OrderedCollection with: 4 with: 3 with: 2 with: 1.      "create collection with up to 4 elements"
x := OrderedCollection new.                                  "allocate collection"
x add: 3; add: 2; add: 1; add: 4; yourself.                  "add element to collection"
y := x addFirst: 5.                                          "add element at beginning of collection"
y := x removeFirst.                                          "remove first element in collection"
y := x addLast: 6.                                           "add element at end of collection"
y := x removeLast.                                           "remove last element in collection"
y := x addAll: #(7 8 9).                                     "add multiple elements to collection"
y := x removeAll: #(7 8 9).                                  "remove multiple elements from collection"
x at: 2 put: 3.                                              "set element at index"
y := x remove: 5 ifAbsent: [].                               "remove element from collection"
b := x isEmpty.                                              "test if empty"
y := x size.                                                 "number of elements"
y := x at: 2.                                                "retrieve element at index"
y := x first.                                                "retrieve first element in collection"
y := x last.                                                 "retrieve last element in collection"
b := x includes: 5.                                          "test if element is in collection"
y := x copyFrom: 2 to: 3.                                    "subcollection"
y := x indexOf: 3 ifAbsent: [0].                             "first position of element within collection"
y := x occurrencesOf: 3.                                     "number of times object in collection"
x do: [:a | Transcript show: a printString; cr].             "iterate over the collection"
b := x conform: [:a | (a >= 1) & (a <= 4)].                  "test if all elements meet condition"
y := x select: [:a | a > 2].                                 "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                 "return collection of elements that fail test"
y := x collect: [:a | a + a].                                "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                      "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                  "sum elements"
sum := 0. 1 to: (x size) do: [:a | sum := sum + (x at: a)]. "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                    "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                    "find max element in collection"
    ifTrue: [a]
    ifFalse: [c]].
y := x shuffled.                                             "randomly shuffle collection"
y := x asArray.                                              "convert to array"
```

Sorted Collection

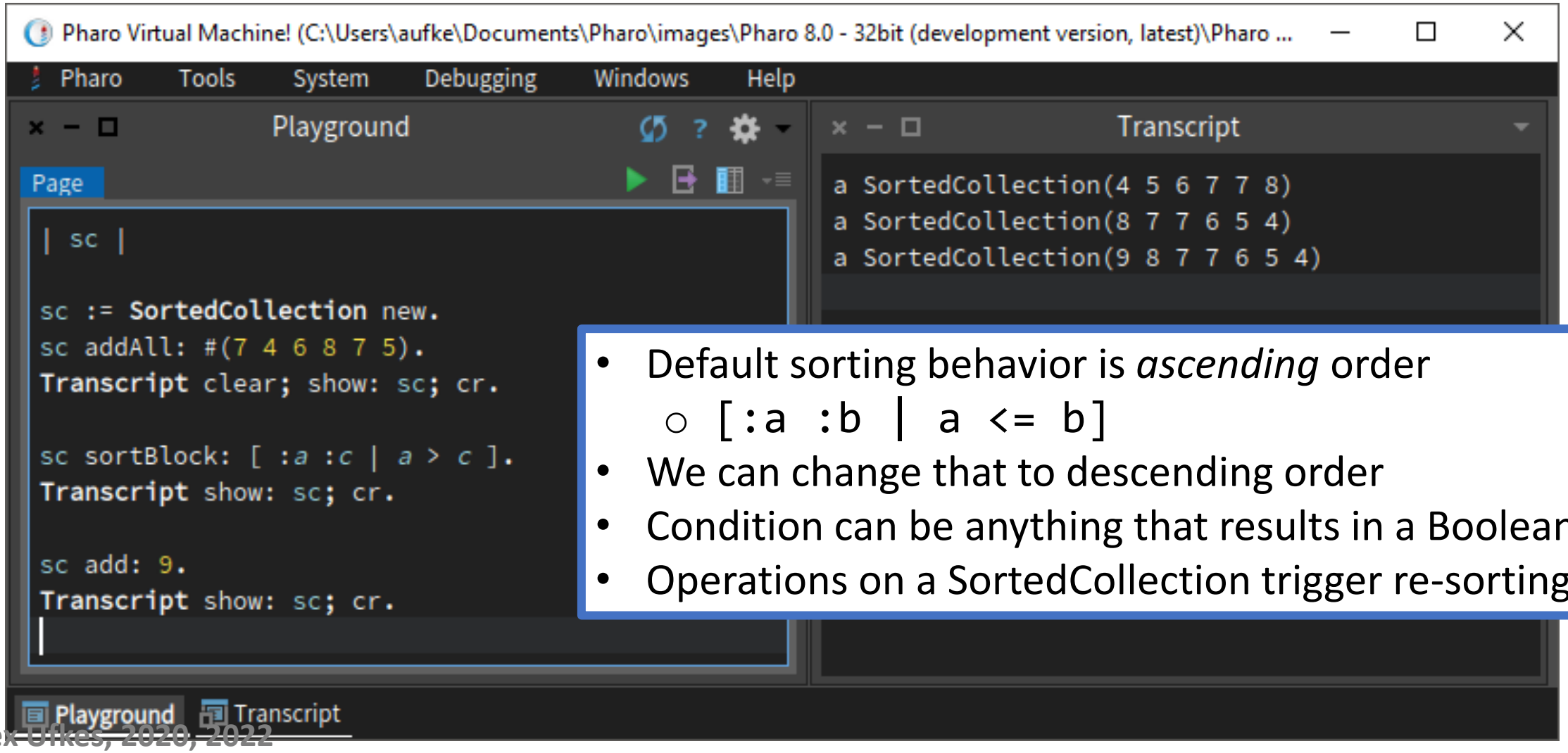
Similar to an OrderedCollection, but, you know, sorted.

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



- 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]`

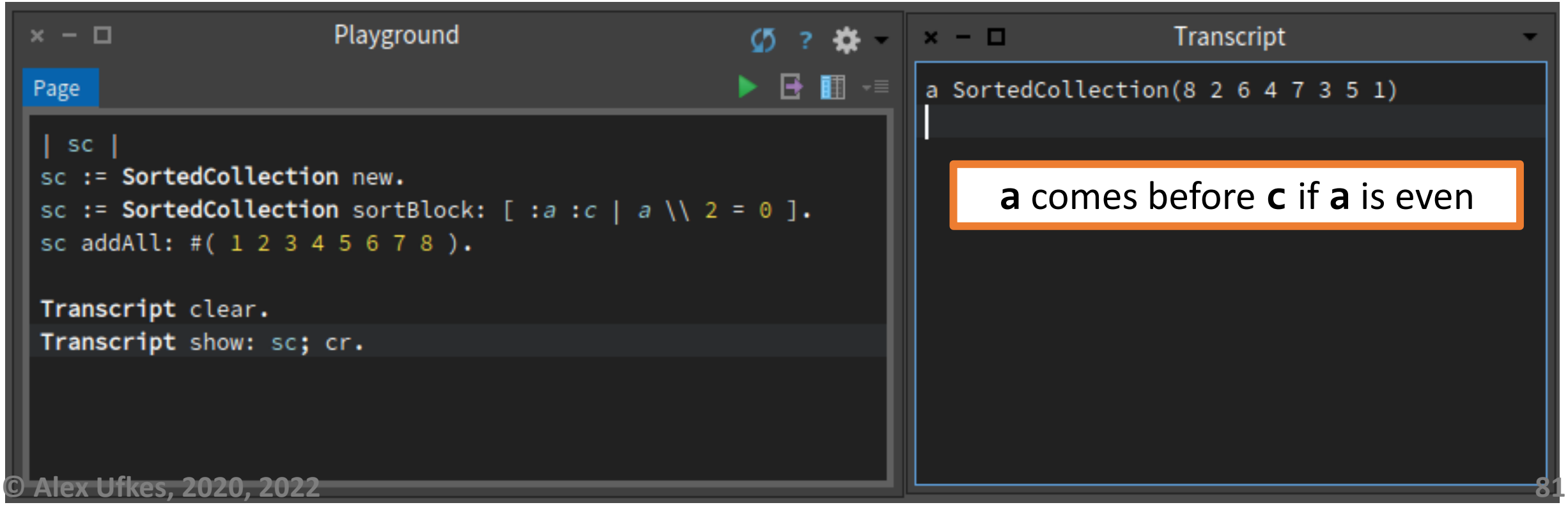
- This is a block that evaluates to Boolean when executed
- We're defining the condition for **a** appearing before **c** in the sequence
- Ascending order: **a** comes before **c** if **a** is less than or equal to **c**
- If block is true, **a** comes first.

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

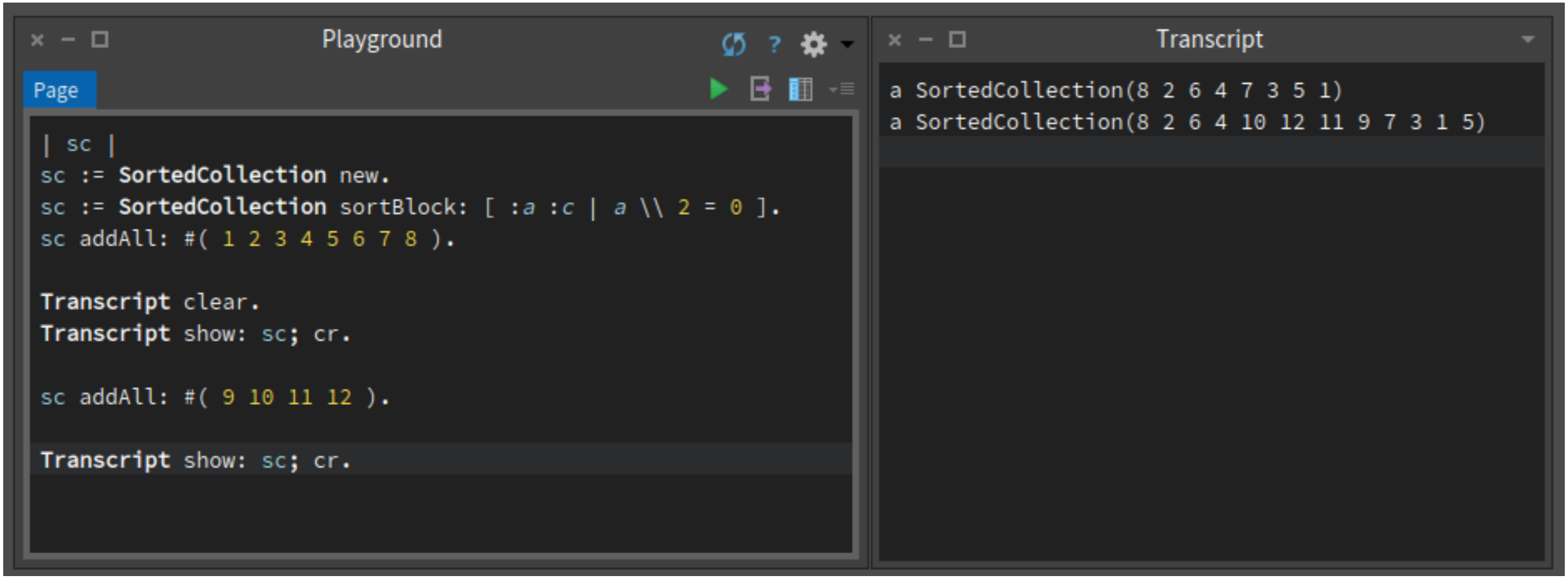
Sorted Collection

Silly sorting criteria?

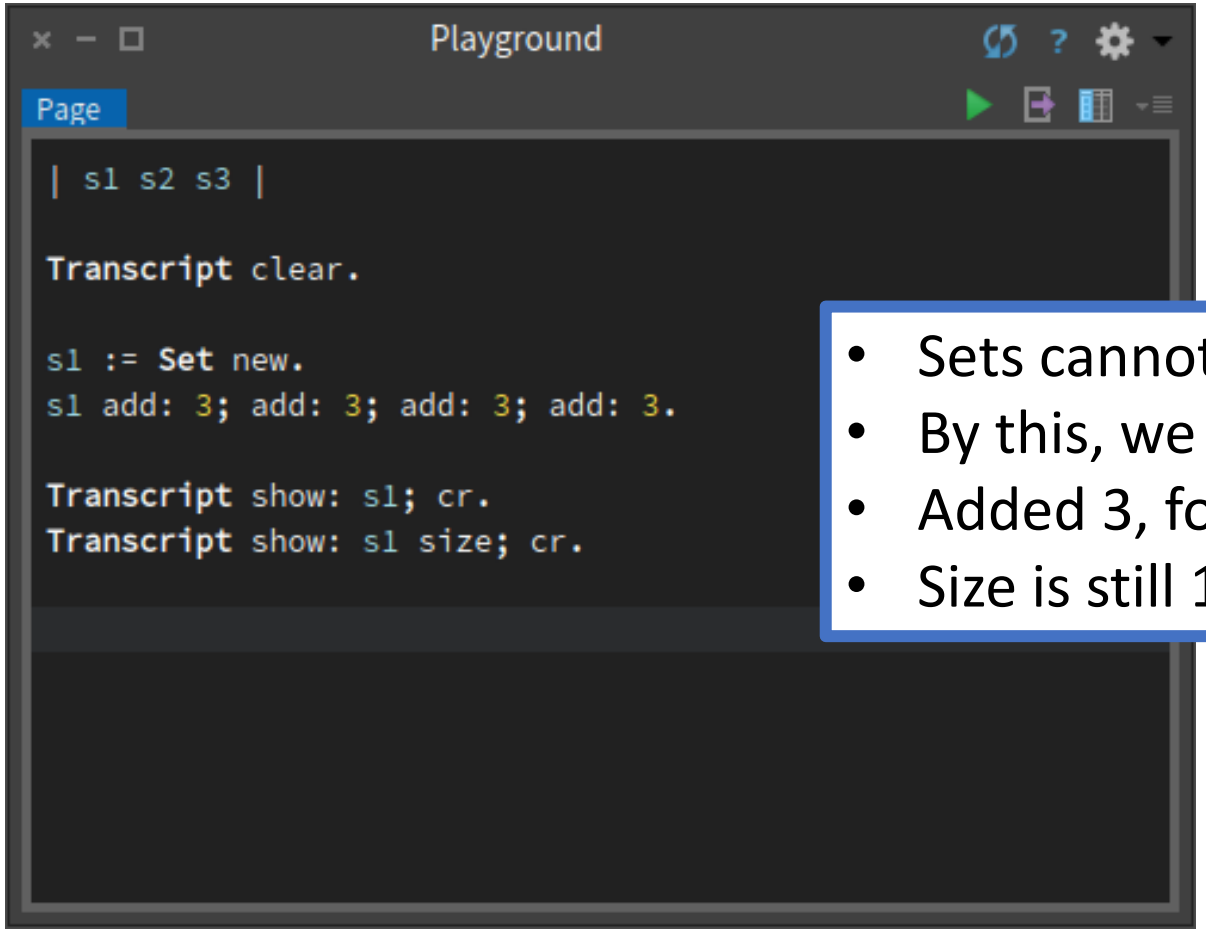
```
[:a :b | a \\ 2 = 0]
```

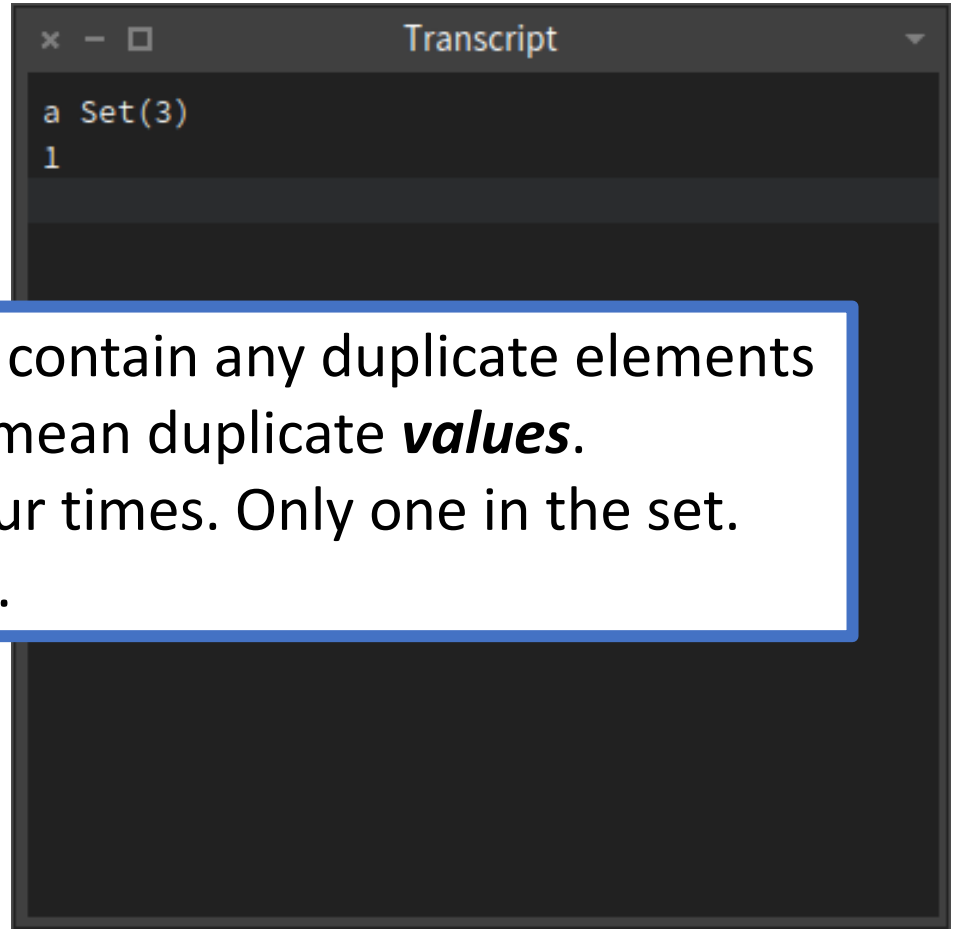


Modifying collection triggers re-sorting



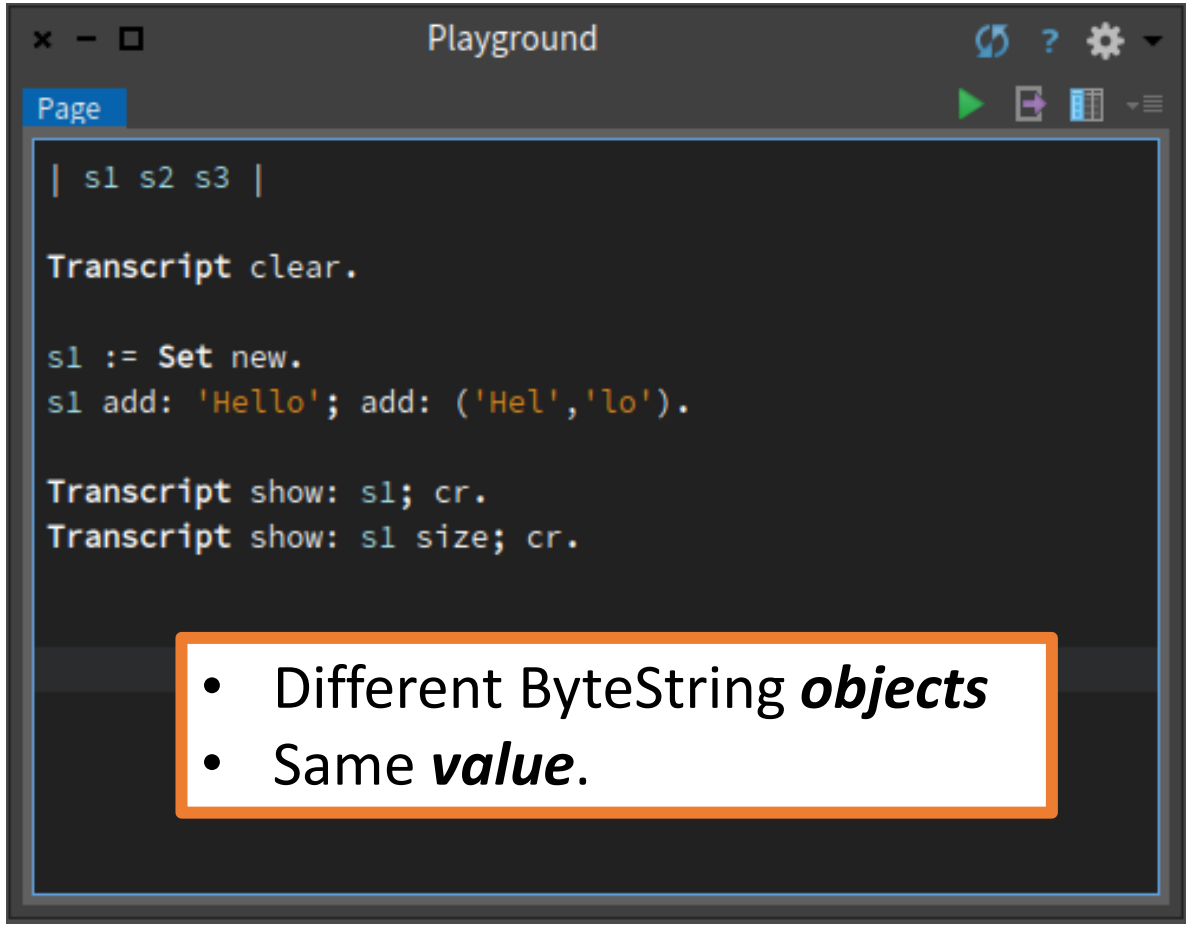
Collections: Sets



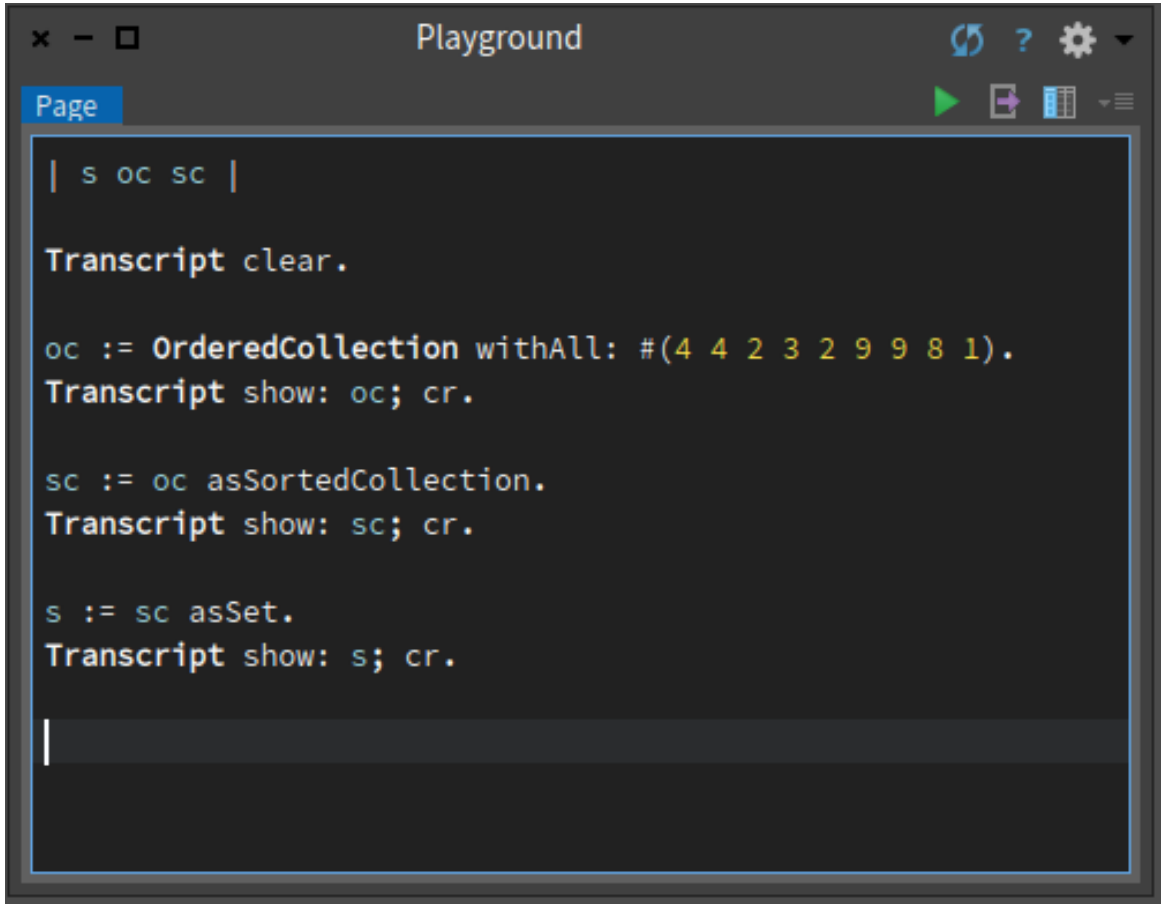


- Sets cannot contain any duplicate elements
- By this, we mean duplicate ***values***.
- Added 3, four times. Only one in the set.
- Size is still 1.

Collections: Sets

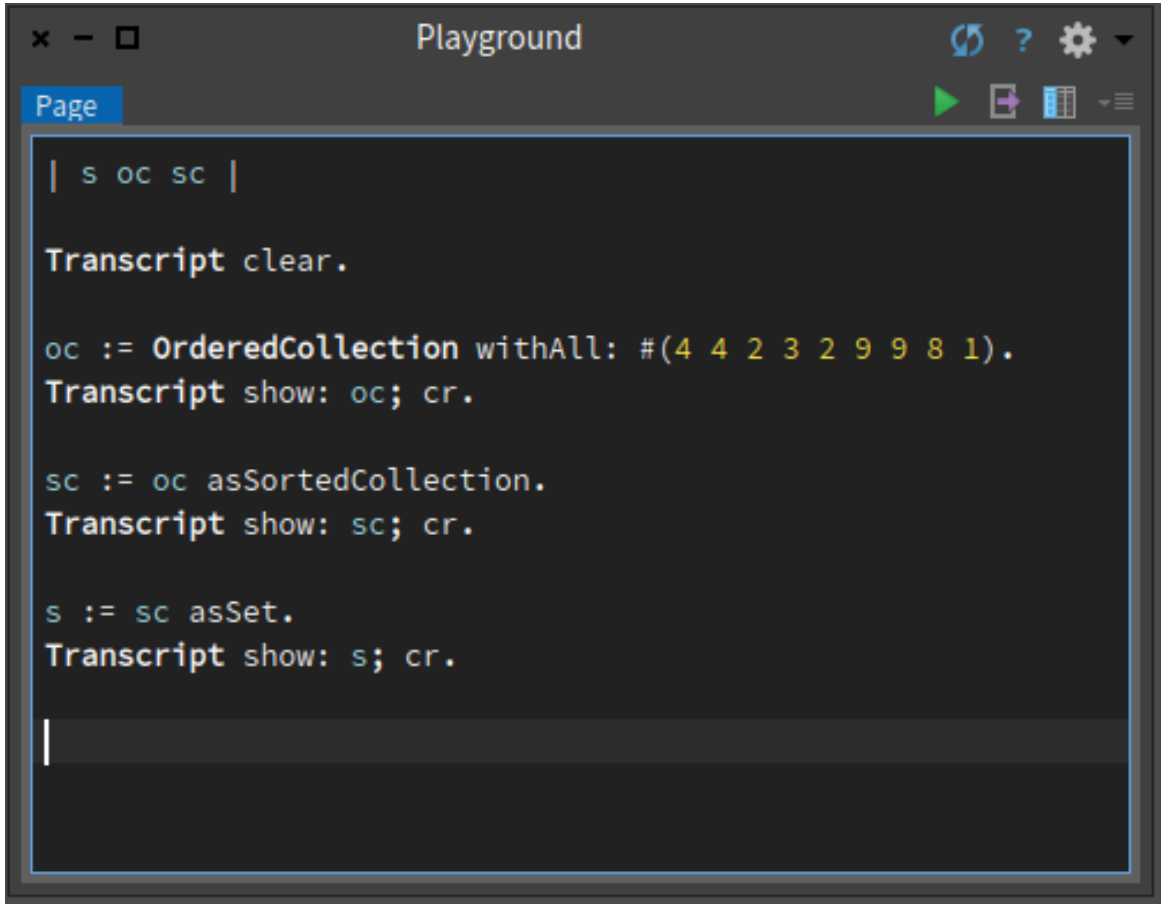


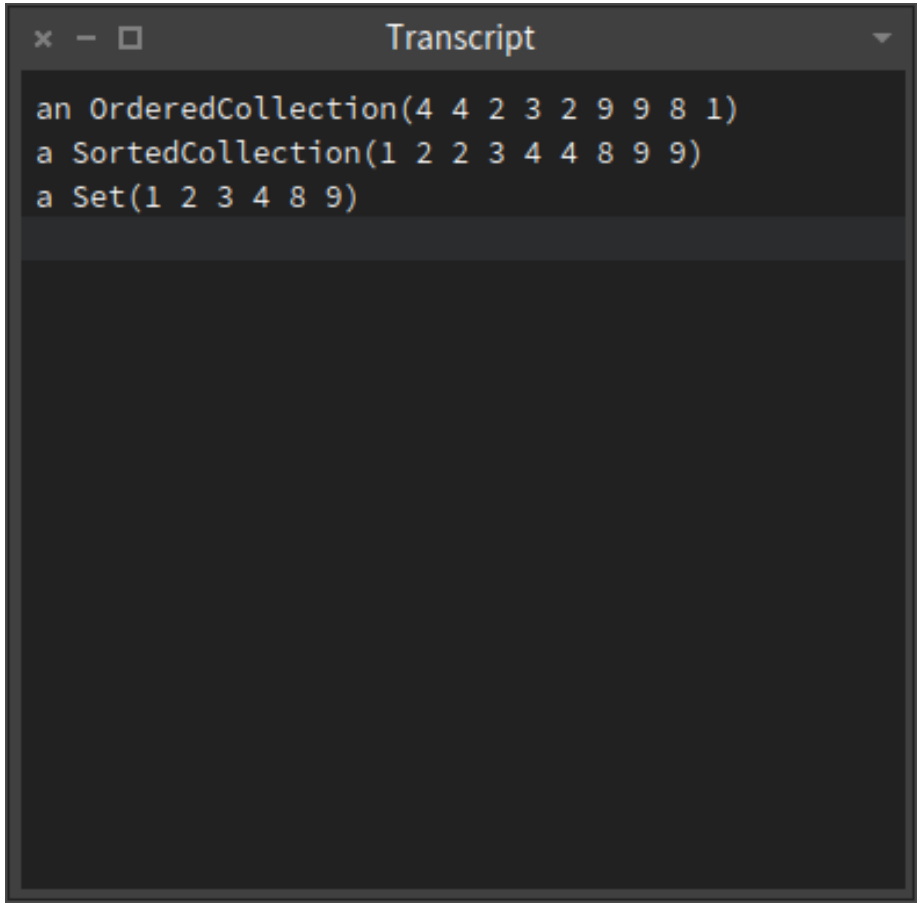
Collections: Sets



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

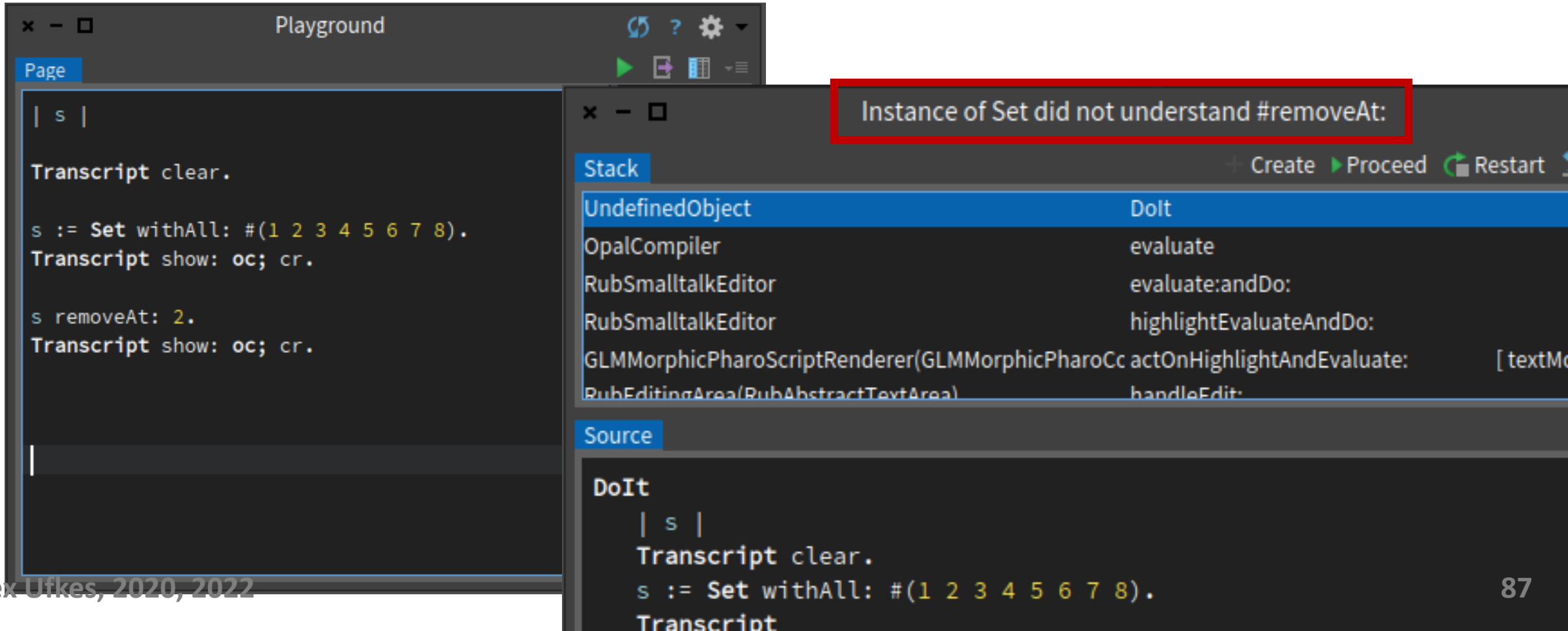
Collections: Sets



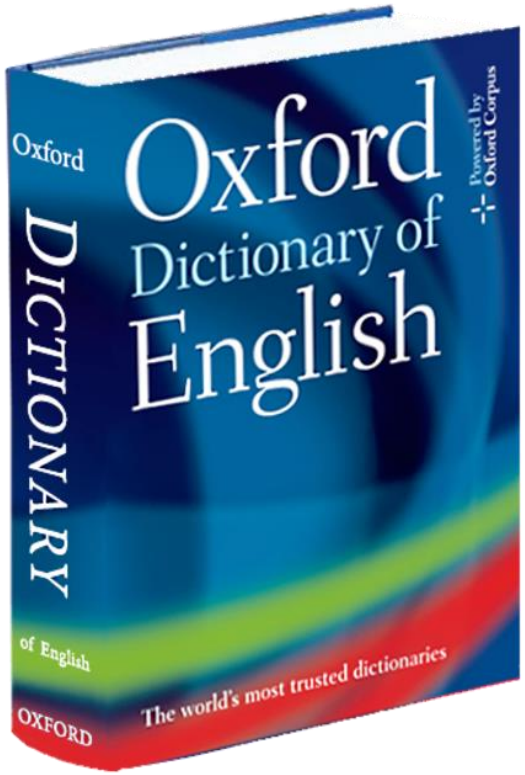


Remove by Index?

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

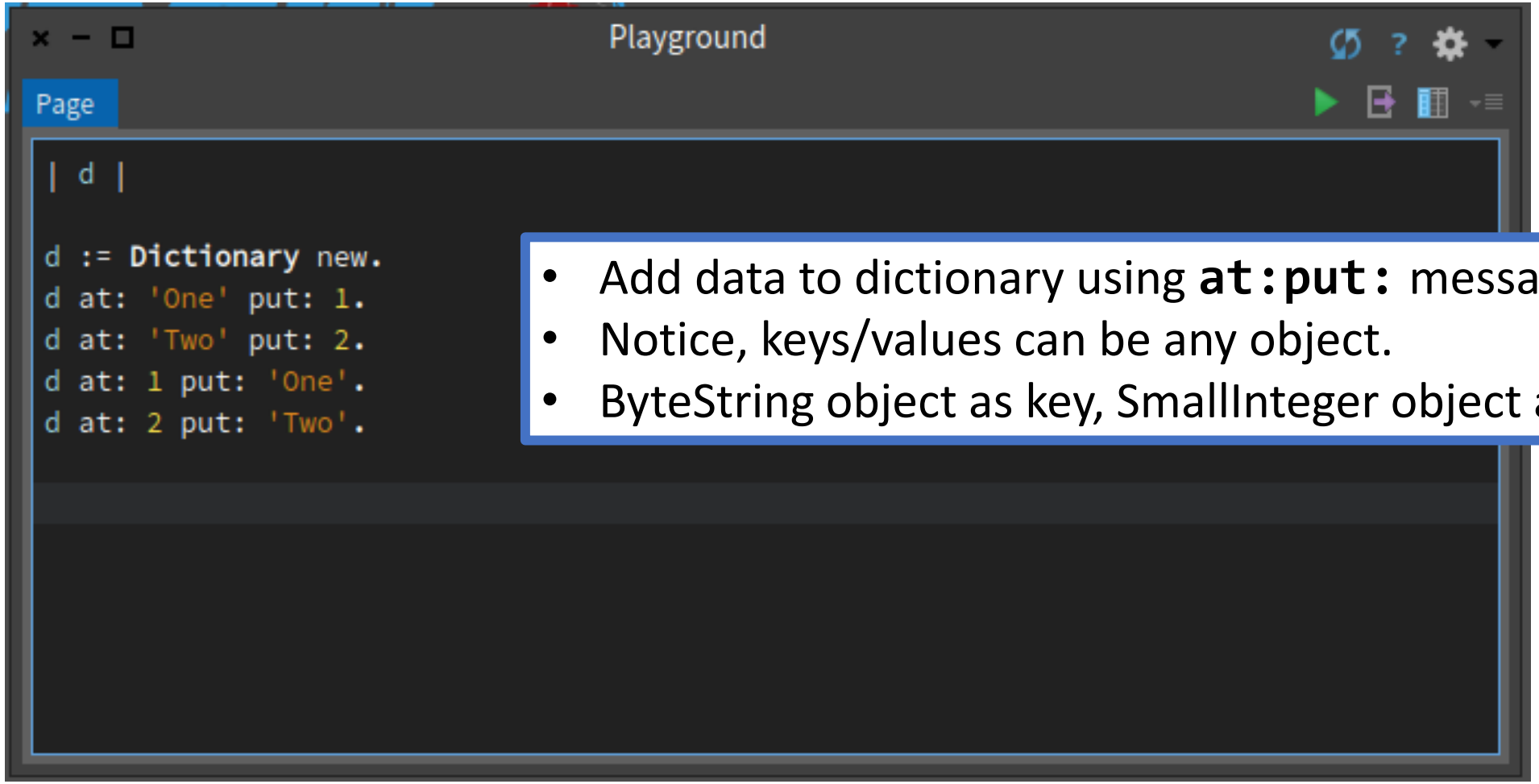


Dictionary



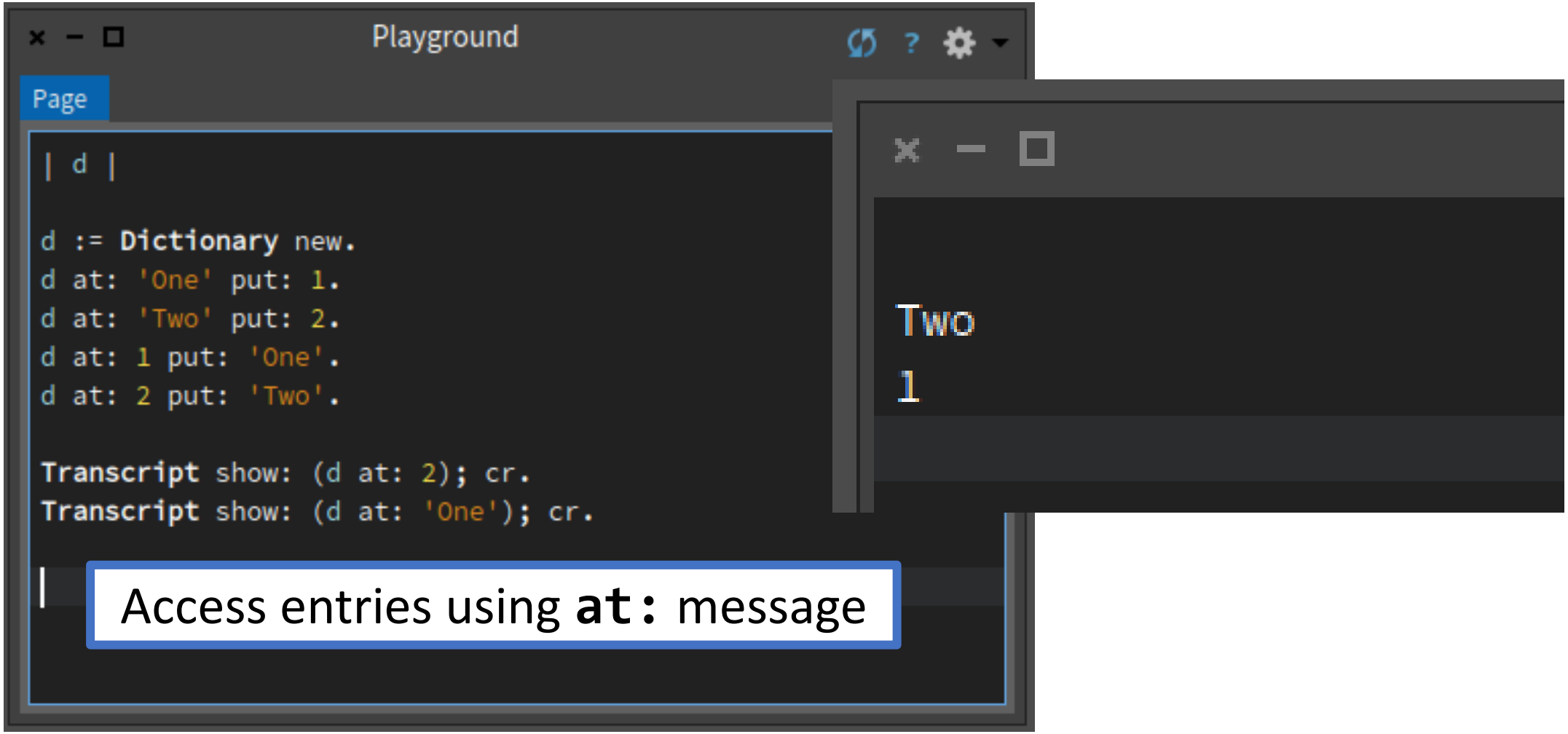
- Arrays are indexed with integers; Dictionaries are indexed with *any object at all.*
- You'll know exactly how this works if you learned Python in C/CPS 109
- Store key/value pairs, key can be anything.
- Very powerful, but we give up ordering (hash table implementation)

Dictionary: Adding Entries

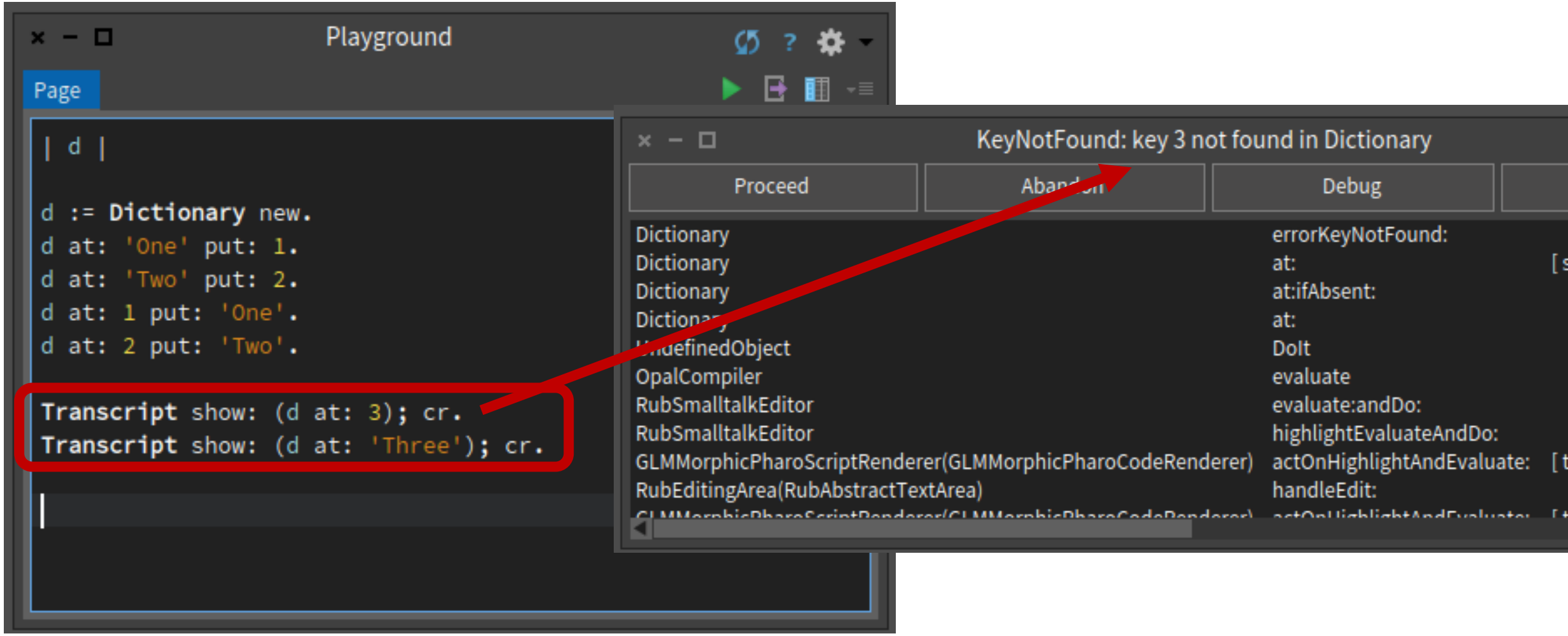


- Add data to dictionary using **`at:put:`** message
- Notice, keys/values can be any object.
- ByteString object as key, SmallInteger object as key.

Dictionary: Getting Value with Key

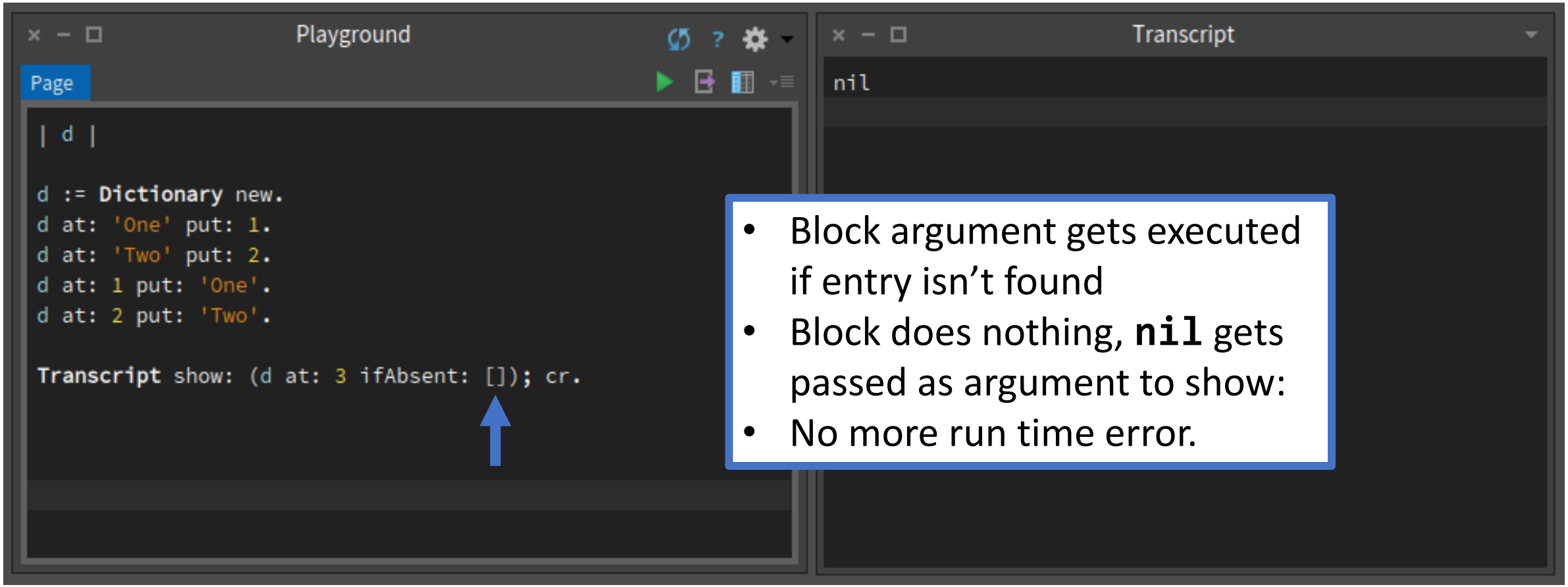


Dictionary: Key not Found?

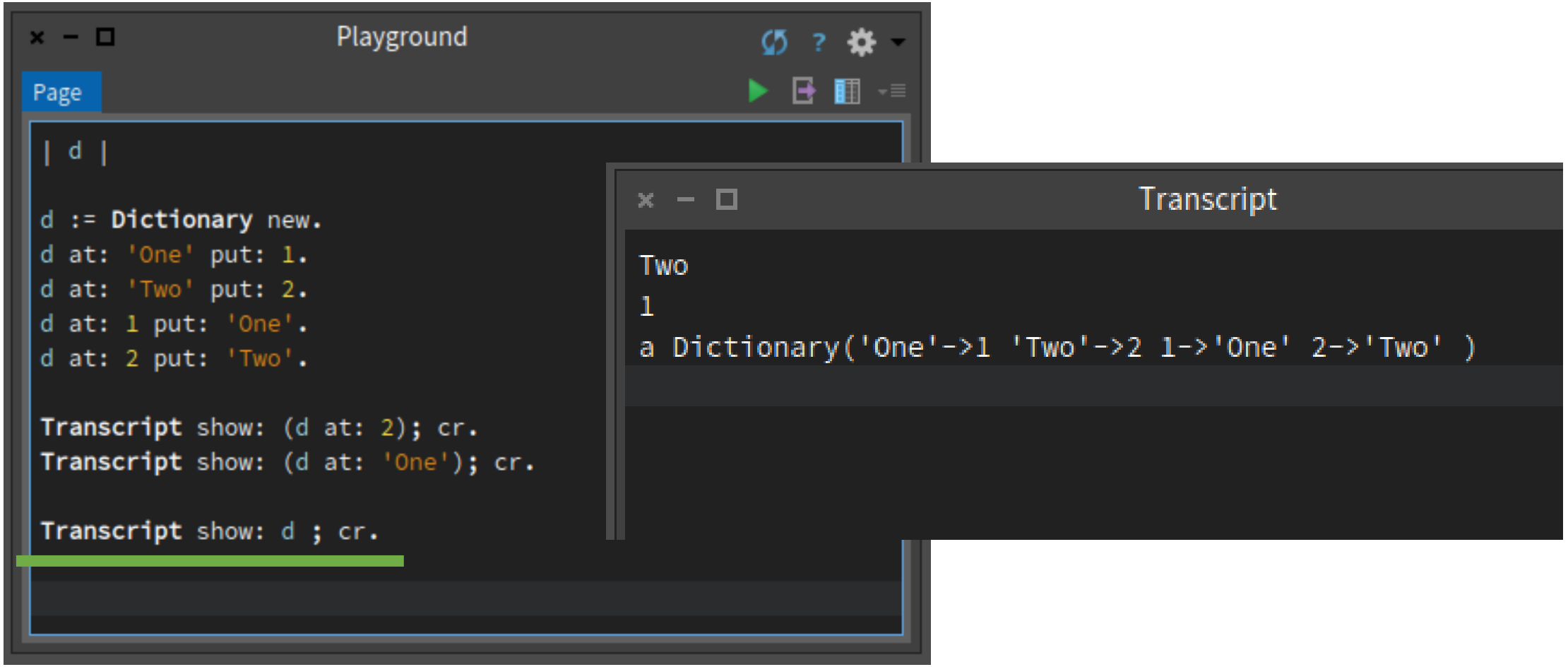


Not very graceful…

Try **`at:ifAbsent:`** instead



Dictionary: Printing



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!

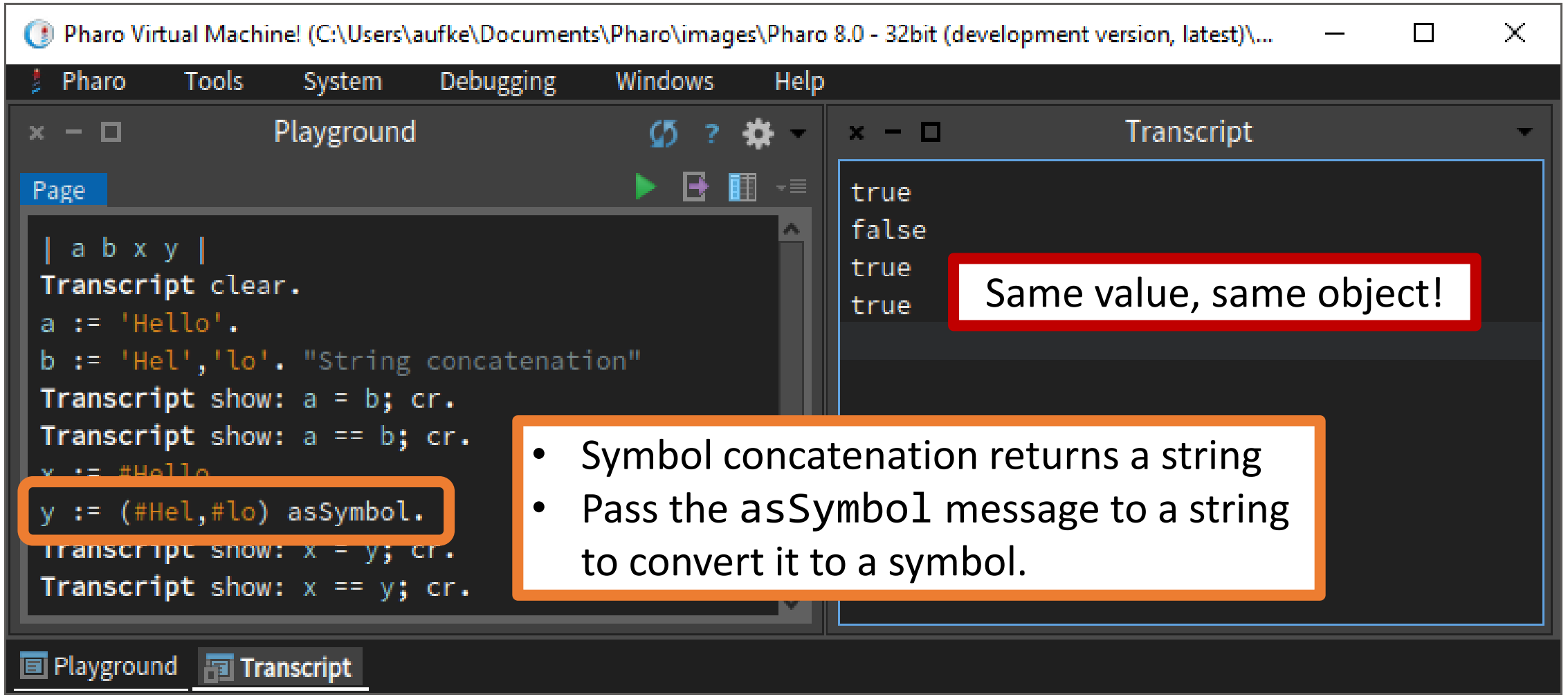
Symbol Reminder

Symbols:

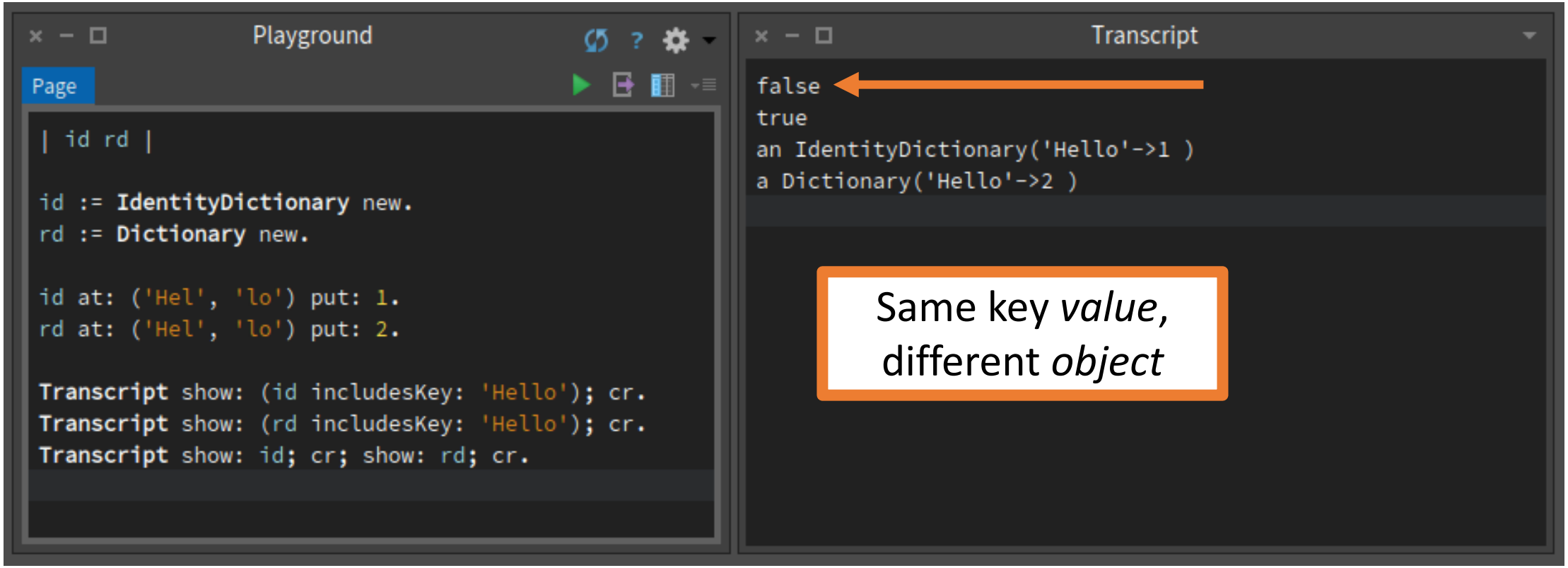
- # followed by string literal
  - **`#‘aSymbol’`** same as **`#aSymbol`** (no whitespace, quotes implied)
  - **`#‘symbol one’ #‘symbol two’`**
- Symbols are ***globally unique***. Strings are *not.*

Meaning:

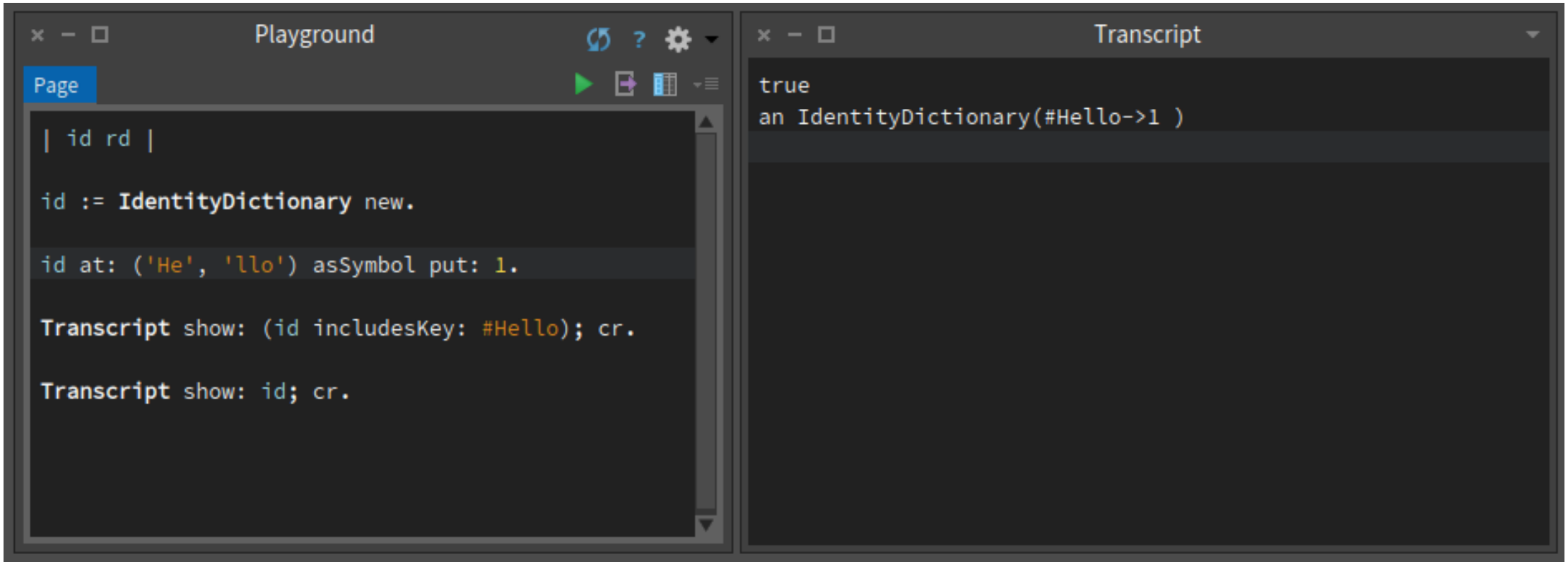
- Two *identical* (value) strings can exist as two *different* objects
- For every *unique* symbol value, there can be only *one* object.



Identity Dictionary



With Symbols?



Dictionaries

And much more!

```
| b x y sum max |
x := Dictionary new.                                          "allocate collection"
x add: #a->4; add: #b->3; add: #c->1; add: #d->2; yourself.   "add element to collection"
x at: #e put: 3.                                              "set element at index"
b := x isEmpty.                                               "test if empty"
y := x size.                                                  "number of elements"
y := x at: #a ifAbsent: [].                                   "retrieve element at index"
y := x keyAtValue: 3 ifAbsent: [].                            "retrieve key for given value with error block"
y := x removeKey: #e ifAbsent: [].                            "remove element from collection"
b := x includes: 3.                                           "test if element is in values collection"
b := x includesKey: #a.                                       "test if element is in keys collection"
y := x occurrencesOf: 3.                                      "number of times object in collection"
y := x keys.                                                  "set of keys"
y := x values.                                                "bag of values"
x do: [:a | Transcript show: a printString; cr].              "iterate over the values collection"
x keysDo: [:a | Transcript show: a printString; cr].          "iterate over the keys collection"
x associationsDo: [:a | Transcript show: a printString; cr]."iterate over the associations"
x keysAndValuesDo: [:aKey :aValue | Transcript                "iterate over keys and values"
   show: aKey printString; space;
   show: aValue printString; cr].
b := x conform: [:a | (a >= 1) & (a <= 4)].                   "test if all elements meet condition"
y := x select: [:a | a > 2].                                  "return collection of elements that pass test"
y := x reject: [:a | a < 2].                                  "return collection of elements that fail test"
y := x collect: [:a | a + a].                                 "transform each element for new collection"
y := x detect: [:a | a > 3] ifNone: [].                       "find position of first element that passes test"
sum := 0. x do: [:a | sum := sum + a]. sum.                   "sum elements"
sum := x inject: 0 into: [:a :c | a + c].                     "sum elements"
max := x inject: 0 into: [:a :c | (a > c)                     "find max element in collection"
   ifTrue: [a]
   ifFalse: [c]].
y := x asArray.                                               "convert to array"
y := x asOrderedCollection.                                   "convert to ordered collection"
y := x asSortedCollection.                                    "convert to sorted collection"
y := x asBag.                                                 "convert to bag collection"
y := x asSet.                                                 "convert to set collection"
```

Date

```
| b x y |
x := Date today.
x := Date dateAndTimeNow.
x := Date readFromString: '01/
x := Date newDay: 12 month: #J
x := Date fromDays: 36000.
y := Date dayOfWeek: #Monday.
y := Date indexOfMonth: #Janua
y := Date daysInMonth: 2 forYe
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 tod
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 * 6
y := Time millisecondClockValu
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: 3.
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.
```

FileStream

```
| b x ios |
ios := FileStream newFileNamed: 'ios.txt'.
ios nextPut: $H; cr.
ios nextPutAll: 'Hello File'; cr.
'Hello File' printOn: ios.
'Hello File' storeOn: ios.
                              "total seconds sinc
                              "seconds past minut
                              "minutes past hour
```

There are many other classes for you to explore:

Interval

```
| b x y sum max |
x := Interval from: 5 to: 10.                    "create
x := 5 to: 10.
x := Interval from: 5 to: 10 by: 2.              "create
x := 5 to: 10 by: 2.
b := x isEmpty.                                  "test if
y := x size.                                     "number
x includes: 9.                                   "test if
x do: [:k | Transcript show: k printString; cr]. "iterate
b := x conform: [:a | (a >= 1) & (a <= 4)].      "test if
y := x select: [:a | a > 7].                     "return
                                                 "return
                                                 "transfo
                                                 "find po
                                                 "sum ele
                                                 "sum ele
                                                 "sum ele
                                                 "find ma
                                                 "convert
                                                 "convert
y := x asSortedCollection.                       "convert
y := x asBag.                                    "convert
y := x asSet.                                    "convert
```

Associations

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

Dictionaries

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

Topic 3: Summary

Continuing study of Smalltalk:

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