C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 1: Imperative paradigm, Smalltalk basics

Ryerson University

Notice!

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

Instructor

Alex Ufkes

aufkes@ryerson.ca

<u>Lecture time (CCPS):</u>

Saturday: 9:00am-12:00pm

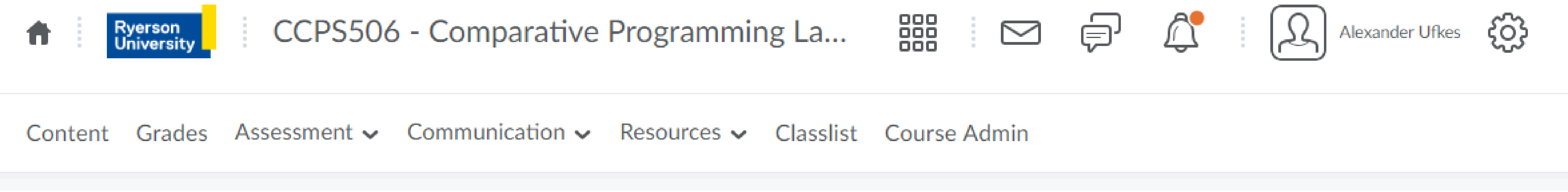
<u>Lab time (CCPS):</u>

Saturday: 12:00-1:00pm

When Contacting…

- E-mail – I check it often (**aufkes@ryerson.ca**)
- Please ***DO NOT*** email me at **aufkes@scs.ryerson.ca**
    - I don't check this one at all.
- Please put CCPS506 in the subject line
- Include your full name, use your Ryerson account

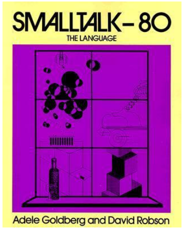
Course Administration



- Announcements related to this course will be made on D2L. Be sure to check regularly!
- Grades, assignments, and labs will be posted to D2L.
- The course outline can also be found there.

Course Synopsis

- Study fundamental concepts in the design of programming languages.
- Explore through four languages: Smalltalk, Elixir, Haskell, and Rust.



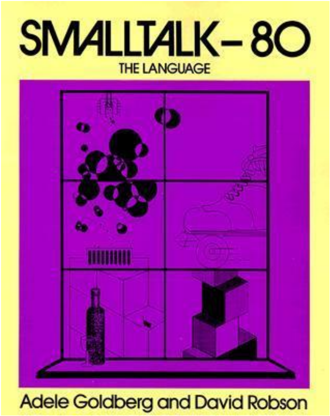
<u>Each of these differs in a number of significant language characteristics:</u>

Type systems: static VS dynamic, strong VS weak typing

Paradigm: object oriented, functional, and imperative

Syntax and semantics: scoping rules, data types, control structures, subprograms, encapsulation, concurrency, and exception handling.

Course Text



No official text for this course.
Save your money!

Lecture slides will be posted every week.

Online resources for each language will also be provided.

Evaluation (CCPS)

Labs:	20%	Two labs per language, 2.5% each
Projects:	40%	One per language, complete 2 of 4
Final Exam:	40%	Released after final lecture

All evaluation details and deadlines can be found in the course outline.

Regarding Deadlines

From the outline:

<u>Late Submissions</u>

Late submissions will be penalized at a rate of 3^n %, where n is the number of days late. One day late is a 3% penalty, two days 9%, three days 27%, four days 81%. Five days or later receives zero.

- The penalty for a couple days late is small, but it ramps up quickly.

Questions So Far?

Today

- Imperative programming paradigm
- Object Oriented Programming
- The Smalltalk programming language

Imperative Language Paradigm

Imperative Language Paradigm

This is what you're familiar with, assuming you've taken C/CPS 109/209

Imperative programming uses ***statements*** to change a program's ***state:*** ?

```
public class Tester
{
    public static void main(String[] args)
    {
        int y = 0, x = 0;
        x = 7;
        y = x*2;
    }
}
```

Statements

Program State

Programs store data in variables

Variables represent locations in the computer's memory

```
public class Tester
{
    public static void main(String[] args)
    {
        int y = 0, x = 0;
        x = 7;
        y = x*2;
    }
}
```

The contents of memory in use by a program, at any given time during its execution, is called the program's ***<u>state</u>***.

Statements can cause a program to change state:

	x	y
State 1)	0	0
State 2)	7	0
State 3)	7	14

```java
public class Tester
{
    public static void main(String[] args)
    {
        int y = 0, x = 0;
        x = 7;
        y = x*2;
    }
}
```

Fundamentally, everything is done by changing values of variables

Everyday Example?

State variables:

- Channel
- Volume

- We must know the current state of the TV, or “Volume Up” and “Channel Down” can’t be properly defined.
- Thus, current volume and channel are part of the TV’s state.

Emulator Save States

- If you've ever played a console emulator with a "save state" option, this is how they work.
- A save state is simply a memory dump of the console's RAM.

Why Imperative?

Recipes, checklists, IKEA instructions, etc. are all familiar concepts.

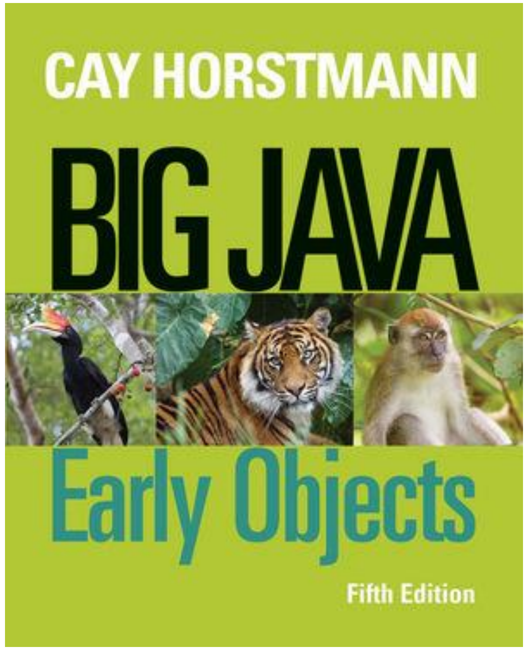
These things are not computer programs but are similar in style to imperative programming.

Understanding imperative programming is thus less of a conceptual leap for the novice programmer.

Evidence?

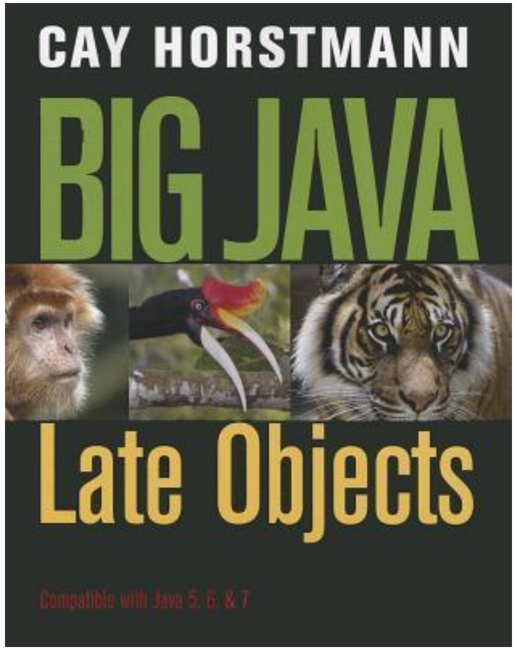
<u>**(Before switching to Python) Ryerson taught multiple versions of CPS109:**</u>

- Objects first (for people with programming experience)
- Objects later (for people new to programming)



Begins straight away with OOP principles, objects and classes.

Focuses on imperative paradigm before introducing OOP abstraction



Why Imperative?

Machine code is imperative, and nearly all computer hardware is designed to execute machine code.

From this low-level perspective, “state” can be described in terms of memory locations and machine instructions.

From a high-level language perspective, state is described in terms of variables and more complex statements

In either case, the paradigm is the same.

```
public static void main(String[] args)
{
    int y = 0, x = 0;
    x = 7;
    y = x*2;
}
```

In other words, we would want a good reason to seek an alternative to imperative programming.

Imperative Drawbacks?

- Fine for small programs, easy to keep track of a small number of variables.
- Difficult to scale up, both in terms of code size and parallelism.
- It gets very hard to model a program’s state in one’s head. This leads to convoluted debugging techniques:

C still dominates in embedded systems

```
for (int i = 0; i < SIZE; i++)
{
    /* Program code here */

    // Print and analyze entire program state each iteration to track down a bug:
    printf("value of a = %d \n", a);
    printf("value of b = %d \n", b);
    printf("value of c = %d \n", c);
    printf("value of d = %d \n", d);
    printf("value of e = %d \n", e);
    printf("value of f = %d \n", f);
    system("pause");
}
```

Procedural Programming

State changes are localized (partially or entirely) to *procedures* (functions/subroutines).

Makes imperative programs far more readable, simplifies coding, and allows for code reuse between programmers.

In C, instead of having 1000 lines of code in our **`main()`** function, we keep **`main()`** as short as possible and add user-defined functions.

```c
float dotProduct(float *vec1, float* vec2, int n)
{
    int i;
    float angle = 0, vec1len = 0, vec2len = 0;

    for (i = 0; i < n; i++) {
        angle += vec1[i] * vec2[i];
        vec1len += vec1[i] * vec1[i];
        vec2len += vec2[i] * vec2[i];
    }

    angle = (float)acos(angle / (sqrtf(vec1len)*sqrtf(vec2len)));

    return (float)(angle*(180.0 / PI));
}

void crossProduct(float *vec1, float* vec2, float *returnVec)
{
    returnVec[0] = vec1[1] * vec2[2] - vec2[1] * vec1[2];
    returnVec[1] = vec2[0] * vec1[2] - vec1[0] * vec2[2];
    returnVec[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
}

void matMul(float *mat1, int r1, int c1, float *mat2, int r2, int c2, float *result)
{
    int i, j, k;

    for (i = 0; i < r1; i++)
        for (j = 0; j < c2; j++) {
            result[(i*c2) + j] = 0;
            for (k = 0; k < r2; k++)
                result[(i*c2) + j] += mat1[(i*c1) + k] * mat2[(k*c2) + j];
        }
}

int main(void)
{
}
```

Example:

- C doesn't have native support for matrix operations.
- Write our own functions rather than duplicating code in `main()`

*"Makes imperative programs far more **readable**, simplifies coding, and allow for code reuse between programmers."*

If procedures are well written, it is often possible to discern what a procedure does based solely on the name and parameter list.

```
float addVectorElements(float* vector, int vectorLength)
{
    float sum = 0;

    for (int i = 0; i < vectorLength; i++)
    {
        sum += vector[i];
    }

    return sum;
}
```

In Summary

Imperative paradigm uses statements to change a program’s state.

- The programmer specifies an explicit sequence of steps for the program to follow.

Adding procedures/functions/subroutines can improve scalability.

- Code can be made more readable, less duplication, easier to reuse.
- Principle of modularity – separate program functionality into independent, interchangeable modules.

Alternatives?

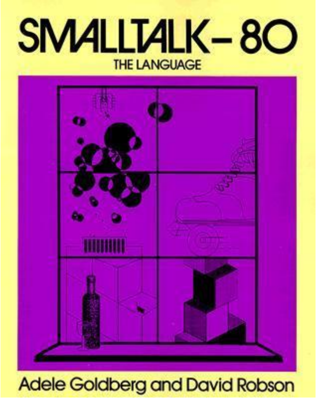
Two widely used paradigms:

<u>Functional Programming:</u>

- Avoid changing state, avoid mutable data
- *Declarative* rather than *imperative*
- Tell the program *where* to go, not *how* to get there.

<u>Object Oriented Programming:</u>

- "Pure" OO languages treat even primitives and operators as objects
- Java/C++ and others support OOP to greater or lesser degrees.



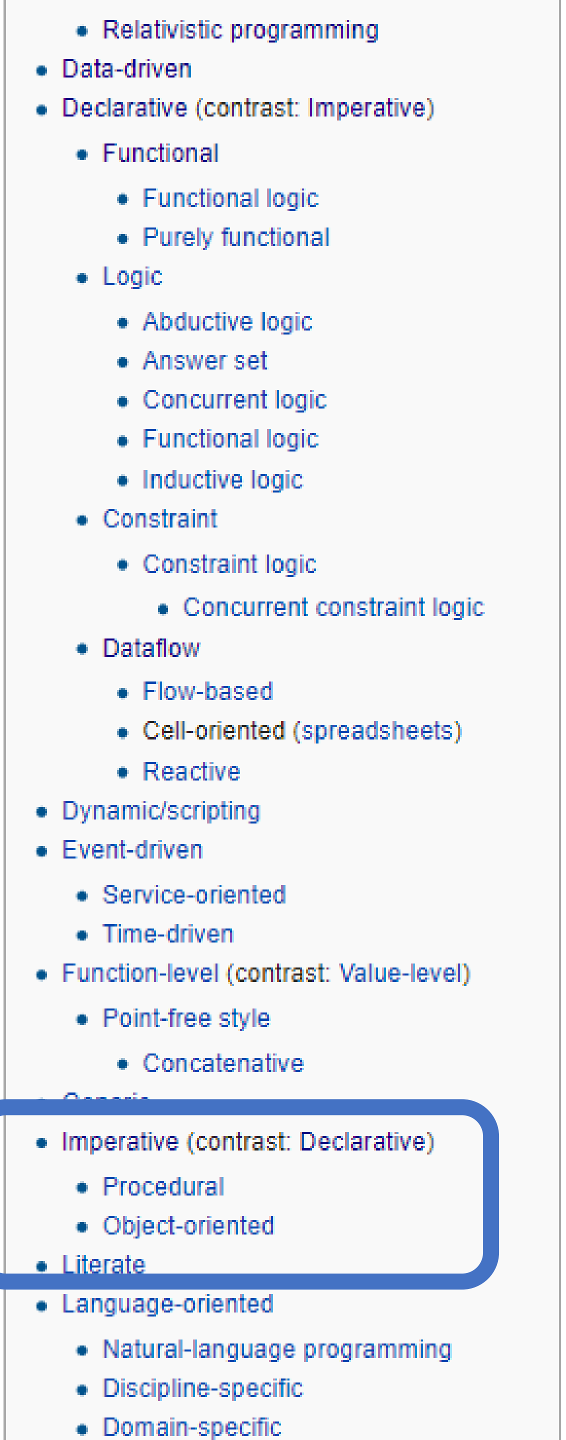
Going forward, always remember:

The line between different paradigms is grey.

Paradigms classify languages based on their features

Any given language can possess features from multiple paradigms and thus belong to all.

C is considered a very imperative language, but it supports ***first class functions*** using function pointers.



Object Oriented Paradigm

Objects?

Broadly speaking, a software construct that implements both ***state*** and ***behavior***.

We can also say that objects have ***identity***. Unique instances of the same class can exist simultaneously.

In Java, behaviors are implemented as methods, C++ as member functions. Same idea.

An object's procedures can access and modify the data fields of that object.

In the OOP paradigm, programs are built up of objects that communicate with each other.

Objects

Broadly speaking, a software construct that implements both state and behavior.

```
public class Tester
{
    public static void main(String[] args)
    {
        int x, y, z;     // Not objects!
        Integer xyz;     // Object!
        double a, b, c;  // Not objects!
        Double abc;      // Object!
        String word;     // Object!
    }
}
```

- These are *primitives*.
- They have a *state*, but no associated *behavior.*
- No associated methods.

Objects

Broadly speaking, a software construct that implements both state and behavior.

```
public class Tester
{
    public static void main(String[] args)
    {
        int x, y, z;      // Not objects!
        Integer xyz;      // Object!
        double a, b, c;   // Not objects!
        Double abc;       // Object!
        String word;      // Object!
    }
}
```

- These are *Objects*.
- They have both a state, and associated behaviors.
- Behaviors implemented via class methods.

Class-Based OOP

- Objects are instances of classes
- The class is the cookie cutter, the object is the cookie.

```
public class Tester
{
    public static void main(String[] args)
    {
        HelloWorld h1 = new HelloWorld();
        HelloWorld h2 = new HelloWorld();
        HelloWorld h3 = new HelloWorld();
        h1.print();
        h2.print();
        h3.print();
    }
}
```

Object instances

```
public class HelloWorld
{
    public void print()
    {
        System.out.println("Hello, World!");
    }
}
```

Class definition

Class-Based OOP

- Objects are instances of classes
- The class is the cookie cutter, the object is the cookie.
- OOP languages typically support notions of inheritance.

Class Integer

java.lang.Object
    java.lang.Number
        java.lang.Integer

- Integer inherits from Number
- Number inherits from Object.

All Implemented Interfaces:

Serializable, Comparable<Integer>

```
public final class Integer
extends Number
implements Comparable<Integer>
```

OOP: In Summary

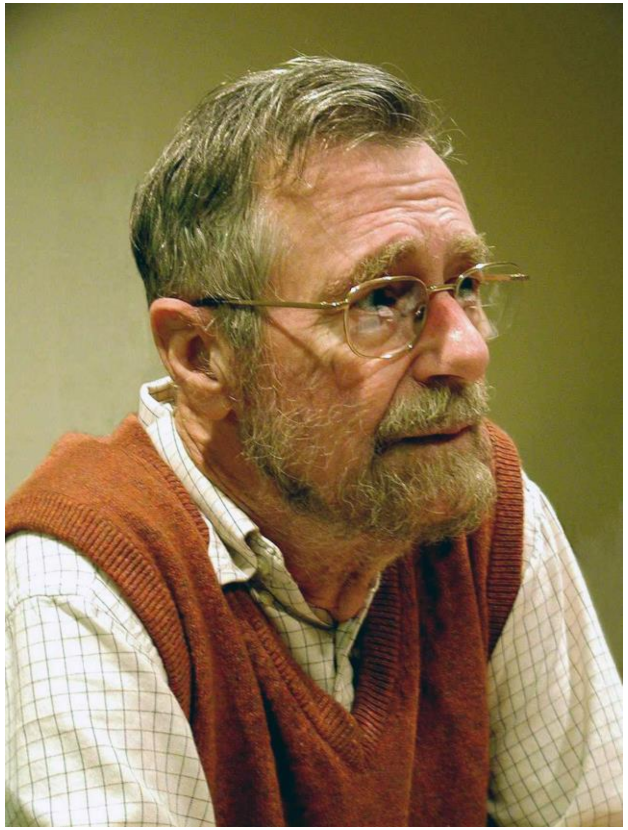
Programs are built up of objects that communicate with each other.

- Objects combine attributes (data, variables) and procedures (functions, methods).
- Most common are class-based OOP languages (C++, Java). Objects are instances of classes.
- Ideas like inheritance provide code reusability.

OOP languages are still largely imperative.

- Class methods can implement behaviors, providing abstraction.

Object Oriented Programming



“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

“Object oriented programs are offered as alternatives to correct ones…”

- Edsger Dijkstra

Smalltalk: OOP cranked up to 11

But first...

Syntax VS Semantics

- The externally visible representation of a program
- Based on sequence of characters (text-based languages)
- Easily understood in the context of a ***syntax error***:

```
public class Tester
{
    public static void main(String[] args)
    {
        int x = 4, y = 6;
        int z = x + y;
        System.out.println(z);
    }
}
```

- This Java code is *syntactically* correct.
- We know this because it compiles.
- The sequence of characters that comprise the source code make sense in the context of the Java language.

<u>Syntax</u> VS Semantics

- The externally visible representation of a program
- Based on sequence of characters (text-based languages)
- Easily understood in the context of a ***syntax error***:

```
public class Tester
{
    public static void main(String[] args)
    {
        in x = 4; y = 6;
        int z = x + y;
        Sys.out.prinln(z;
    }
}
```

- This Java code contains syntax errors. It does *not* compile.
- The sequence of characters that comprise this source code does **NOT** make sense!

A few more things relating to syntax…

Simplicity - How *much* to learn:

- Size of grammar. How "much" syntax is there?
- Complexity of navigating modules or classes
- Complexity of type system (how many types?)

Orthogonality - How *hard* to learn, how do features interact:

- How many ways can we combine grammar elements
- Type system overall (static, dynamic)

Extensibility:

- Do mechanisms exist to extend the language?
- Functionally, syntactically, defining literals, overloading, etc.

Syntax VS Semantics

- If syntax is the form, semantics is the meaning. What does the code do?
- Can be understood by showing relationship between input and output
- Code can be syntactically correct but have an unclear meaning.

```java
public class Tester
{
    public static void main(String[] args)
    {
        if (1 == 1)
            System.out.println("Hello");
        else
            System.out.println("World");
    }
}
```

- This code is syntactically correct.
- Semantically, it is somewhat confusing.

1)

```java
public class Tester
{
    public static void main(String[] args)
    {
        if (1 == 1)
            System.out.println("Hello");
        else
            System.out.println("World");
    }
}
```

- This code is syntactically correct.
- Semantically, it is confusing.
- Semantically, It is the same as:

2)

```java
public class Tester
{
    public static void main(String[] args)
    {
        System.out.println("Hello");
    }
}
```

- An understanding of a language's semantics allows us to look at **1)**, and understand it as being the same as **2)**
- Leads to more efficient machine code.

"A compiler will complain about syntax, your coworkers will complain about semantics"

Pragmatics

- What can a particular language construct be used ***for***.
- Consider the humble assignment operator (=):

```
public class Tester
{
    public static void main(String[] args)
    {
        int a = 1, b = 2, c = 3, sum;
        int d = a + b;
        sum = d + c;
        System.out.println(sum);
    }
}
```

1. Initialize variables with constants
2. Initialize variable with result of sum of two other variables.
3. Store sum of two variables in a variable

However! The assignment operator *can't* typically be used to clone arrays/objects.

Implementation

- A particular set of pragmatics that makes a program executable
- Multiple unique implementations can solve the same problem

```java
public class Tester
{
    public static void main(String[] args)
    {
        int a = 1, b = 2, c = 3, sum;
        int d = a + b;
        sum = d + c;
        System.out.println(sum);
    }
}
```

```java
public class Tester
{
    public static void main(String[] args)
    {
        int a = 1, b = 2, c = 3, sum;
        sum = a + b + c;
        System.out.println(sum);
    }
}
```

These implementations are slightly different but solve the same problem of summing three numbers and printing the result

Programming Language Characteristics

<u>Syntax – Language form:</u>

- Simplicity, how much to learn
- Orthogonality, how hard to learn, how do features interact
- Extensibility, can the language be extended by the programmer

<u>Semantics – Language meaning:</u>

- What does a block of code actually do/mean

<u>Pragmatics:</u>

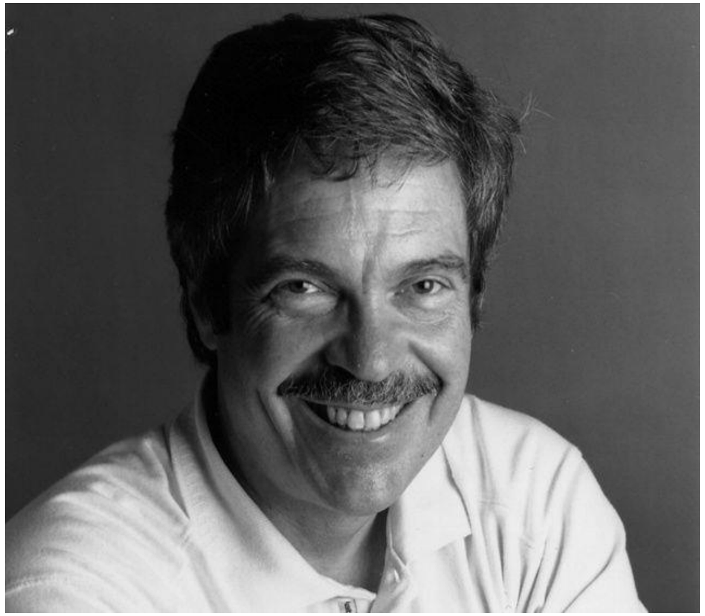
- What can a particular language construct be used for.

<u>Implementation:</u>

- A particular set of pragmatics that makes a program executable.


Smalltalk
ifTrue: [car honk]

Alan Kay

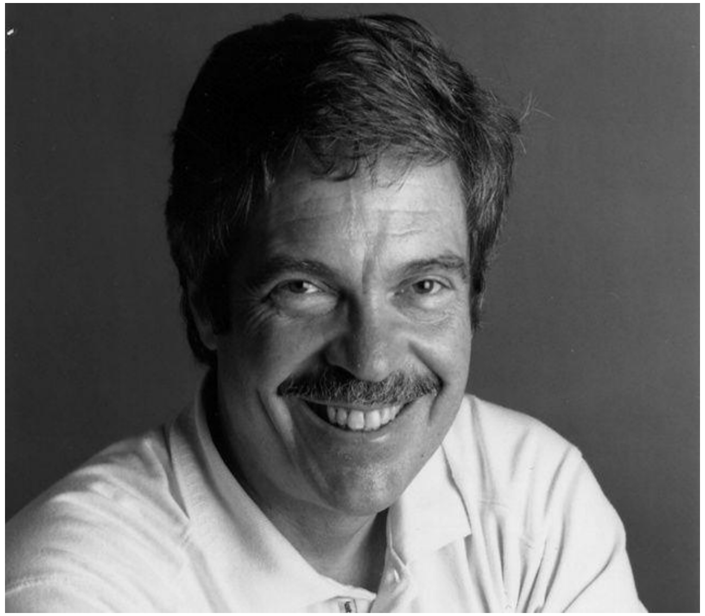


Coined the term *Object Oriented Programming* in grad school, 1966/67

Big idea:

- Use encapsulated “mini computers” in software
- Communicate via message passing, rather than direct data sharing
- Each mini computer has its own isolated state
- Inspired by biology, cellular communication.
- Avoid breaking down programs into separate data structures and procedures.

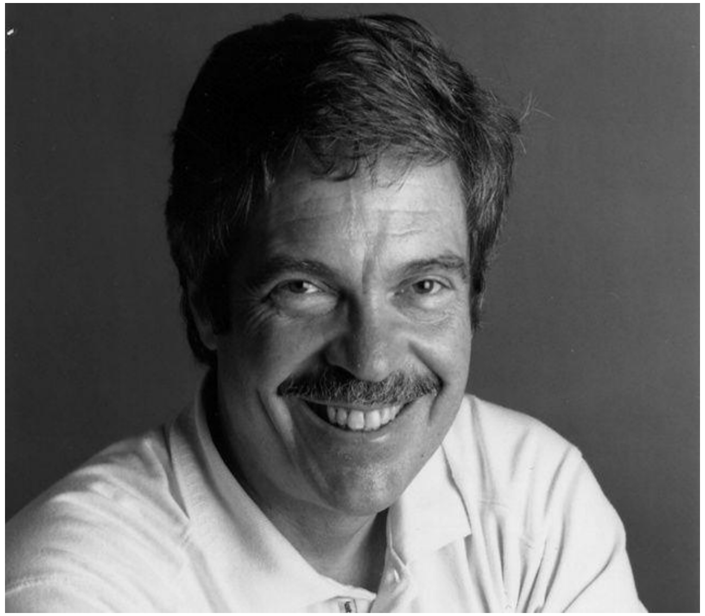
Alan Kay



In pursuit of this idea:

- Developed Smalltalk along with Dan Ingalls, Adele Goldberg, and others at Xerox PARC.
- Originally, Smalltalk did not feature sub-classing.
- Kay considers sub-classing a distraction from OOP's true benefits: ***message passing***.

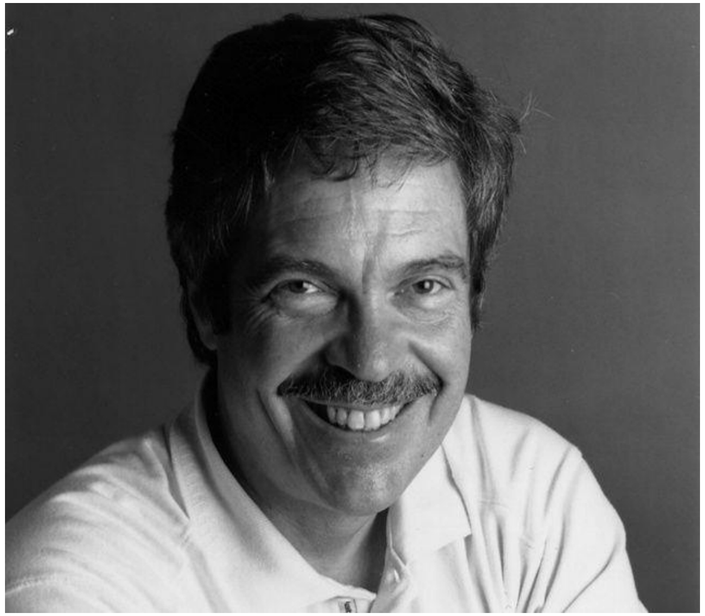
Alan Kay



“I’m sorry that I long ago coined the term “objects” for this topic because it gets many people to focus on the lesser idea. The big idea is messaging.”

“OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things..”

Alan Kay

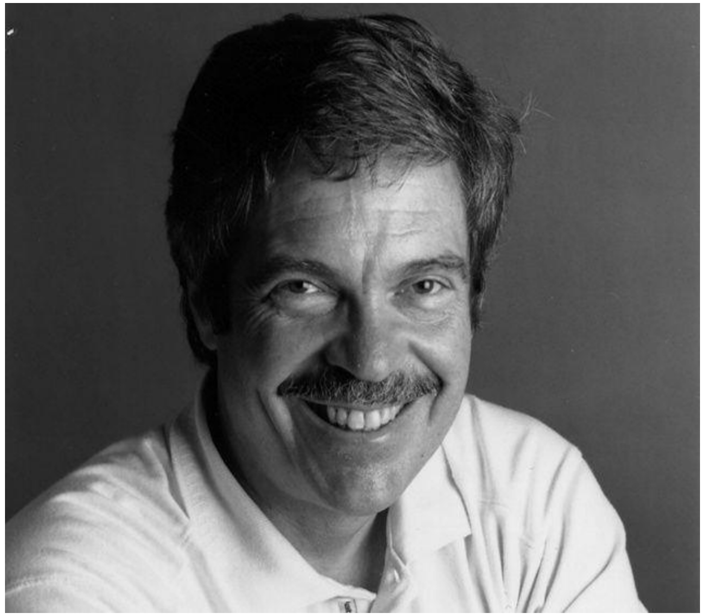


According to Kay, the essential ingredients of OOP are:

1. **Message passing**
2. **Encapsulation**
3. **Dynamic binding**

<u>Conspicuously missing from this list?</u>
Inheritance, sub-class polymorphism

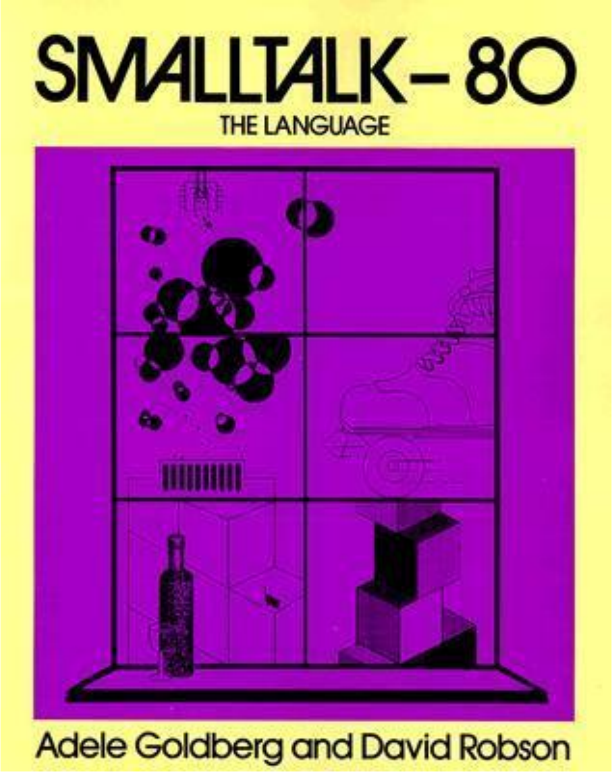
Alan Kay



“Java is the most distressing thing to happen to computing since MS-DOS.”

“I made up the term ‘object-oriented’, and I can tell you I didn’t have C++ in mind.”

Smalltalk



History:

- "Smalltalk" typically refers to Smalltalk-80
- However, first version was Smalltalk-71
- Created in a few mornings of work by Kay on a bet that it could be implemented in a "page of code".
- Smalltalk-72 was more full-featured, used for research at Xerox PARC
- Smalltalk-76 saw performance-enhancing revisions
- Smalltalk-80 V1 was given to select companies for peer review
- Smalltalk-80 V2 was released to the public in 1983.

Overview

Smalltalk is the prototypical class-based, object-oriented language.

There are no primitives: No `int x`, `double y`, etc.

Control structures are methods:

- No `if`/`else`/`while`/`for` syntax constructs.
- Control flow implemented via <u>blocks</u> and <u>message passing</u>.

- Its syntax is very minimal – famously fits on a postcard
- Objects (and message passing!) are central – Unlike Java and C++, there are no primitives. Everything is an object.
- ***Pure*** object-oriented.

Pure Object-Oriented

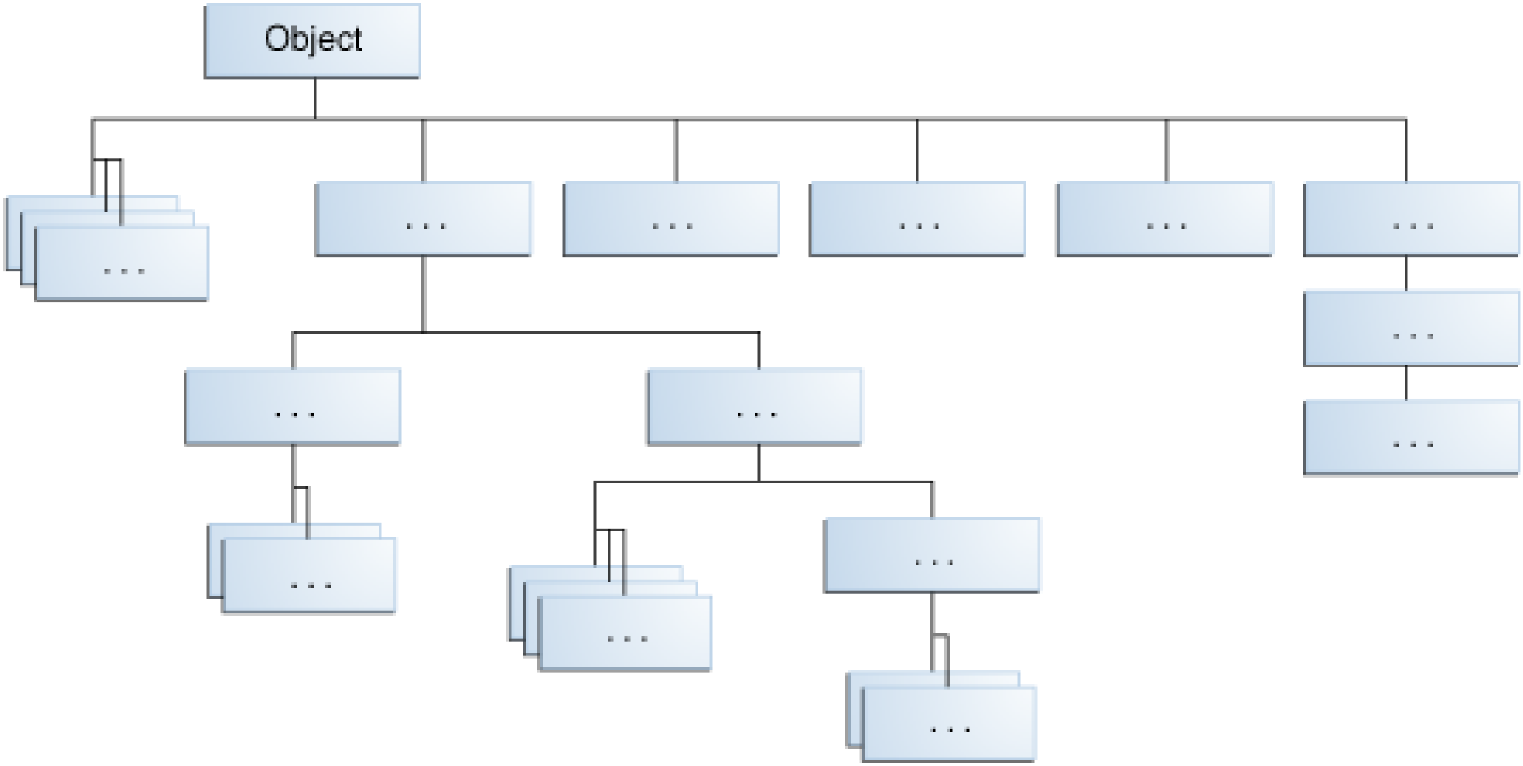
- Everything is an object. Everything is an instance of a corresponding class. Recall cookie/cookie cutter analogy.
- Class-based. Every object has a class that defines the structure of that object
- *Classes* (the cookie cutter!) themselves are also *objects*.
    - Each class is an instance of the *metaclass* of that object.
    - Each metaclass is an instance of a class called ***Metaclass***

Your brain right now:

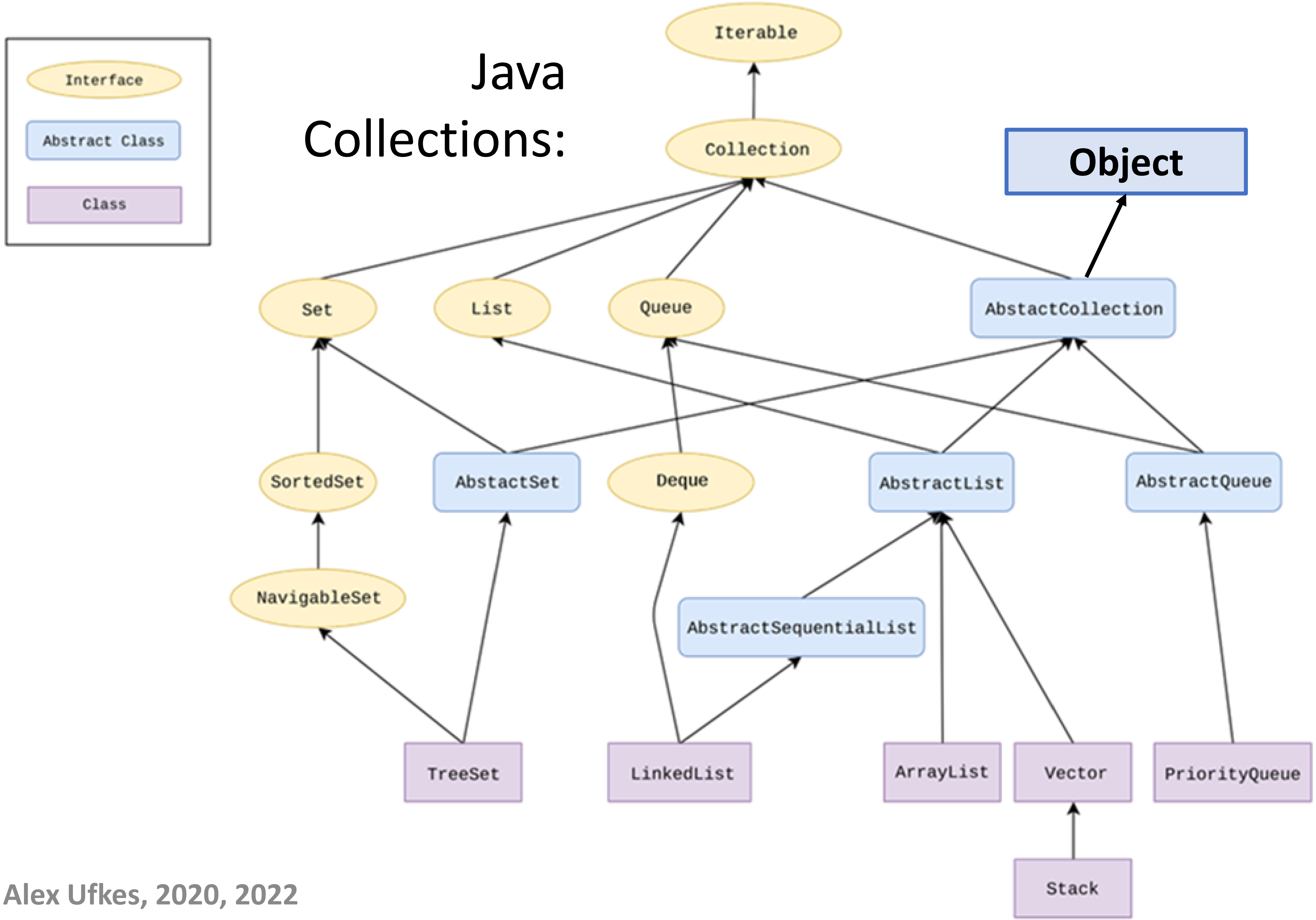
SMALLTALK-80
THE LANGUAGE

Class Hierarchy

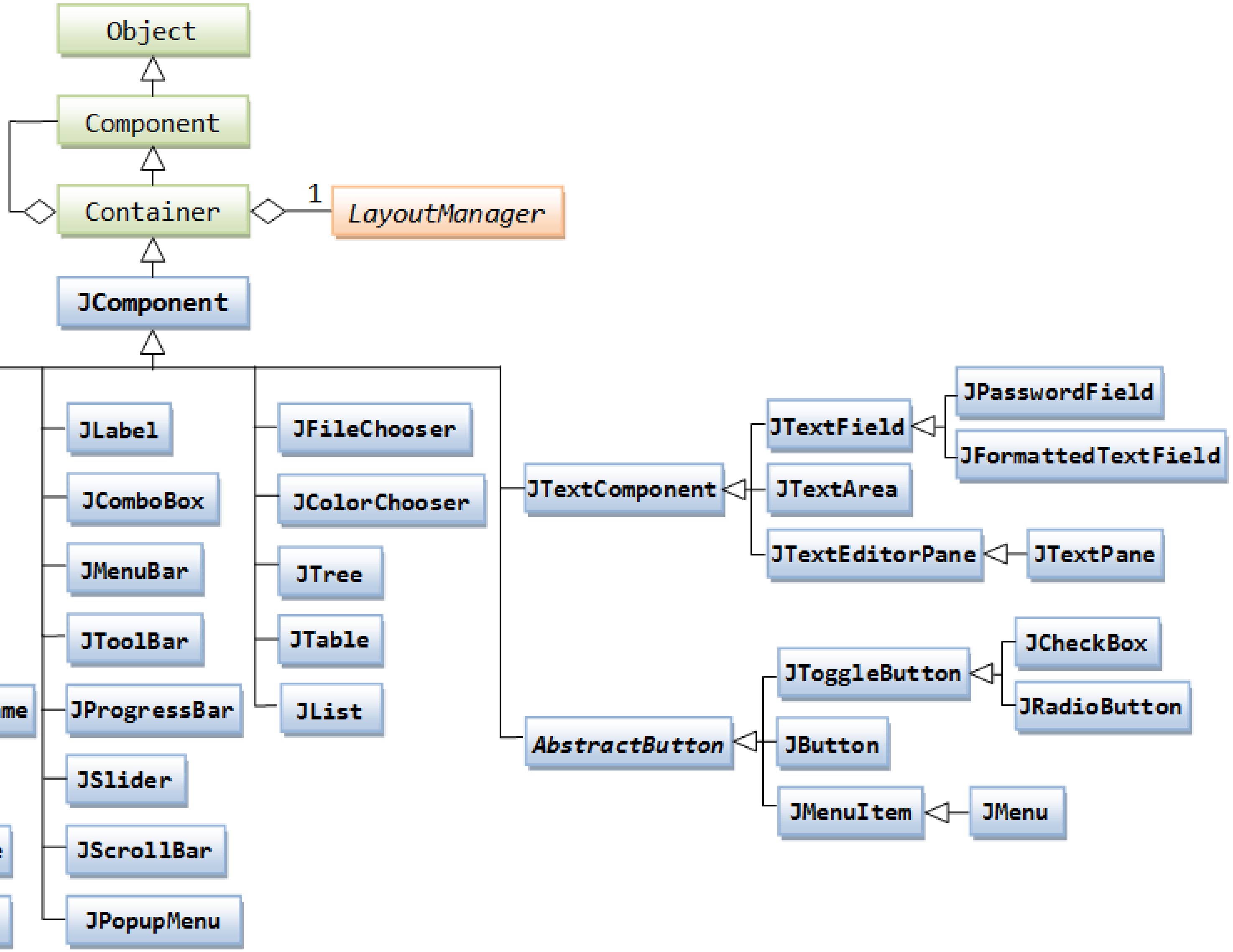
You’ve seen Java’s:



Java Collections:

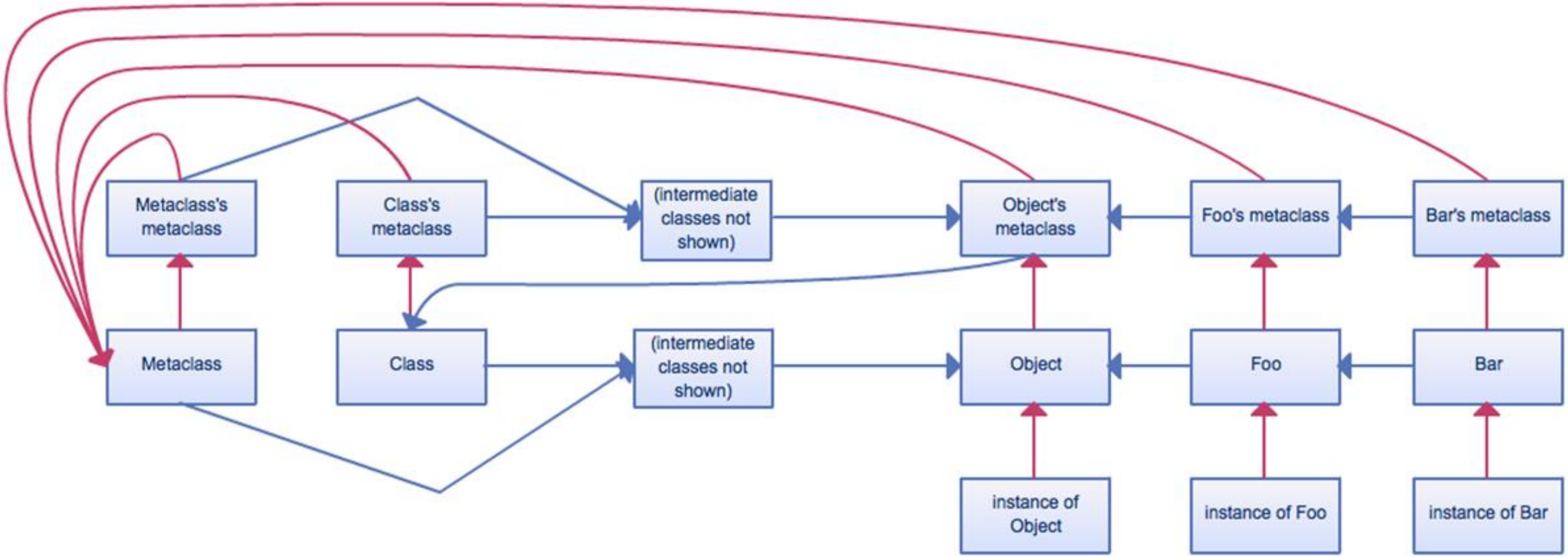


Java Swing Components:

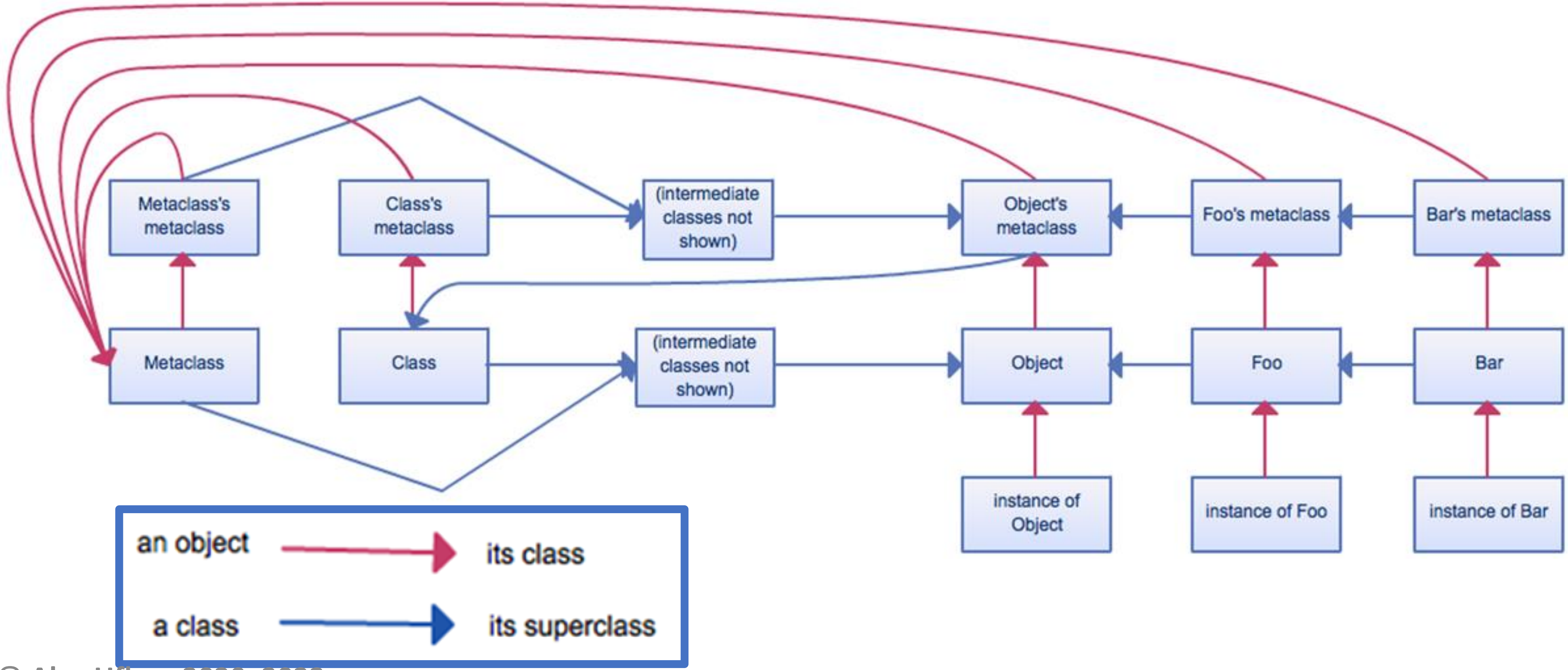


Class Hierarchy

In Smalltalk?



- Classes (the cookie cutter!) themselves are also objects.
    - Each class is instance of the *metaclass* of that object.
    - Each metaclass is an instance of a class called ***Metaclass***



Objects in Smalltalk

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

<u>A Smalltalk object can do exactly three things:</u>

1. Hold state (assignment)
2. Receive a *<u>message</u>* (from itself or another object)
3. Send *<u>message</u>* (to itself or another object)

Message passing is **central** in Smalltalk. Understand message passing, understand Smalltalk.

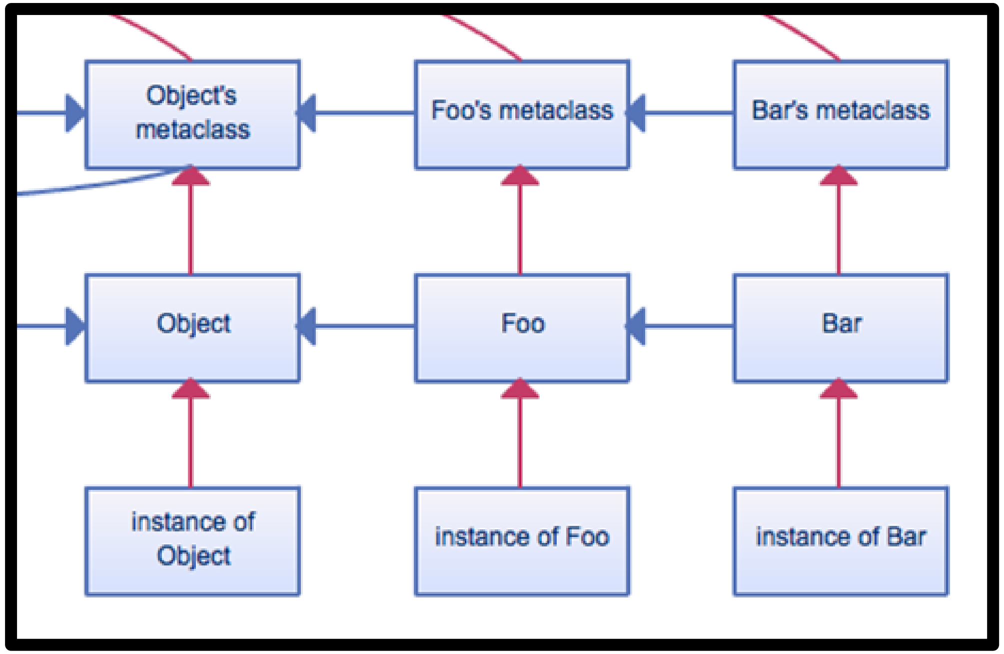
SMALLTALK-80
THE LANGUAGE

Message Passing

Passing a message to an object is semantically equivalent to invoking one of its methods:

<u>When an object receives a message:</u>

- Search the object's class for an appropriate method to deal with the message.
- Not found? check superclass (inheritance!)
- Repeat until method is found, or we hit class "Object". Much like Java.
- Still not found? Throw exception.



Message Passing

Message passing drives all computation in Smalltalk.

For every snippet of Smalltalk code we see, look at it in terms of message passing.

What messages are being sent? What objects are they being sent to?

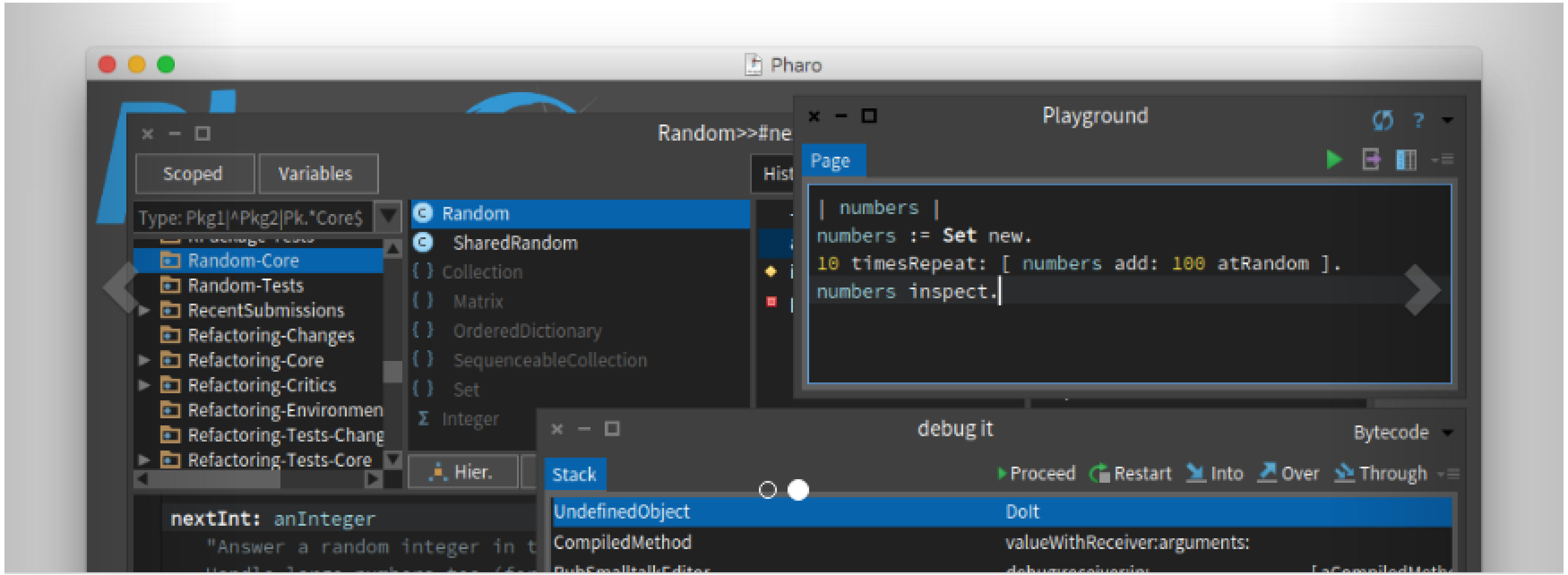
Understand message passing, understand Smalltalk.

> *"I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is messaging."*
>
> **- Alan Kay**

https://pharo.org/

The immersive programming experience

Pharo is a pure object-oriented programming language *and* a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one).



- Pharo is a GUI-based programming environment for the Smalltalk language.
- Smalltalk is based on a virtual machine, similar to Java, which interprets bytecode and makes it platform independent.
- One of the unique features of Smalltalk is that all development and changes are done in the Smalltalk environment itself.
- All classes (including their code) and objects (including their state) are stored inside an image that encapsulates the complete state of the system.
- When you save the image, close the VM, and then re-open it again, perhaps on another machine, everything will be exactly as you left it.

List of implementations [edit]

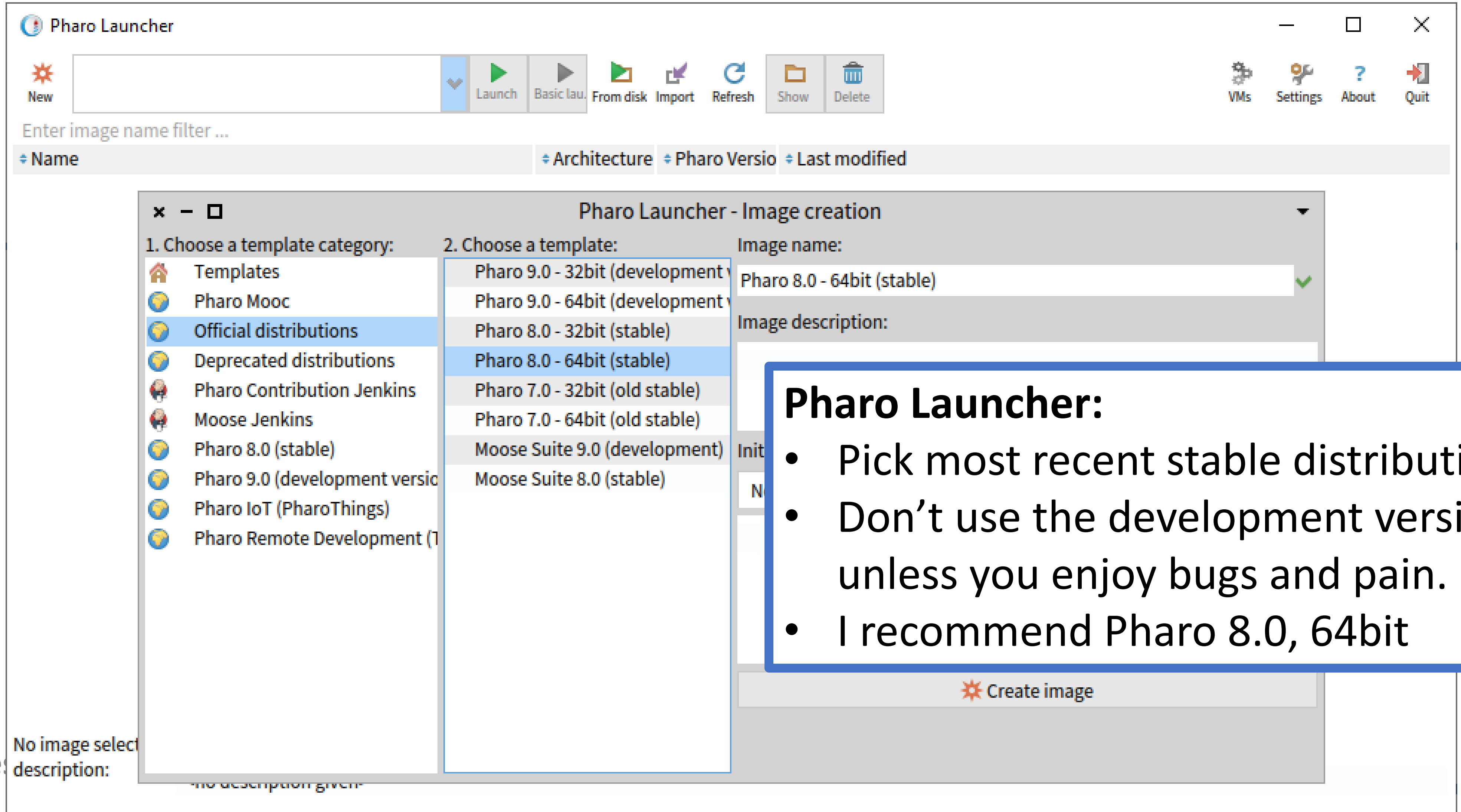
- Amber Smalltalk Smalltalk running atop JavaScript
- Athena, Smalltalk scripting engine for Java ≥ 1.6
- Bistro
- Cincom has the following Smalltalk products: ObjectStudio, VisualWorks and WebVelocity.
- Visual Smalltalk Enterprise, and family, including Smalltalk/V
- Cuis Smalltalk, open source, modern Smalltalk-80 [3]
  - Cog, JIT VM written in Squeak Smalltalk
- F-Script
- GemTalk Systems, GemStone/s
- GNU Smalltalk
  - Étoilé Pragmatic Smalltalk, Smalltalk for Étoilé, a GNUstep-based user environment
  - StepTalk, GNUstep scripting framework uses Smalltalk language on an Objective-C runtime
- Gravel Smalltalk, a Smalltalk implementation for the JVM
- Instantiations, VA Smalltalk being the follow-on to IBM VisualAge Smalltalk
  - VisualAge Smalltalk
- Little Smalltalk
- Object Arts, Dolphin Smalltalk
- Object Connect, Smalltalk MT Smalltalk for Windows
- Objective-Smalltalk, Smalltalk on Objective-C runtime with extensions for Software Architecture
  - LSW Vision-Smalltalk have partnered with Object Arts
- Panda Smalltalk, open source engine, written in C, has no dependencies except libc
- Pharo Smalltalk, Pharo Project's open-source multi-platform Smalltalk
  - Cog, JIT VM written in Squeak Smalltalk
- Pocket Smalltalk, runs on Palm Pilot
- Redline Smalltalk, runs on the Java virtual machine[33]
- Refactory, produces #Smalltalk
- Smalltalk YX
- Smalltalk/X[34]
- Squeak, open source Smalltalk
  - Cog, JIT VM written in Squeak Smalltalk
    - CogDroid, port of non-JIT variant of Cog VM to Android
  - eToys, eToys visual programming system for learning
  - iSqueak, Squeak interpreter port for iOS devices, iPhone/iPad
  - JSqueak, Squeak interpreter written in Java
    - Potato, Squeak interpreter written in Java, a direct derivative of JSqueak
  - RoarVM, RoarVM is a multi- and manycore interpreter for Squeak and Pharo
- Strongtalk, for Windows, offers optional strong typing
- Vista Smalltalk

There are many different Smalltalk implementations.

Each may have subtle differences in their syntax and major differences in their class organization.

When/if Googling for help, it's useful to specify the specific implementation (Pharo for this course).

Pharo: Smalltalk IDE



Pharo Launcher:

- Pick most recent stable distribution
- Don’t use the development version, unless you enjoy bugs and pain.
- I recommend Pharo 8.0, 64bit

Nifty Pharo Reference:

http://files.pharo.org/media/pharoCheatSheet.pdf

Nifty Squeak Reference:

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

- Squeak is a different Smalltalk implementation.
- Most of the syntax is the same, and this terse guide is very conveniently laid out as a reference to use while coding.
- (Pharo is a commercial derivative of Squeak)

```
Transcript clear.                                 "clear to transcript window"
Transcript show: 'Hello World'.                   "output string in transcript window"
Transcript nextPutAll: 'Hello World'.             "output string in transcript window"
Transcript nextPut: $A.                           "output character in transcript window
Transcript space.                                 "output space character in transcript
Transcript tab.                                   "output tab character in transcript wi
Transcript cr.                                    "carriage return / linefeed"
'Hello' printOn: Transcript.                      "append print string into the window"
'Hello' storeOn: Transcript.                      "append store string into the window"
Transcript endEntry.                              "flush the output buffer"
```

Assignment

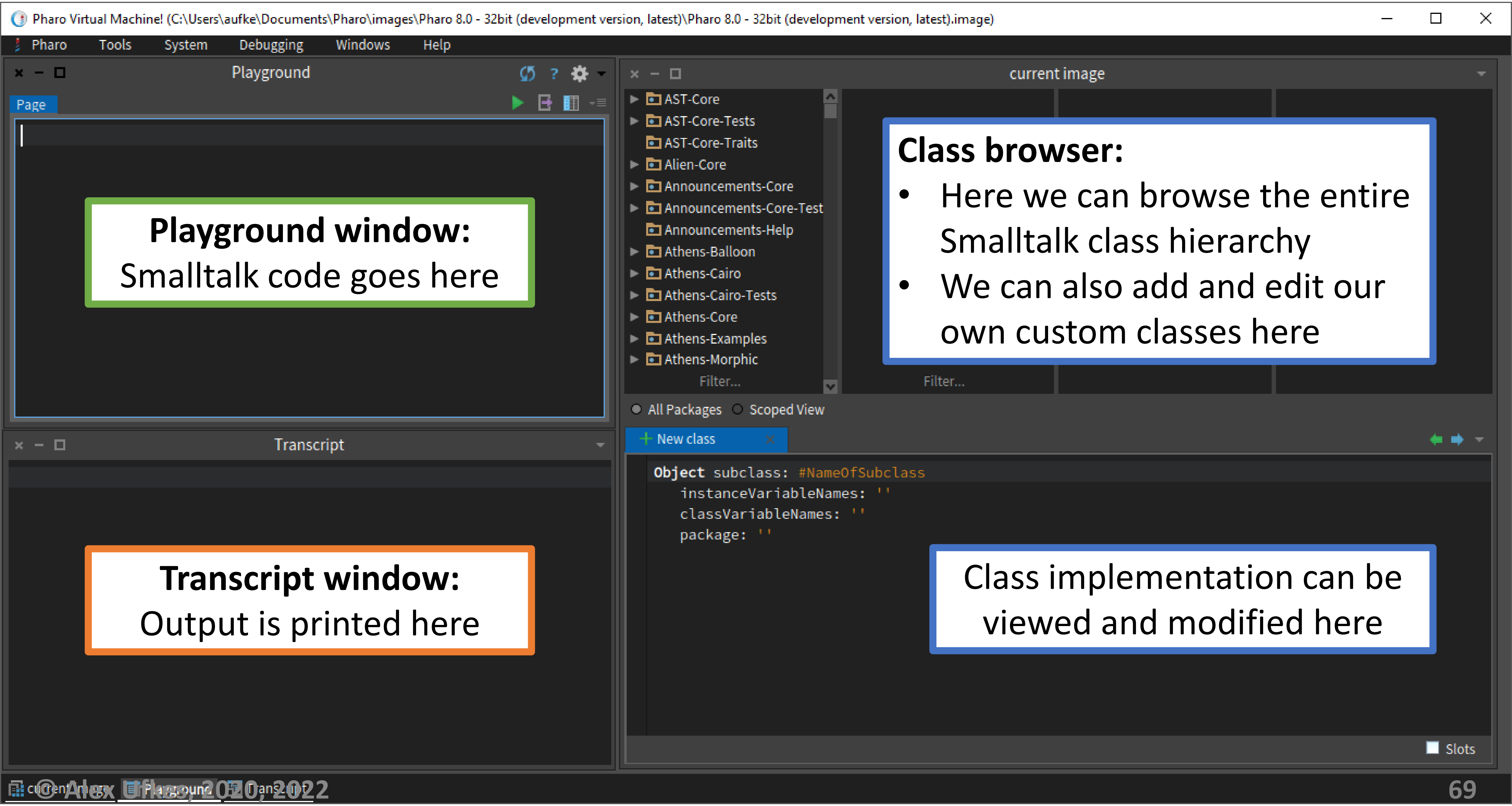
```
| x y |
x _ 4.                                            "assignment (Squeak) <-"
x := 5.                                           "assignment"
x := y := z := 6.                                 "compound assignment"
x := (y := 6) + 1.
x := Object new.                                  "bind to allocated instance of a class
x := 123 class.                                   "discover the object class"
x := Integer superclass.                          "discover the superclass of a class"
x := Object allInstances.                         "get an array of all instances of a cl
x := Integer allSuperclasses.                     "get all superclasses of a class"
x := 1.2 hash.                                    "hash value for object"
y := x copy.                                      "copy object"
y := x shallowCopy.                               "copy object (not overridden)"
y := x deepCopy.                                  "copy object and instance vars"
y := x veryDeepCopy.                              "complete tree copy using a dictionary
```

Constants

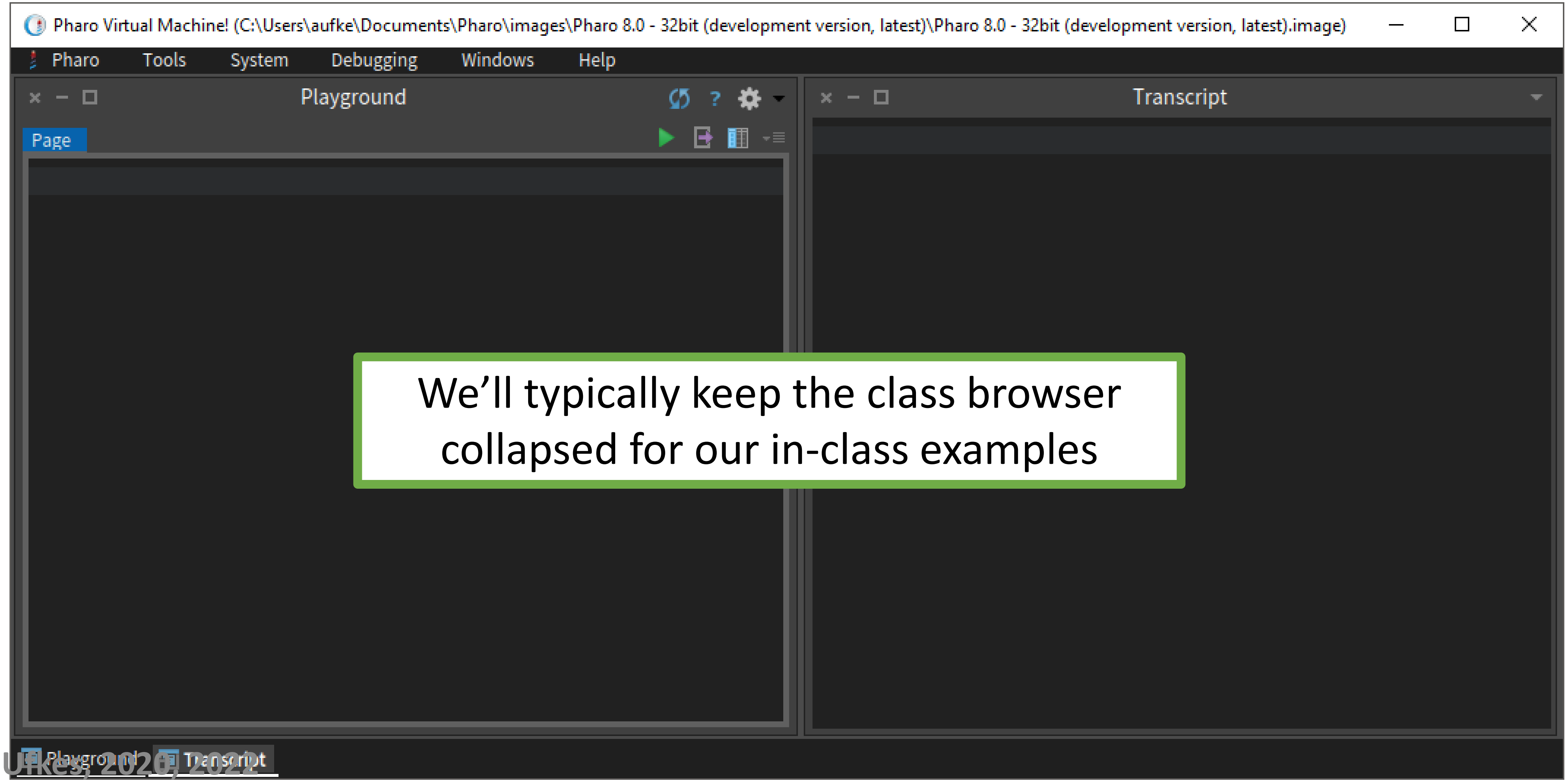
```
| b x |
b := true.                                        "true constant"
b := false.                                       "false constant"
x := nil.                                         "nil object constant"
x := 1.                                           "integer constants"
x := 3.14.                                        "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"
```

Booleans

```
| b x y |
x := 1. y := 2.
b := (x = y).                                     "equals"
```

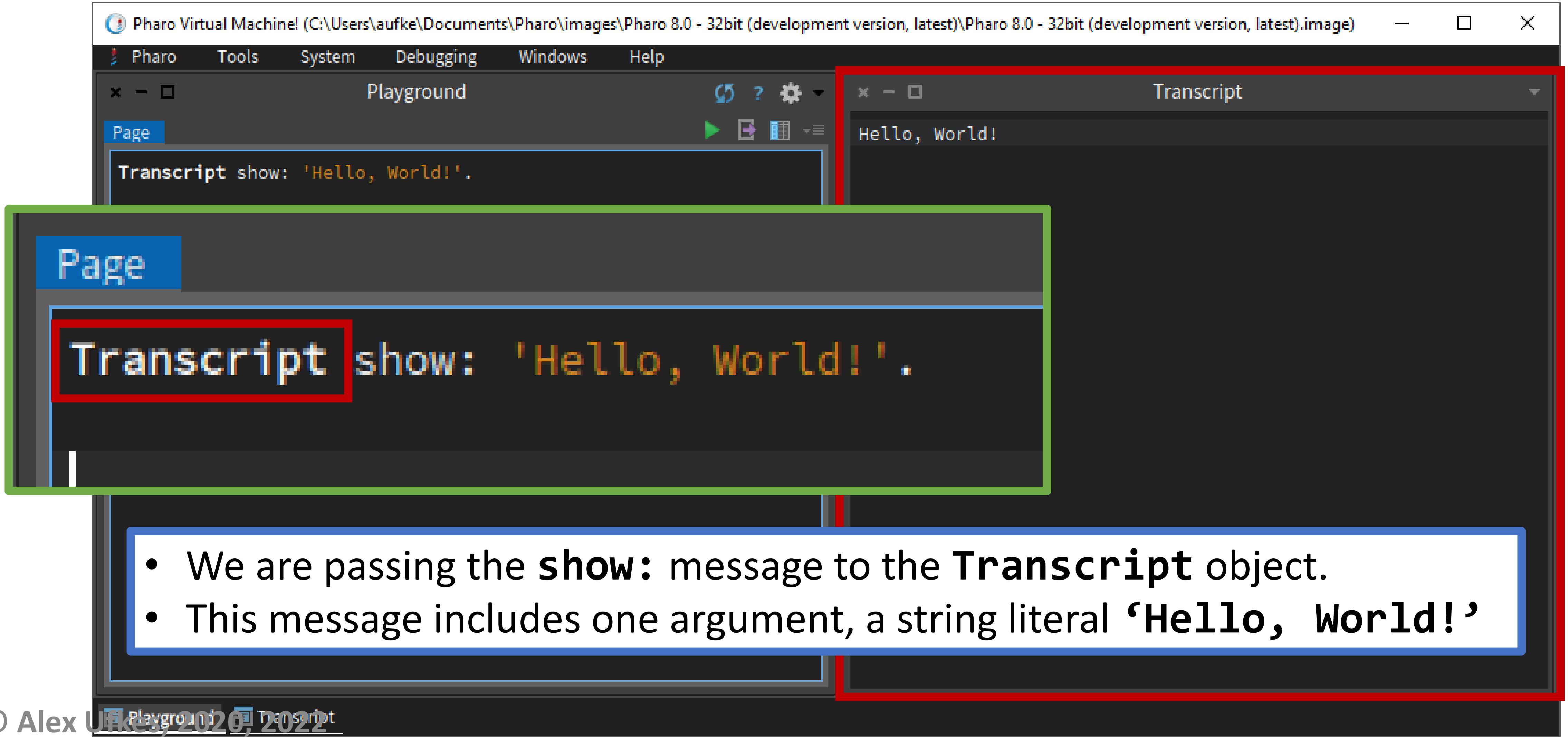

Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo 8.0 - 32bit (development version, latest).image)
Pharo Tools System Debugging Windows Help
Playground
Page
Playground window:
Smalltalk code goes here
Transcript
Transcript window:
Output is printed here
current image
AST-Core
AST-Core-Tests
AST-Core-Traits
Alien-Core
Announcements-Core
Announcements-Core-Test
Announcements-Help
Athens-Balloon
Athens-Cairo
Athens-Cairo-Tests
Athens-Core
Athens-Examples
Athens-Morphic
Filter...
Filter...
Class browser:
• Here we can browse the entire Smalltalk class hierarchy
• We can also add and edit our own custom classes here
All Packages
Scoped View
New class
Object subclass: #NameOfSubclass
instanceVariableNames: ''
classVariableNames: ''
package: ''
Class implementation can be viewed and modified here
Slots
current image
Playground
Transcript

Pharo: Smalltalk IDE



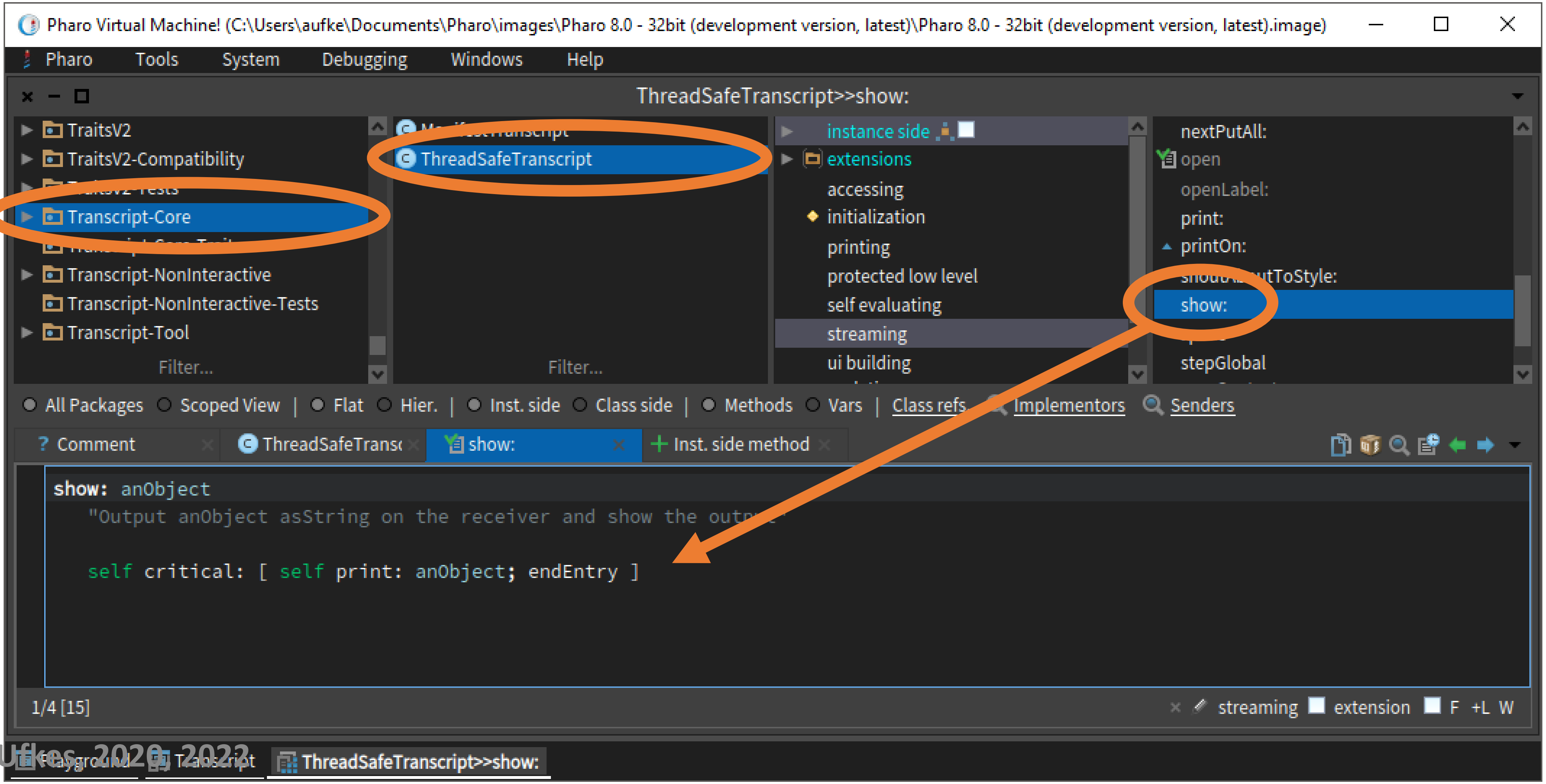
We'll typically keep the class browser collapsed for our in-class examples

Hello, World!



- We are passing the **`show:`** message to the **`Transcript`** object.
- This message includes one argument, a string literal **`'Hello, World!'`**

Transcript show:




Send message

Messages: Unary

Think of every Smalltalk statement in terms of message passing:

```
x := 16 sqrt.
```

<u>Only 3 operations:</u>
- Assignment
- Send message
- Receive message

The message **`sqrt`** is sent to the object **16**

In Java, we'd say: **`x = Math.sqrt(16);`**

Messages: Unary

Think of every Smalltalk statement in terms of message passing:

```
x := 16 sqrt.
```

<u>Only 3 operations:</u>
- Assignment
- Send message
- Receive message

The message **`sqrt`** is sent to the object **16**

- **`16`** is an instance of the **`SmallInteger`** class.
- **`SmallInteger`** handles the message (if it knows how)
- Returns the result of the square root (in this case 4)
  - 4 is an object!

Messages: Unary

Think of every Smalltalk statement in terms of message passing:

```
x := 16 sqrt.
```

<u>Only 3 operations:</u>

- Assignment
- Send message
- Receive message

Result assigned to x

- **`16`** is an instance of the **`SmallInteger`** class.
- **`SmallInteger`** handles the message (if it knows how)
- Returns the result of the square root (in this case 4)
- x now references the result – a **`SmallInteger`** object, **`4`**

Messages: Unary

Think of every Smalltalk statement in terms of message passing:

```
x := 16 sqrt.
```

Unary messages are passed without arguments

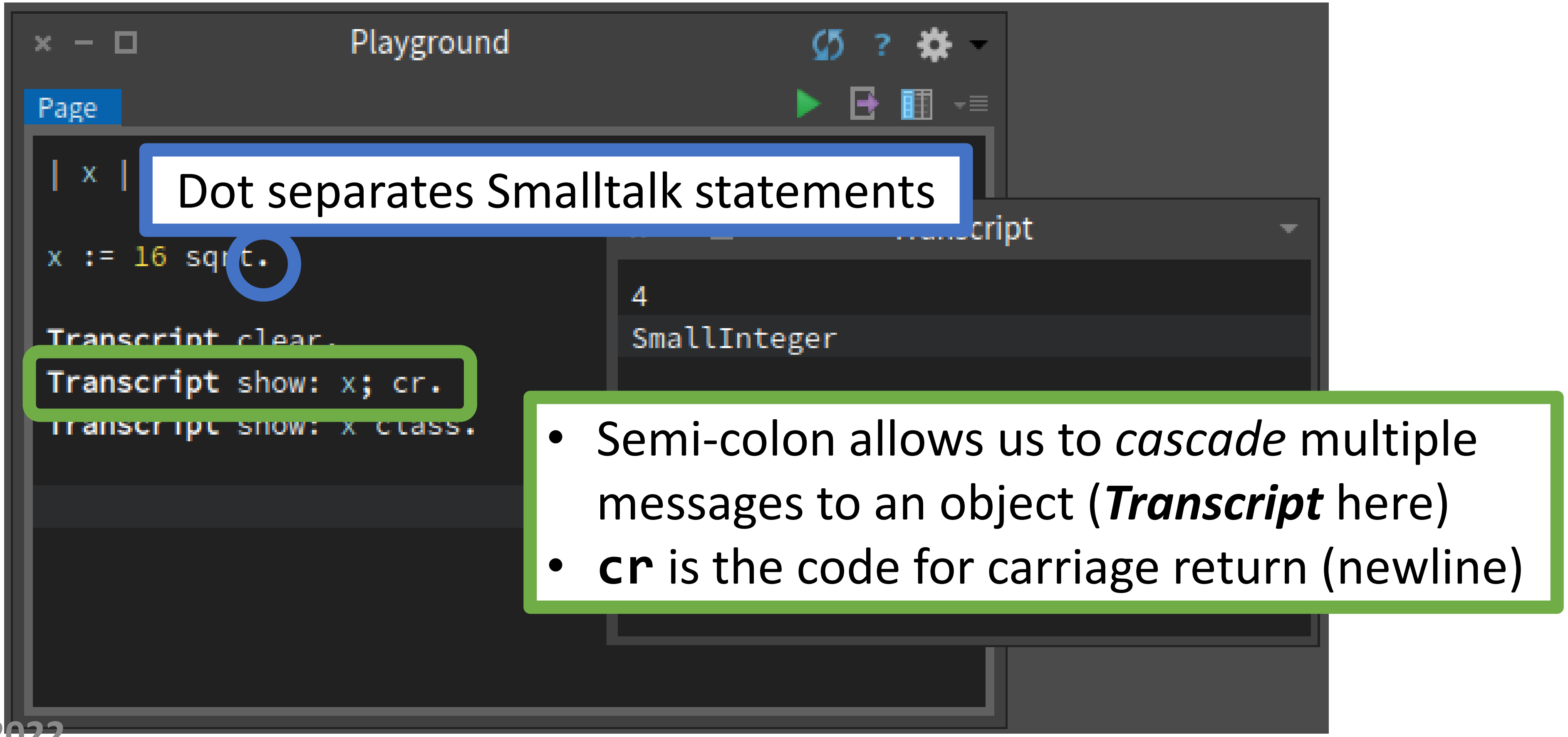
Unary Messages:

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

Any message without argument(s)

Messages in Smalltalk

Think of every Smalltalk statement in terms of message passing:



Messages: Binary

Three kinds:
1. Unary
2. **Binary**
3. Keyword

```
x := 3 + 4
```

The message **+** is passed to object **3** with the argument **4**

Binary messages are strictly between **two** objects.
Symbolic operators are binary messages.

Binary Messages:

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

Arithmetic, comparison, etc.

Messages: Keyword

Three kinds:
1. Unary
2. Binary
3. **Keyword**

```
x := 2 raisedTo: 4.
```

- **2** is the receiving object
- **`raisedTo:`** is the message
- **4** is the argument
- This is called a "*keyword*" message

Keyword messages can contain **any number of arguments**.

Keyword messages include a colon. Quick and easy way to differentiate.

Multiple Arguments

```
x := ‘Hello’ indexOf: $o startingAt: 2.
```

- The actual message is **`indexOf:startingAt:`**
- Smalltalk interleaves arguments.
- Meant to improve readability.

Multiple Arguments: Interleaving

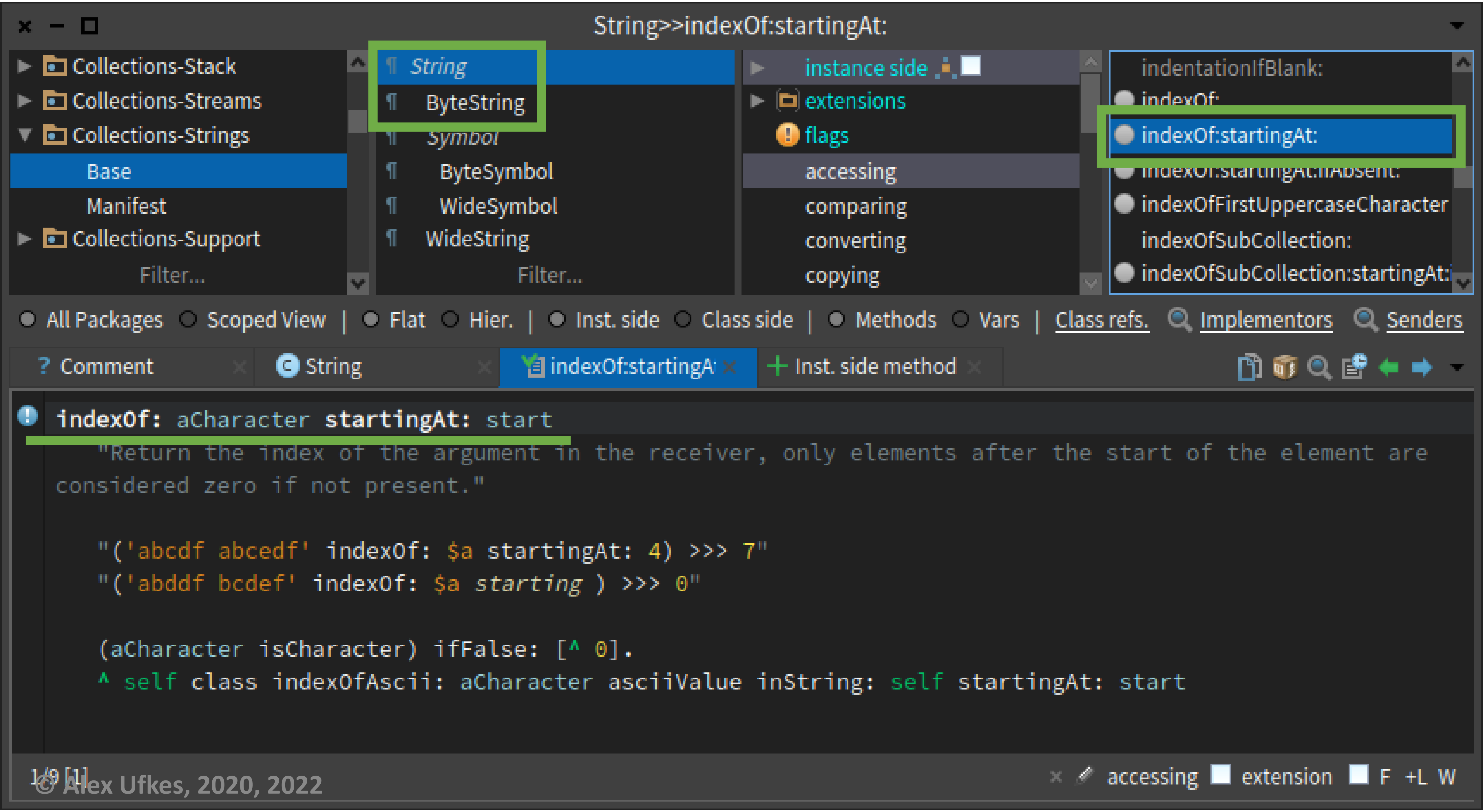
Don’t be confused!

```
x := ‘Hello’ indexOf: $o startingAt: 2.
```

Semantically identical Java syntax is as follows:

```
x = “Hello”.indexOf(‘o’, 2);
```

Argument interleaving has other implications that we’ll explore later.


String>>indexOf:startingAt:
Collections-Stack
Collections-Streams
Collections-Strings
Base
Manifest
Collections-Support
Filter...
String
ByteString
Symbol
ByteSymbol
WideSymbol
WideString
Filter...
instance side
extensions
flags
accessing
comparing
converting
copying
indentationIfBlank:
indexOf:
indexOf:startingAt:
indexOf:startingAt:ifAbsent:
indexOfFirstUppercaseCharacter
indexOfSubCollection:
indexOfSubCollection:startingAt:
All Packages
Scoped View
Flat
Hier.
Inst. side
Class side
Methods
Vars
Class refs.
Implementors
Senders
Comment
String
indexOf:startingAt:
Inst. side method
indexOf: aCharacter startingAt: start
"Return the index of the argument in the receiver, only elements after the start of the element are considered zero if not present."
"('abcdf abcedf' indexOf: $a startingAt: 4) >>> 7"
"('abddf bcdef' indexOf: $a starting) >>> 0"
(aCharacter isCharacter) ifFalse: [^ 0].
^ self class indexOfAscii: aCharacter asciiValue inString: self startingAt: start
1/9 [1]
accessing
extension
F
+L
W

Message Summary

Unary Messages:

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

Any message without argument(s)

Binary Messages:

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

Arithmetic, comparison, etc.

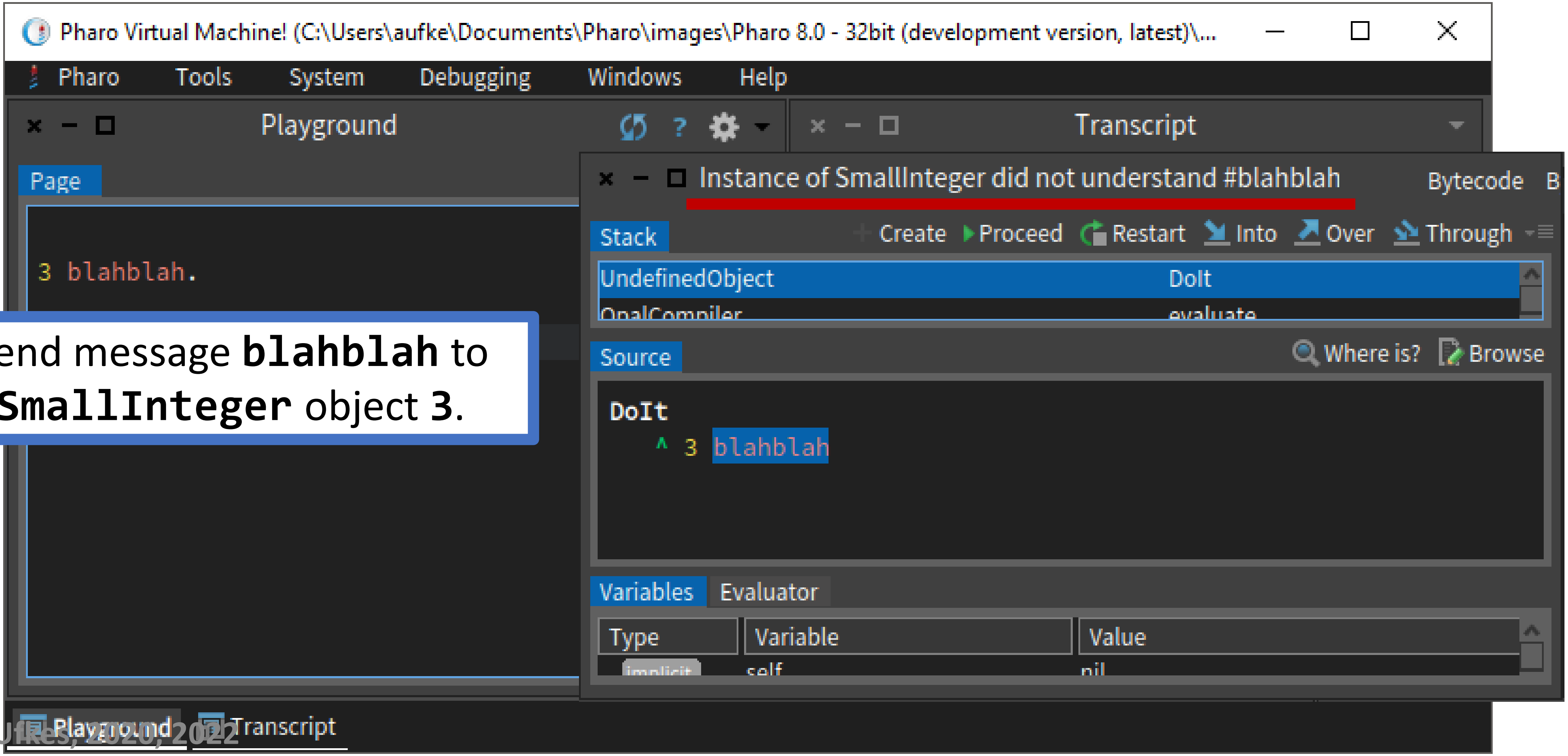
Keyword Messages:

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

Message with one or more arguments, ending in colon:

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

In Smalltalk, you can send ***any*** message to ***any*** object. If the object doesn't know what to do with the message, a run-time error occurs.



Send message **blahblah** to **SmallInteger** object **3**.

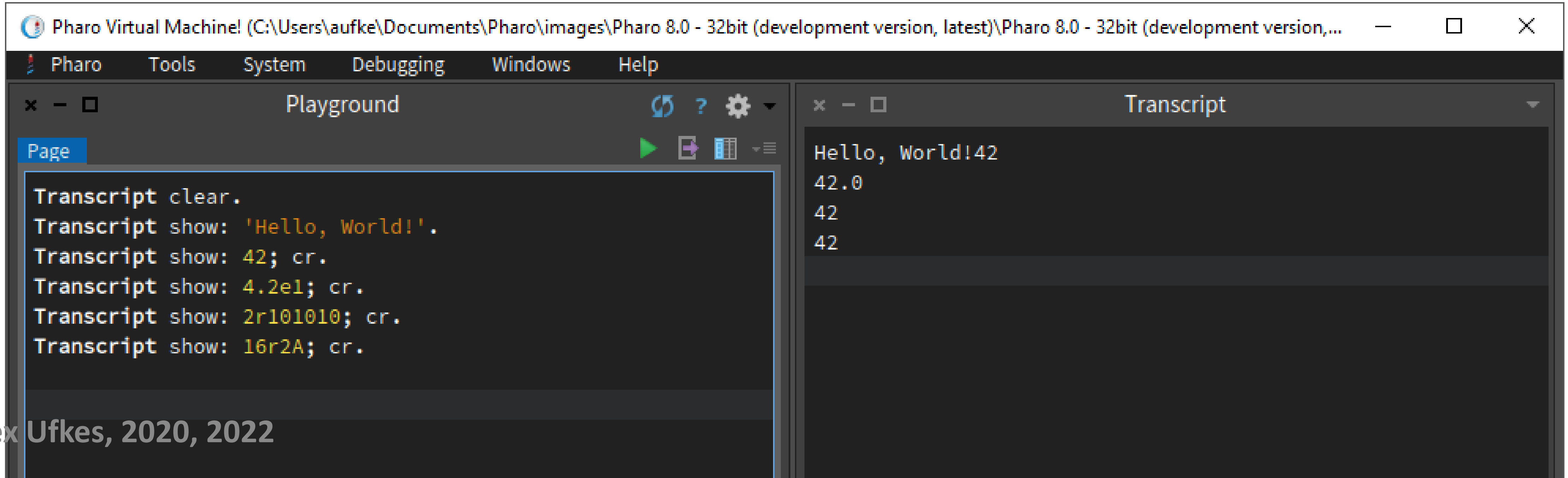
Smalltalk Literals

Numbers: `42, -42, 123.45, 1.2345e2, 2r10010010, 16rA000`

Characters: Denoted by a $ - `$A, $8, $?`

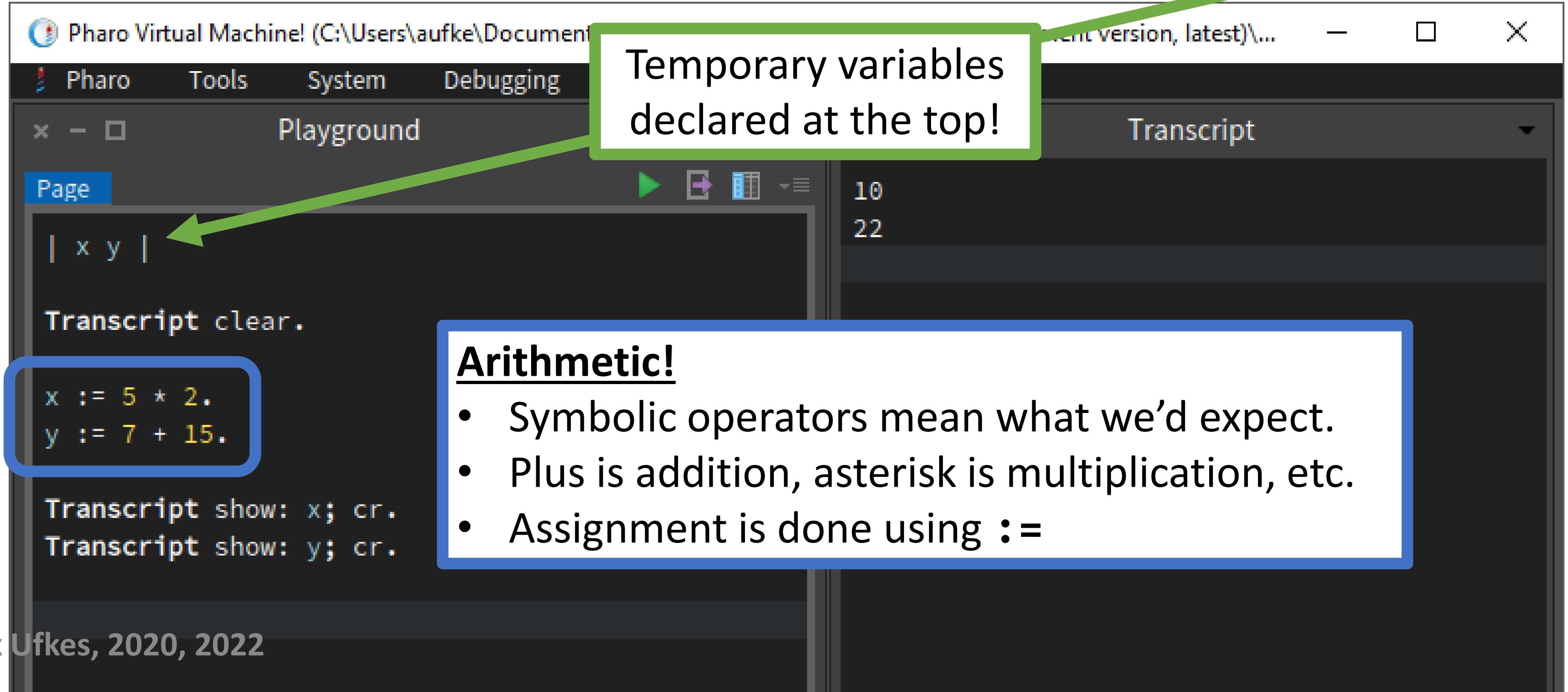
Strings: Denoted with single quotes: **‘Hello, World!’**

Comments: Double quotes - **“This is a Smalltalk comment”**



Smalltalk Variables

- Must be declared before use.
- Variables are references to objects.
- Most common are instance and temporary variables.
- Temporary variables declared inside vertical bars: **| x y |**



Temporary variables declared at the top!

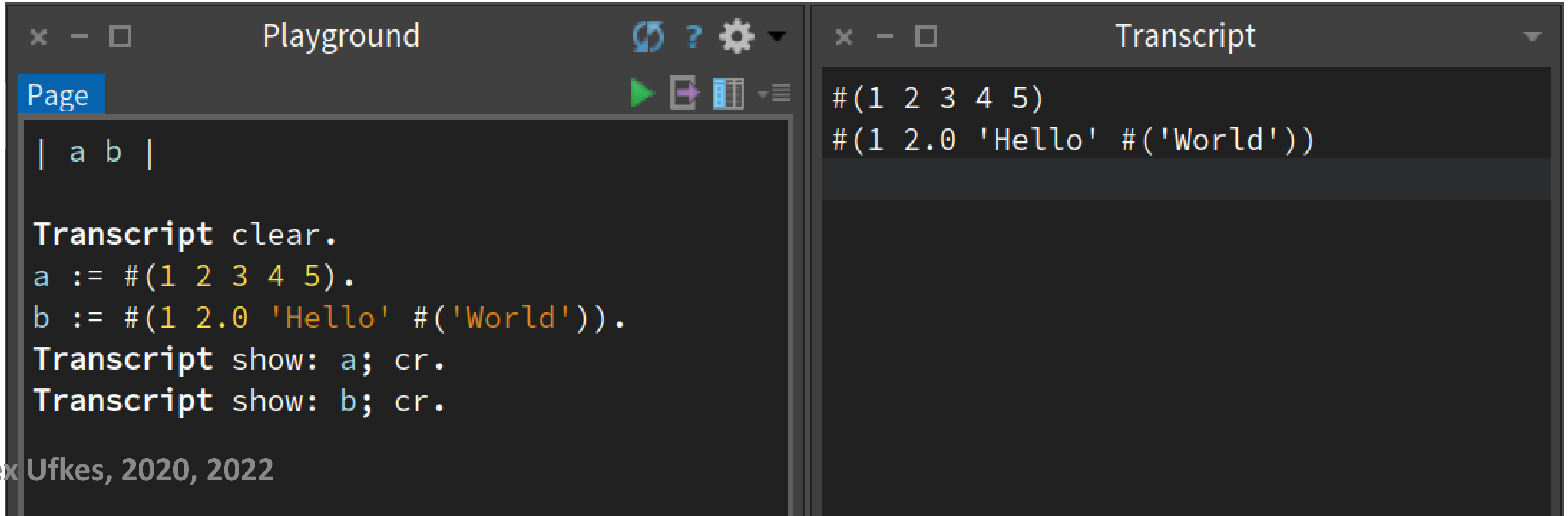
Arithmetic!

- Symbolic operators mean what we'd expect.
- Plus is addition, asterisk is multiplication, etc.
- Assignment is done using **:=**

#(Arrays)

Array of literals (static):

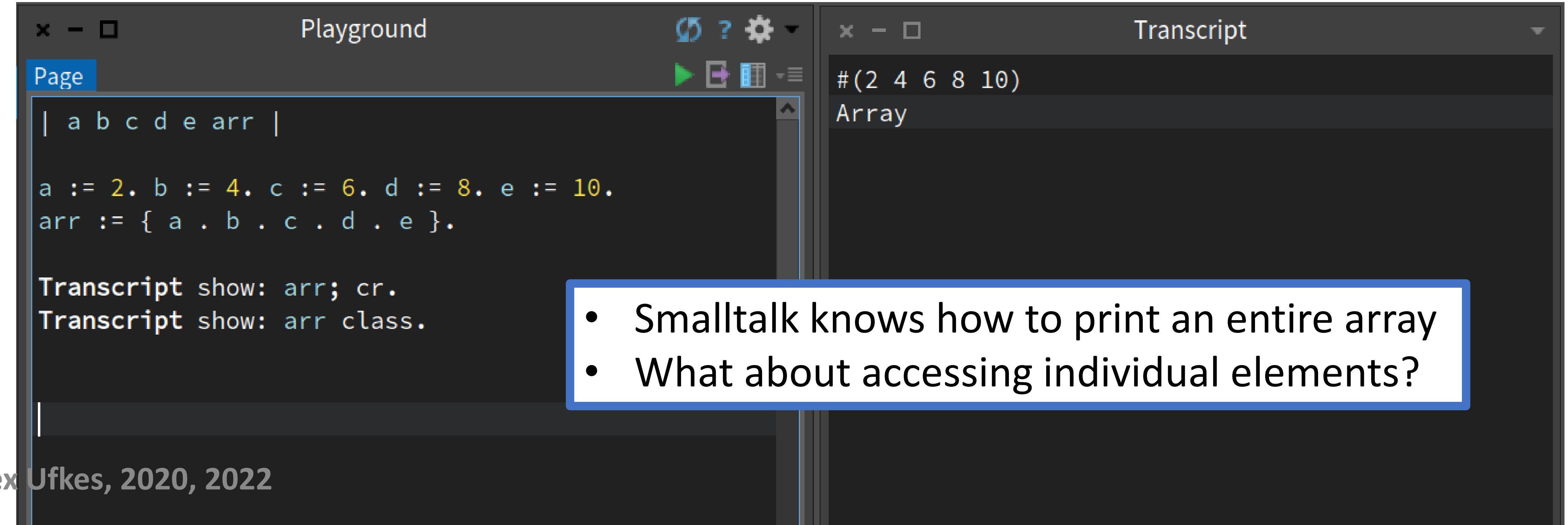
- `#(1 2 3 4 5)`   Array of integers, numbers separated by spaces
- `#(1 2.0 ‘Hello’ #(‘World’))`
- Arrays in Smalltalk can contain any object. Heterogeneous.



#{Arrays}

<u>Array of variables (dynamic):</u>

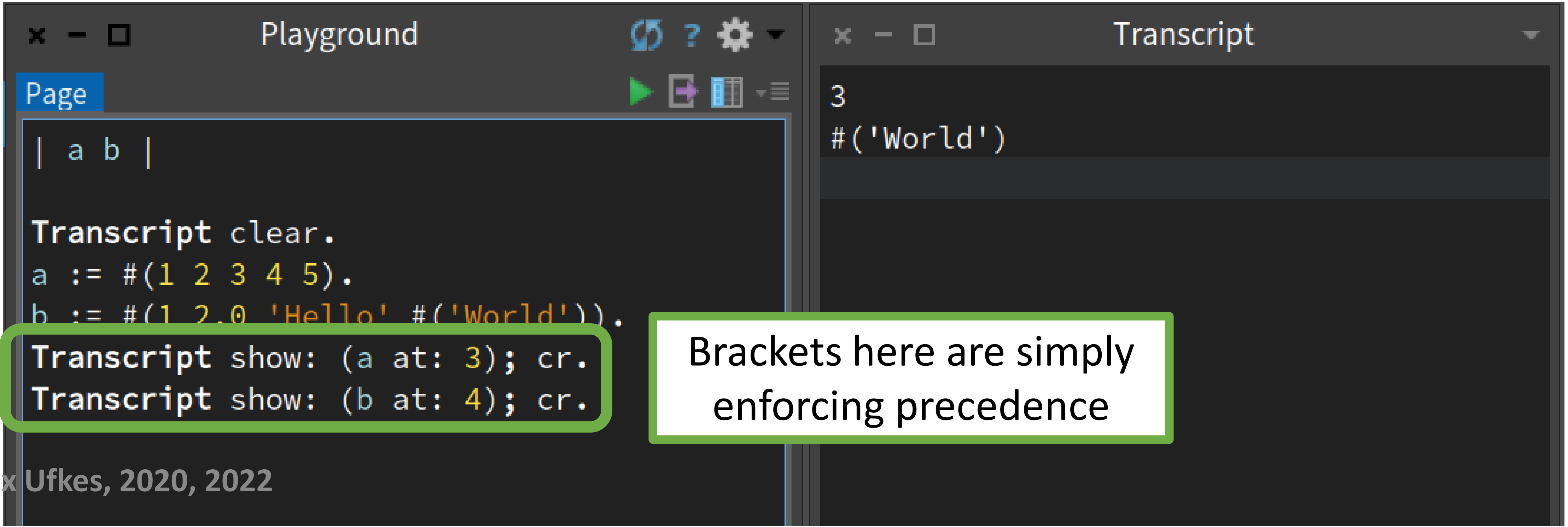
- `#{a . b . c . d . e}`   Array of variables
- Defined with curly braces, periods between elements.



- Smalltalk knows how to print an entire array
- What about accessing individual elements?

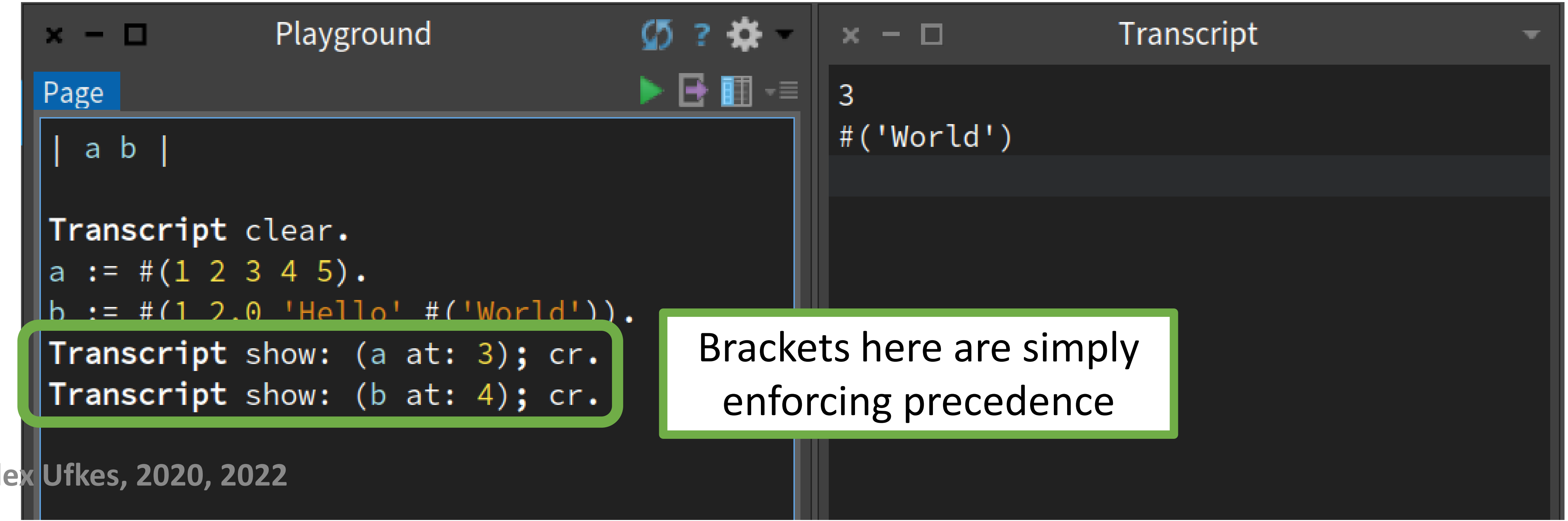
Accessing Array Elements

- Use **at:** message with single argument indicated index
- Based on what is printed, we see that indexing in Smalltalk starts at 1!
- We need parentheses – Otherwise Pharo will read the message as **`show:at:`** instead of **`show:`** and **`at:`** as separate messages



Accessing Array Elements

- We need parentheses – Otherwise Pharo will read the message as **`show:at:`** instead of **`show:`** followed by **`at:`**
- Send **`at:`** message to **`a`** with argument **`3`**, that result becomes the argument of the **`show:`** message, sent to **`Transcript`**.



Brackets here are simply enforcing precedence

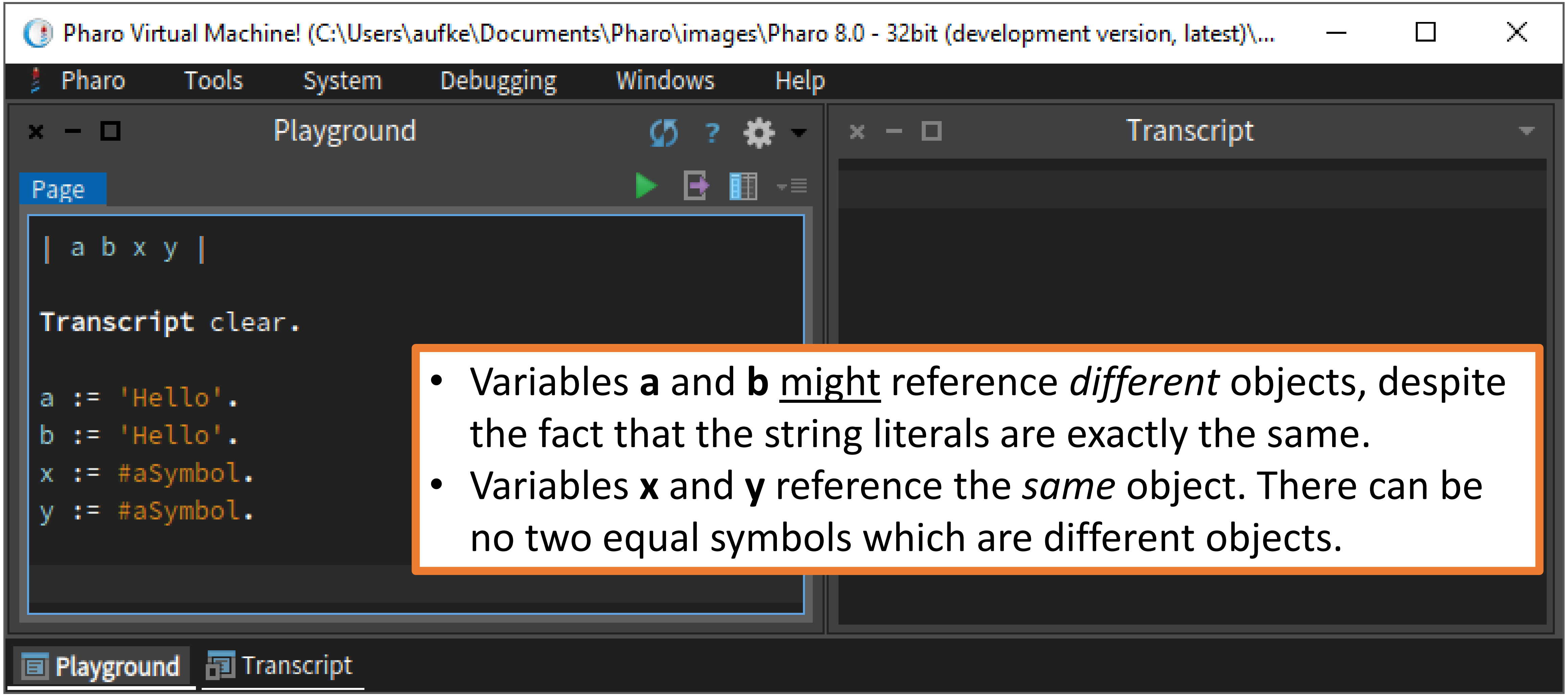
#Symbols

followed by a ***string literal***

- **`#‘aSymbol’`** same as **`#aSymbol`** (quotes implied)
- **`#‘symbol one’ #‘symbol two’`**
- Symbol objects are globally *unique.* Strings are *not.*

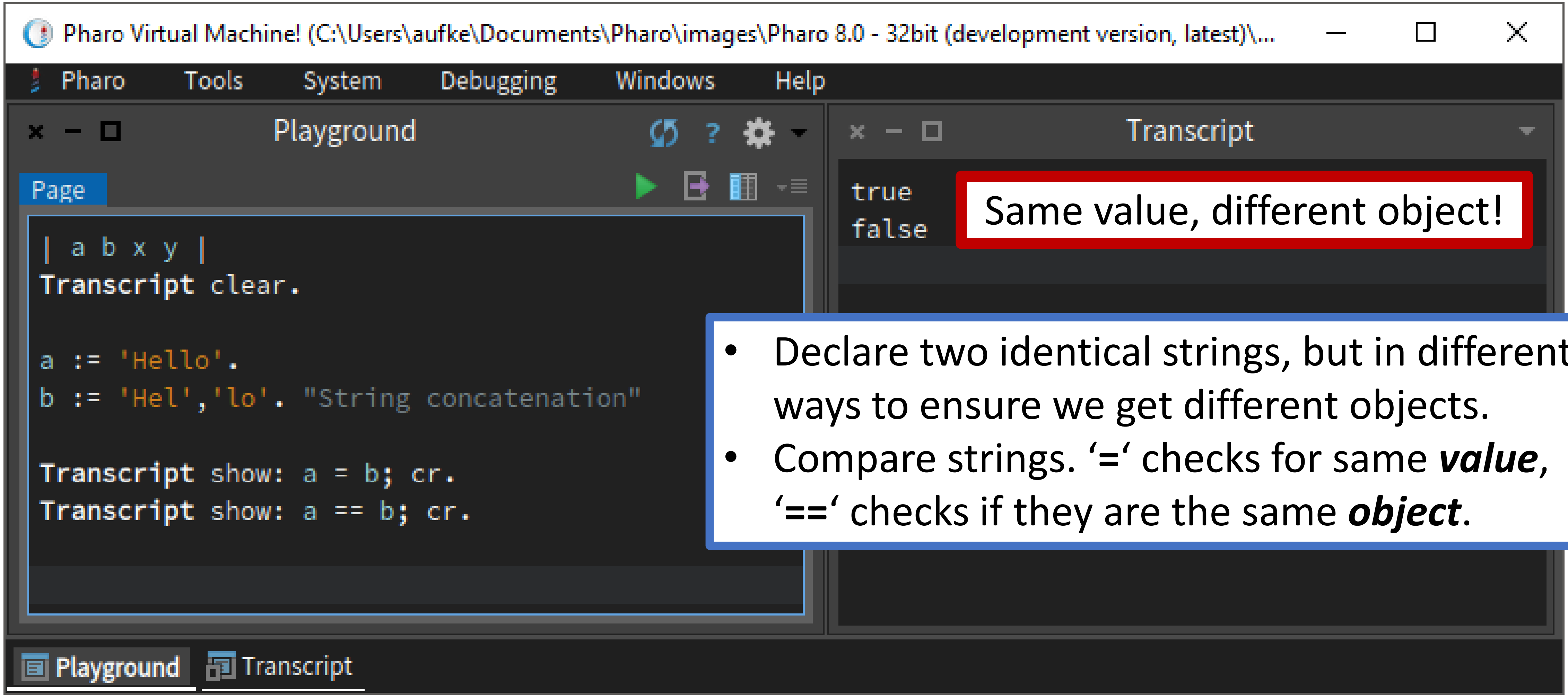
<u>Meaning:</u>

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

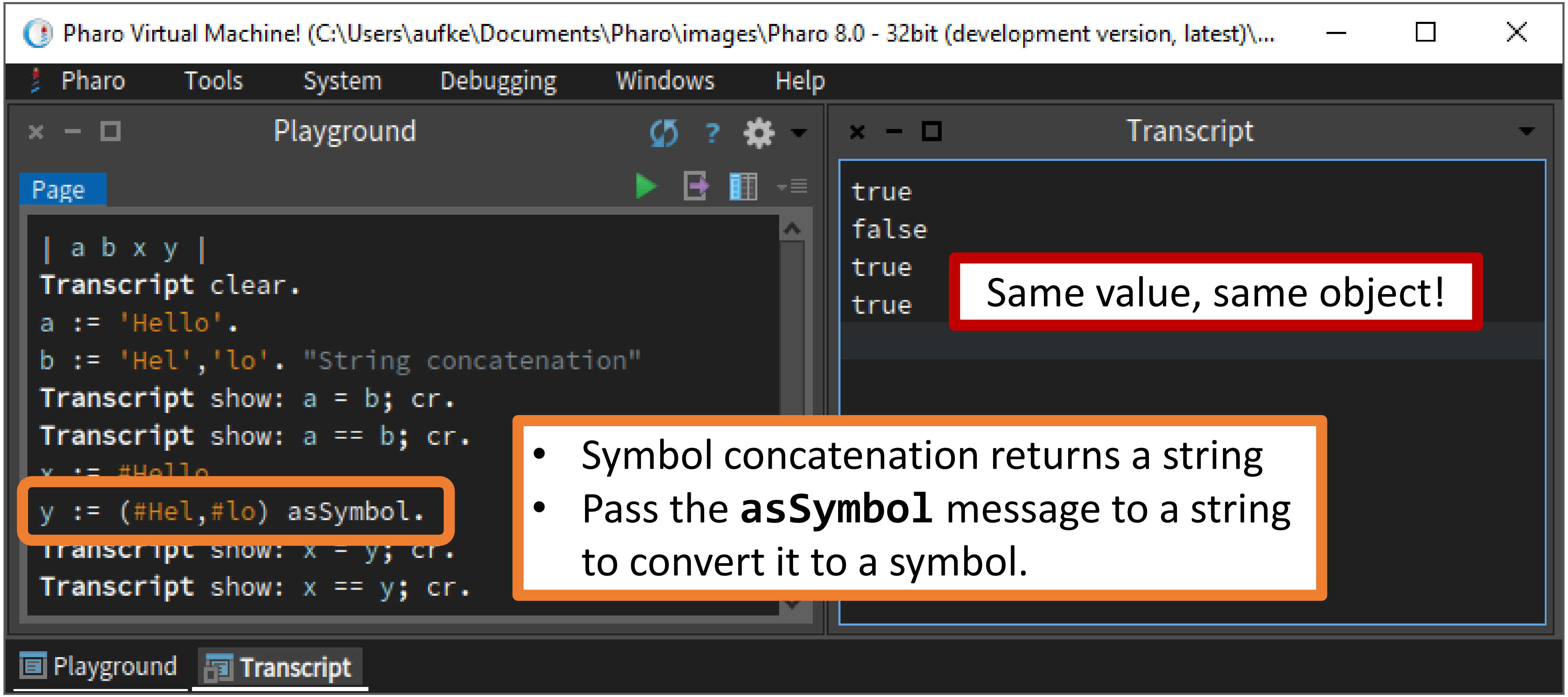


- Variables **a** and **b** <u>might</u> reference *different* objects, despite the fact that the string literals are exactly the same.
- Variables **x** and **y** reference the *same* object. There can be no two equal symbols which are different objects.

Let’s prove it!



- Declare two identical strings, but in different ways to ensure we get different objects.
- Compare strings. ‘=‘ checks for same ***value***, ‘==‘ checks if they are the same ***object***.


Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\...
Pharo
Tools
System
Debugging
Windows
Help
Playground
Page
| a b x y |
Transcript clear.
a := 'Hello'.
b := 'Hel','lo'. "String concatenation"
Transcript show: a = b; cr.
Transcript show: a == b; cr.
y := (#Hel,#lo) asSymbol.
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

Symbols: What's the point?

Checking for equal string value involves comparing individual characters. This can be costly if the strings are long. Linear time operation.

Checking if two variables reference the same object is fast – single integer comparison between addresses.

With symbols, if they reference different objects, they have different values. The same cannot be said of strings.

Symbols: What’s the point?

<u>Messages are symbols!</u>

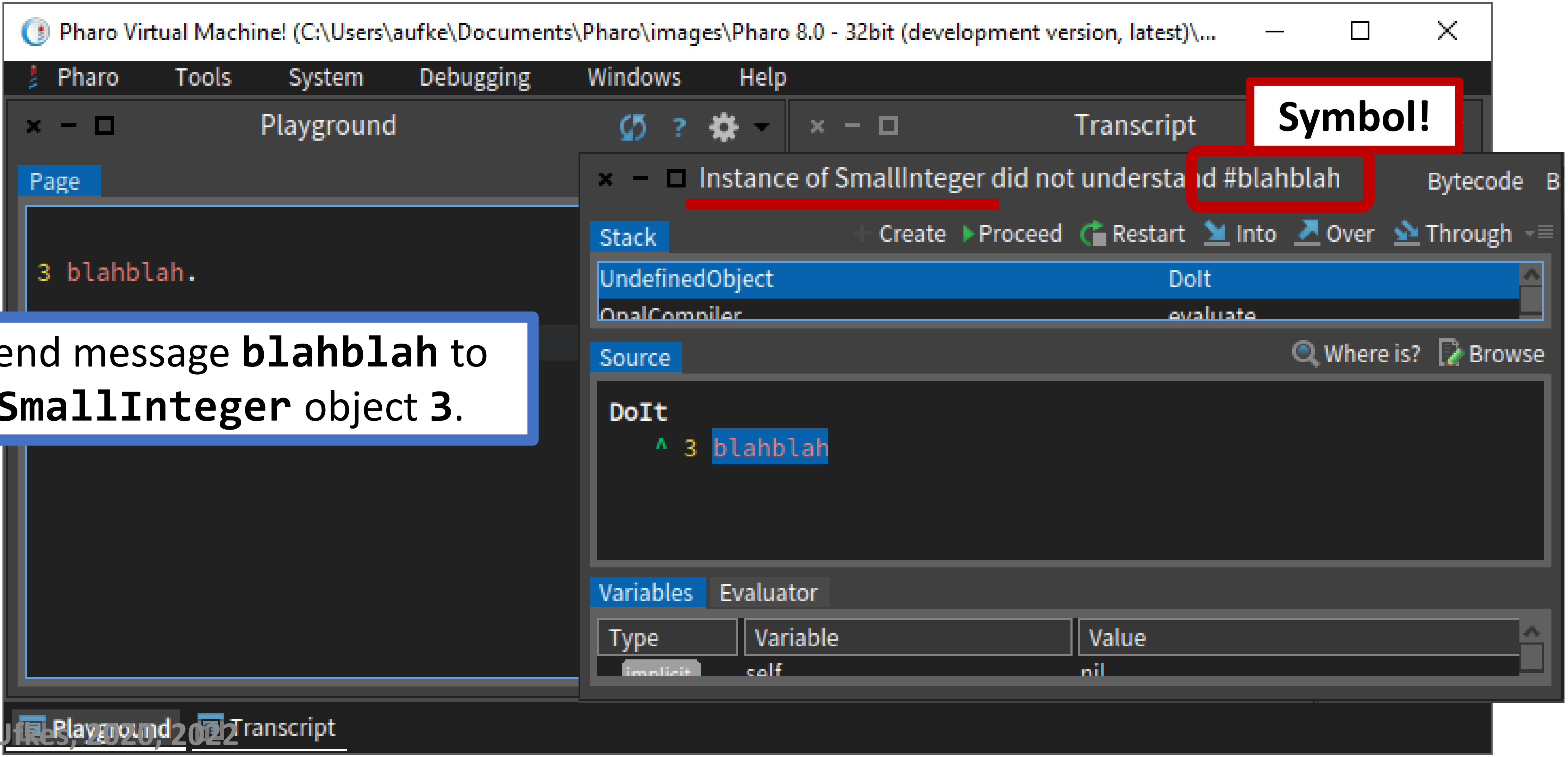
Given that message passing is central in Smalltalk,
we would expect to be doing a lot of it.

<u>When a message is sent to an object:</u>

- *Search the object’s class for an appropriate method*
    - *(Method whose name matches message.)*

Symbols make each check constant time as opposed
to linear time. Very valuable!

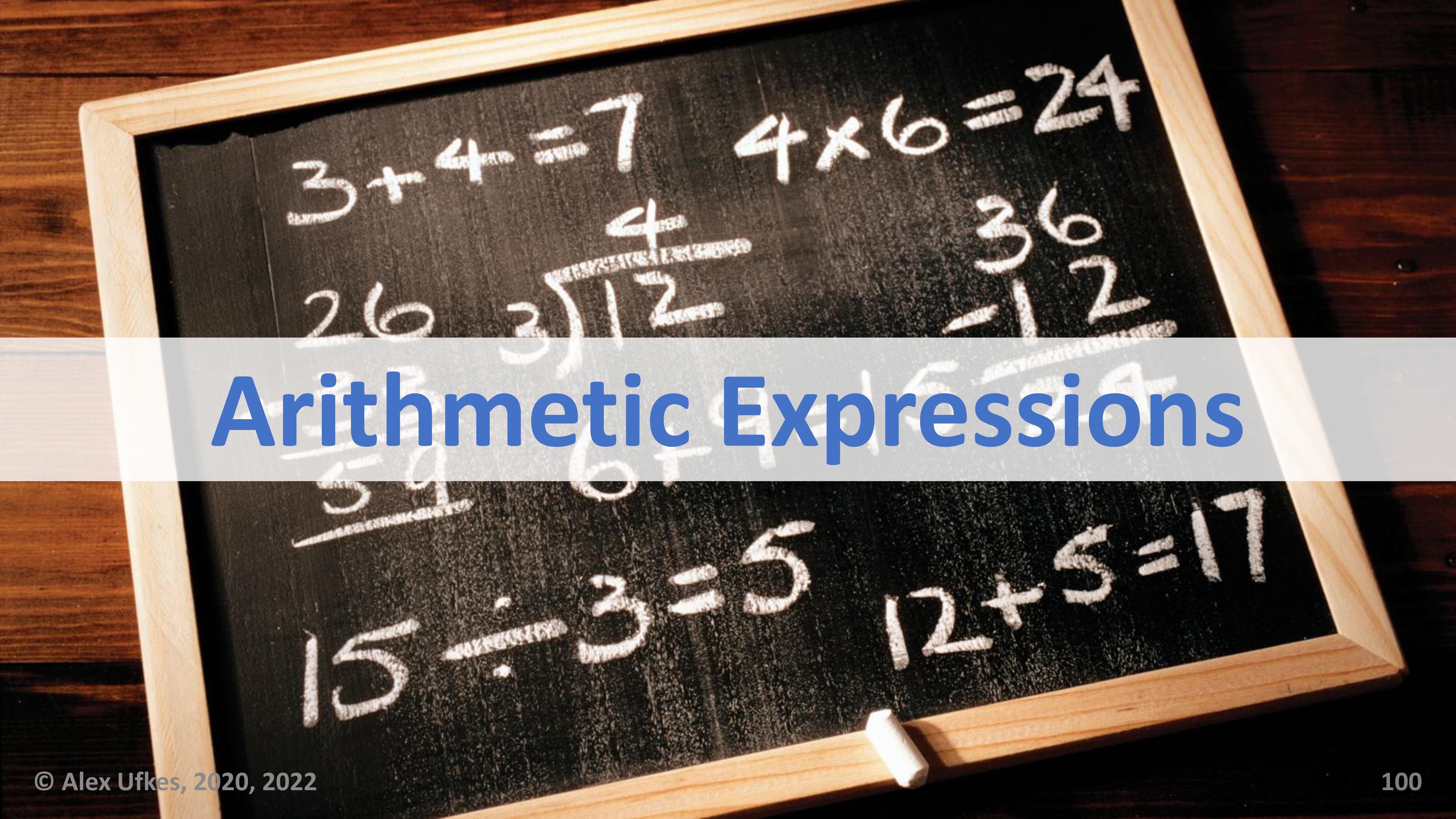
*In Smalltalk, you can send **any** message to **any** object. If the object doesn't know what to do with the message, a run-time error occurs.*



Send message **blahblah** to **SmallInteger** object **3**.

Summary: Literals

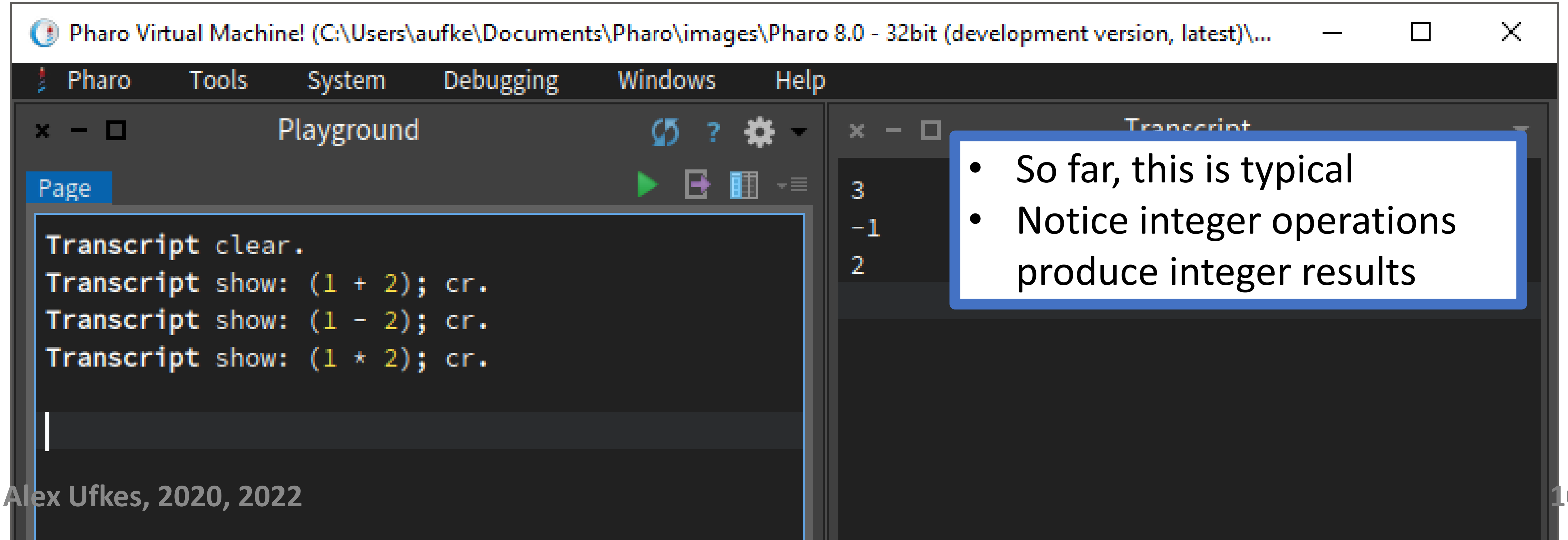
```
| b x |
b := true.                          "true constant"
b := false.                         "false constant"
x := nil.                           "nil object constant"
x := 1.                             "integer constants"
x := 3.14.                          "float constants"
x := 2e-2.                          "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"
```



Arithmetic Expressions

SMALLTALK-80 THE LANGUAGE **VS.** Java

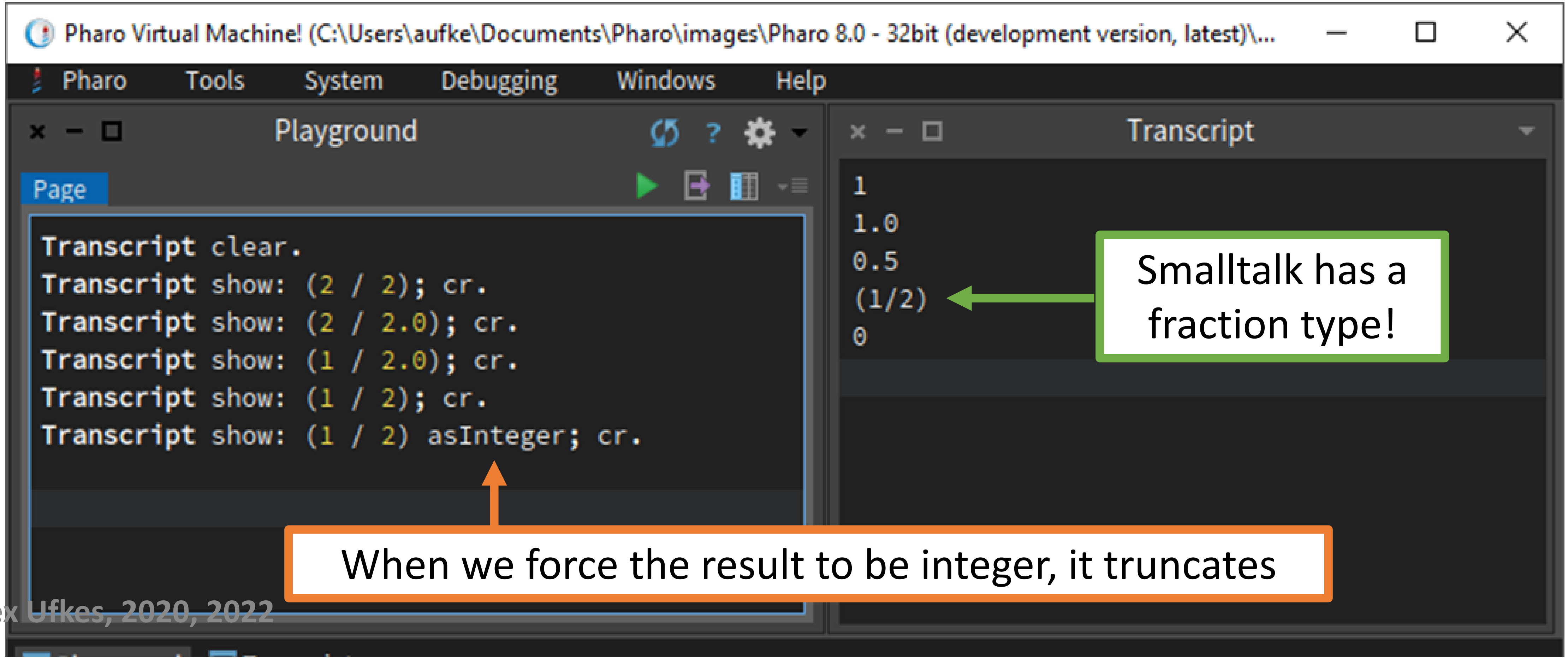
Arithmetic is largely the same in every language. Math is math.



- So far, this is typical
- Notice integer operations produce integer results

Division

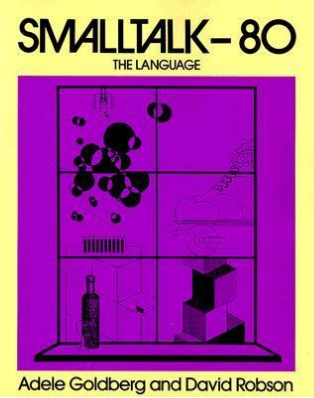
Division is a coin toss. Truncate? Convert to float?



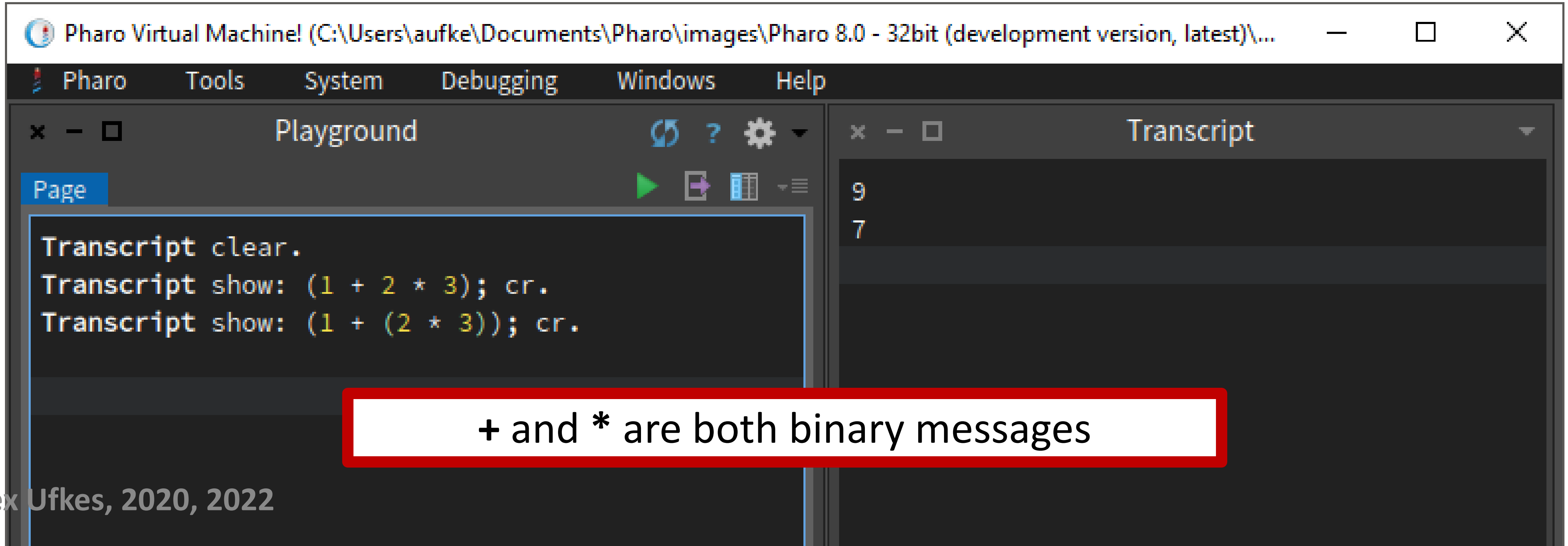
Operator Precedence in Java

Level	Operator	Description	Associativity
16	[] . ()	access array element access object member parentheses	left to right
15	++ --	unary post-increment unary post-decrement	not associative
14	++ -- + - ! ~	unary pre-increment unary pre-decrement unary plus unary minus unary logical NOT unary bitwise NOT	right to left
13	() new	cast object creation	right to left
12	* / %	multiplicative	left to right
11	+ - +	additive string concatenation	left to right
10	<< >> >>>	shift	left to right
9	< <= > >= instanceof	relational	not associative
8	== !=	equality	left to right
7	&	bitwise AND	left to right
6	^	bitwise XOR	left to right
5	|	bitwise OR	left to right
4	&&	logical AND	left to right
3	||	logical OR	left to right
2	?:	ternary	right to left
1	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	right to left

Operator/Message Precedence in



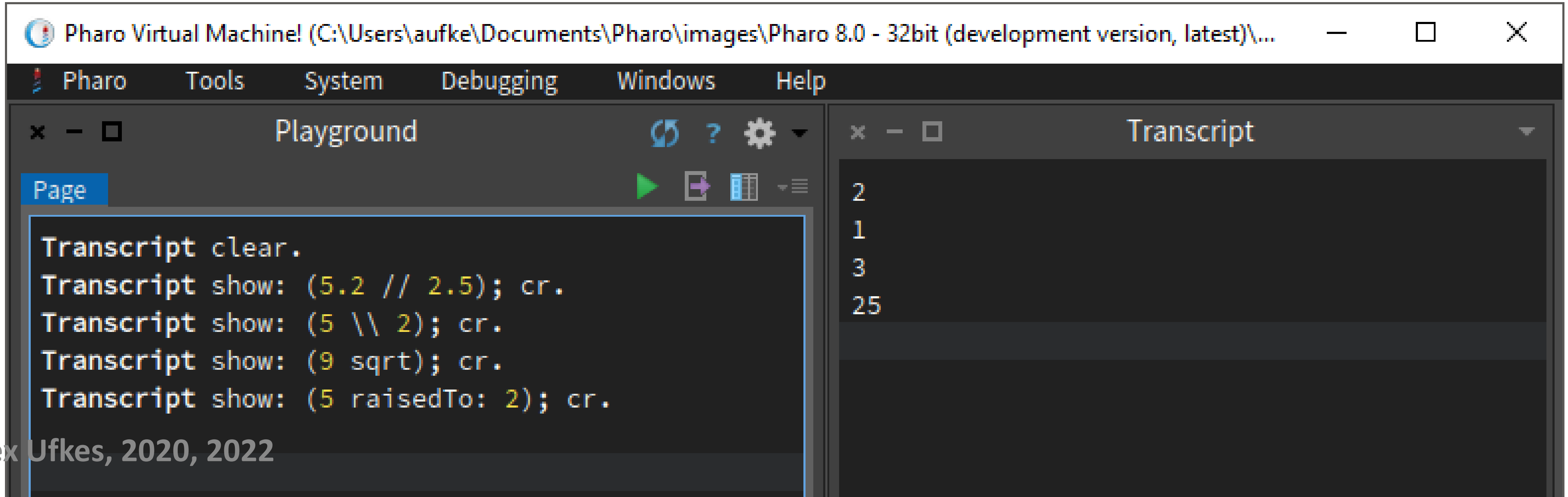
- ***Three*** levels! Unary -> Binary -> Keyword
- After that, ordering goes from left to right
- Brackets ***must*** be used to specify ordering outside of this.



+ and * are both binary messages

New or Differing Operators

`//`	Integer division
`\\`	Integer remainder
`sqrt`	Square root
`raisedTo:`	Exponentiation



```
x := 5 sign.                     "numeric sign (1, -1 or 0)"
x := 5 negated.                  "negate receiver"
x := 1.2 integerPart.            "integer part of number (1.0)"
x := 1.2 fractionPart.           "fractional part of number (0.2)"
x := 5 reciprocal.               "reciprocal function"
x := 6 * 3.1.                    "auto convert to float"
x := 5 squared.                  "square function"
x := 25 sqrt.                    "square root"
x := 5 raisedTo: 2.              "power function"
x := 5 raisedToInteger: 2        "power function with integer"
x := 5 exp.
x := -5 abs.
x := 3.99 rour
x := 3.99 trur
x := 3.99 rour
x := 3.99 trur
x := 3.99 floo
x := 3.99 ceil
x := 5 factori
x := -5 quo: 3
x := -5 rem: 3
x := 28 gcd: 1
x := 28 lcm: 1
x := 100 ln.
x := 100 log.
x := 100 log:
x := 100 floor
x := 180 degreesToRadians.       "convert degrees to radians"
```

... and much, much more:

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

```
x := 100 floorLog: 10.                  "floor of the log"
x := 180 degreesToRadians.              "convert degrees to radians"
x := 3.14 radiansToDegrees.             "convert radians to degrees"
x := 0.7 sin.                           "sine"
x := 0.7 cos.                           "cosine"
x := 0.7 tan.                           "tangent"
x := 0.7 arcSin.                        "arcsine"
x := 0.7 arcCos.                        "arccosine"
x := 0.7 arcTan.                        "arctangent"
x := 10 max: 20.                        "get maximum of two numbers"
x := 10 min: 20.                        "get minimum of two numbers"
x := Float pi.                          "pi"
x := Float e.                           "exp constant"
x := Float infinity.                    "infinity"
x := Float nan.                         "not-a-number"
x := Random new next; yourself. x next. "random number stream (0.0 to 1.0)"
x := 100 atRandom.                      "quick random number"
```

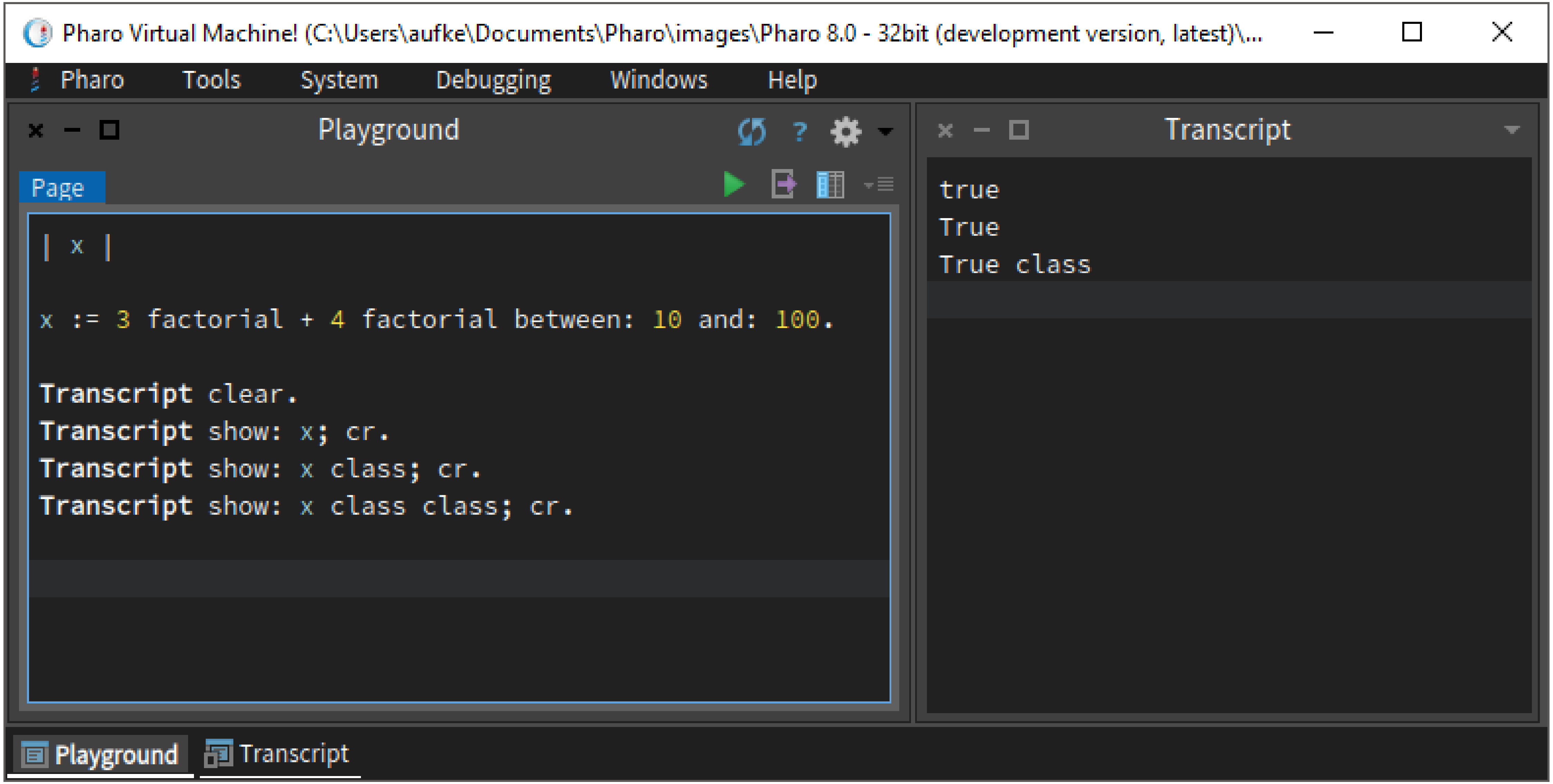
Example: What is the Result?

Which messages are unary? Binary? Keyword?

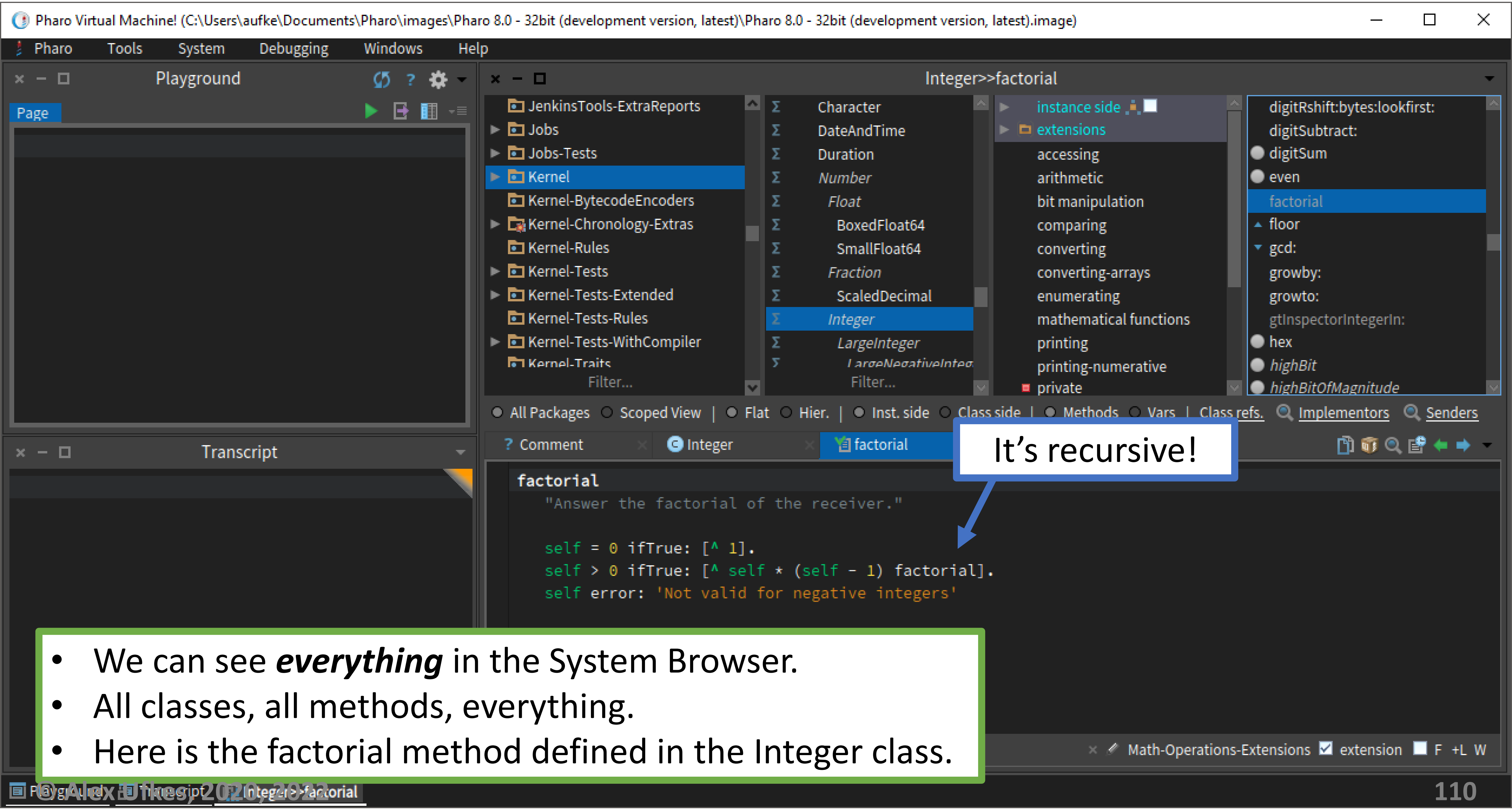
```
3 factorial + 4 factorial between: 10 and: 100
```

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

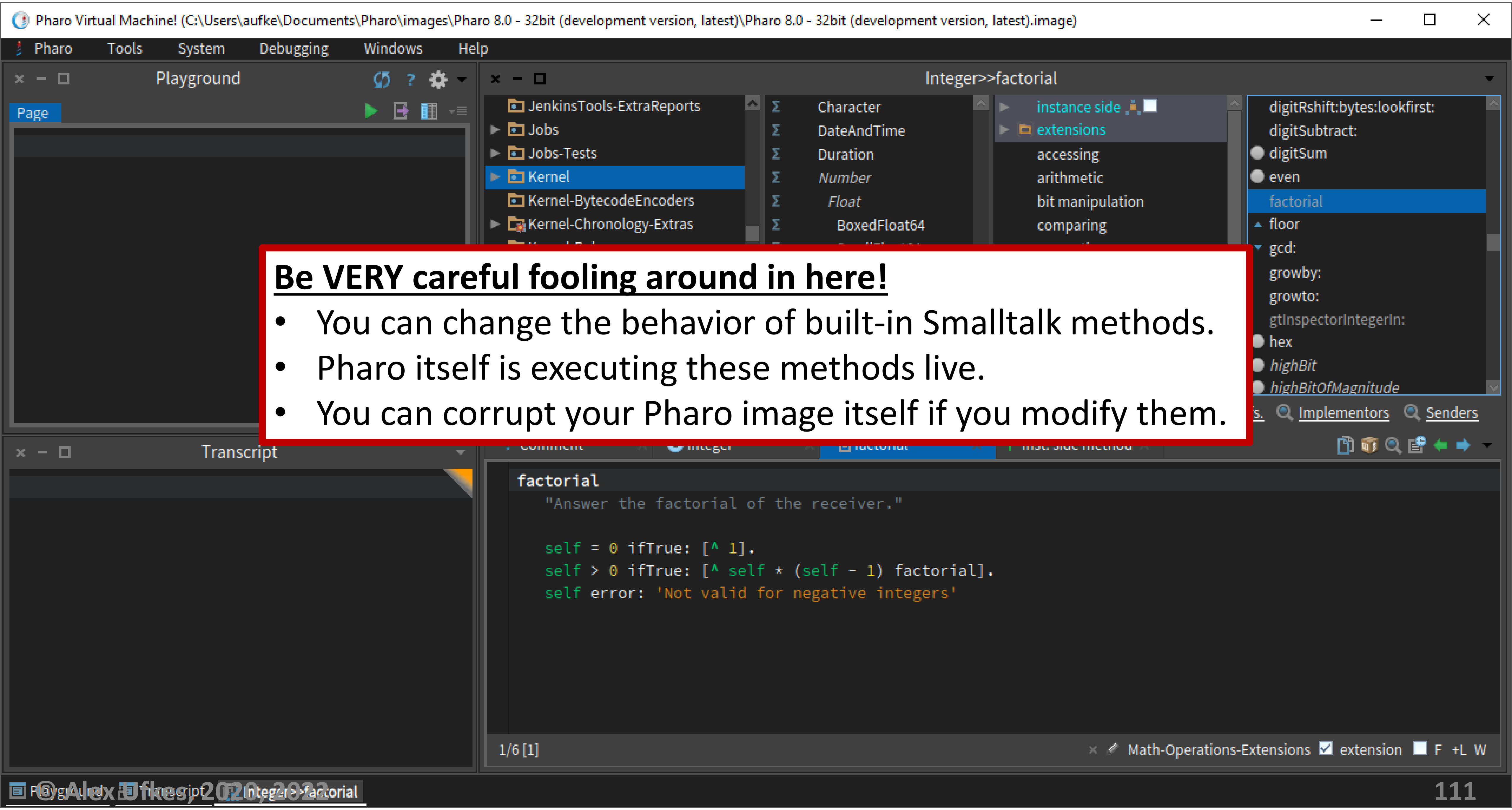
```
6 + 24 between: 10 and: 100

30 between: 10 and: 100

true
```


Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\...
Pharo Tools System Debugging Windows Help
Playground
Page
| x |
x := 3 factorial + 4 factorial between: 10 and: 100.
Transcript clear.
Transcript show: x; cr.
Transcript show: x class; cr.
Transcript show: x class class; cr.
Transcript
true
True
True class
Playground
Transcript

Classes



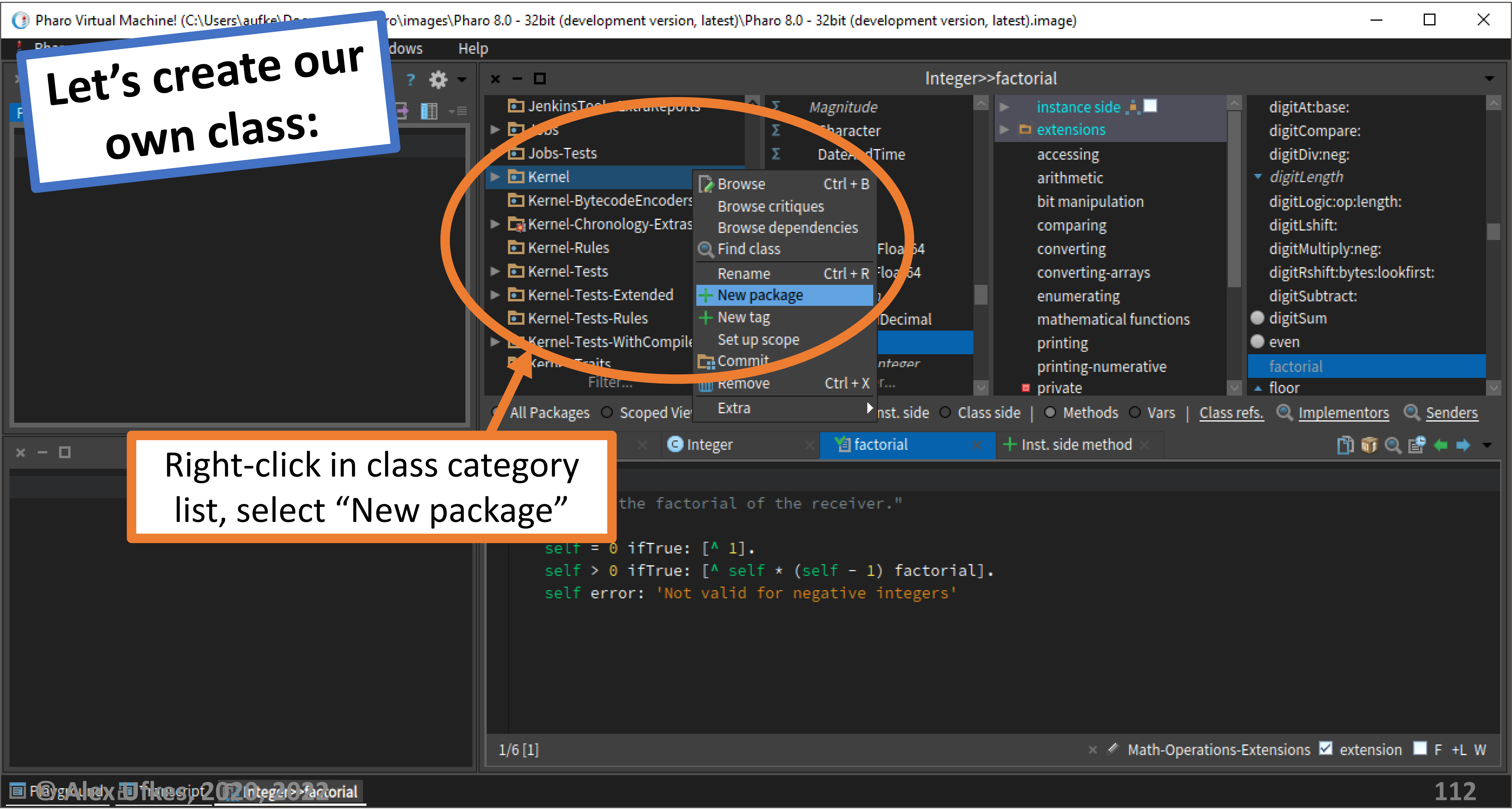
- We can see ***everything*** in the System Browser.
- All classes, all methods, everything.
- Here is the factorial method defined in the Integer class.



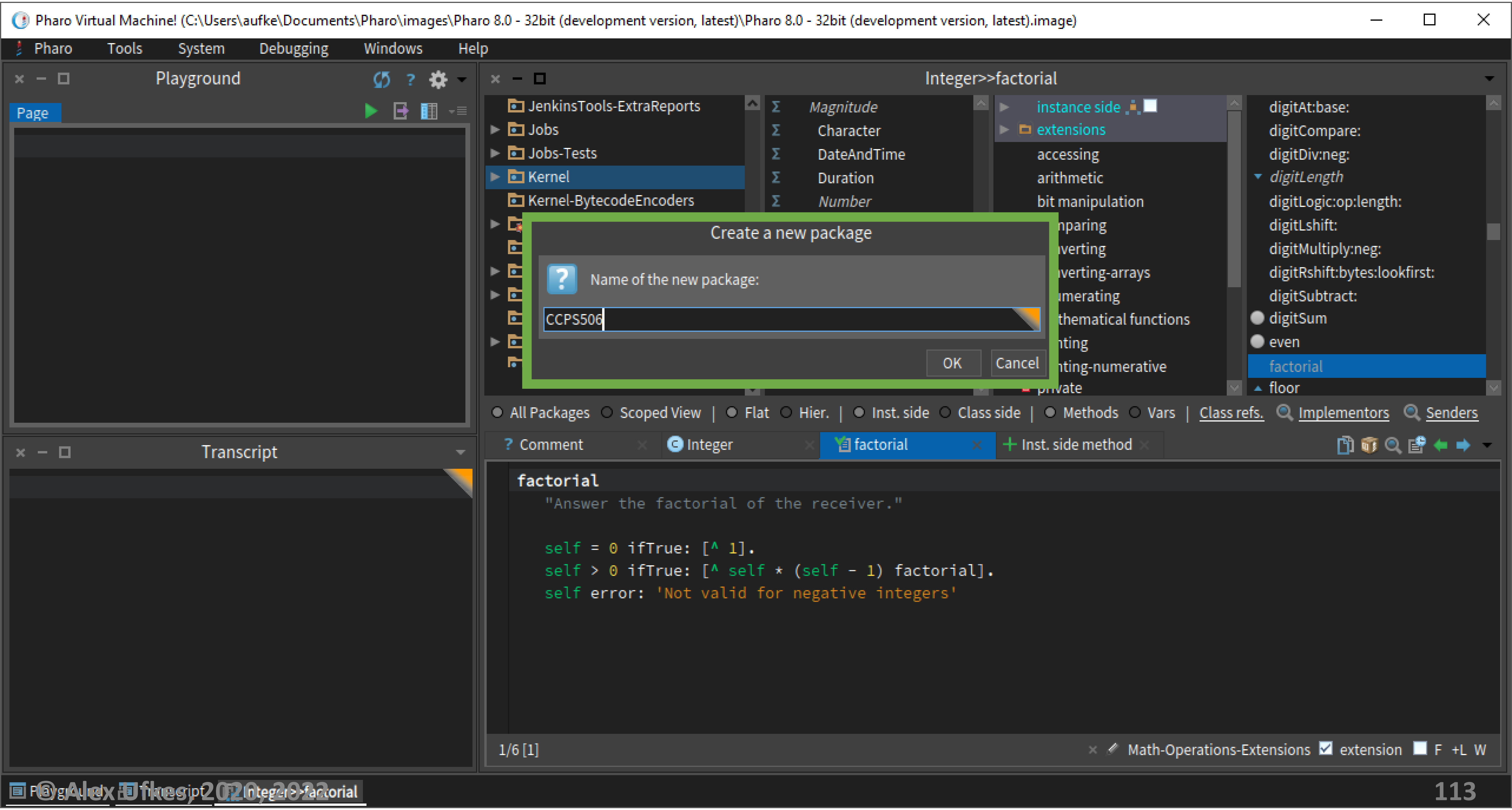
Be VERY careful fooling around in here!

- You can change the behavior of built-in Smalltalk methods.
- Pharo itself is executing these methods live.
- You can corrupt your Pharo image itself if you modify them.

Let's create our own class:

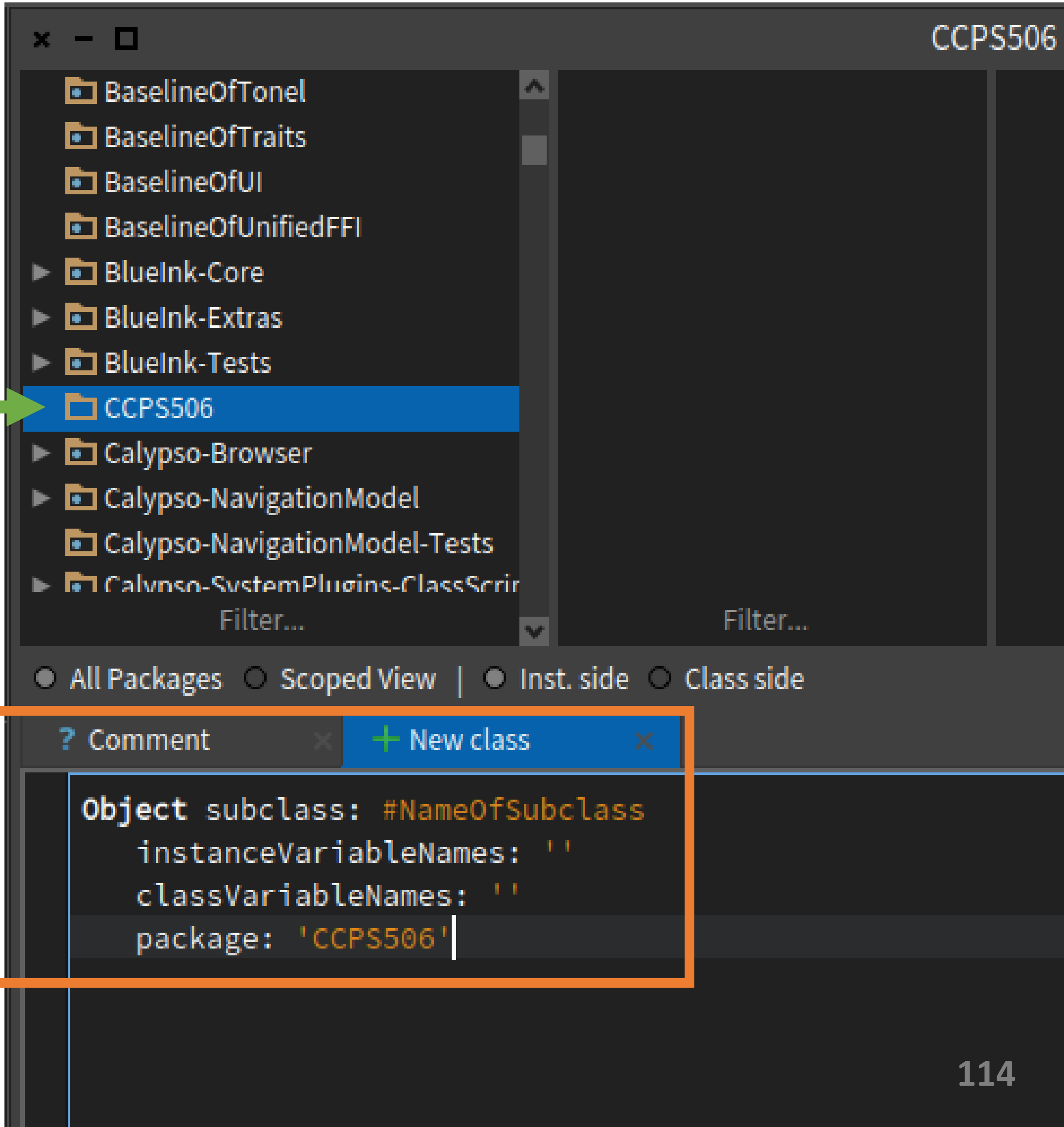


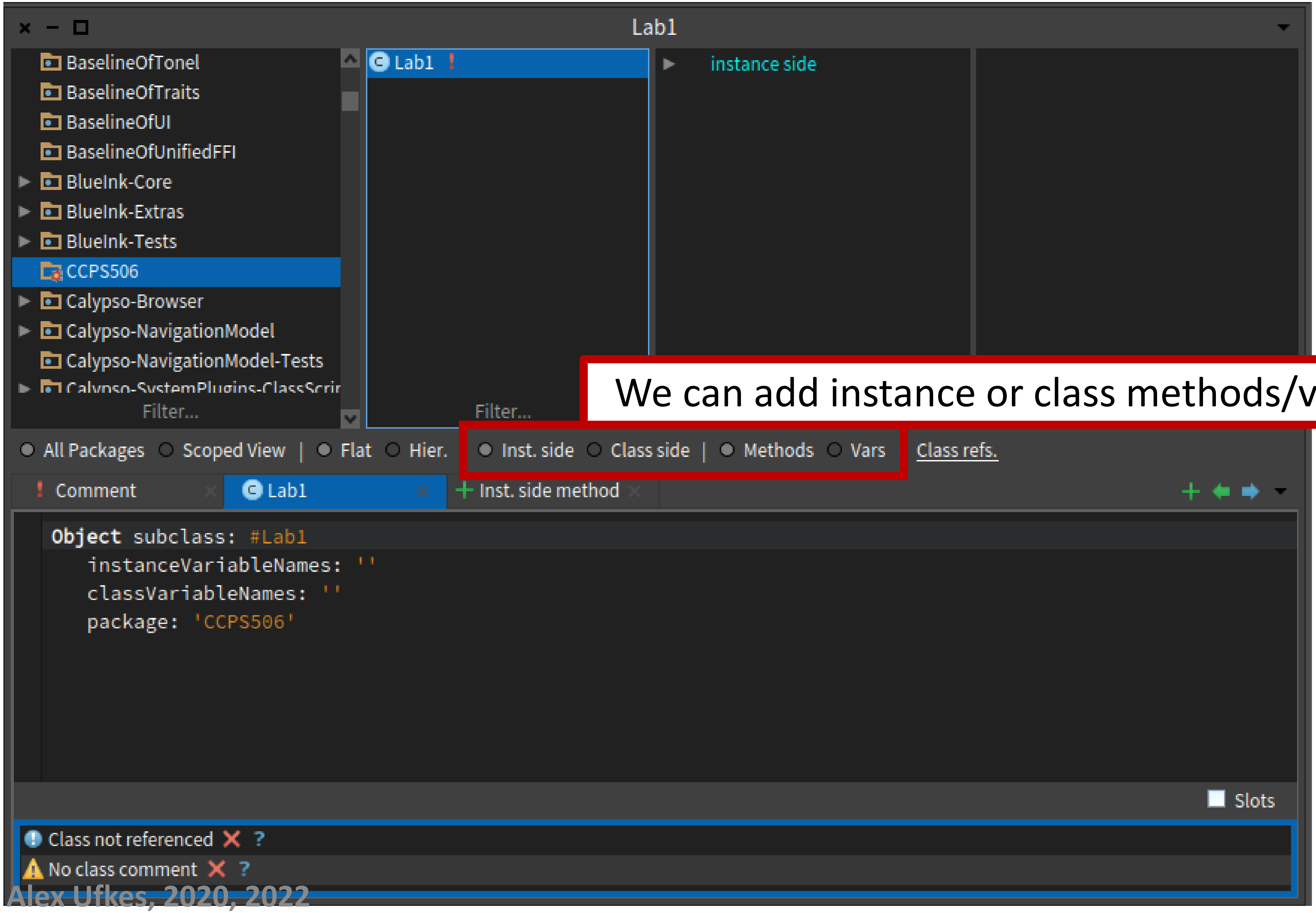
Right-click in class category list, select "New package"


Pharo Virtual Machine! (C:\Users\aufke\Documents\Pharo\images\Pharo 8.0 - 32bit (development version, latest)\Pharo 8.0 - 32bit (development version, latest).image)
Pharo Tools System Debugging Windows Help
Playground
Page
Transcript
Integer>>factorial
JenkinsTools-ExtraReports
Jobs
Jobs-Tests
Kernel
Kernel-BytecodeEncoders
Magnitude
Character
DateAndTime
Duration
Number
instance side
extensions
accessing
arithmetic
bit manipulation
digitAt:base:
digitCompare:
digitDiv:neg:
digitLength
digitLogic:op:length:
digitLshift:
digitMultiply:neg:
digitRshift:bytes:lookfirst:
digitSubtract:
digitSum
even
factorial
floor
Create a new package
Name of the new package:
CCPS506
OK
Cancel
All Packages
Scoped View
Flat
Hier.
Inst. side
Class side
Methods
Vars
Class refs.
Implementors
Senders
Comment
Integer
factorial
Inst. side method
factorial
"Answer the factorial of the receiver."
self = 0 ifTrue: [^ 1].
self > 0 ifTrue: [^ self * (self - 1) factorial].
self error: 'Not valid for negative integers'
1/6 [1]
Math-Operations-Extensions
extension
F +L W
Playground
Transcript
Integer>>factorial

Select your new package

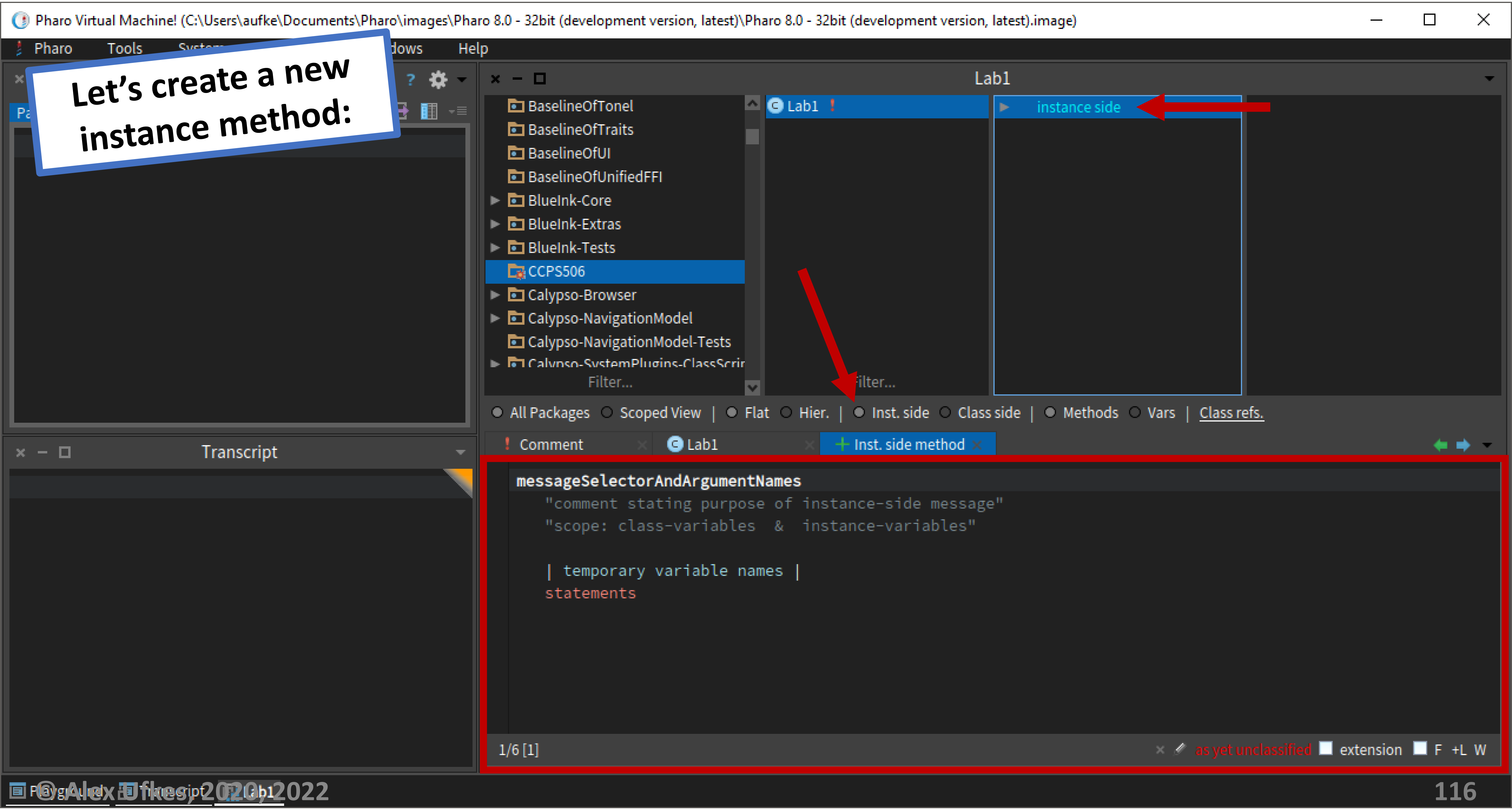
- Under “New class” is a class template
- Give your subclass a catchy name
- Ctrl-S to save

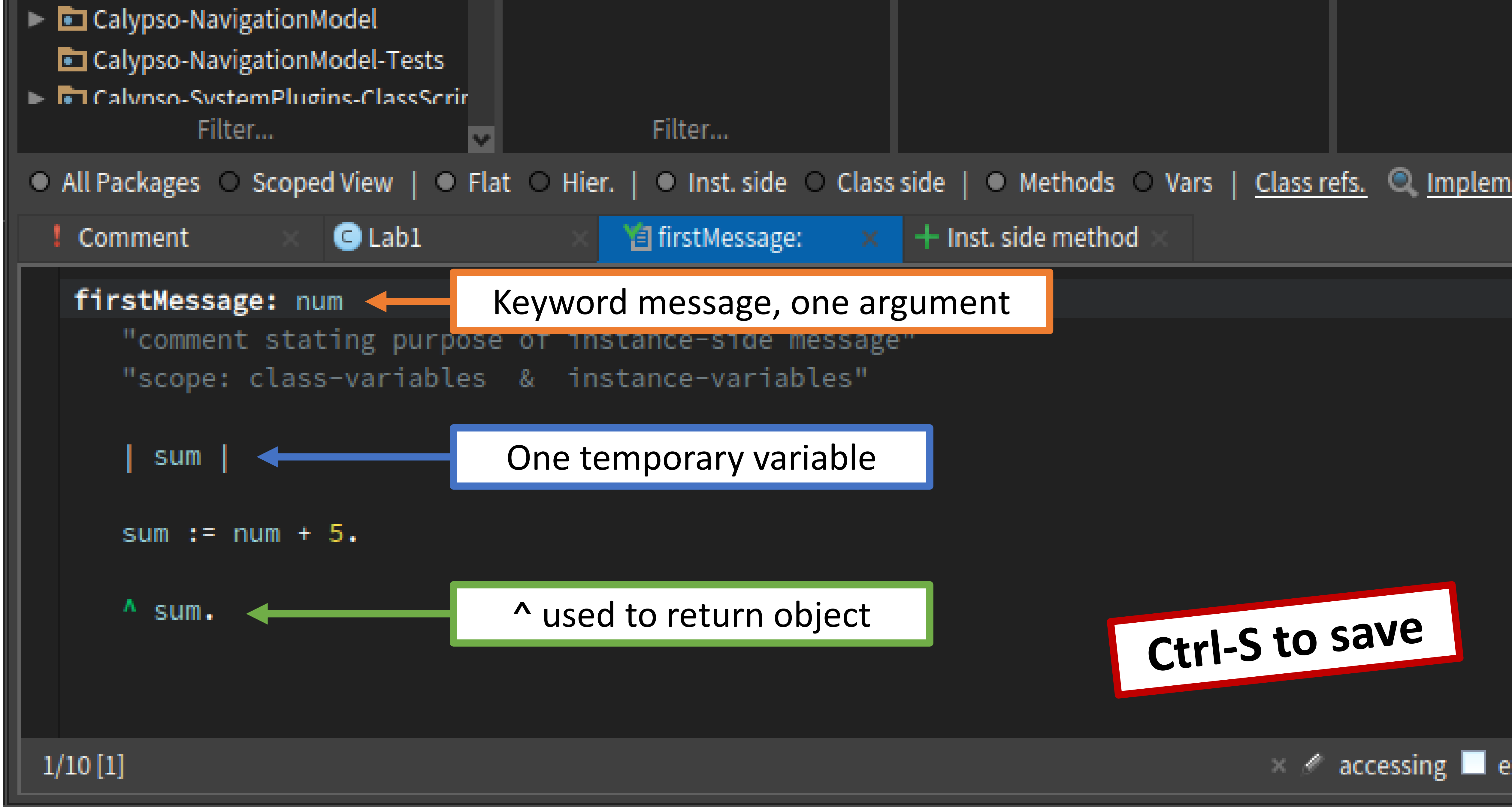




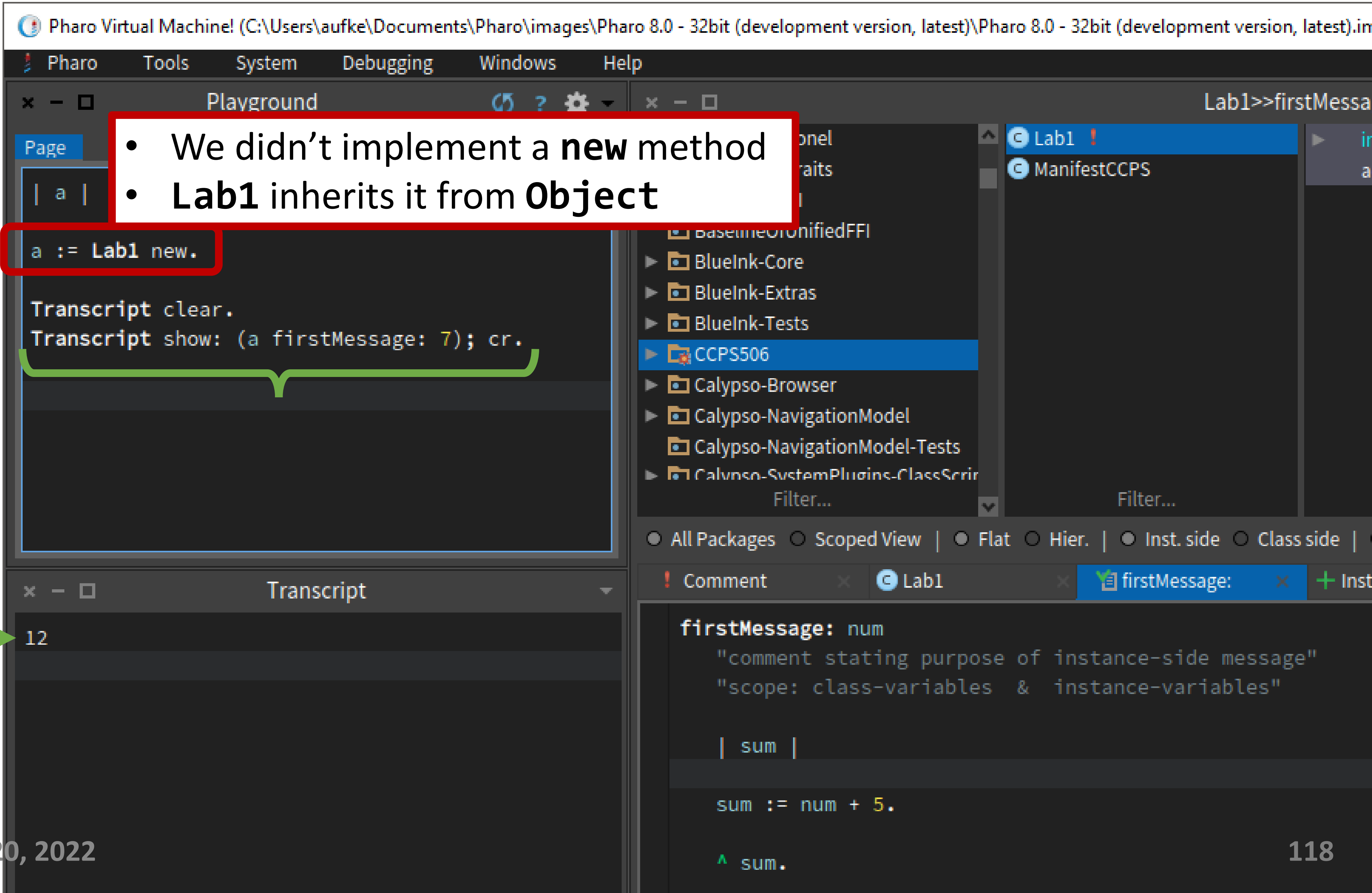
We can add instance or class methods/variables

Let's create a new instance method:




Calypso-NavigationModel
Calypso-NavigationModel-Tests
Filter...
Filter...
All Packages
Scoped View
Flat
Hier.
Inst. side
Class side
Methods
Vars
Class refs.
Comment
Lab1
firstMessage:
Inst. side method
firstMessage: num
"comment stating purpose of instance-side message"
"scope: class-variables & instance-variables"
| sum |
sum := num + 5.
^ sum.
Keyword message, one argument
One temporary variable
^ used to return object
Ctrl-S to save
1/10 [1]
accessing

- We didn’t implement a **new** method
- **Lab1** inherits it from **Object**



Summary

- Imperative programming paradigm
- Object Oriented Programming
- Smalltalk:
    - **Message Passing**
    - Objects, literals
    - Arithmetic
- Classes and methods in Pharo

Next week…

Blocks & more

*(The **fun** stuff!)*