C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 3: Out with Smalltalk, in with Elixir

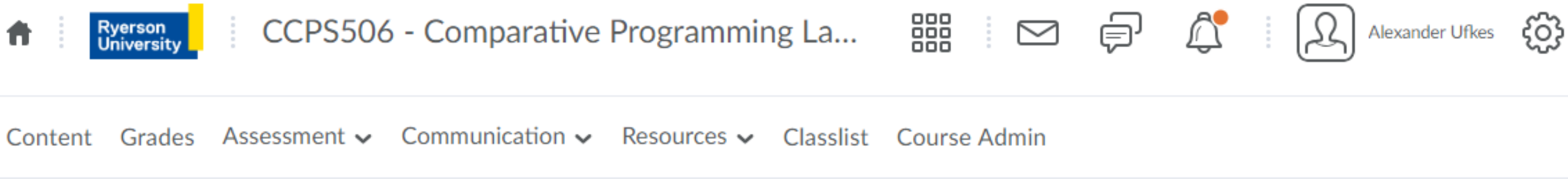
Ryerson University

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)



- Assignment description is posted!
- If you liked Smalltalk, you could start working on the Smalltalk version.

Today

- Double dispatch
- Smalltalk conclusion
- Functional paradigm
- Getting started with Elixir

Method “Overloading”

Method Overloading

Methods are overloaded through differing parameter lists:

```
public class ArrayListTest
{
    public static int add (int x)
    {
        return x + x;
    }

    public static int add (int x, int y)
    {
        return x + y;
    }

    public static void main(String[] args)
    {
        System.out.println( add(5) );
        System.out.println( add(5, 2) );
    }
}
```

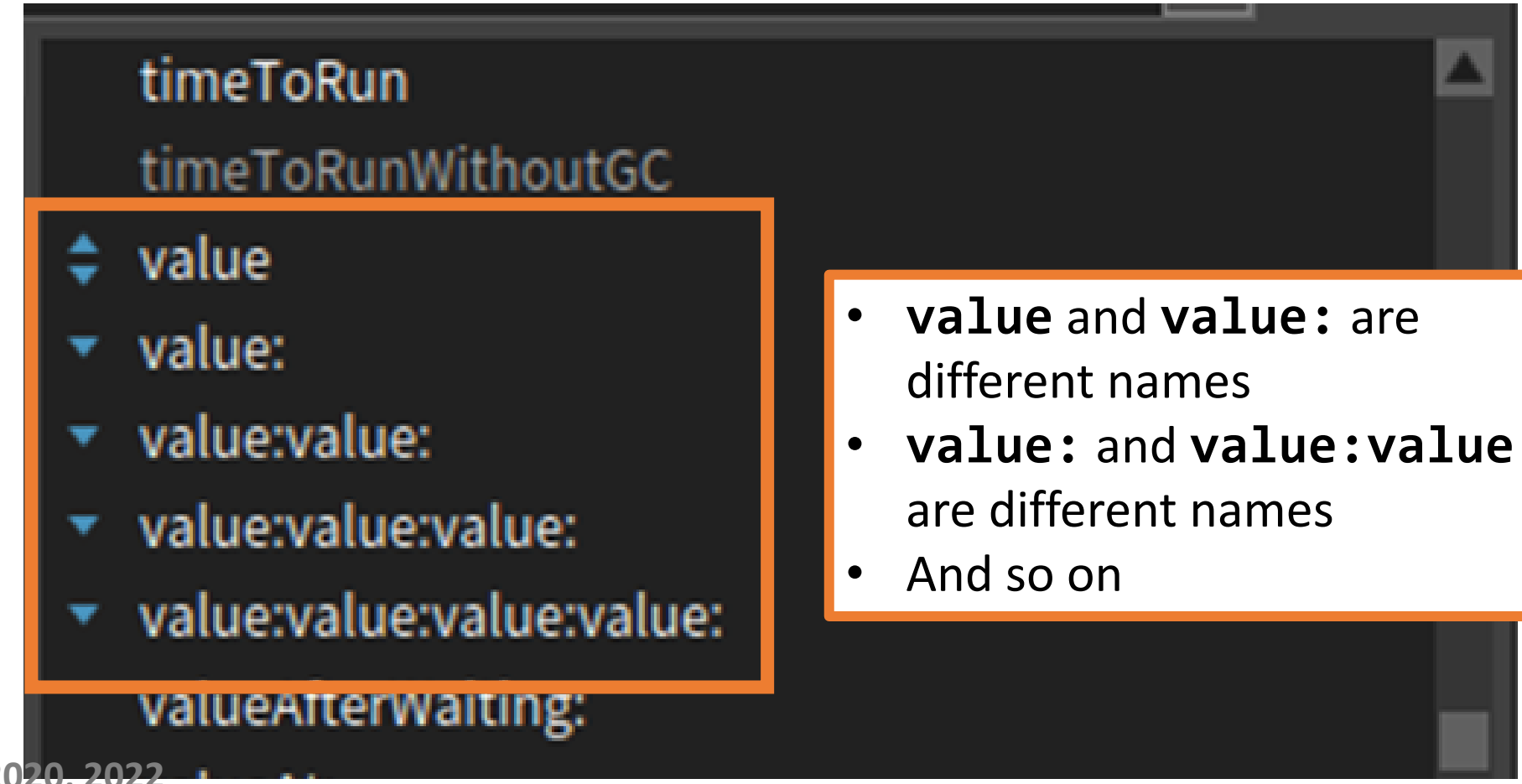
- In Java, method name and parameter list are independent.
- In Smalltalk, they are fundamentally linked

Method Overloading

- In Smalltalk, there is no overloading in this fashion.
- We cannot have a single message that optionally accepts differing numbers of arguments.
- When we add another argument, the method name changes.

Method Overloading

When we add another argument, the method/message name changes.



- **`value`** and **`value:`** are different names
- **`value:`** and **`value:value:`** are different names
- And so on

Method Overloading

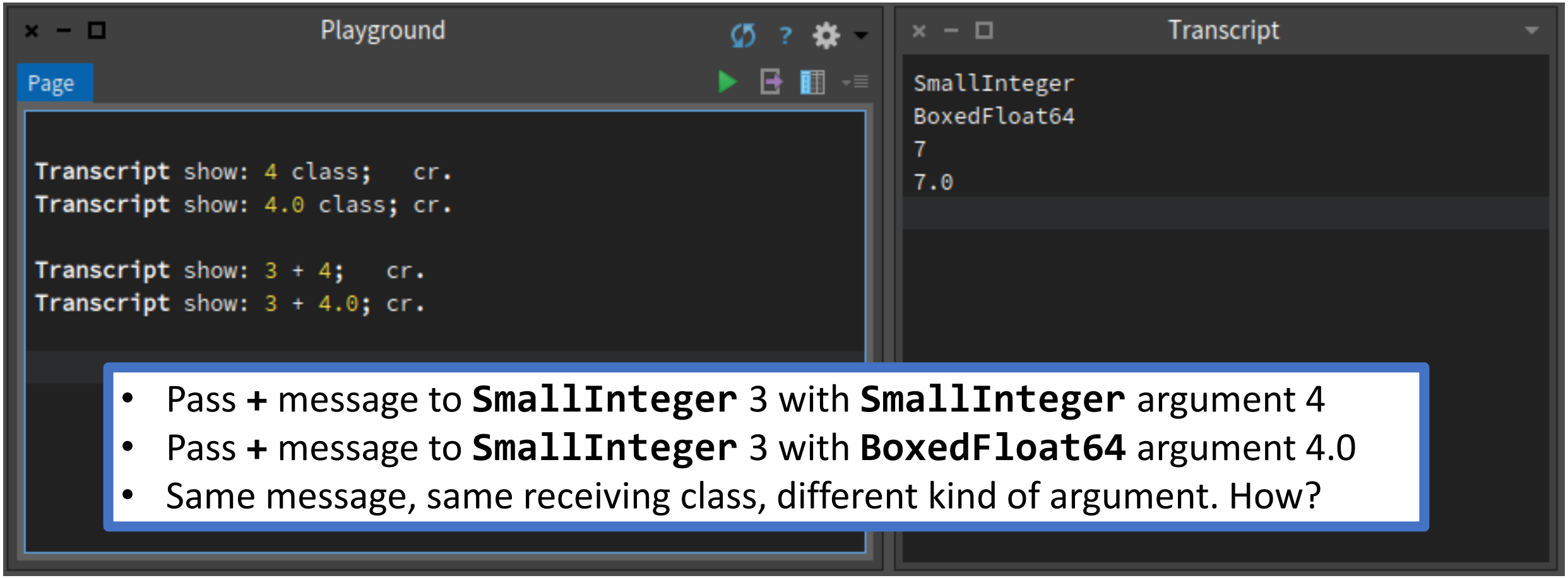
- All well and good, but what about the same number of arguments with different types?
- In Java, compiler sees these as different:

```
public int add(int x, int y)

public double add(double x, double y)
```

- In Smalltalk, argument types aren't checked upon message pass.
- Invoking a method (passing a message) only fails when receiving object can't handle the message.
- We get a Did Not Understand error (DNU).

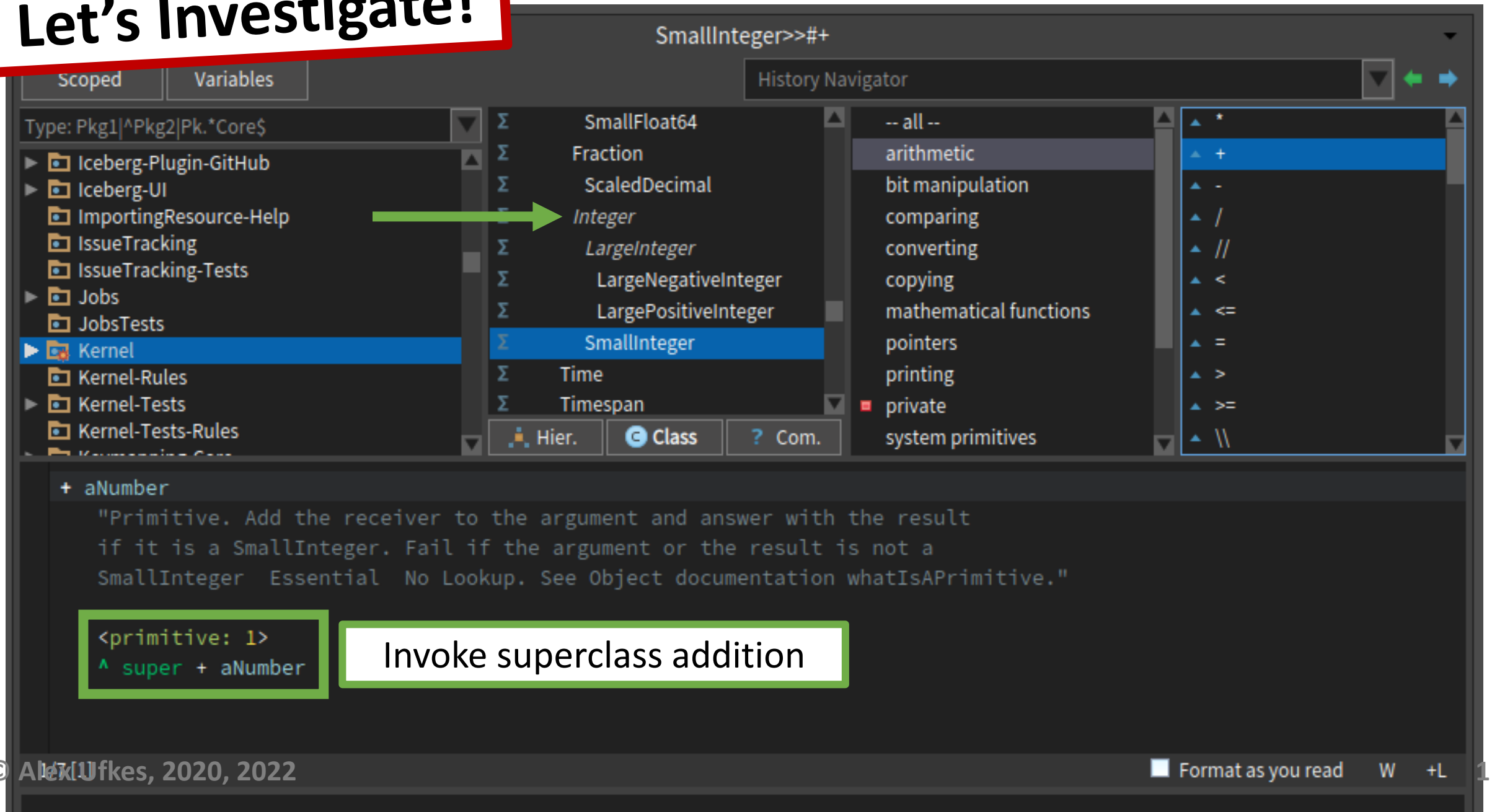
Method Overloading

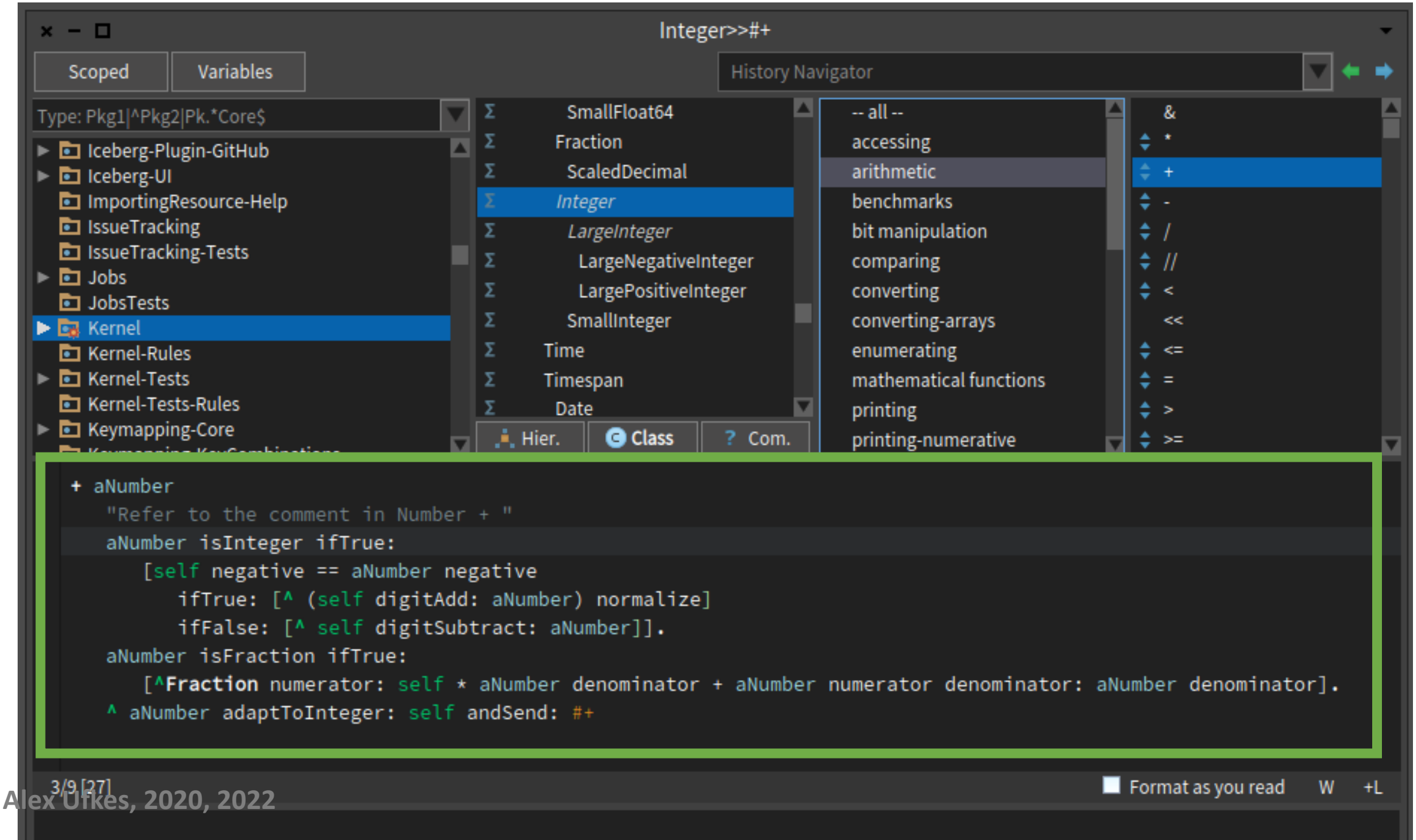
However! The following code succeeds!



- Pass **+** message to **`SmallInteger`** 3 with **`SmallInteger`** argument 4
- Pass **+** message to **`SmallInteger`** 3 with **`BoxedFloat64`** argument 4.0
- Same message, same receiving class, different kind of argument. How?

Let's Investigate!

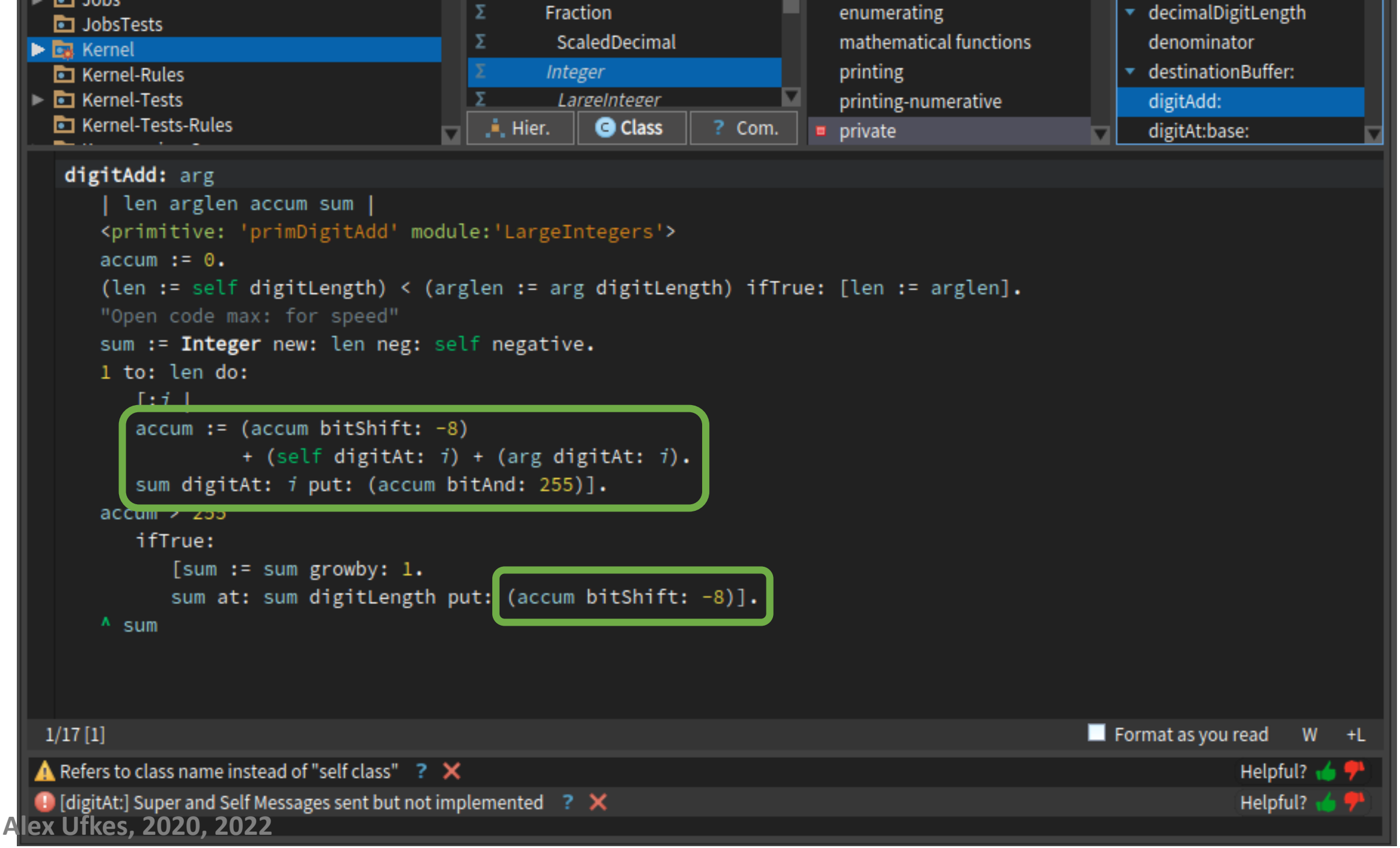




- Check if argument is integer.
- If so, it's an integer expression and we can react accordingly

- Check if operands have same sign
- **`negative`** returns Boolean

```
+ aNumber
    "Refer to the comment in Number + "
    aNumber isInteger ifTrue:
        [self negative == aNumber negative
            ifTrue: [^ (self digitAdd: aNumber) normalize]
            ifFalse: [^ self digitSubtract: aNumber]].
    aNumber isFraction ifTrue:
        [^Fraction numerator: self * aNumber denominator + aNumber numerator denominator: aNumber denominator].
    ^ aNumber adaptToInteger: self andSend: #+
```

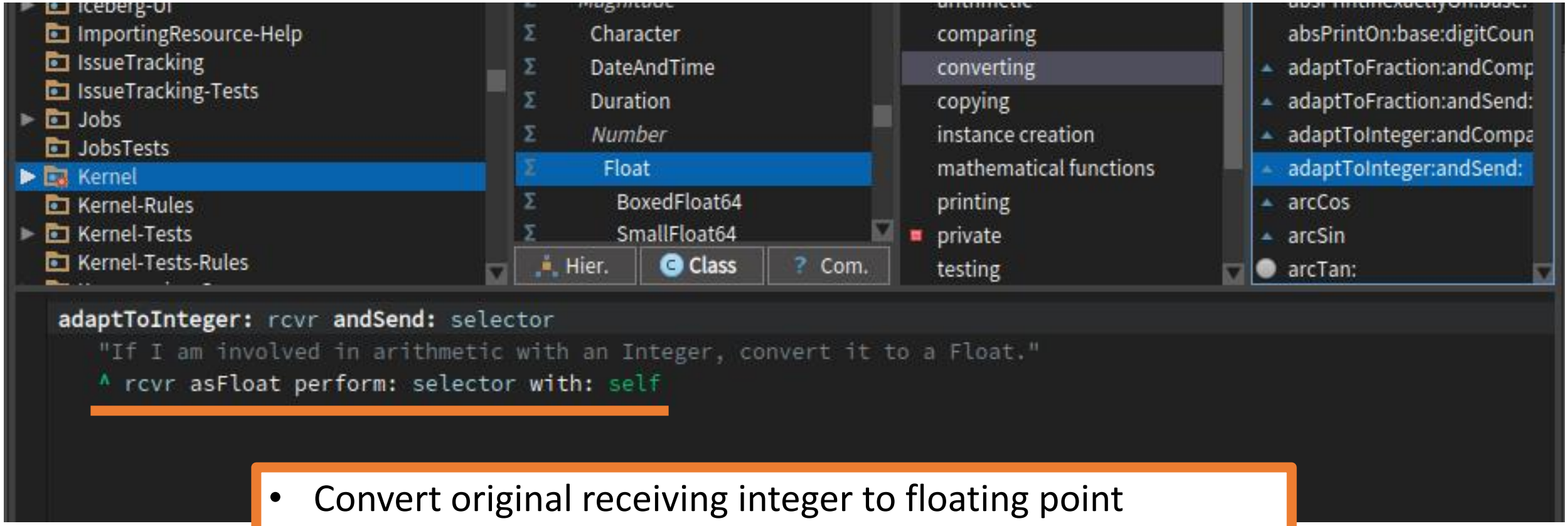


```
+ aNumber
    "Refer to the comment in Number + "
    aNumber isInteger ifTrue:
        [self negative == aNumber negative
            ifTrue: [^ (self digitAdd: aNumber) normalize]
            ifFalse: [^ self digitSubtract: aNumber]]
    aNumber isFraction ifTrue:
        [^Fraction numerator: self * aNumber denominator + aNumber numerator denominator: aNumber denominator].
    ^ aNumber adaptToInteger: self andSend: #+
```

If arg **`isFraction`**, we can still add precisely using numerator and denominator

If arg is neither fraction nor integer, we send it **`adaptToInteger:andSend:`** message

adaptToInteger:andSend:



- Convert original receiving integer to floating point
- Perform addition between two **BoxedFloat64**
- It's now a problem for the Float class implementation of **+**!

Double Dispatch

- Method overloading not possible in Smalltalk.
- Uses the previous technique instead, called ***double dispatch***
- Double/multiple dispatch is ***not unique to Smalltalk***.

Double Dispatch:

- Broadly: Make additional method/function calls based on the **types** of the objects involved in the original call at runtime.
- I.e., if arg is float, invoke method for floating point addition.
- Overloading is done at compile time; double dispatch occurs at runtime.

Double Dispatch

Overloading is decided at compile time, double dispatch at runtime.

- In Smalltalk (double dispatch), the same method gets invoked regardless of the argument type. Same message regardless!
- Secondary method call(s) occur in the body of the first method, depending on argument type.

- In Java, a different method gets invoked from the very start depending on the type of the argument.
- Decided at *compile time (early binding)*.

Explore this on your own for some of the other types and operators

Late VS Early Binding

Dynamic/late binding VS static/early binding

Early binding

- Method to be called is found at compile time
- Method not found = compile error
- More efficient at runtime

Late binding

- Method is looked up at runtime
- Often as simple is searching name
- Symbol comparison in Smalltalk
- Method not found = runtime error
- Costlier at runtime

Double dispatch happens at runtime, late binding

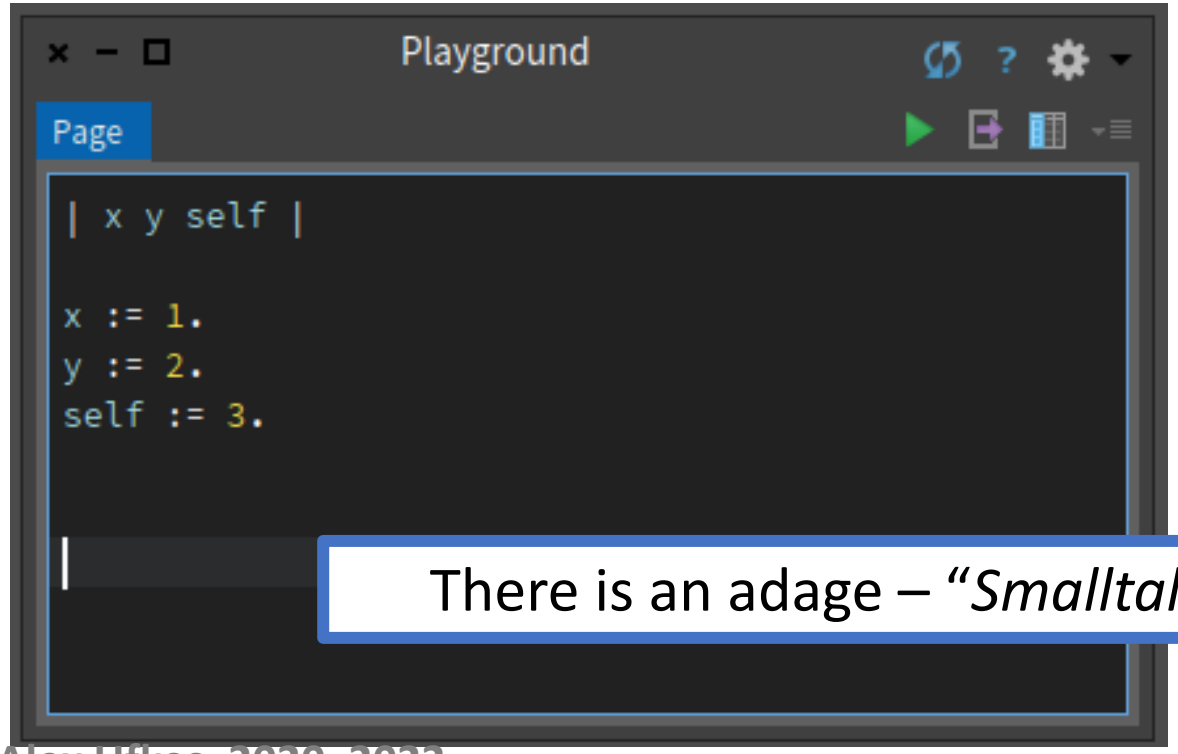
This concludes your Smalltalk crash course!

Let’s finish with a high-level summary.

Smalltalk Syntax

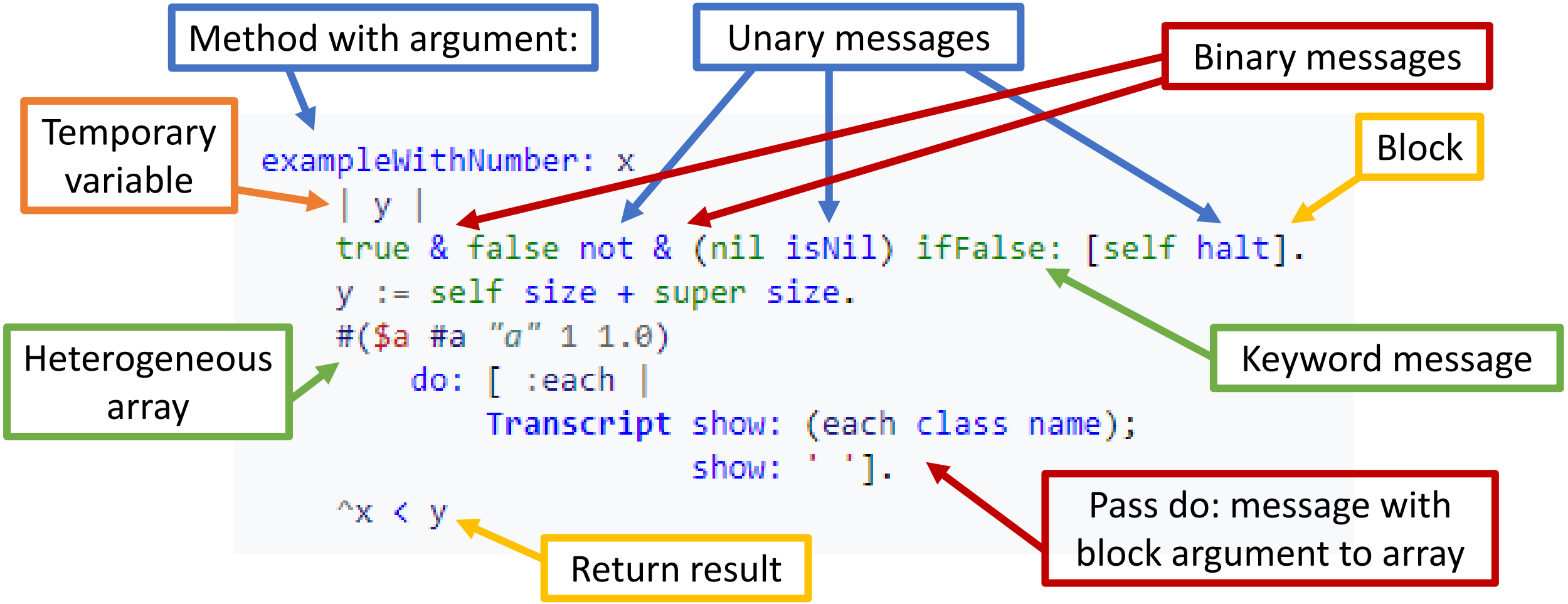
Extremely minimalist:

- Only five reserved "keywords": **true**, **false**, **nil**, **self**, **super**
- Java has 50, C++ has 82, C has 32



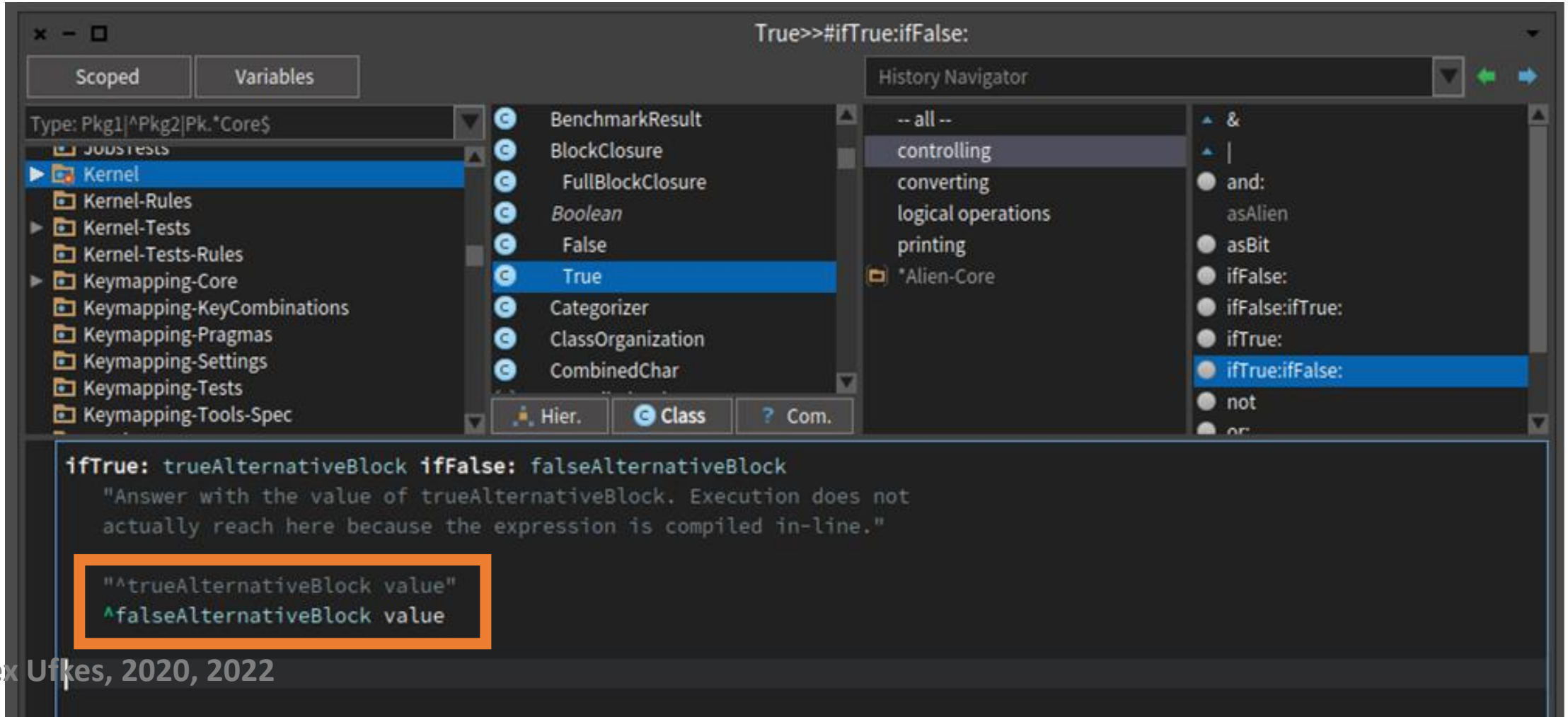
There is an adage – "*Smalltalk syntax fits on a postcard*"

"Smalltalk syntax fits on a postcard"



Smalltalk Extensibility

We are free to modify core Smalltalk classes and methods:



Amuse your friends! Confound your enemies!

We are free to modify core Smalltalk classes and methods:

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

    "^trueAlternativeBlock value"
    ^falseAlternativeBlock value
```

Beware: You **WILL** cause havoc and be forced to create a new Pharo image.

- The Pharo live environment is using these very methods in a JIT fashion.
- Changing **`True`** to **`False`** means Pharo itself thinks **`True`** is **`False`**.

Smalltalk Semantics

“A compiler will complain about syntax, your coworkers will complain about semantics”

Always remember:

1. Everything is an object
2. Computation is done by passing messages to objects.
3. Message precedence: unary, binary, keyword.
    - Equal precedence evaluates left to right

Everything else follows from these principles.

Smalltalk

- Has garbage collection (like Java)
- Best-in-class IDE (according to fans).
  - Class browser, playground, debugger, transcript, etc.
- Just-in-time compilation (JIT)
  - Code executed in a live environment
- Image-based development

Smalltalk popularity? A dead language?

https://medium.com/smalltalk-talk/who-uses-smalltalk-c6fdaa6319a

JPMorgan Chase:

"Around the time that Java was just being introduced to the world, the banking giant rolled out a new financial risk management and pricing system called Kapital, written entirely in Cincom Smalltalk."

"JPMorgan estimates that developing and maintaining this system in any other language would've required at least three times the amount of resources."

Smalltalk popularity? A dead language?

https://medium.com/smalltalk-talk/smalltalk-and-the-future-of-the-software-industry-3f69cac13e5a

"One of the goals of Smalltalk was to make it very easy to teach programming to children*."*

"I believe the best way to teach beginners how to program is with a good teaching language. Languages like Java, Python, JavaScript, C#, C/C++, PHP, Ruby are all industrial languages; they carry a lot of industrial baggage that can get in the way of a beginner."

Smalltalk popularity? A dead language?

https://www.quora.com/Why-did-the-Smalltalk-programming-language-fail-to-become-a-popular-language

"The popular story that's been around is that Sun killed Smalltalk with Java. That seems to be partly true…"

"I think what prevented Smalltalk from increasing in popularity was the popularity of developing for the internet…"

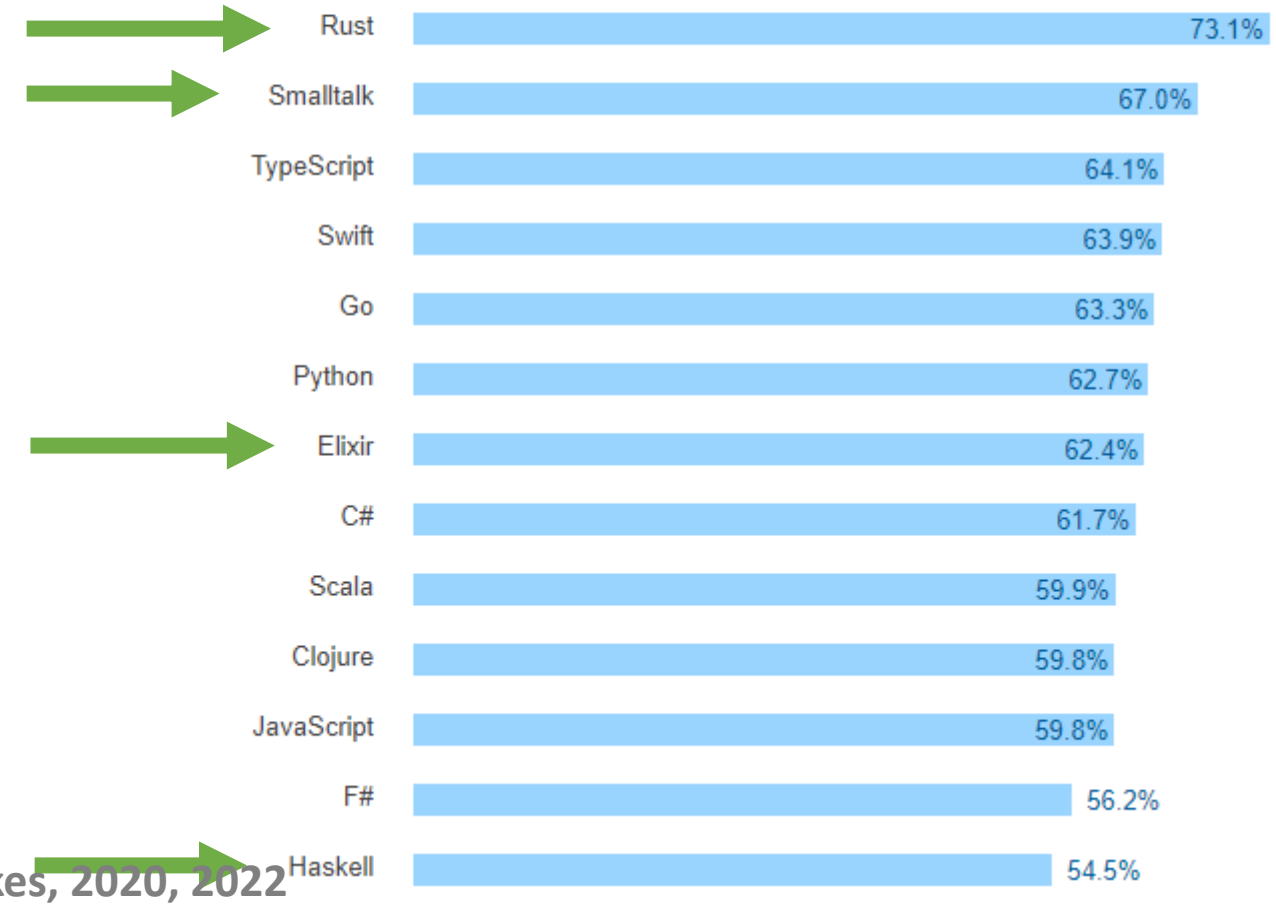
"The dominant idea in the Smalltalk community in the '90s was that it was a GUI desktop platform, and that's where it should stay."

Most Loved, Dreaded, and Wanted

2017

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted



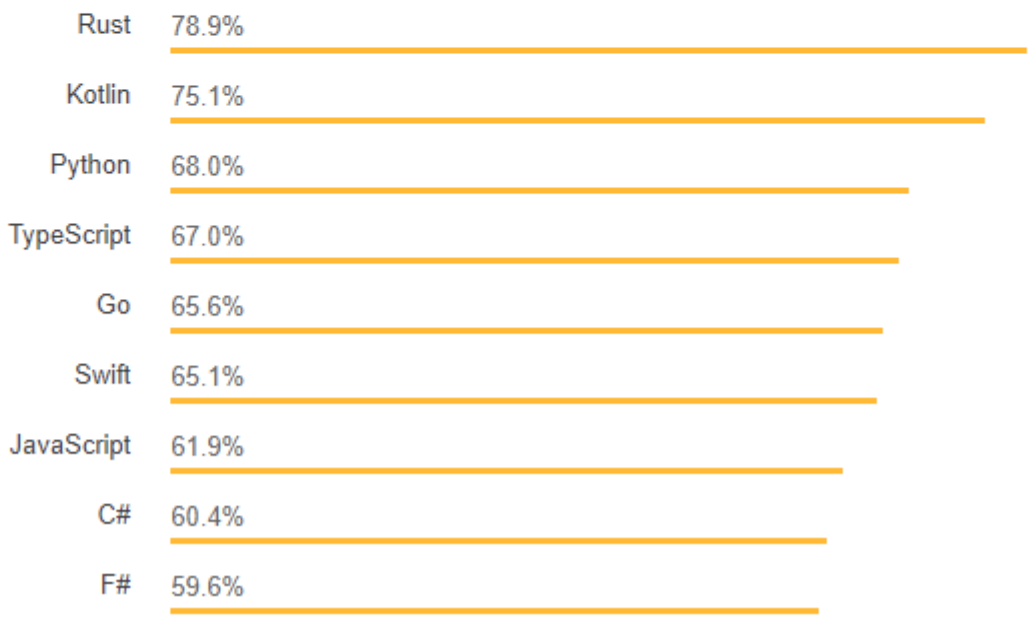
Loved: % of people currently using that want to keep using

2018

Most Loved, Dreaded, and Wanted

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted



- Smalltalk disappears.
- Not sure why, but I think they didn’t include it in the survey.
- Rust still on top!

Most Loved, Dreaded, and Wanted

2018

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted

Visual Basic 6	89.9%
Cobol	84.1%
CoffeeScript	82.7%
VB.NET	80.9%
VBA	80.0%
Matlab	77.4%
Assembly	71.4%
Perl	71.3%
Objective-C	70.3%
Lua	68.2%
Groovy	66.4%
Delphi/Object Pascal	65.1%
C	62.6%

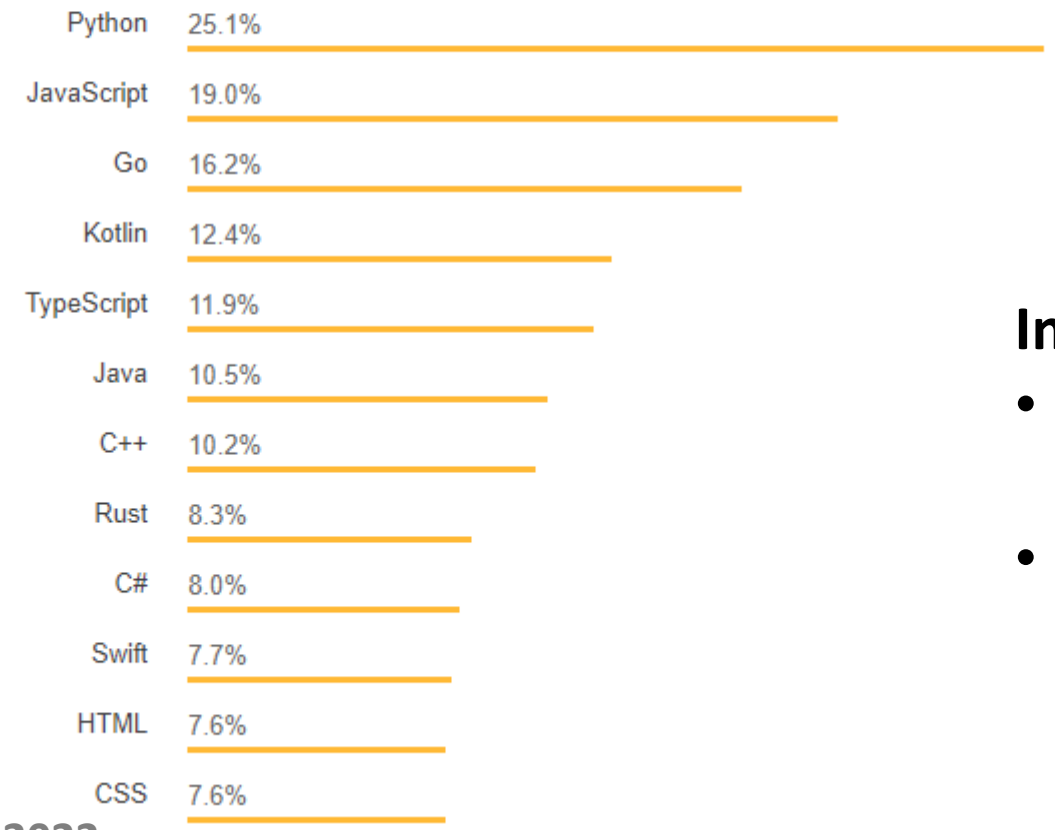
Dreaded: % of people currently using that no longer want to.

Most Loved, Dreaded, and Wanted

2018

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted

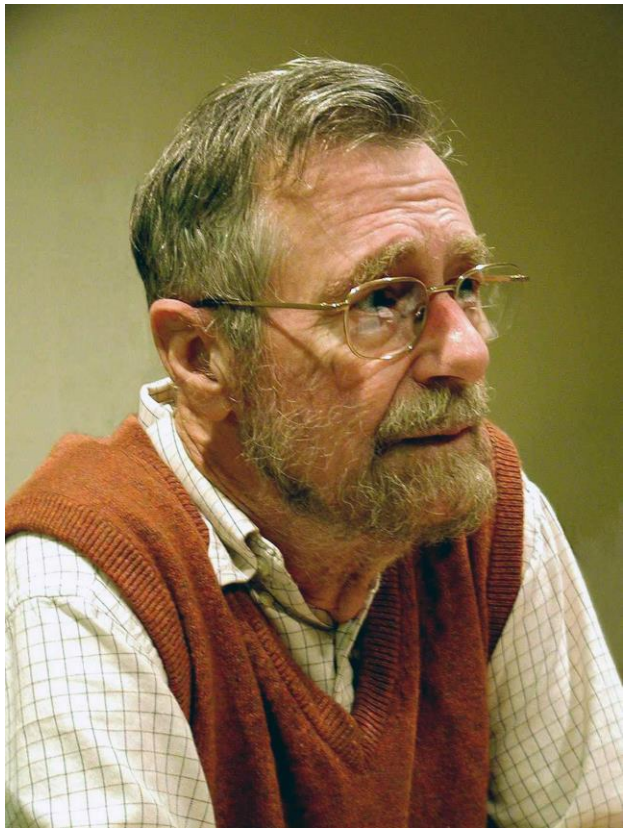


Wanted: % of people not using who want to use

Important!

- This refers to what individual developers want to use.
- It does not necessarily reflect what the ***market*** wants

Leaving OOP Behind



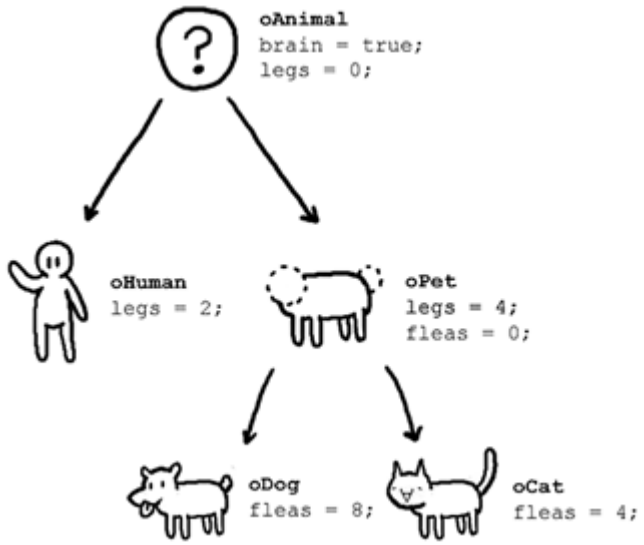
“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”

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

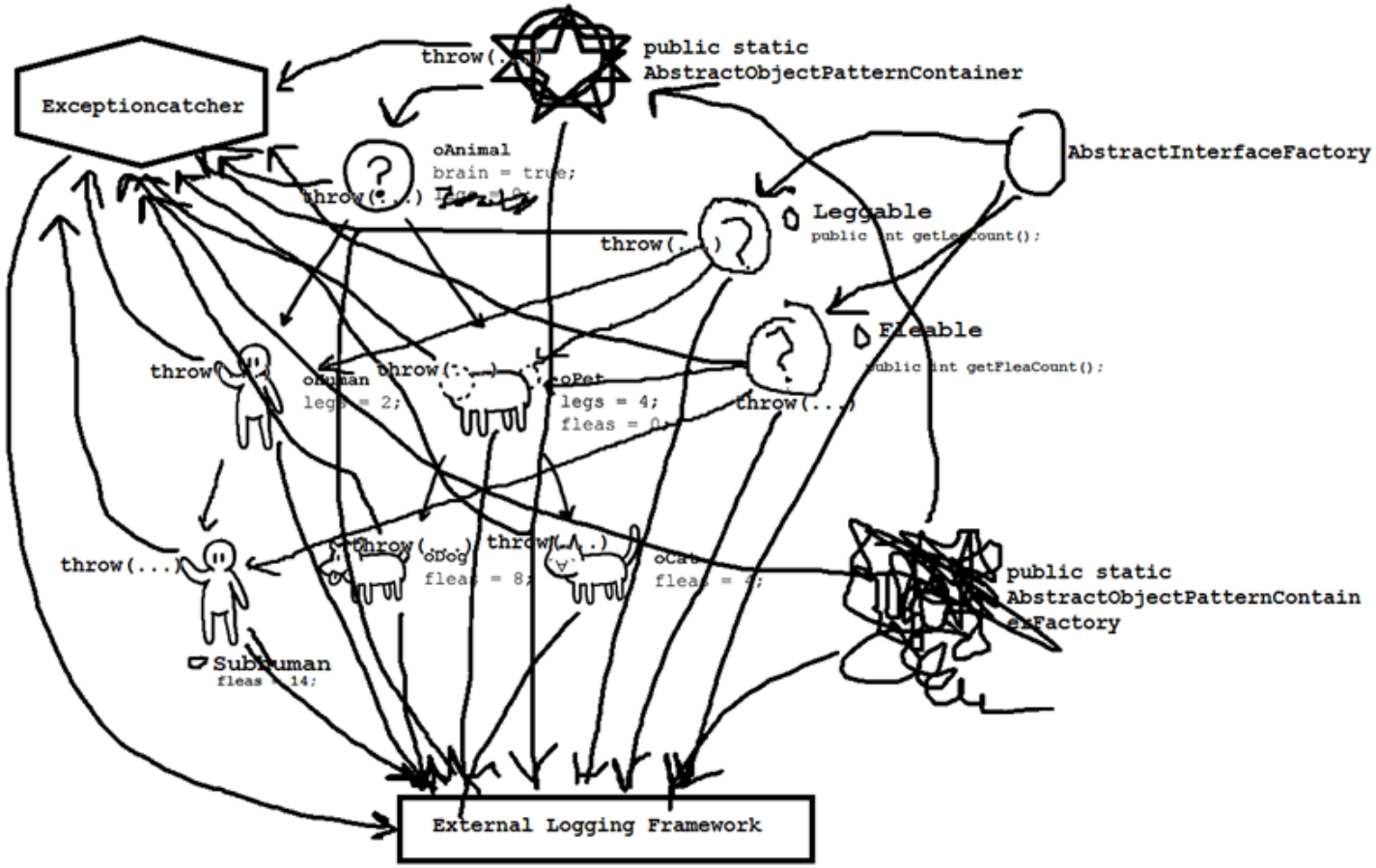
- Edsger Dijkstra

Leaving OOP Behind

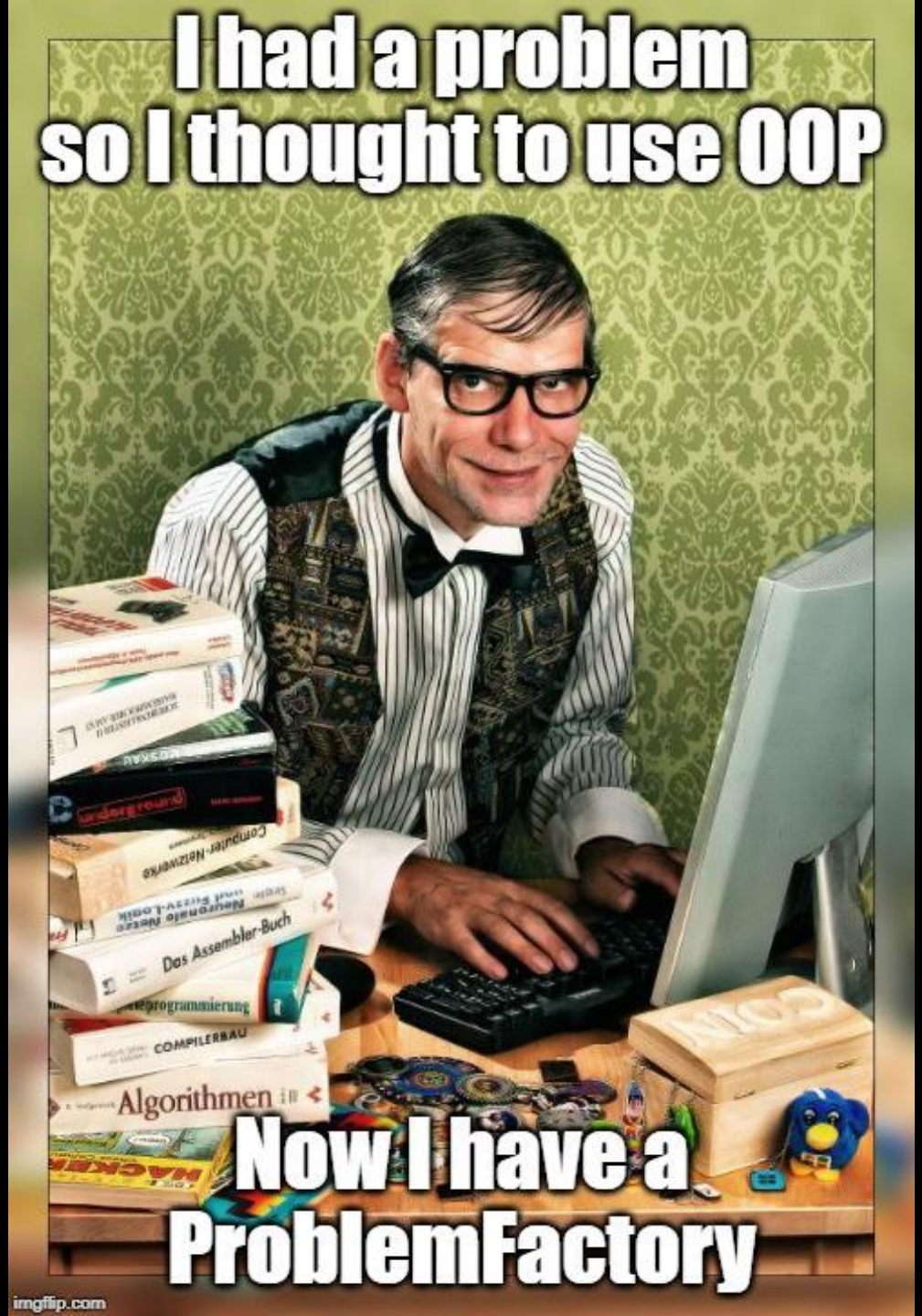
What OOP users claim:



What actually happens:



https://www.reddit.com/r/ProgrammerHumor/comments/418x95/theory_vs_reality/

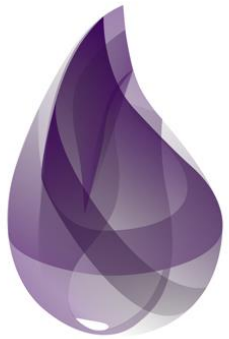


Alternatives to Imperative?

Two widely used paradigms:

Functional Programming:

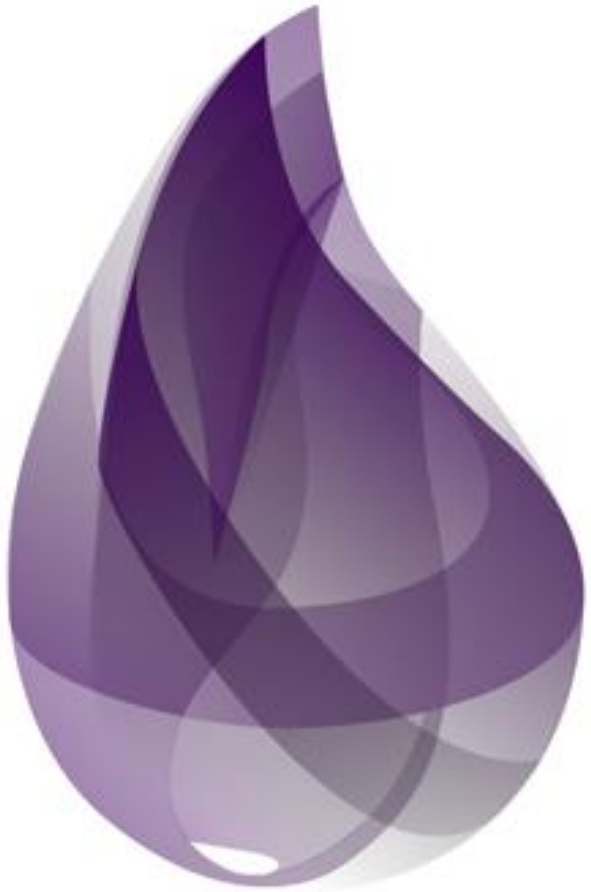
- Avoid changing state, avoid mutable data
- *Declarative* rather than *imperative*
- Tell the program *where* to go, not *how* to get there.



Object Oriented Programming:

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

Functional Programming



Declarative VS Imperative

Declarative programming paradigm:

- Style of building and structuring computer programs.
- Functional programming languages are characterized by a *declarative* style.
- Express the logic of a computation rather than explicit control flow. ?

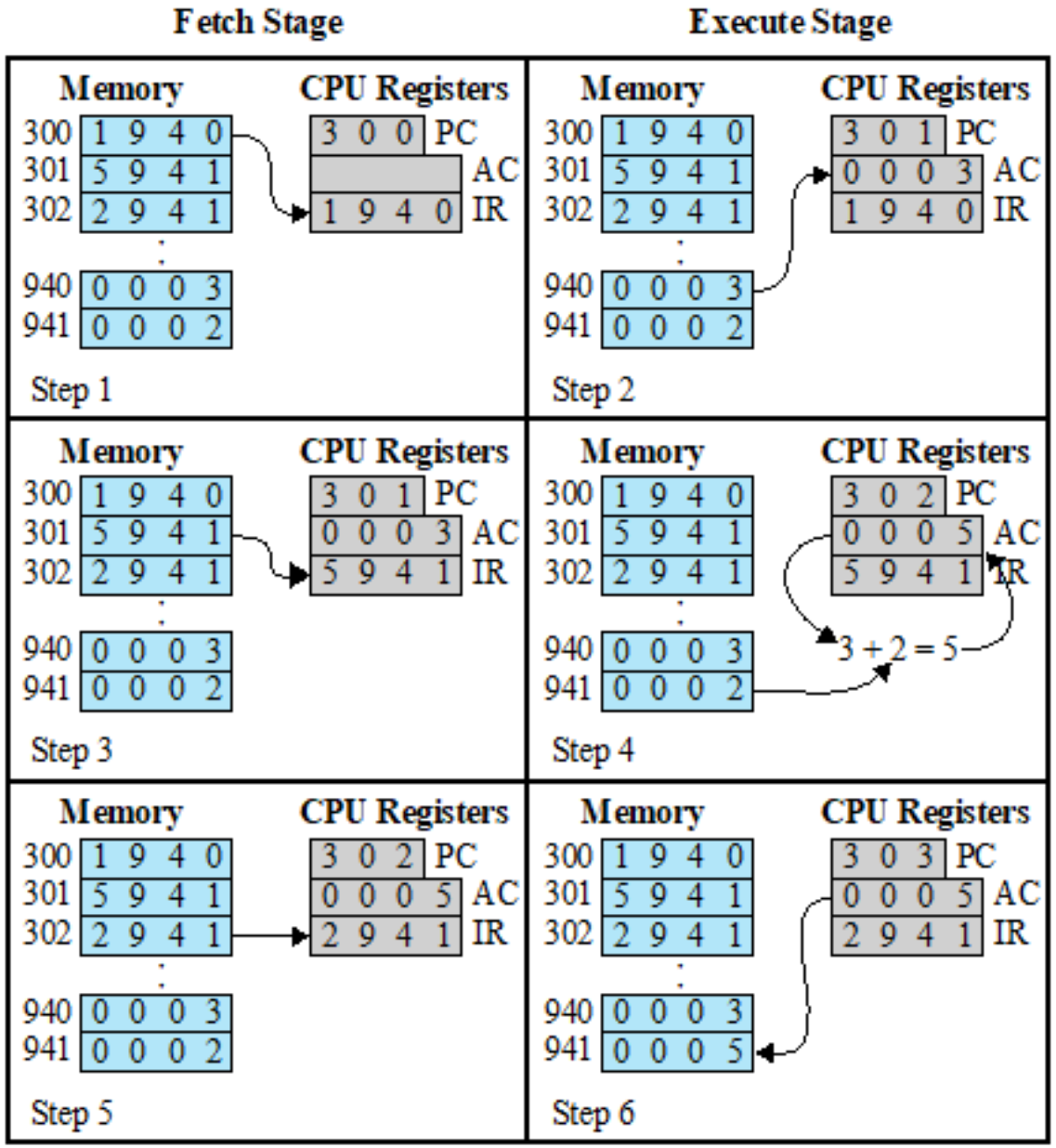
*The order in which individual statements, instructions or function calls of an **imperative** program are executed or evaluated.*

Emphasis on explicit control flow is one thing that separates ***imperative*** languages from ***declarative*** languages.

Control Flow

```
public static void main(String[] args)
{
    Scanner in = new Scanner(System.in);

    System.out.print("Enter a temperature: ");
    int temp = in.nextInt();

    if (temp >= 20)
        System.out.println("Warm outside!");
    else
        System.out.println("Cool outside!");
}
```

Determined using control structures in imperative languages.



CCPS 310/590 Example

At the machine instruction level, control flow works by altering the program counter.

Program counter tells the OS which instruction to fetch next.

Declarative VS Imperative

Imperative languages implement algorithms as a sequence of explicit steps (statements, control flow)

Declarative language syntax describes the *logic* of an algorithm

The declarative paradigm allows developers to worry about the ***what***, not the ***how***.

The how is left up to the language's implementation (compiler/interpreter)

Always Remember!

Machine code is ***imperative***.

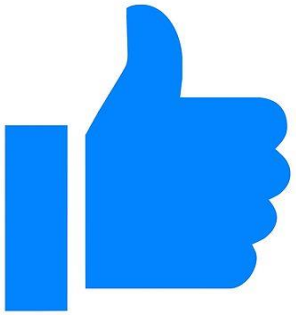
Functional programs compile into machine code, just like imperative ones.

The distinction is in what the programmer is **required to think about**, and what the language **hides behind the scenes.**

Declarative VS Imperative

The actual, practical difference between these two paradigms can be very hard to grasp.

How can we program without thinking about control flow?

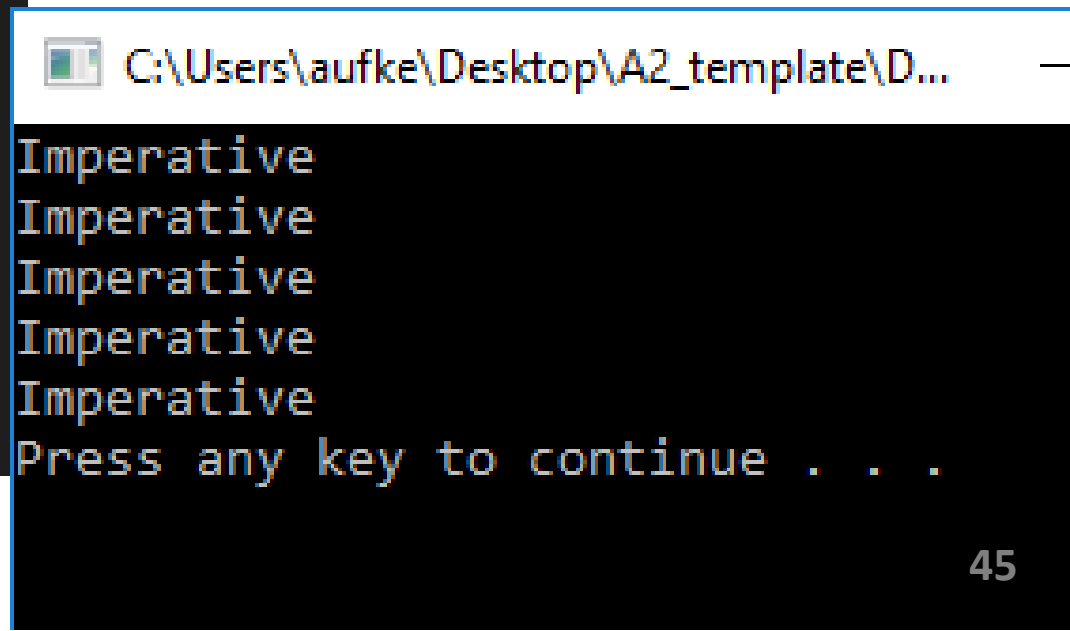


What makes a language declarative?
The fact that it's not imperative.

Faking it in C++

```cpp
#include <iostream>
using namespace std;

int main(void)
{
    for (int i = 0; i < 5; i++)
    {
        cout << "Imperative" << endl;
    }
}
```

- This C++ code is imperative
- We use a control structure (**for** loop) to tell the program to iterate.
- Even specify how this iteration is done.
- Initialization, condition, update



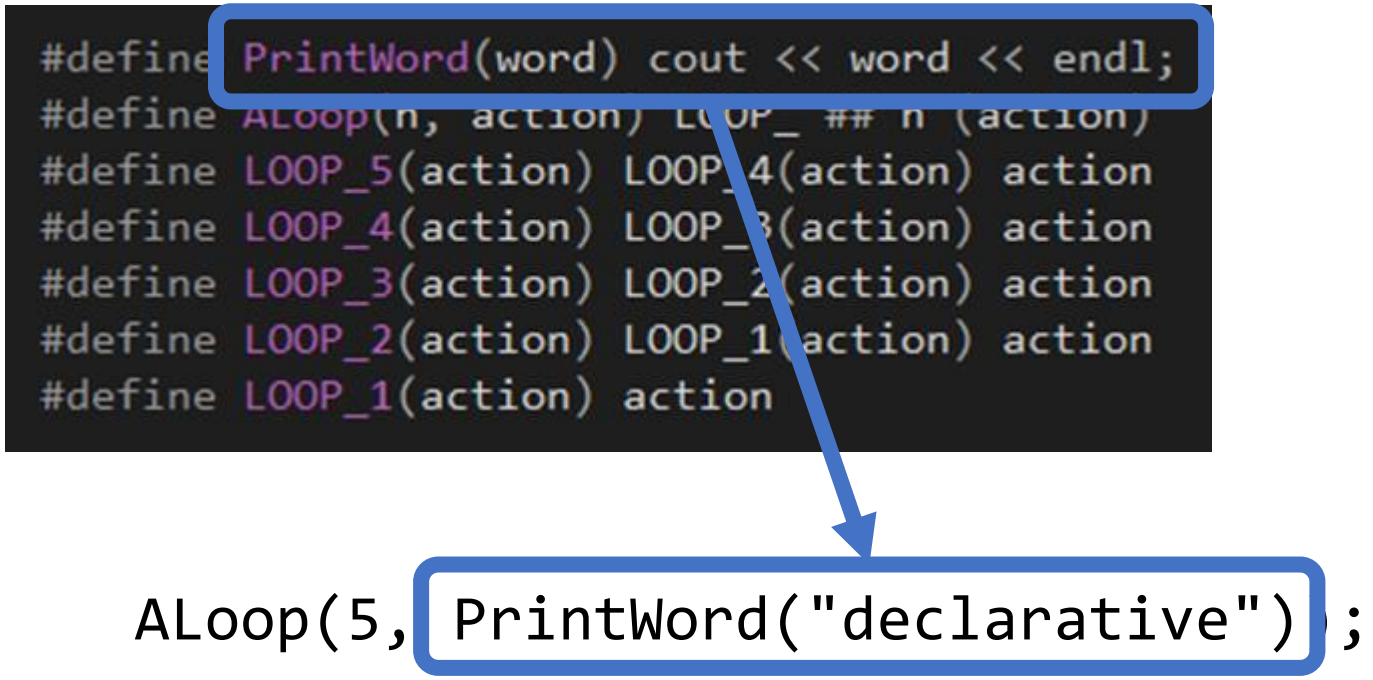
Faking it in C++

```
#include <iostream>
using namespace std;

#define PrintWord(word) cout << word << endl;
#define ALoop(n, action) LOOP_ ## n (action)
#define LOOP_5(action) LOOP_4(action) action
#define LOOP_4(action) LOOP_3(action) action
#define LOOP_3(action) LOOP_2(action) action
#define LOOP_2(action) LOOP_1(action) action
#define LOOP_1(action) action

int main(void)
{
    ALoop(5, PrintWord("declarative"));
}
```

- Simulate declarative programming using preprocessor directives.
- Directives here are basically a list of substitutions
- Assume that this is done by the programming language behind the scenes.

Faking it in C++

Directives here are basically a list of substitutions



```
#define PrintWord(word) cout << word << endl;
#define ALoop(n, action) LOOP_ ## n (action)
#define LOOP_5(action) LOOP_4(action) action
#define LOOP_4(action) LOOP_3(action) action
#define LOOP_3(action) LOOP_2(action) action
#define LOOP_2(action) LOOP_1(action) action
#define LOOP_1(action) action
```

Faking it in C++

Directives here are basically a list of substitutions

```
ALoop(5, PrintWord("declarative"));
```

↓

```
LOOP_5 (action)
LOOP_5 (PrintWord("declarative"))
LOOP_5 (cout << word << endl;)
```

```
action
```

↓

```
cout << word << endl;
```

```
#define PrintWord(word) cout << word << endl;
#define ALoop(n, action) LOOP_ ## n (action)
#define LOOP_5(action) LOOP_4(action) action
#define LOOP_4(action) LOOP_3(action) action
#define LOOP_3(action) LOOP_2(action) action
#define LOOP_2(action) LOOP_1(action) action
#define LOOP_1(action) action
```

Faking it in C++

Directives here are basically a list of substitutions

```
ALoop(5, PrintWord("declarative"));
```

```
LOOP_5 (action)
LOOP_4 (action) action
LOOP_3 (action) action action
LOOP_2 (action) action action action
LOOP_1 (action) action action action action
action action action action action
```

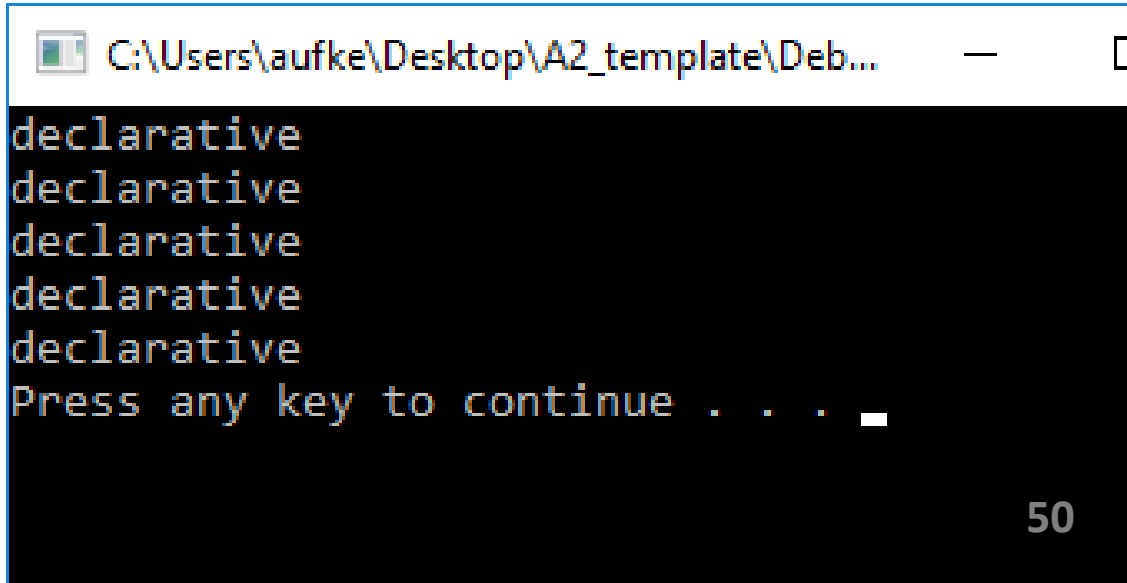
```
action
```

```
cout << word << endl;
```

Faking it in C++

```
#include <iostream>
using namespace std;

#define PrintWord(word) cout << word << endl;
#define ALoop(n, action) LOOP_ ## n (action)
#define LOOP_5(action) LOOP_4(action) action
#define LOOP_4(action) LOOP_3(action) action
#define LOOP_3(action) LOOP_2(action) action
#define LOOP_2(action) LOOP_1(action) action
#define LOOP_1(action) action

int main(void)
{
    ALoop(5, PrintWord("declarative"));
}
```

Imagine this was all done behind the scenes by the implementation of the programming language



Declarative VS Imperative

```
for (int i = 0; i < 5; i++)
{
    cout << "Imperative" << endl;
}
```

```
ALoop(5, PrintWord("declarative"));
```

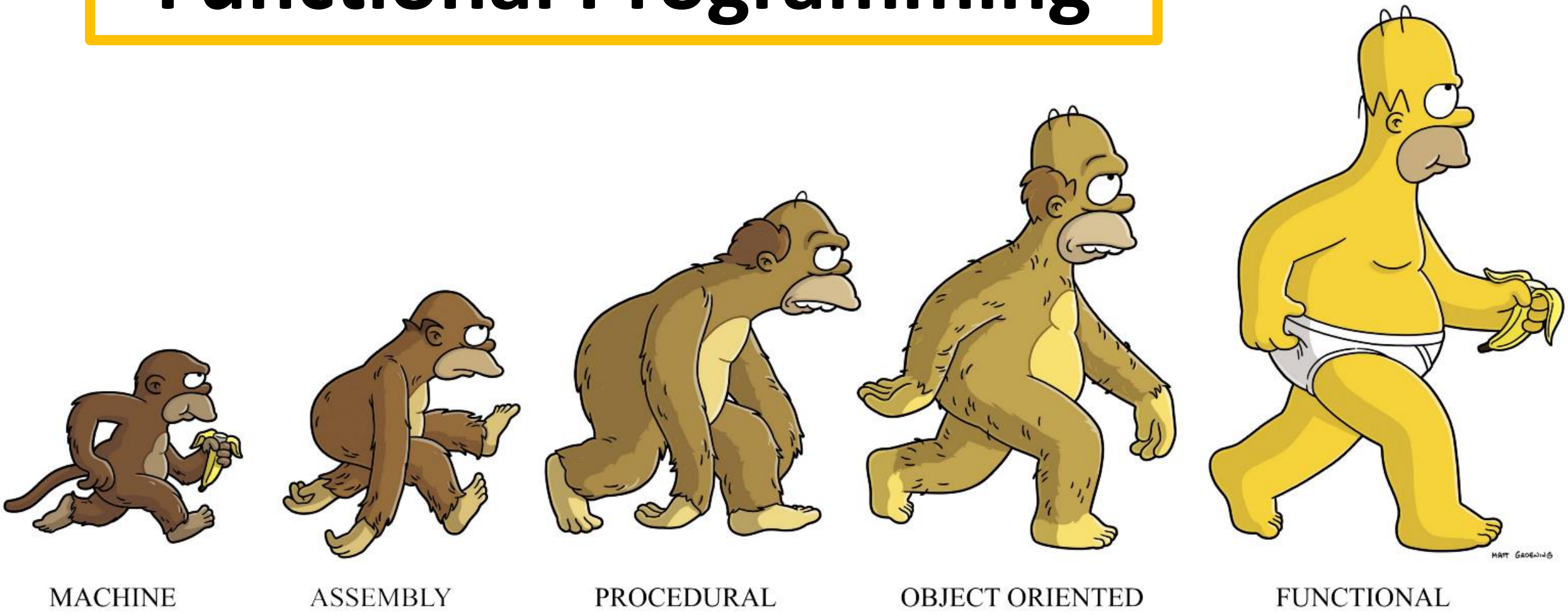
Imperative:

- Programmer specifies control flow
- Each loop iteration explicitly defined

Declarative (*C++ fakery*):

- Programmer says what they want
- Nevermind how it's done

Functional Programming

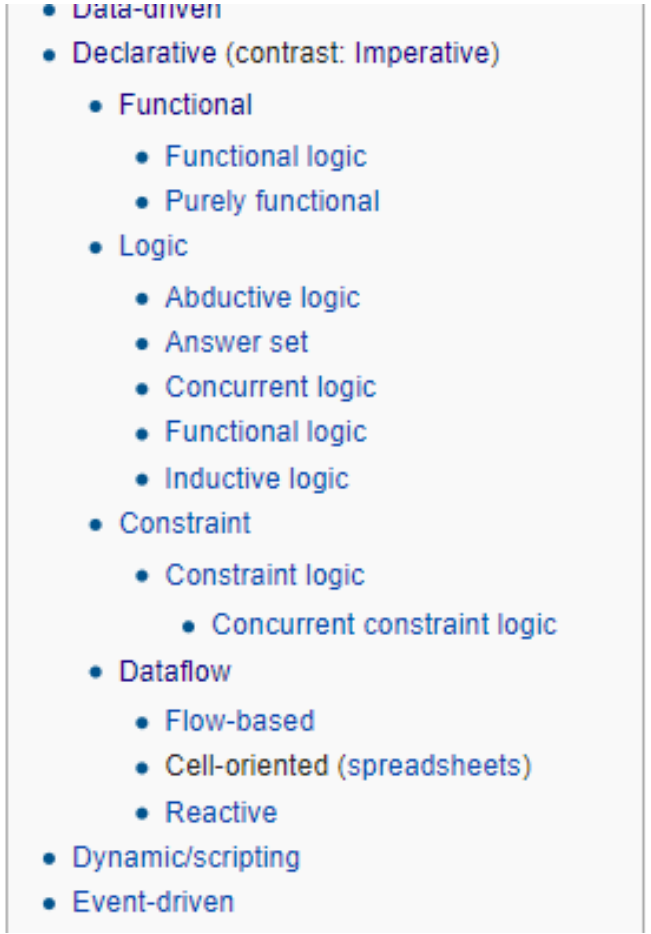


Depends what you're doing, depends who you ask…

Functional Programming

Functional programming languages are characterized by a **declarative** style.

Functional are not the only declarative languages:



Always Remember!

The line between imperative/OOP and functional programming is grey.

Code can be written in a functional *style* using a language not specifically designed for functional programming.

Some languages are designed to be functional, but still contain imperative elements.

Functional Language Characteristics

Things that are generally foreign to imperative programming:

- Avoids changing global state, no state to reason about.
- Avoid global variables, keep scope as tight/local as possible

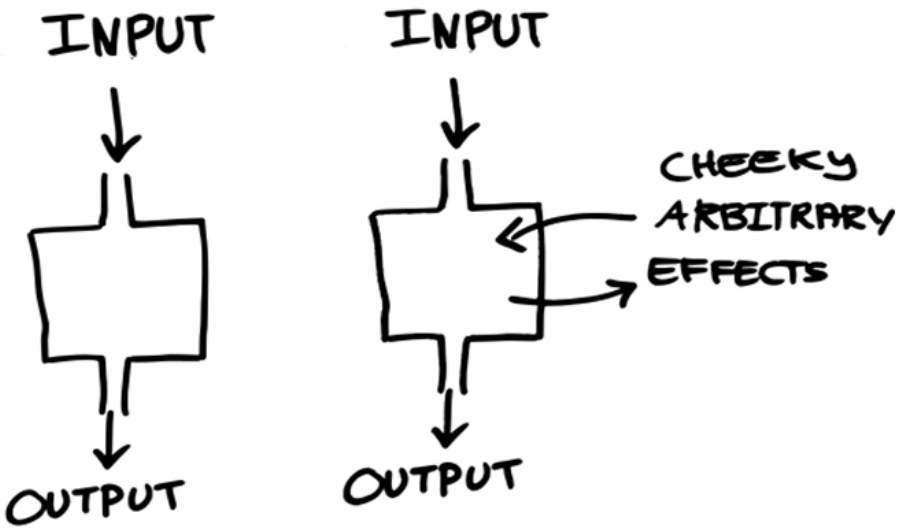
```
for (int i = 0; i < SIZE; i++)
{
    /* Program code here */

    // 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");
}
```

No *side effects*!

Side Effects?

Things that are generally foreign to imperative programming:

A function can be said to have a side effect if it has an observable interaction with the outside world aside from returning a value.



- Modify global variable
- Raise an exception
- Write data to display or file

Side Effects

Function/method output can depend on history (or current state):

```
public class SideEffect
{
    private static int n = 0;

    public static int retNum(int number) {
        return (n += number);
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < 5; i++)
            System.out.println(retNum(1));
    }
}
```

- Call the same function, with the same argument, 5 times.
- Different result each time.
- Common in imperative languages.
- Rare in functional languages.

Side Effects

Declarative/functional languages avoid side effects

- The output of a function depends ***solely*** on the input arguments
  - No side effects, no dependence on global or local state.
- This makes it much easier to predict the behavior of a program
  - Primary motivation for developing functional programming
- With no state to be concerned of, parallel processing becomes much easier. No race conditions!
  - Functions can be spawned as separate threads/processes

Functional Language Characteristics

Things that are generally foreign to imperative programming:

Pure functions are emphasized/enforced:

- Pure function? A function without side effects
- If the return value of a pure function is not used, the function can be safely removed.
- Output depends solely on input (***referential transparency***).
- Pure functions without a data dependency can be executed in any order. Safe to parallelize (*thread-safe*).

Quick Note

- In practice it's unreasonable to have a programming language containing only pure functions.
- This would preclude things like file I/O and user input.
- Common to have a pure function "core" surrounded by impure functions that interact with the outside world
- This is true of Elixir, but depends on the language.
- Pure functions can be written in any language, but functional languages enforce them in various ways.

Functional Language Characteristics

Functions and recursion are **central**:

Flow control accomplished with functions calls.

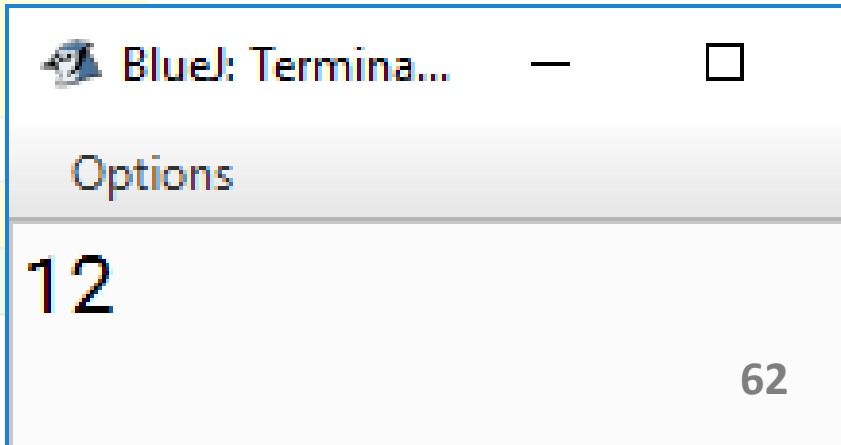
- We already saw in Smalltalk how this is possible
- Much lower focus on loop/if-else/case constructs.
- Collections are operated upon using recursion.

Specifically, ***tail*** recursion in Elixir. Tail recursion is recognized and optimized by the compiler into iterative machine code. ***Tail Call Optimization***.

Recursion

```
public class Recursion
{
    // Assume args > 0
    public static int mult(int n1, int n2) {
        if (n2 == 0)
            return 0;
        return n1 + mult(n1, n2-1);
    }

    public static void main(String[] args)
    {
        System.out.println(mult(3,4));
    }
}
```

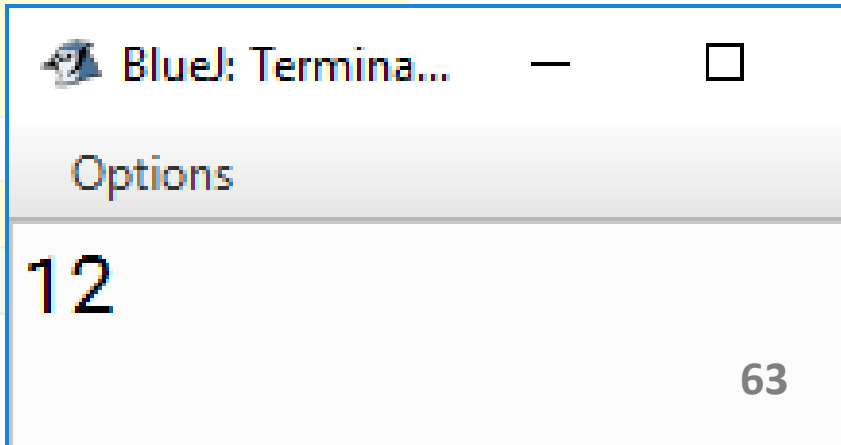
```
mult(3, 4)
3+mult(3, 3)
3+(3+mult(3, 2))
3+(3+(3+mult(3, 1)))
3+(3+(3+(3+mult(3, 0))))
3+(3+(3+(3+0)))
12
```



Tail Recursion

```
public class Recursion
{
    // Assume args > 0
    public static int tail_mult(int n1, int n2, int t
        if (n2 == 0)
            return total;
        return tail_mult(n1, n2-1, total+n1);
    }

    public static void main(String[] args)
    {
        System.out.println(tail_mult(3,4,0));
    }
}
```

```
tail_mult(3, 4, 0)
tail_mult(3, 3, 3)
tail_mult(3, 2, 6)
tail_mult(3, 1, 9)
tail_mult(3, 0, 12)
12
```



Tail Recursion

```
mult(3, 4)
3 + mult(3, 3)
3 + (3 + mult(3, 2))
3 + (3 + (3 + mult(3, 1)))
3 + (3 + (3 + (3 + mult(3, 0))))
3 + (3 + (3 + (3 + 0)))
12
```

Every recursive call must complete before we even begin adding values

```
tail_mult(3, 4, 0)
tail_mult(3, 3, 3)
tail_mult(3, 2, 6)
tail_mult(3, 1, 9)
tail_mult(3, 0, 12)
12
```

Here, total is updated each call. This version looks a lot more like iteration. Optimizable.

Functional Language Characteristics

Things that are generally foreign to imperative programming:

First class functions and higher order functions:

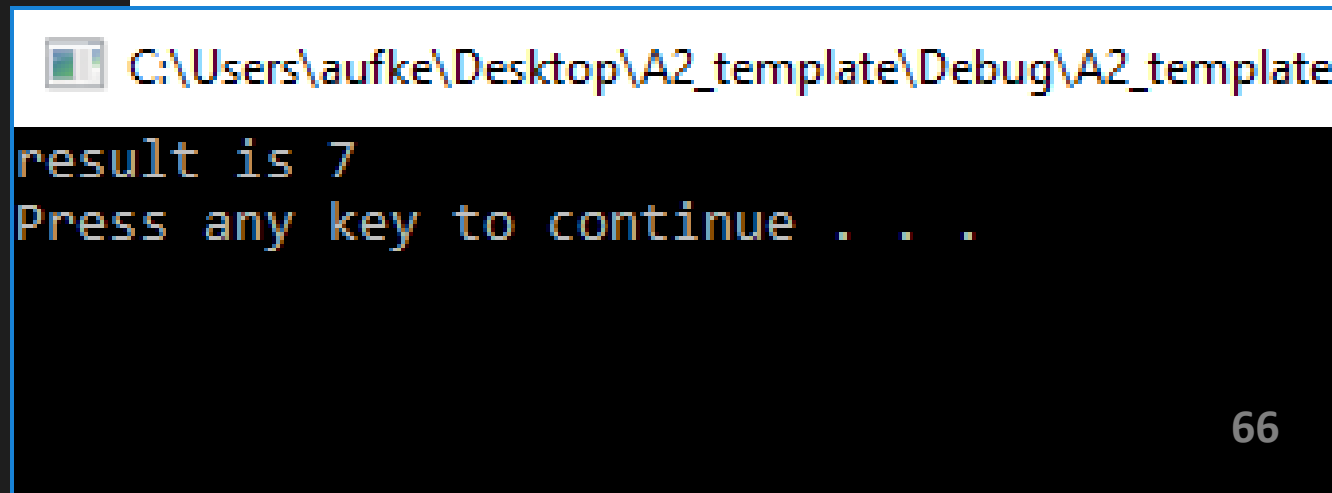
- Functions that return functions or accept them as arguments
  - I.e., differential operator. Derivative of function is a function.
- "First class" describes programming language entities that have no restriction on their use.
- I.e., first class functions can appear anywhere in the program that other first-class entities (such as numbers) can.
  - Functions as arguments & return values, function literals, etc.

Always Remember…

```c
#include <stdio.h>

int addInt(int n, int m)
{
    return n + m;
}

void printInt(int(*x)(int, int), int a, int b)
{
    printf("result is %d \n", x(a, b));
}

int main(void)
{
    int(*fPtr)(int, int);
    fPtr = &addInt;

    printInt(fPtr, 3, 4);
}
```

The line between imperative and functional programming is grey.

C supports passing functions as arguments via function pointers.



Functional Language Characteristics

Things that are generally foreign to imperative programming:

Strict (eager) VS. non-strict (lazy) evaluation:

- **Strict:** Always evaluate function arguments before invoking the function.
- **Lazy:** Evaluates arguments if their value is required to invoke the function.

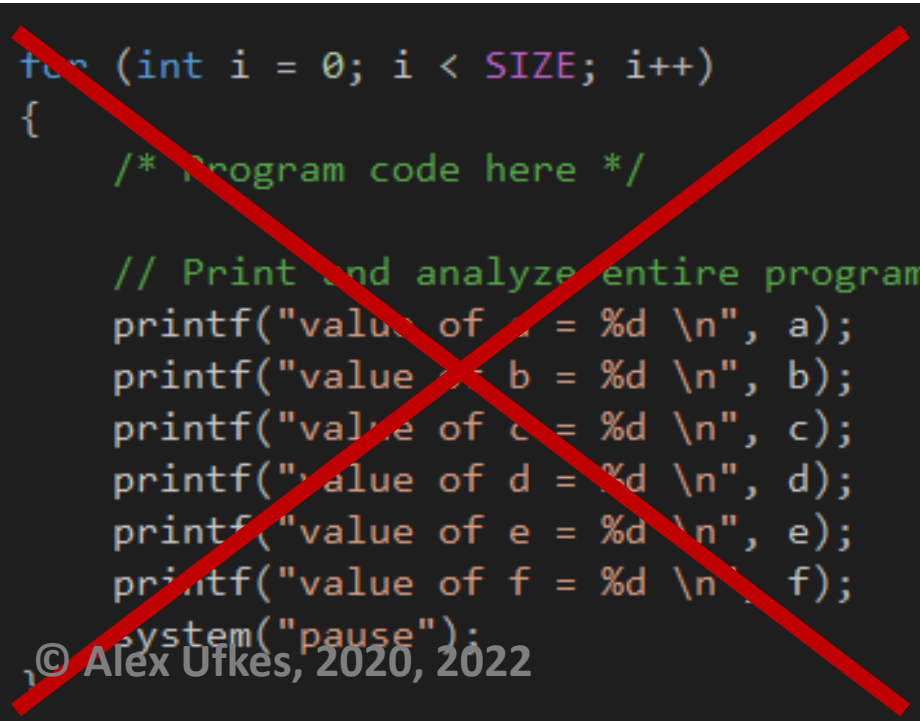
```
print length( [2+1, 3*2, 1/0, 5-4] );
```

- Fails under strict evaluation, can't divide by zero.
- Under lazy evaluation we get the correct value of 4. We don't need to know the actual values of the array elements to know how many there are.

Functional Programming: Advantages

Easier to reason about pure functions:

- If the function is internally consistent, it is *always* correct.
- No tracking down global variables, tracing pointers/references, etc.



Functional Programming: Advantages

Concurrent programming is easier:

- No side effects, functions can be spawned as processes/threads.
- There is no state to be shared between different threads.
- No need for semaphores (or similar) if you don't have side effects!
  - Pure functions never access or modify things outside their scope
  - No such thing as a race condition when values are immutable.

Functional Programming: Advantages

Programs are easier to understand:

```
for (int i = 0; i < 5; i++)
{
  cout << "Imperative" << endl;
}
```

- Allocate space for variable `i`
- Initialize `i` to 0
- Iterate as long as `i` is less than 5
- Increment `i` after each iteration

```
ALoop(5, PrintWord("declarative"));
```

- Do `something` 5 times

Functional Programming: Disadvantages

Recursion can cause memory use to explode:

- Operating on a list with 10000 items requires 10000 recursive calls. Stack explodes
- Tail recursion mitigates this but using tail recursion can often require inelegant code gymnastics.

```
public static int tail_mult(int n1, int n2, int total) {
    if (n2 == 0)
        return total;
    return tail_mult(n1, n2-1, total+n1);
}
```

Functional Programming: Disadvantages

Recursion can cause memory use to explode:

- Operating on a list with 10000 items requires 10000 recursive calls. Stack explodes
- Tail recursion mitigates this as we saw, but using tail recursion can often require inelegant code gymnastics.

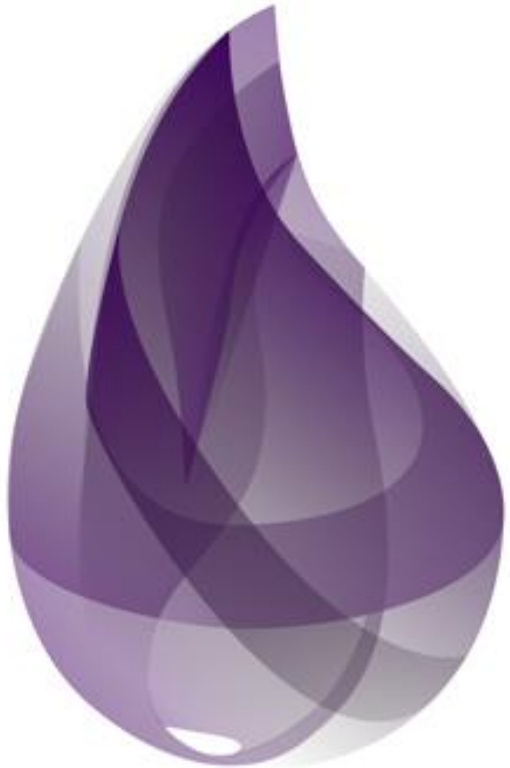
No assignment statements, data is immutable:

- Performing actions requires allocating new memory.
- Remember strings in Java – Changing the value of a string creates a new string object with the new value.
- ***Garbage collection very important!***

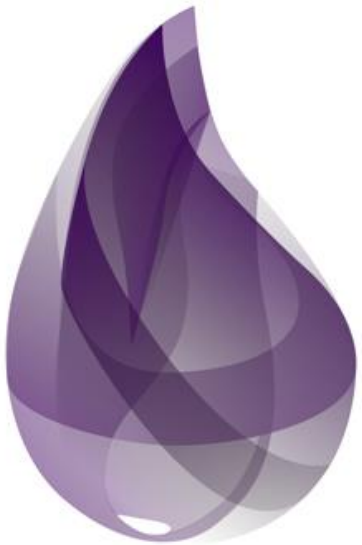
When FP?

- LISP - 1957 - first-class functions
- APL - 1962 - no globals
- ML - 1973 - Hindley-Milner type inference
- Hope*! - early 1970s - call-by-pattern, algebraic data types
- Miranda* - 1985 - proprietary
- Haskell* - 1990

Only predated by FORTRAN

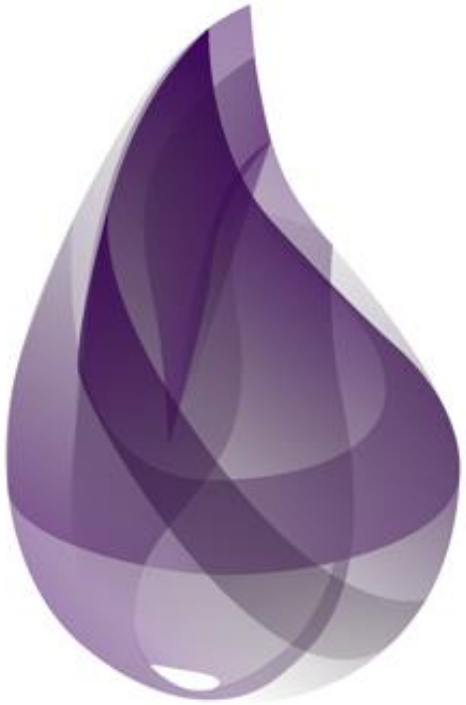


elixir



History: Erlang

- Proprietary language used at Ericsson, developed by Joe Armstrong
- Initially implemented in Prolog at Ericsson
- By 1988, it had been proven suitable for prototyping telephone exchanges but…
- Prolog interpreter was much too slow, needed to be 40x faster.
- In 1992 work began on BEAM VM
  - Compiles Erlang to C
  - Balance performance and disk space.
- Went from lab to real applications by 1995
- In 1998, Ericsson banned internal use of Erlang, causing Armstrong to quit.
  - Rehired in 2004, after ban was lifted.



elixir

History: Elixir

- Builds on Erlang, runs on BEAM VM
- Erlang was prolog-like, Elixir is more conventional.
- First appeared in 2011
- Developed by Jose Valim as an R&D project at Plataformatec (consulting firm)
- Used at Pinterest, and for web development by Discord.

Elixir: Overview

- Elixir is a functional programming language
  - Mostly immutable, rich support for concurrency
- Everything is an expression.
  - Everything evaluates to ***something***.
- Elixir compiles into Erlang bytecode.
  - Thus, Erlang functions can be called from Elixir
- Emphasizes recursion and higher-order functions
  - As opposed to side-effect-based looping

Elixir: Processes

- Elixir code runs inside lightweight threads of execution.
  - Isolated, exchange information via message passing.
- Not uncommon to have hundreds of thousands of processes running *concurrently* in same VM.
  - Note: These are NOT *operating system* processes!
  - Extremely lightweight in terms of CPU and memory
  - A process need not be an expensive resource

Installing Elixir

https://elixir-lang.org/

elixir

HOME INSTALL GUIDES LEARNING DOCS DEVELOPMENT BLOG PACKAGES

```
defprotocol String.Inspect
  only: [BitString, List,

defimpl String.Inspect, fo
  def inspect(false), do:
  def inspect(true),  do:
  def inspect(nil),   do:
  def inspect(:""),   do:

  def inspect(atom) do
```

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

To learn more about Elixir, check our getting started guide and our learning page for other resources. Or keep reading to get an overview of the platform, language and tools.

News: Elixir v1.6 released

Search...

ElixirConf

ElixirConf™ US is being held in Bellevue, WA, September 4-7, 2018.
ElixirConf EU is being held in Warsaw, Poland, April 16-18, 2018.

elixir

HOME INSTALL GUIDES LEARNING DOC

Installing Elixir

Windows

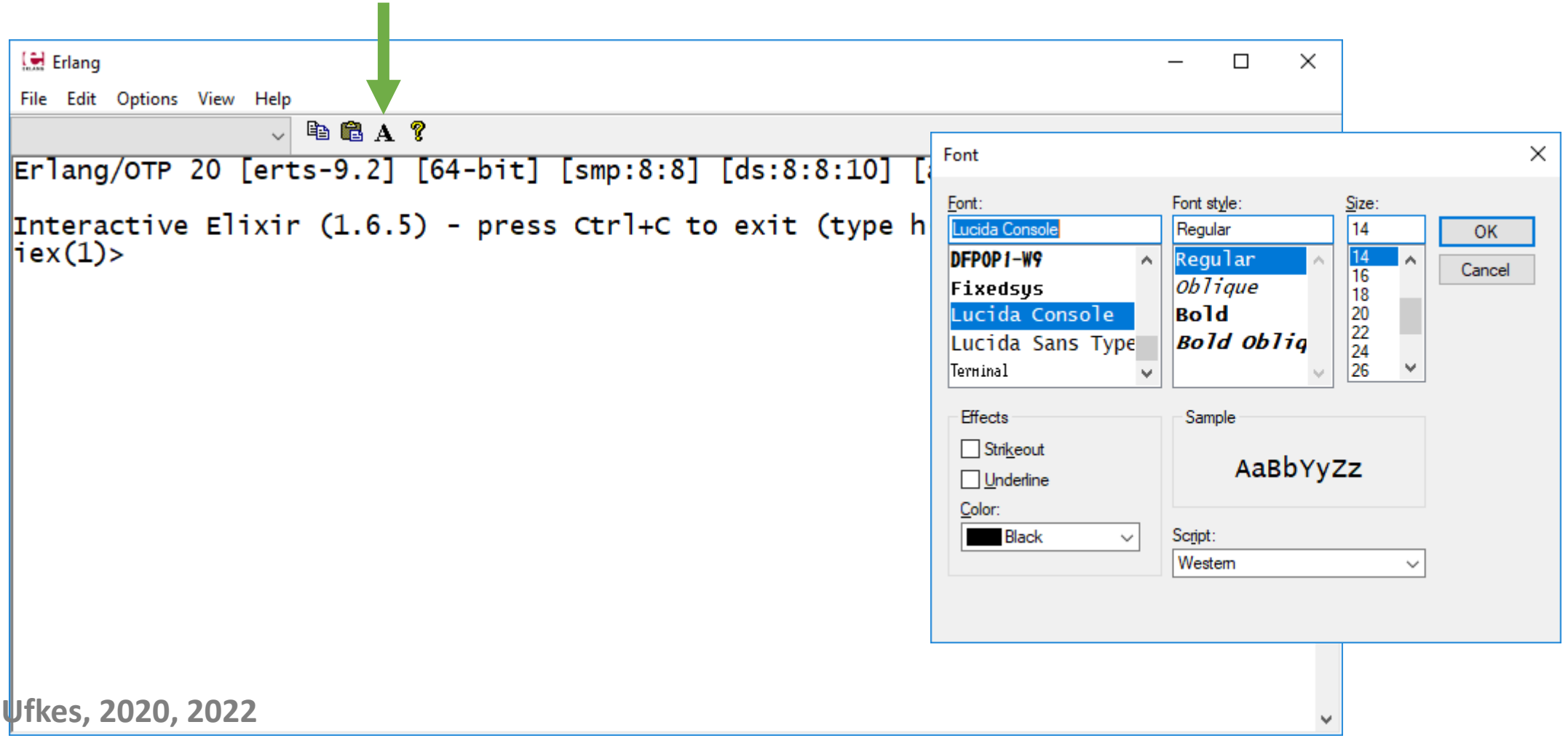
- Web installer
  - Download the installer
  - Click next, next, ..., finish

Mac OS X

- Homebrew
  - Update your homebrew to latest: `brew update`
  - Run: `brew install elixir`
- Macports
  - Run: `sudo port install elixir`

Erlang Shell

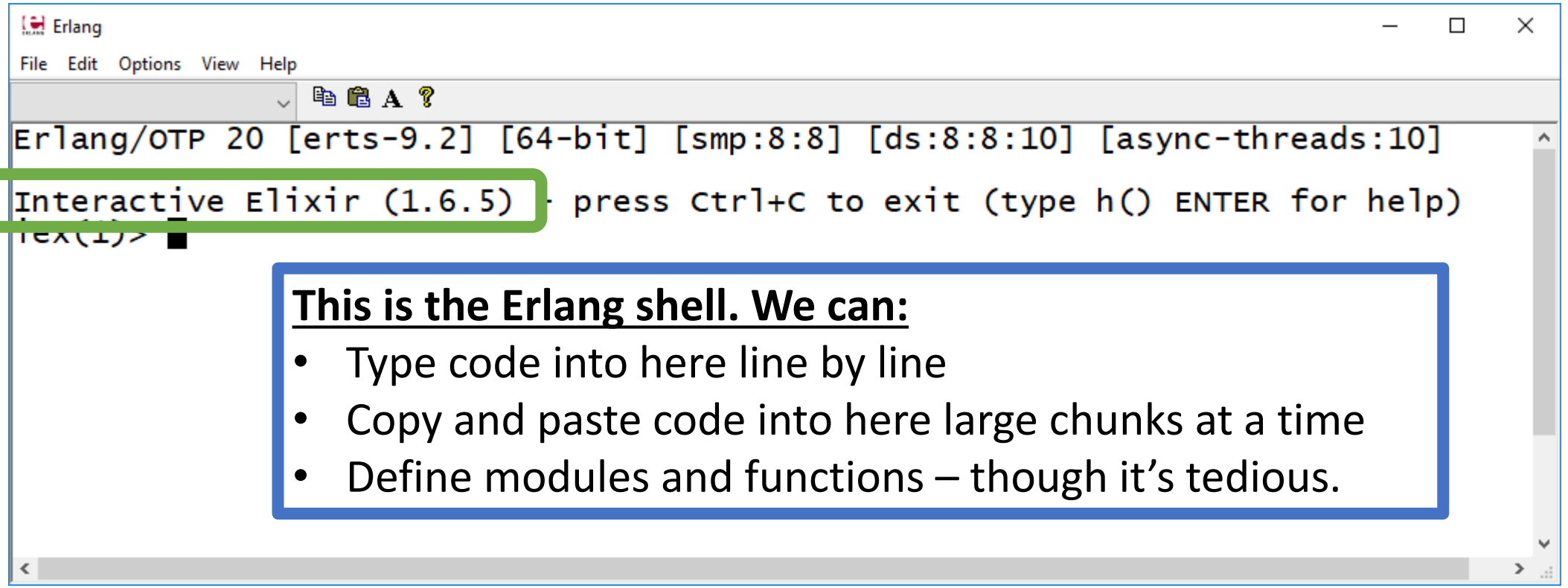
Better than just a terminal window



Writing and Compiling Elixir

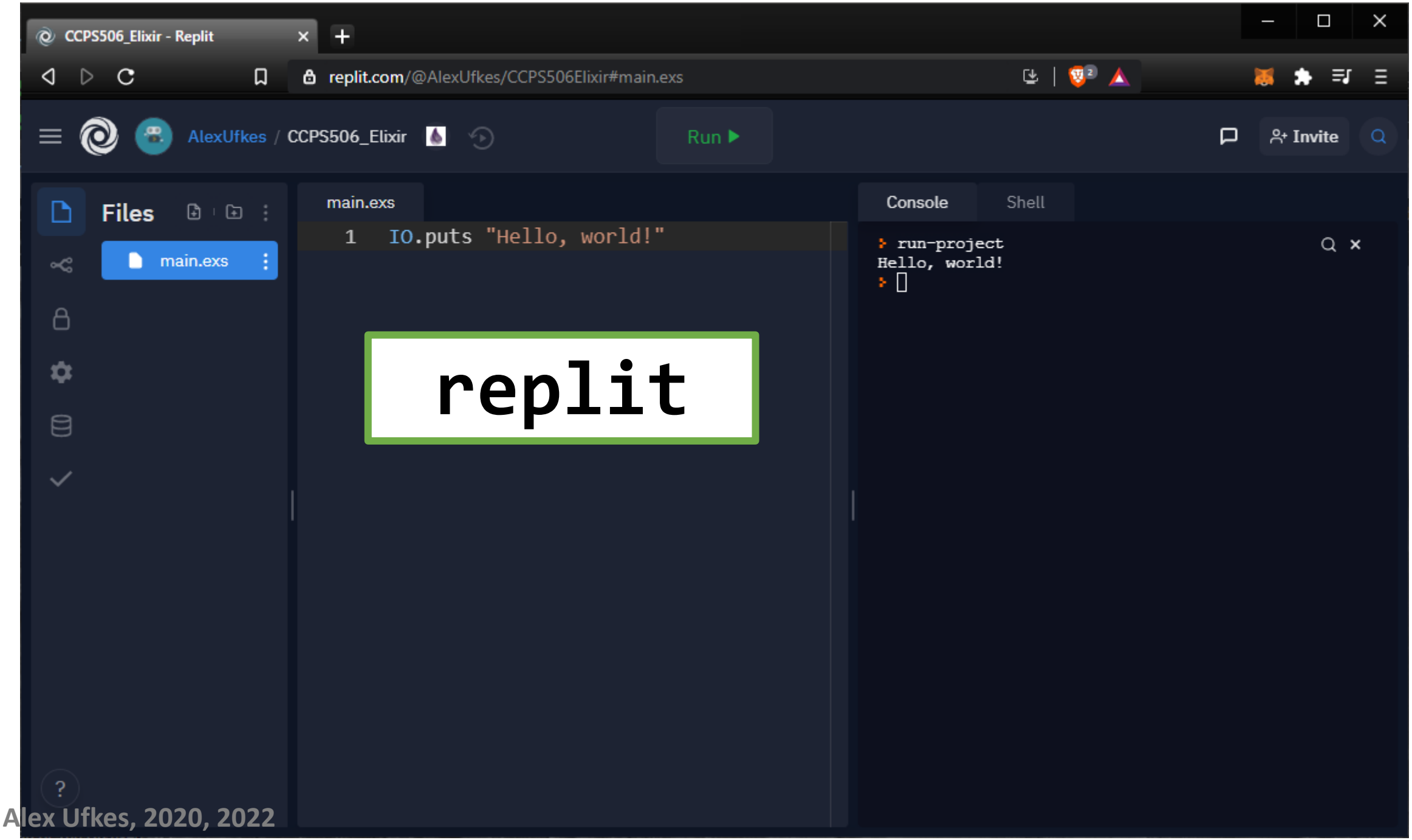
Play around in the interactive shell or do things from the command line.

IDEs exist, but you're on your own. I won't help troubleshoot IDE-related problems.



This is the Erlang shell. We can:

- Type code into here line by line
- Copy and paste code into here large chunks at a time
- Define modules and functions – though it's tedious.



Elixir References

https://media.pragprog.com/titles/elixir/ElixirCheat.pdf

https://elixir-lang.org/getting-started/introduction.html

https://hexdocs.pm/elixir/master/api-reference.html#content

Summary

- Double dispatch
- Smalltalk conclusion
- Functional paradigm
- Getting started with Elixir