COMING UP TODAY

x86 Assembly

```
mov ecx, 1
xor eax, eax
jmp .check
```

First, a little preview of what's coming up. We'll be seeing Assembly, C, Lambda Calculus, Haskell, JavaScript, Ramda, and a whole lot more.

👩‍🎓 Much of today will be *academic / context / jargon.*

👩‍💻 Later parts will be `practical / required / curricular.`

★

we will mark those parts like this!

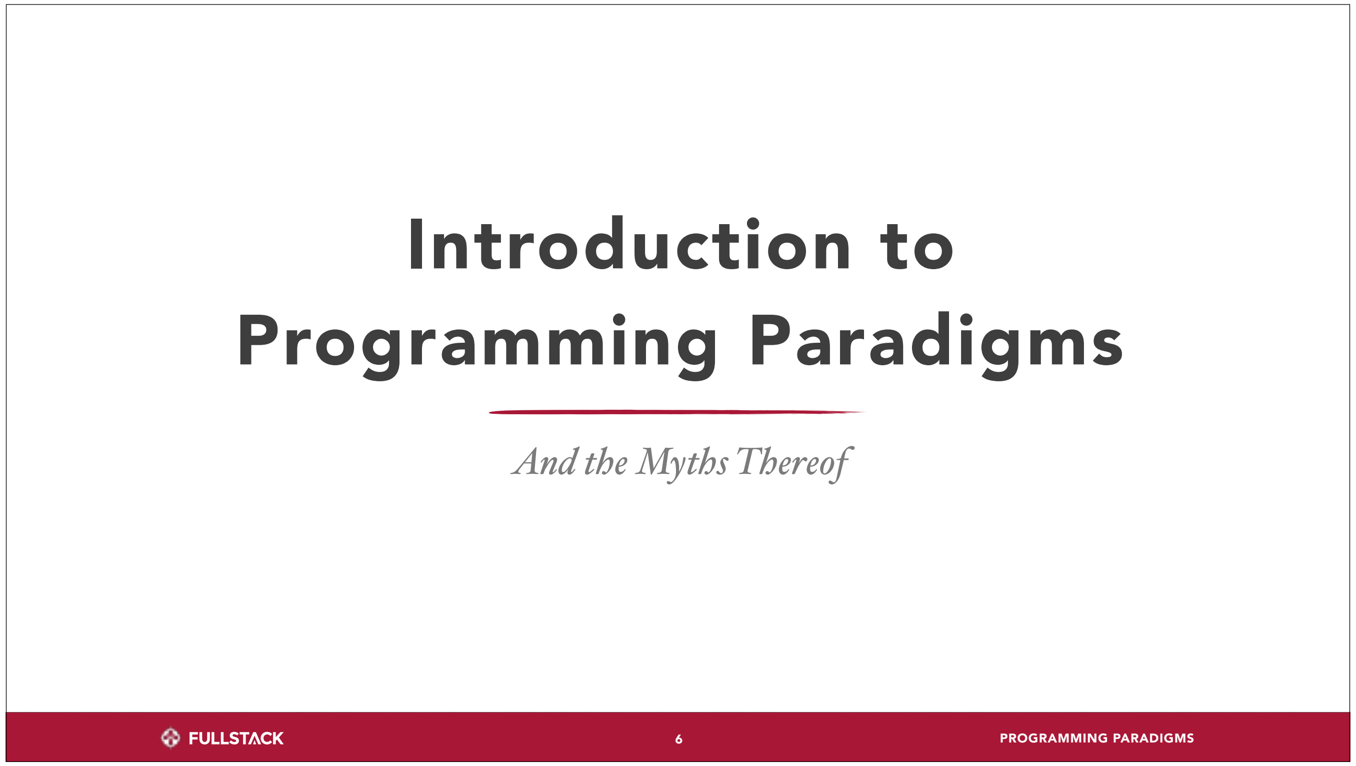
🚀 Trajectory

stuff for context / fun

- 🍱 **Programming Paradigms**
  - Imperative (📚 Assembly)
  - Declarative
- λ **Concepts of FP**
  - Features (📚 Haskell)
  - History (📚 Lambda Calculus)
- 👩‍💻 **FP Fundamentals in JS**
  - Pure Functions (💻 Jamda)
  - Function Composition
  - Currying & Partial Application (💻 Pointfree)
  - Immutability (💻 Immutable List)

stuff we want you to start learning

Welcome to the Functional Programming *Power Monday*! This first lecture in the series begins not with functional programming per se, but a bit of background context.



Introduction to Programming Paradigms

And the Myths Thereof

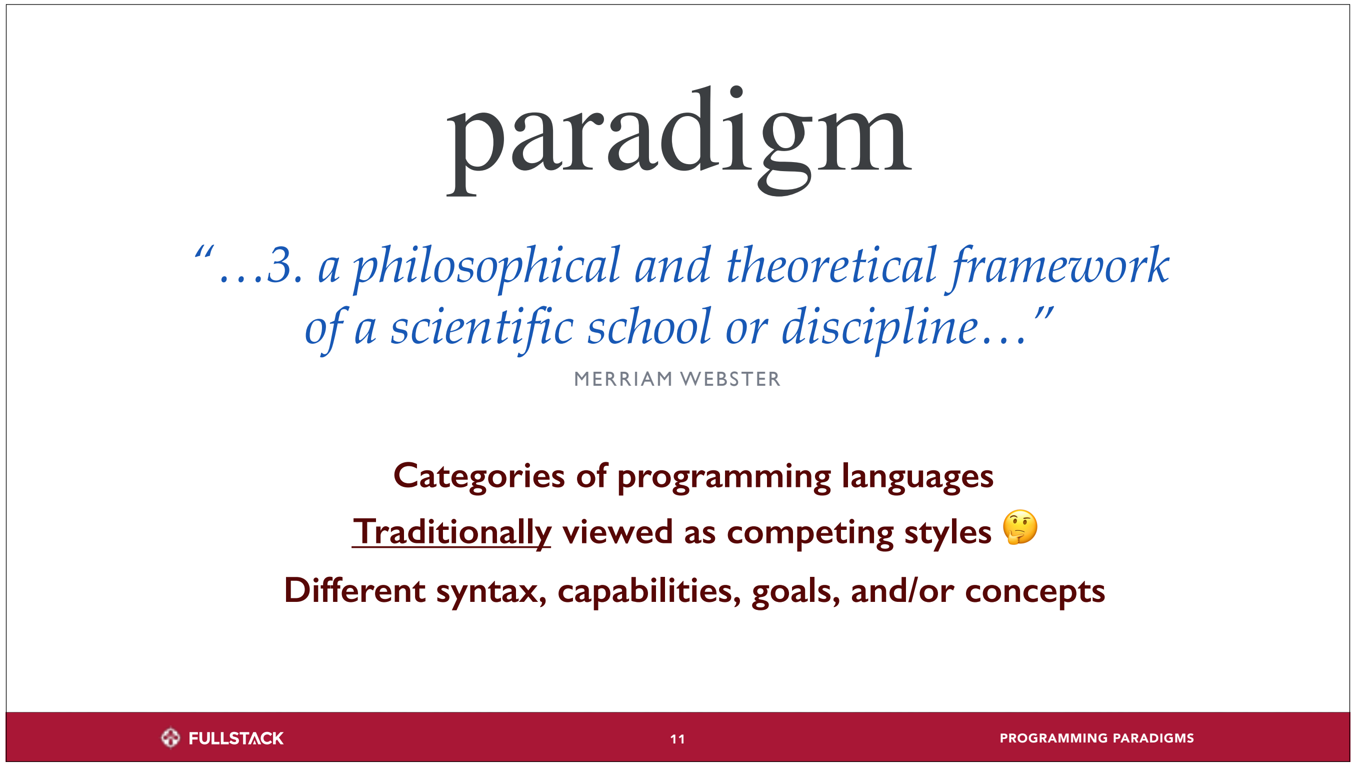
Today we're going to begin by discussing broad categories or styles of programming languages, called paradigms.

You may have heard terms like this…

Functional!
Object Oriented!
Procedural!
Declarative!

Perhaps you felt like this…

What do these words even mean?



paradigm

"…3. a philosophical and theoretical framework of a scientific school or discipline…"

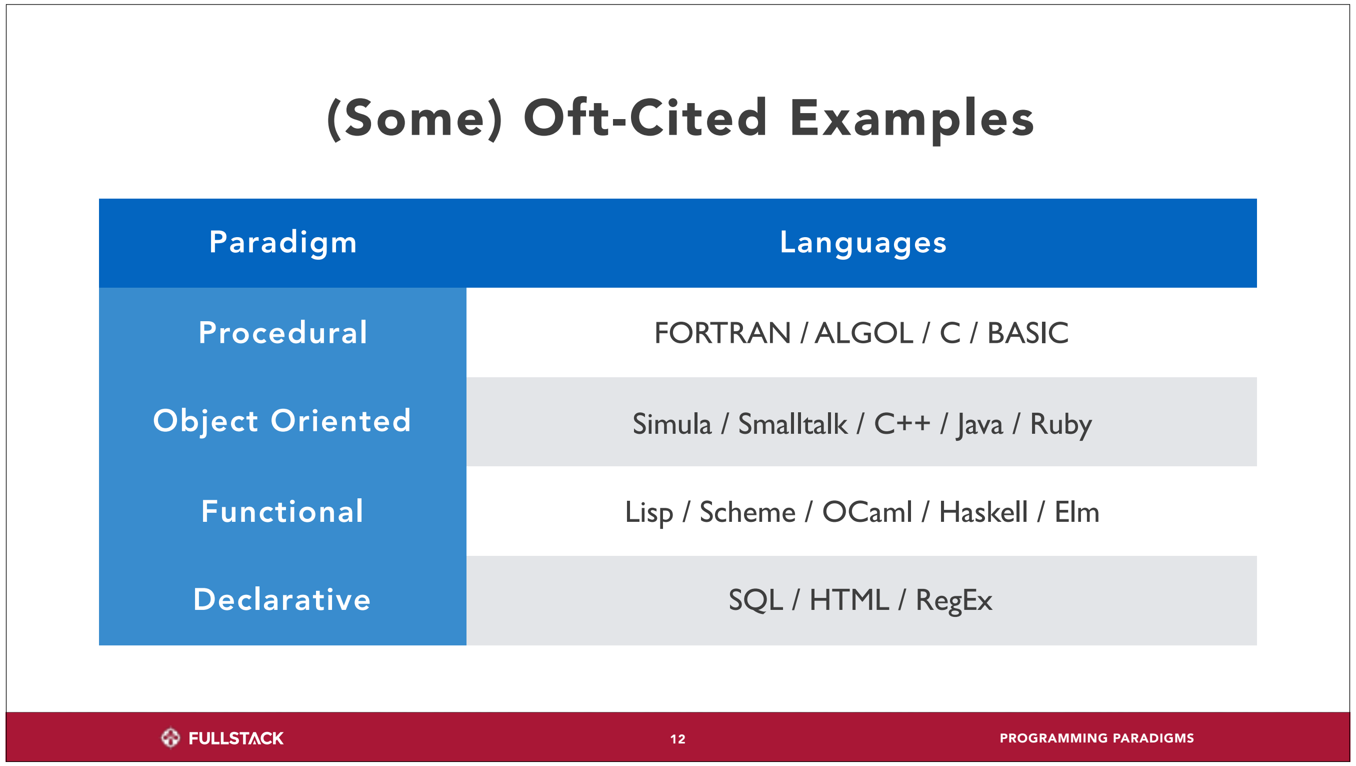
MERRIAM WEBSTER

Categories of programming languages

Traditionally viewed as competing styles 🤔

Different syntax, capabilities, goals, and/or concepts

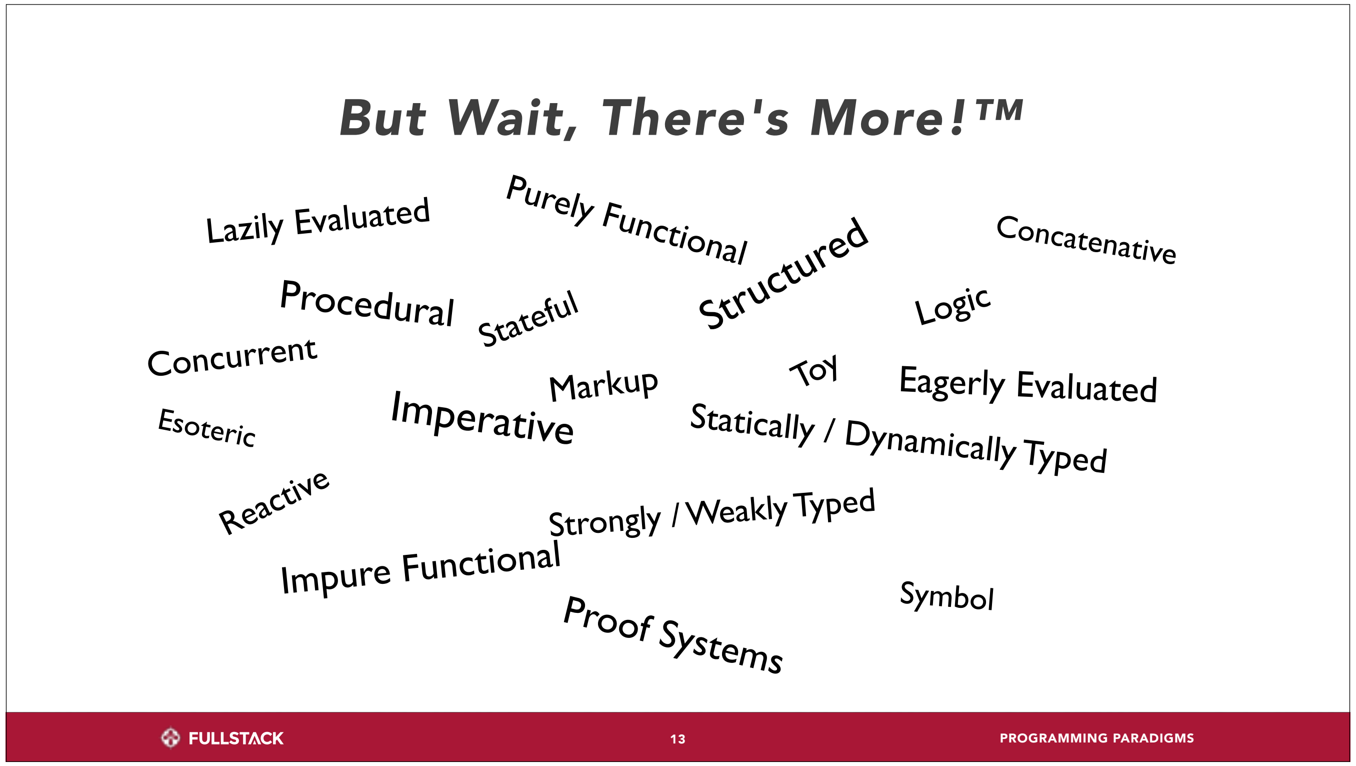
The buzzwords on previous slides are examples of *paradigms*.



(Some) Oft-Cited Examples

Paradigm	Languages
Procedural	FORTRAN / ALGOL / C / BASIC
Object Oriented	Simula / Smalltalk / C++ / Java / Ruby
Functional	Lisp / Scheme / OCaml / Haskell / Elm
Declarative	SQL / HTML / RegEx

Some languages were either built to be, or are considered to be, archetypes of particular paradigms.



But Wait, There's More!™

Lazily Evaluated

Purely Functional

Concatenative

Procedural

Stateful

Structured

Logic

Concurrent

Markup

Toy

Eagerly Evaluated

Esoteric

Imperative

Statically / Dynamically Typed

Reactive

Strongly / Weakly Typed

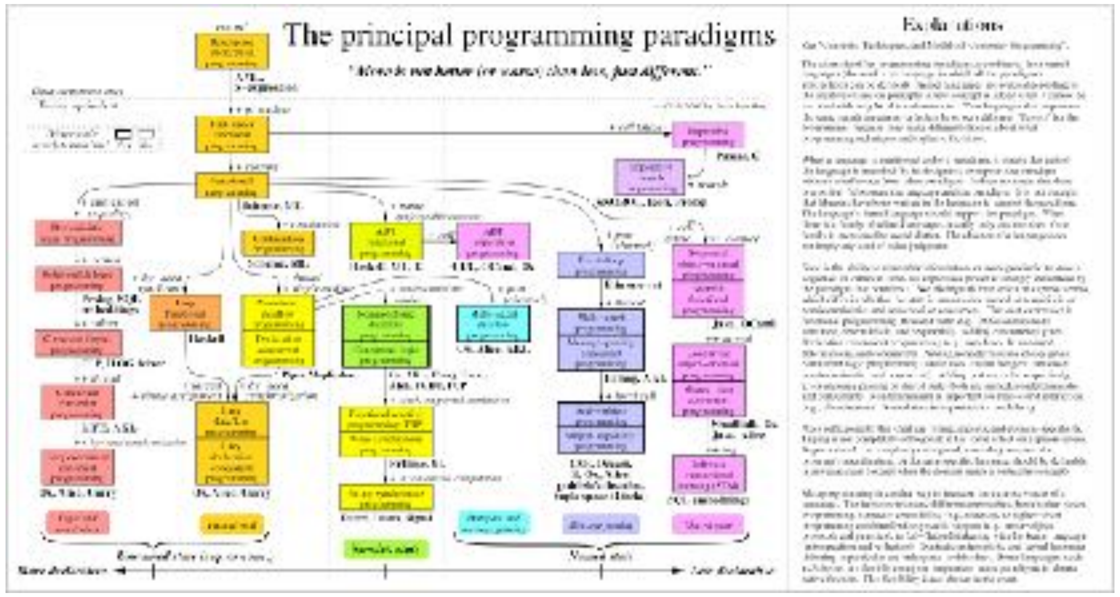
Impure Functional

Symbol

Proof Systems

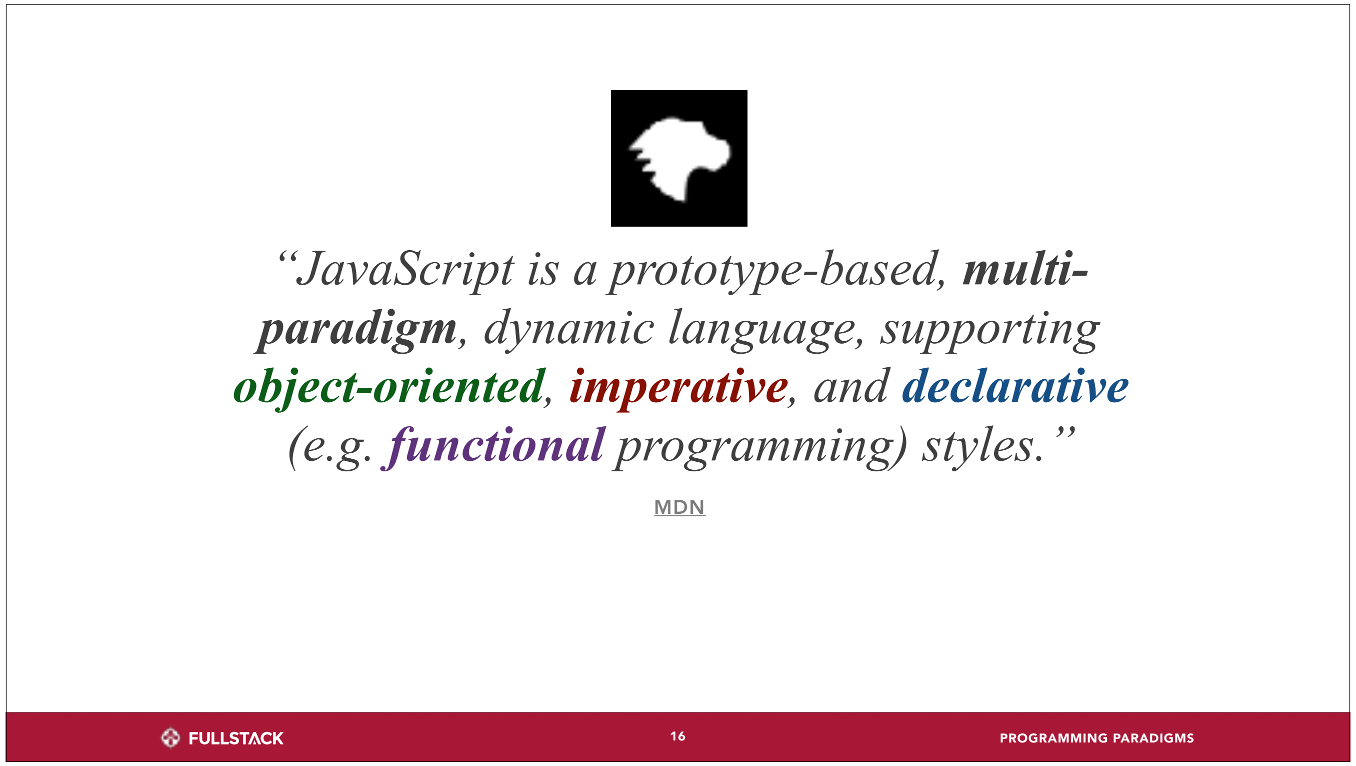
We are barely scratching the surface.

A Large & Arbitrary Topic



A disclaimer. Classifying programming languages into neat little boxes is doomed; the boxes are not well defined and languages don't play by the rules. *(Image from Peter Van Roy's book "Concepts, Techniques, and Models of Computer Programming")*

WHAT ABOUT JS?

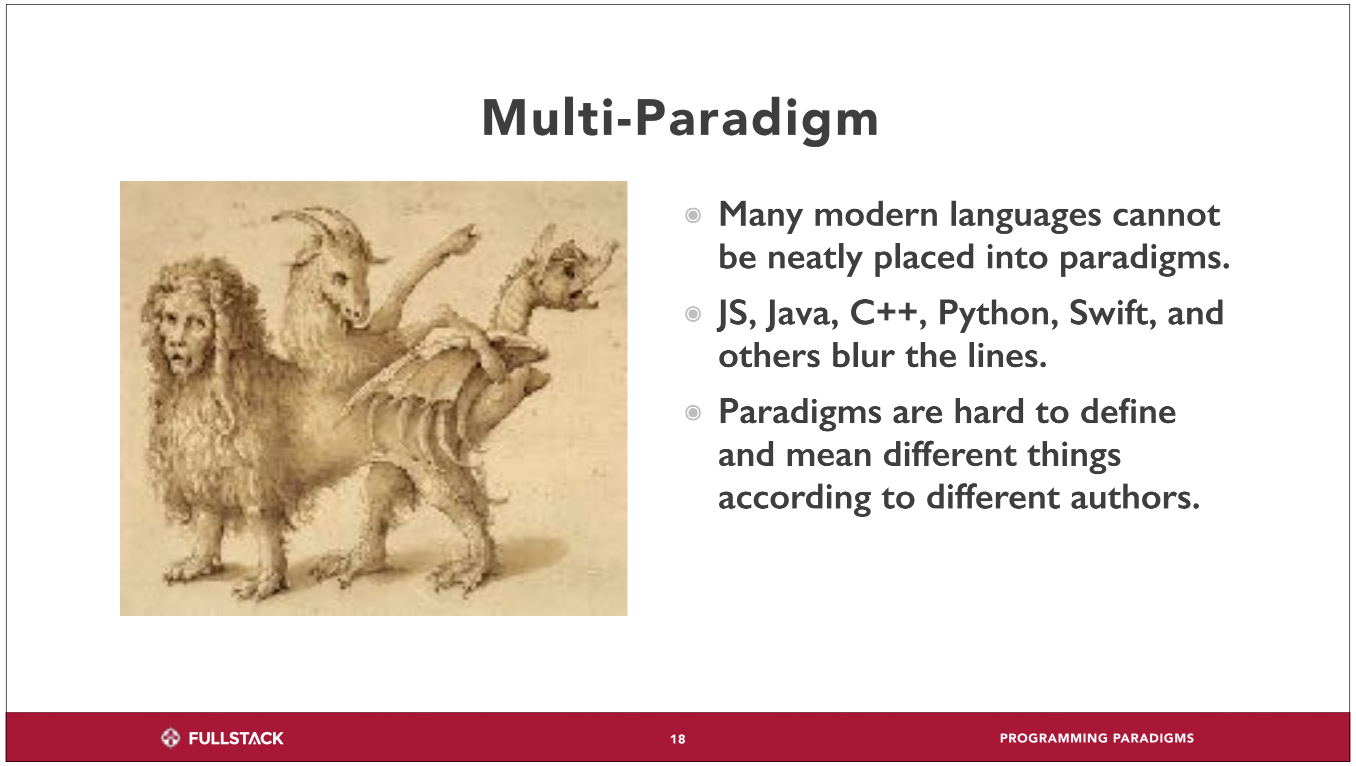


> *“JavaScript is a prototype-based, **multi-paradigm**, dynamic language, supporting **object-oriented**, **imperative**, and **declarative** (e.g. **functional** programming) styles.”*

MDN

Literally the first paragraph on the MDN page introducing JS.

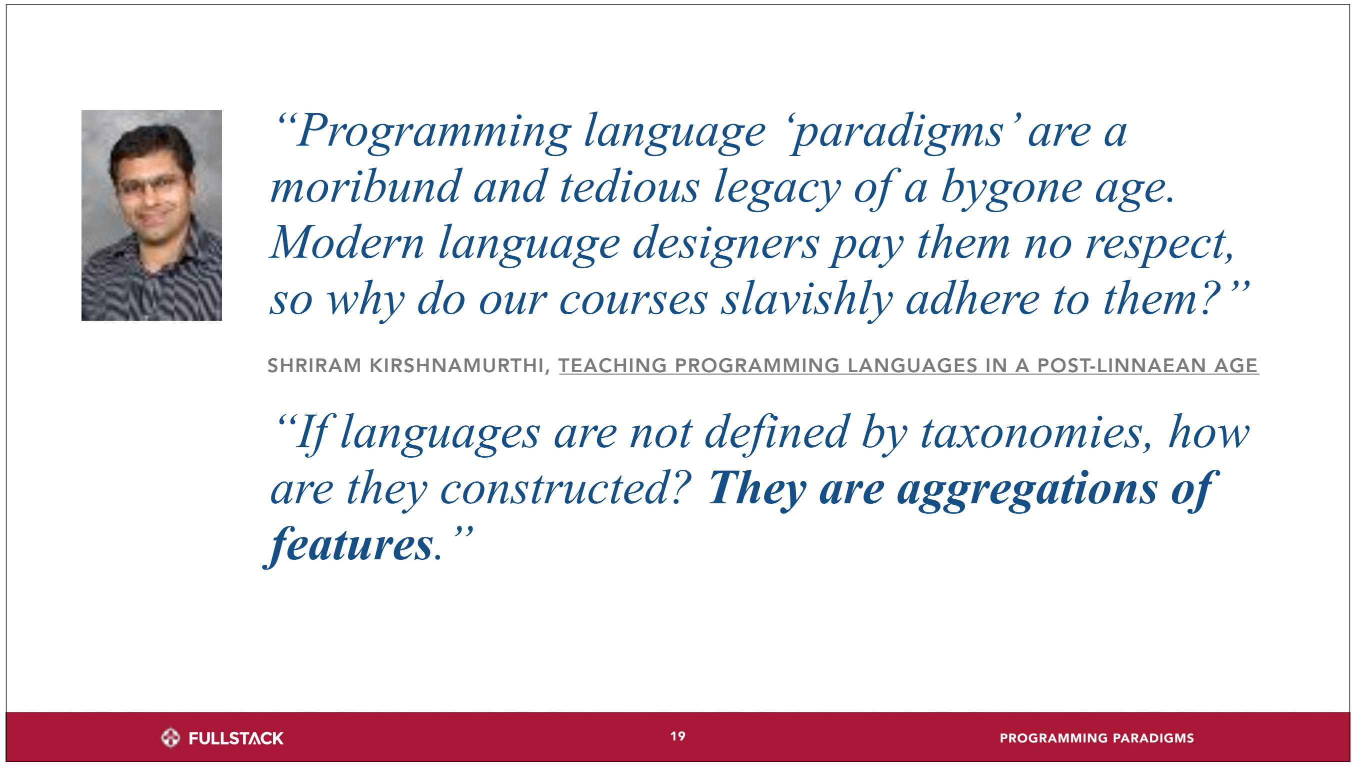
SAY WHAT!?



Multi-Paradigm

- Many modern languages cannot be neatly placed into paradigms.
- JS, Java, C++, Python, Swift, and others blur the lines.
- Paradigms are hard to define and mean different things according to different authors.

Image is of a chimera, mythological beast combining features from several animals.



> *“Programming language ‘paradigms’ are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?”*

SHRIRAM KIRSHNAMURTHI, TEACHING PROGRAMMING LANGUAGES IN A POST-LINNAEAN AGE

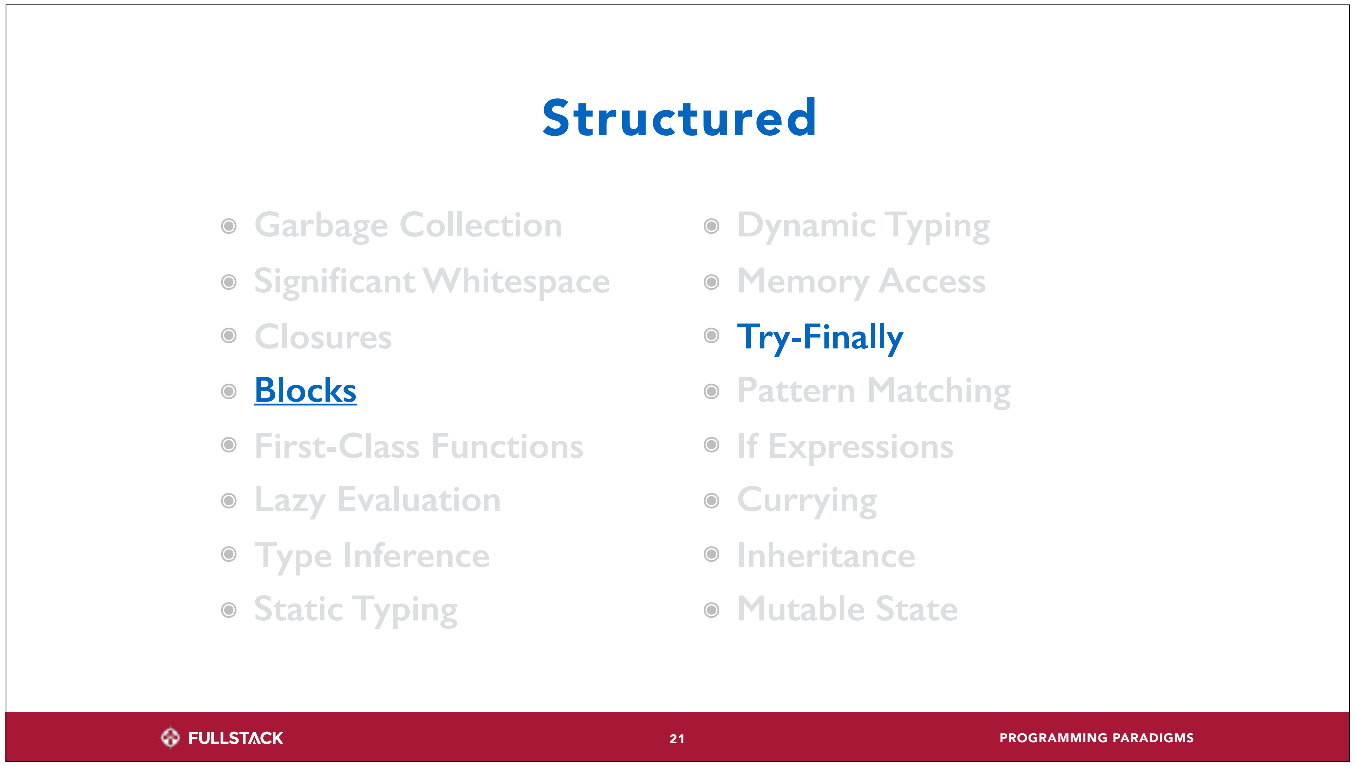
> *“If languages are not defined by taxonomies, how are they constructed? **They are aggregations of features**.”*

Real languages are less **classified**, more **composed**.

(*Side note, this is actually similar to how some programming trends favor mixins over class inheritance.*)

A Random Set of Language Features

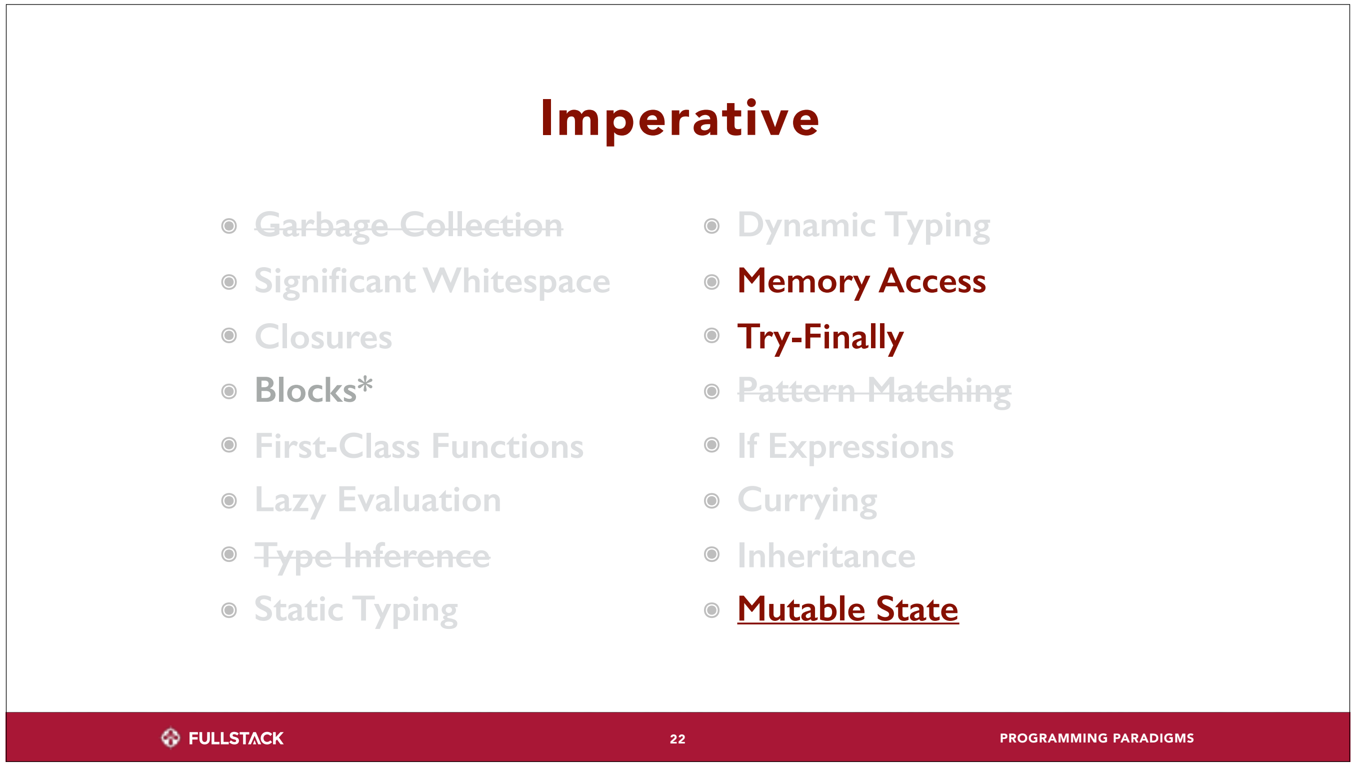
- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State



Structured

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

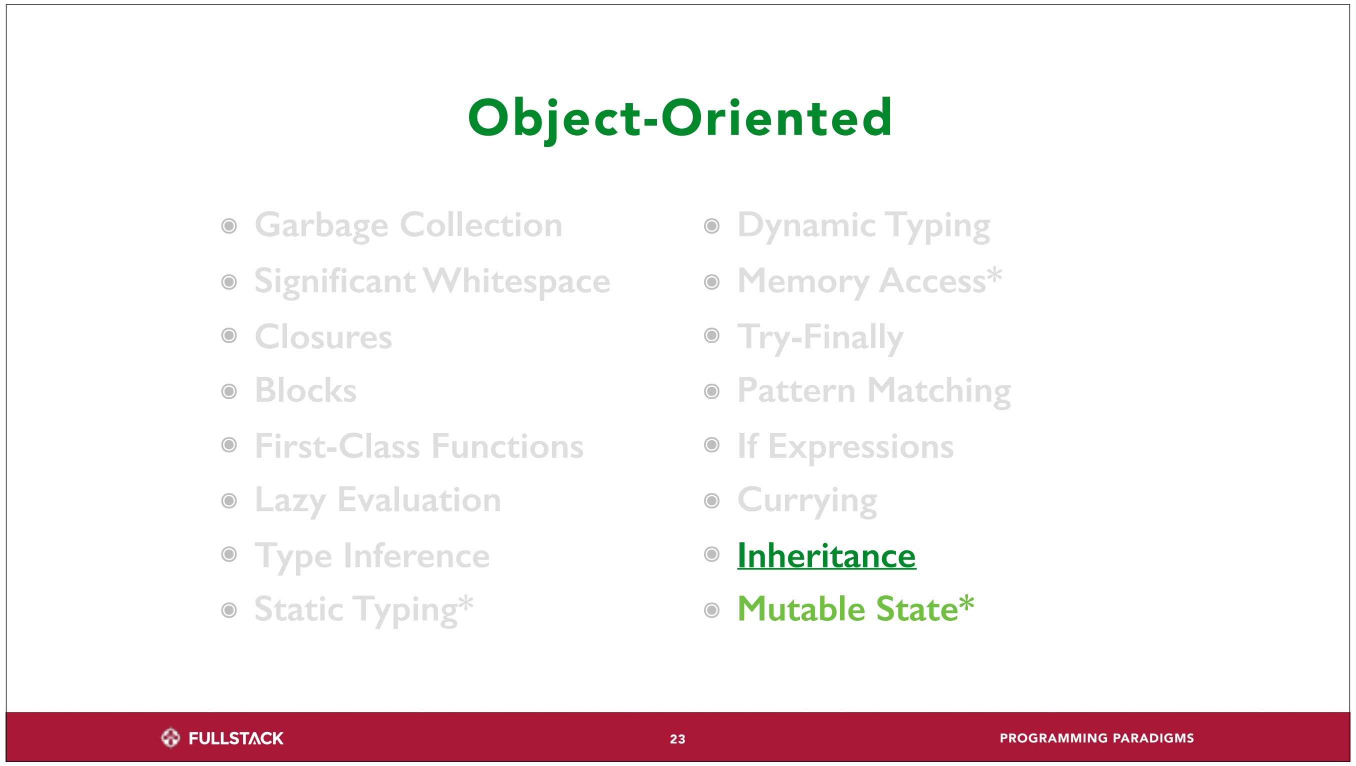
[SKIPPABLE] Some of these features might be considered to fall within a given paradigm



Imperative

- ~~Garbage Collection~~
- Significant Whitespace
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- ~~Type Inference~~
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- ~~Pattern Matching~~
- If Expressions
- Currying
- Inheritance
- Mutable State

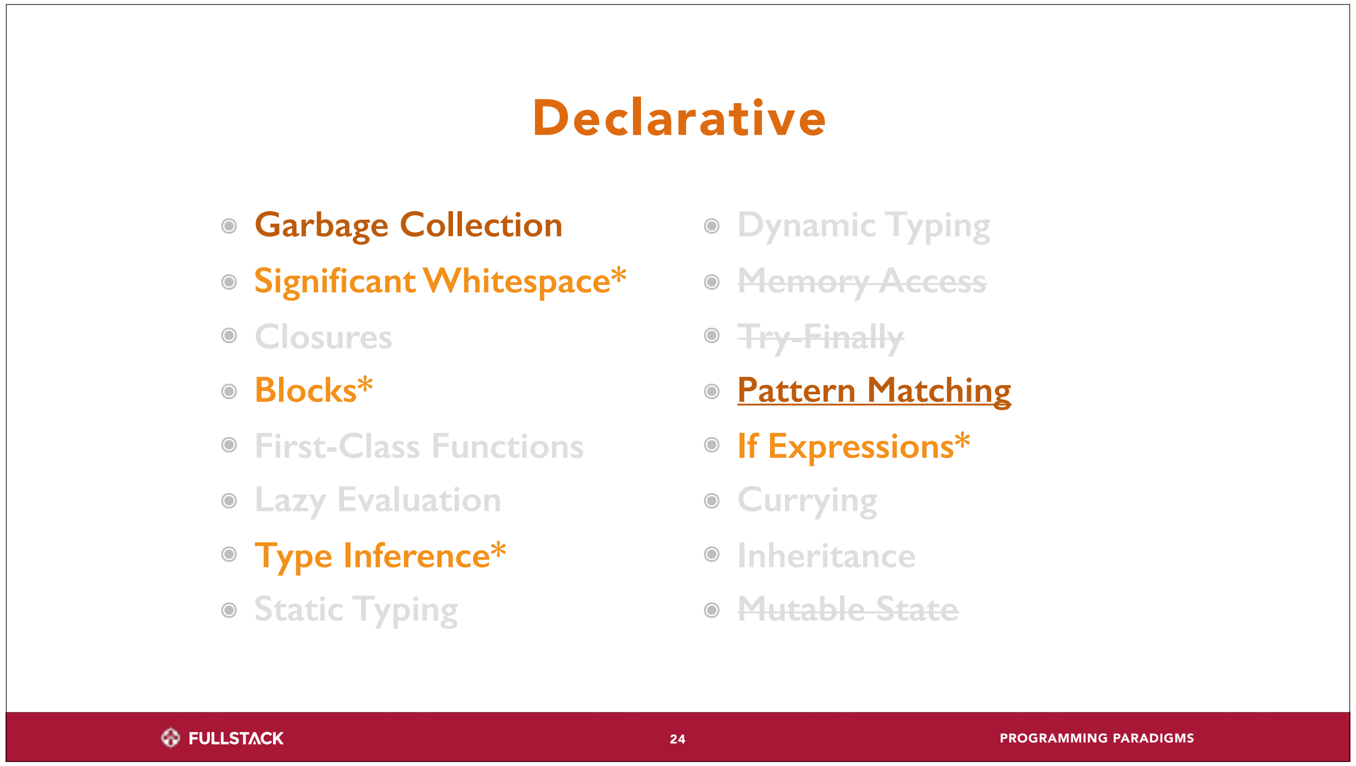
[SKIPPABLE] But a given feature might be often found across multiple paradigms



Object-Oriented

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing*
- Dynamic Typing
- Memory Access*
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State*

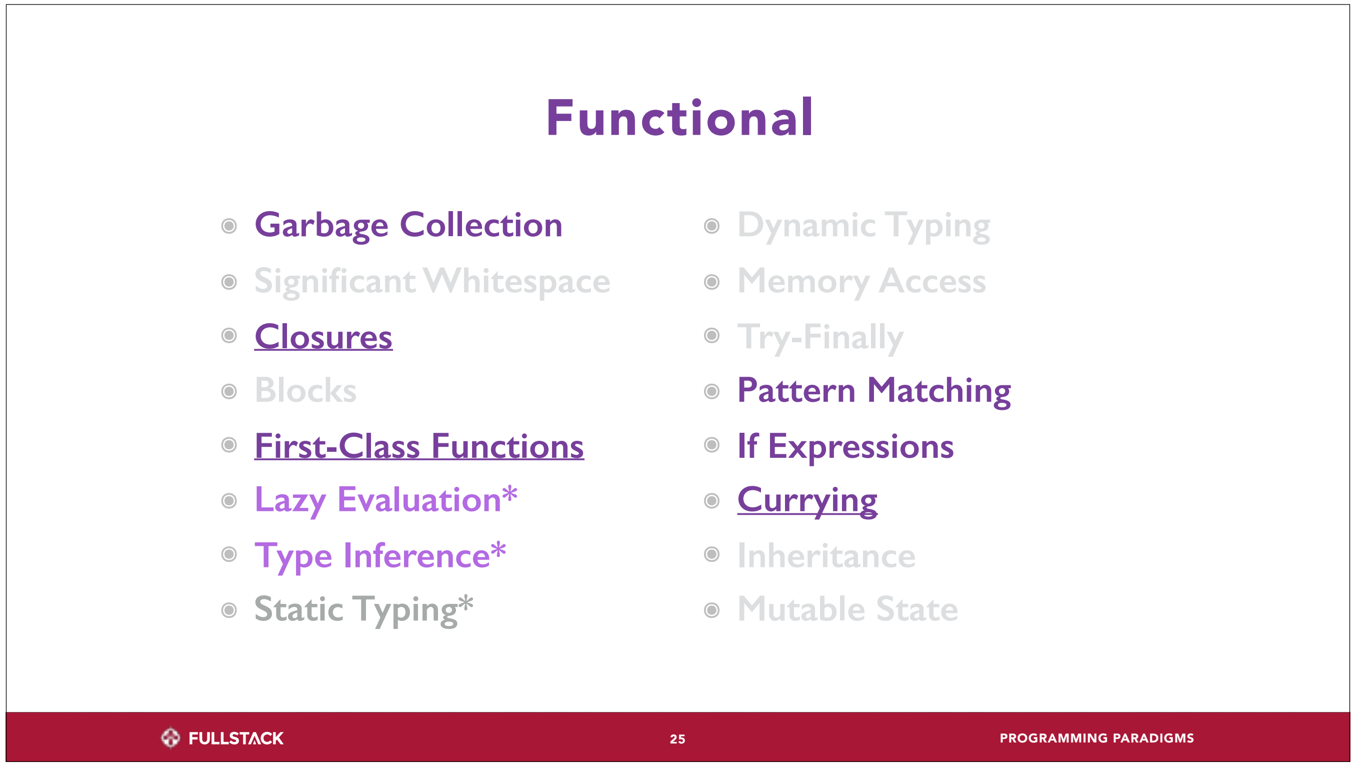
[SKIPPABLE]



Declarative

- Garbage Collection
- Significant Whitespace*
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- Type Inference*
- Static Typing
- Dynamic Typing
- ~~Memory Access~~
- ~~Try-Finally~~
- Pattern Matching
- If Expressions*
- Currying
- Inheritance
- ~~Mutable State~~

[SKIPPABLE] Is significant whitespace declarative? What about if-expressions (which produce a value)? These features may be found in non-declarative contexts.



Functional

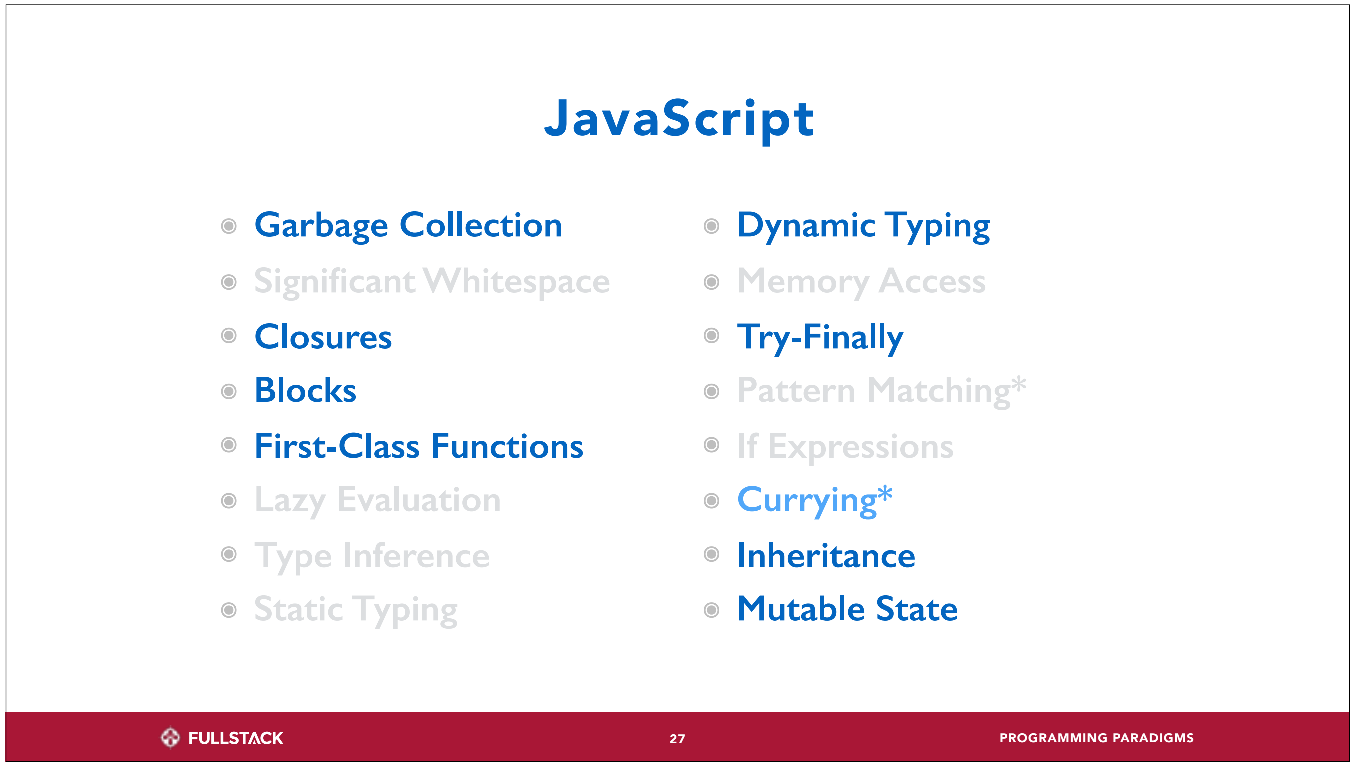
- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation*
- Type Inference*
- Static Typing*

- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State

[SKIPPABLE] If a programming language lacks pattern matching, is it no longer functional? Trying to define paradigms in terms of features is a flawed approach.

C

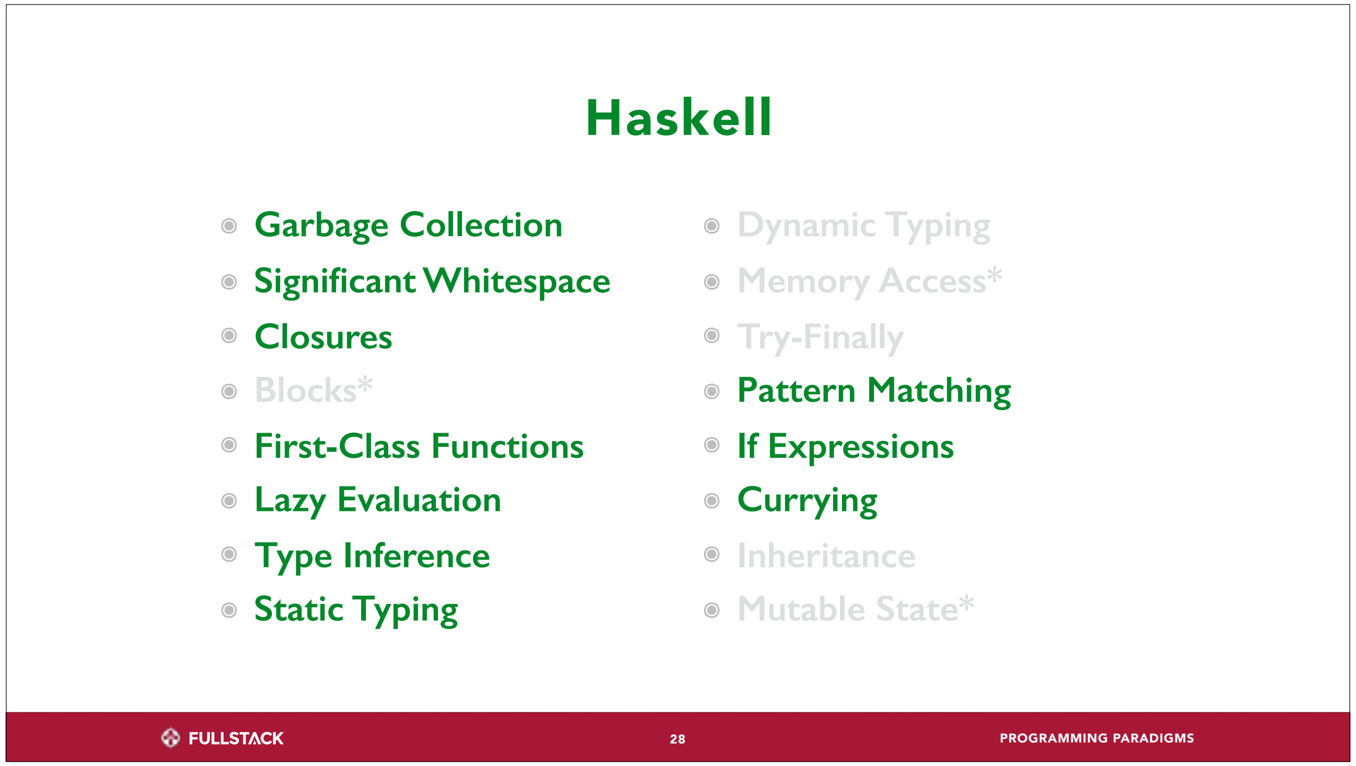
- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions*
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally*
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State



JavaScript

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access
- Try-Finally
- Pattern Matching*
- If Expressions
- Currying*
- Inheritance
- Mutable State

In contrast, languages definitely **DO** or **DO NOT** have a certain feature, or **CAN** emulate a feature (asterisks). This makes it a lot more concrete to classify a language via features.

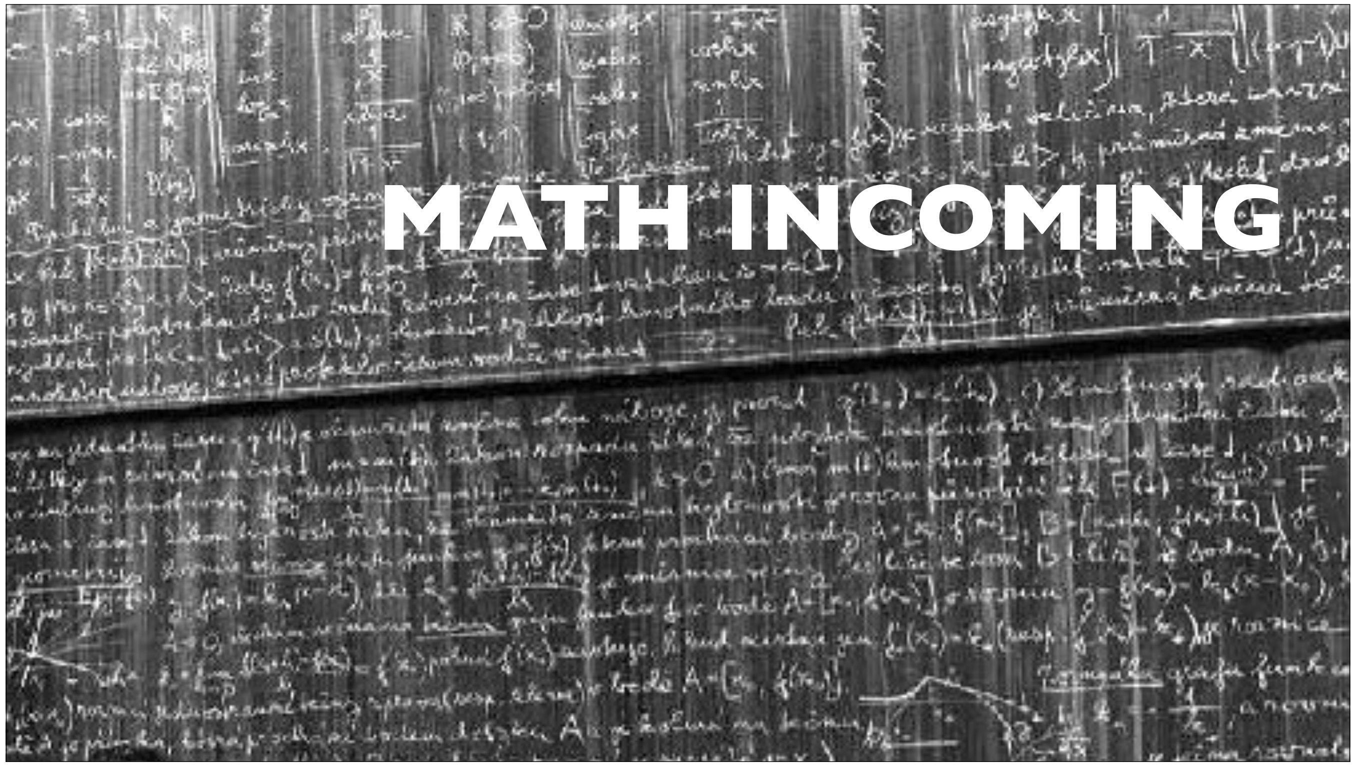


Haskell

- Garbage Collection
- Significant Whitespace
- Closures
- Blocks*
- First-Class Functions
- Lazy Evaluation
- Type Inference
- Static Typing
- Dynamic Typing
- Memory Access*
- Try-Finally
- Pattern Matching
- If Expressions
- Currying
- Inheritance
- Mutable State*

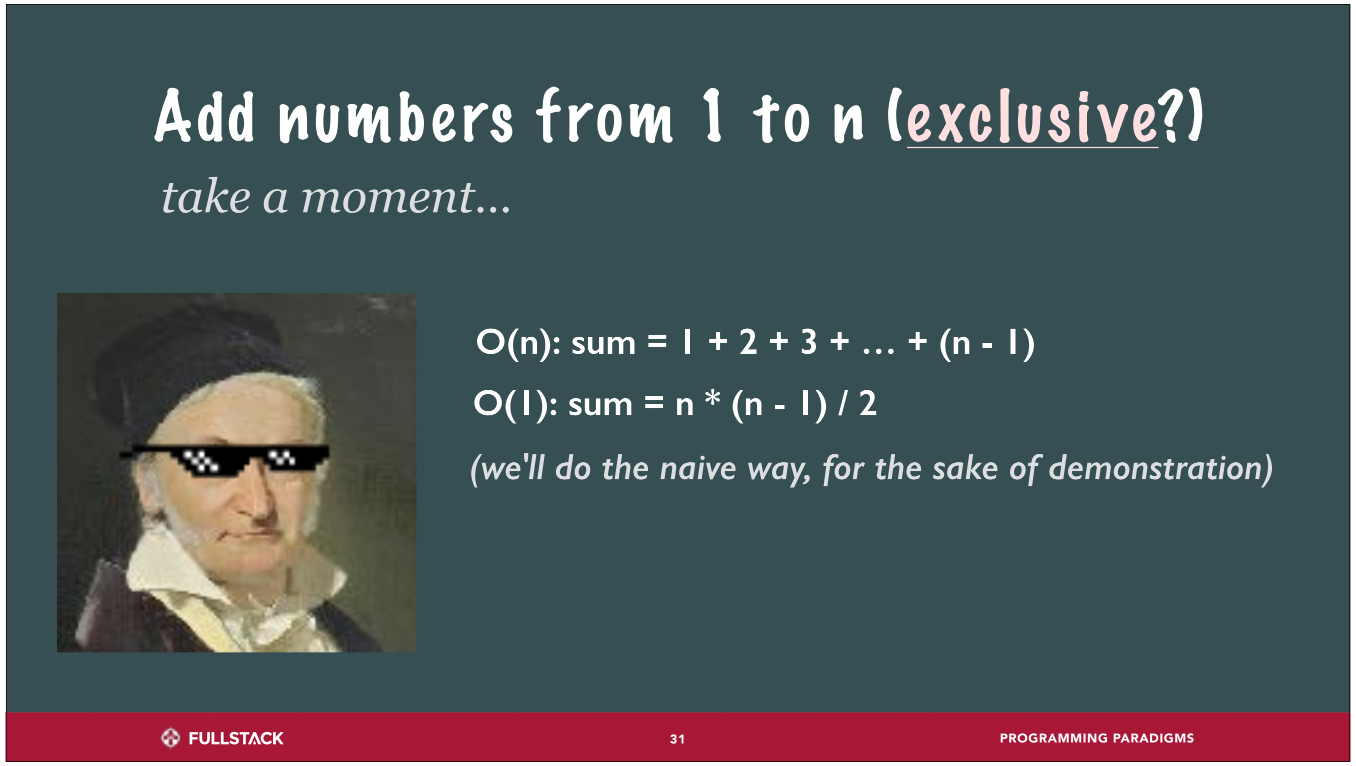
Notice that Haskell and JS share some features, typically considered functional. But they definitely have differences too. And Haskell has things beyond the "functional" features.

Paradigms: Imperative, Structured, Procedural



MATH INCOMING

To help us understand how computers and by extension computer languages work, let's use a small example algorithm.



Add numbers from 1 to n (exclusive?)

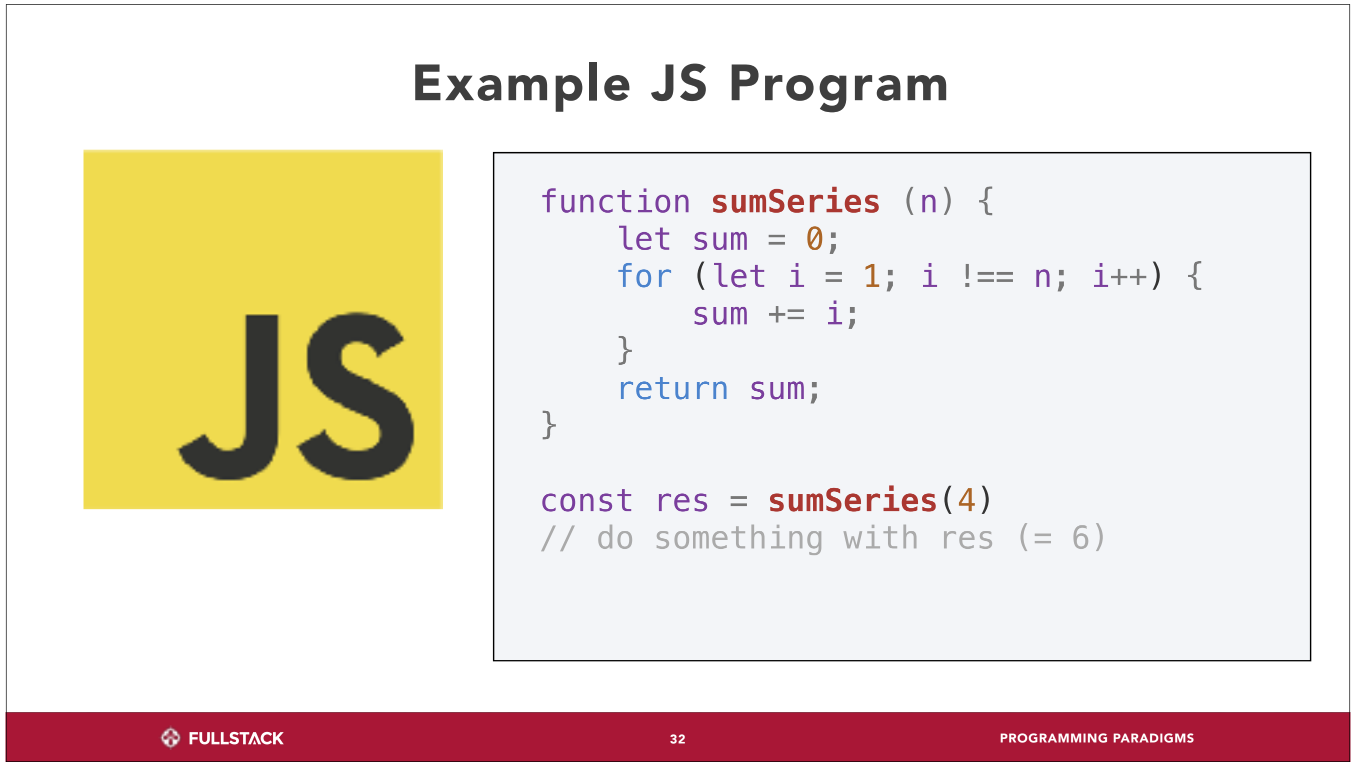
take a moment...

O(n): sum = 1 + 2 + 3 + … + (n - 1)

O(1): sum = n * (n - 1) / 2

(we'll do the naive way, for the sake of demonstration)

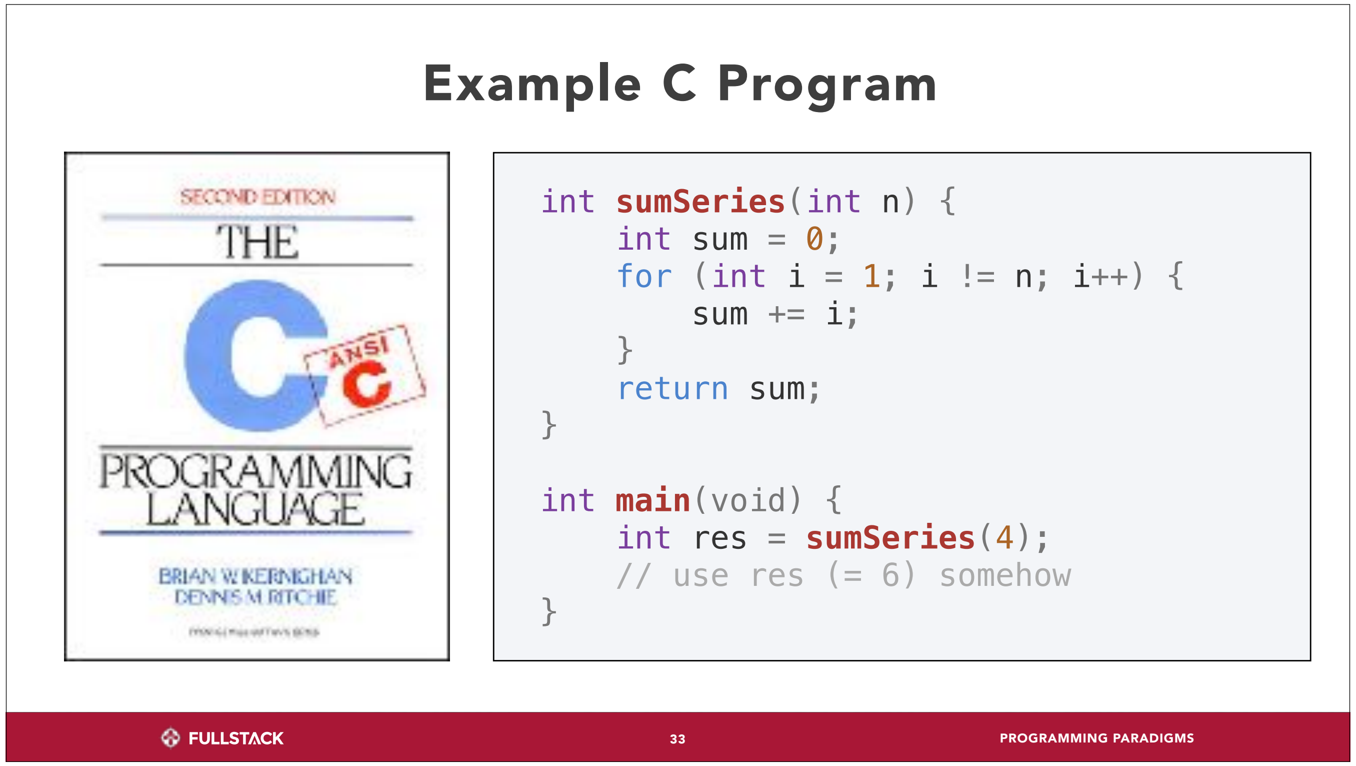
This is a straightforward problem. The mathematician Carl Friedrich Gauss (1777-1855) famously solved it as a young boy in primary school. We'll use a more naive solution for demonstration purposes.



Example JS Program

```
function sumSeries (n) {
    let sum = 0;
    for (let i = 1; i !== n; i++) {
        sum += i;
    }
    return sum;
}

const res = sumSeries(4)
// do something with res (= 6)
```

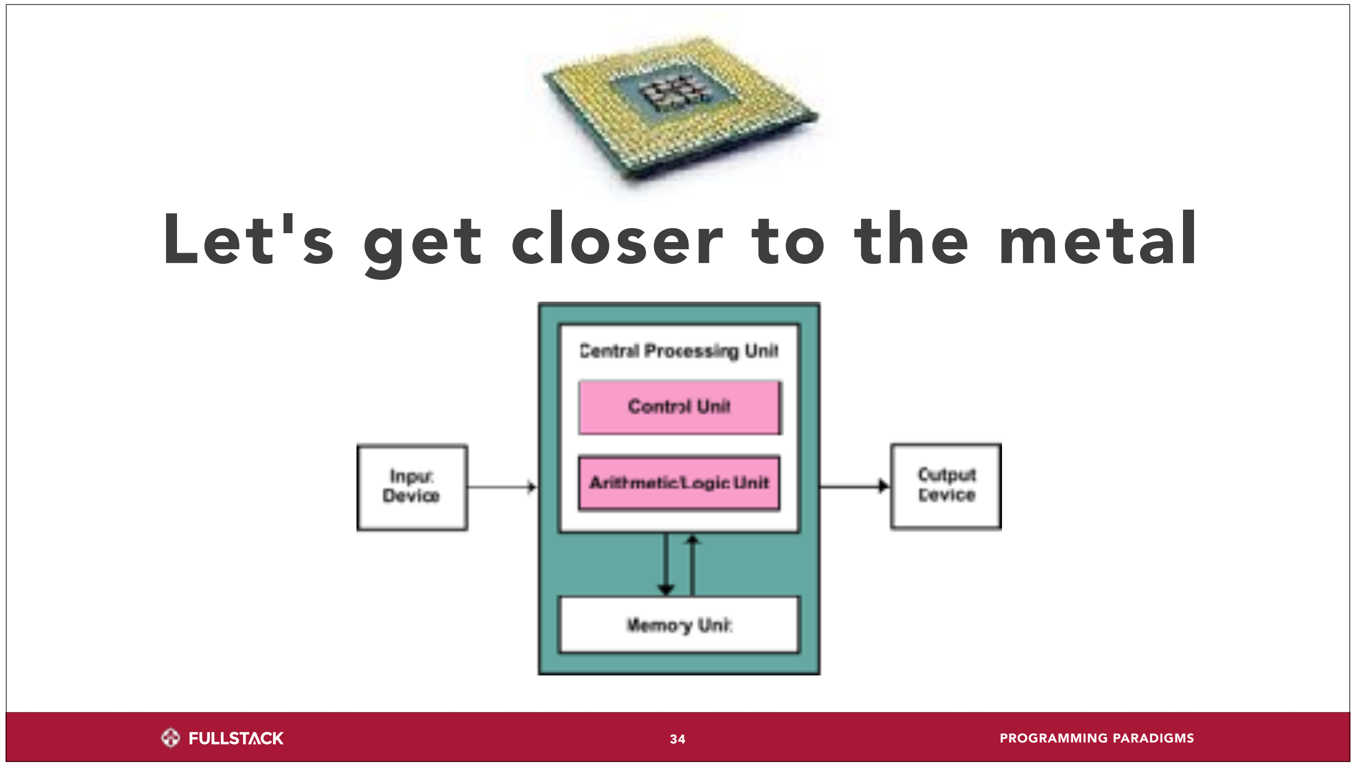
Here's how one might code the naive solution in JS.



Example C Program

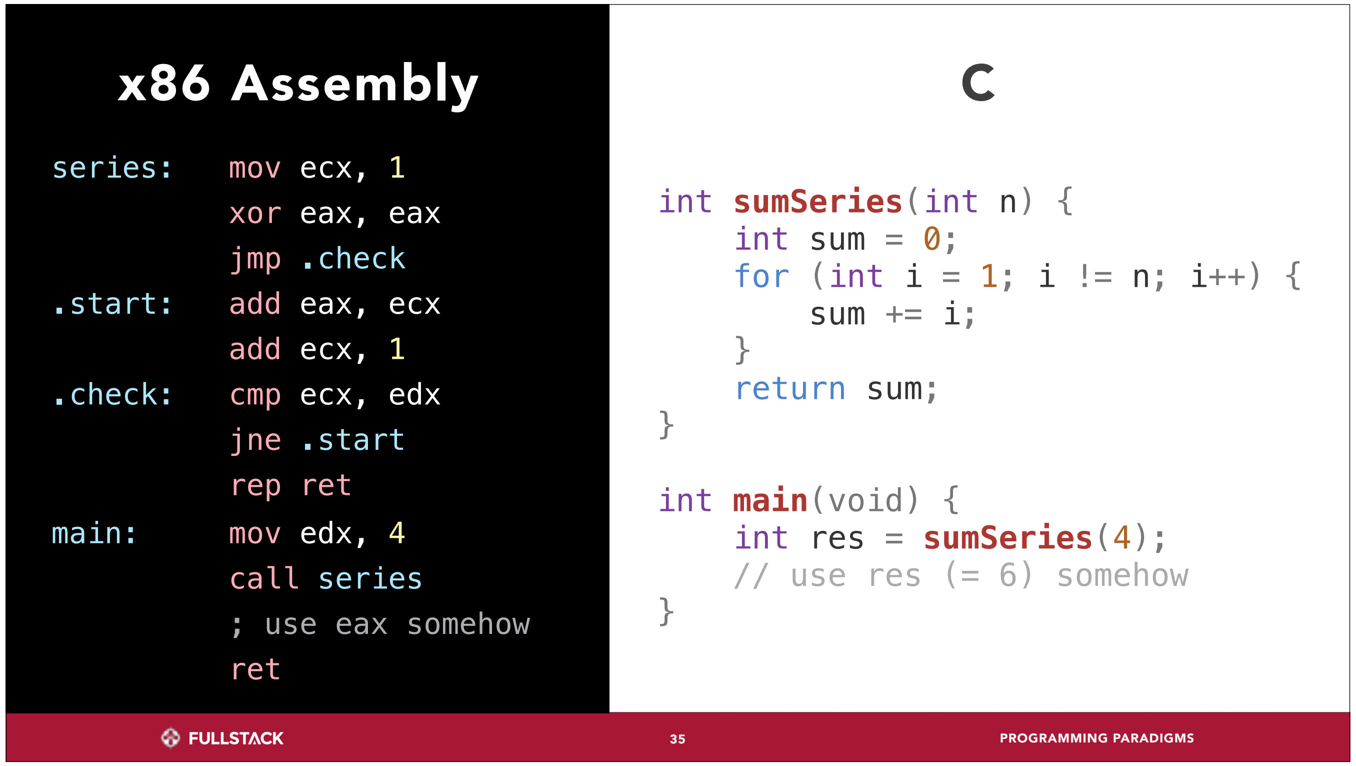
```c
int sumSeries(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = sumSeries(4);
    // use res (= 6) somehow
}
```

…and here is an equivalent solution in C. Note that JS borrowed its syntax from Java, which borrowed its syntax from C. So it's no wonder that the two snippets are almost identical.



Let's get closer to the metal

“The metal” refers the CPU. Virtually all commercial computers use the Von Neumann Architecture, in which the CPU accesses addressable memory – fetching data, computing results, and storing data.



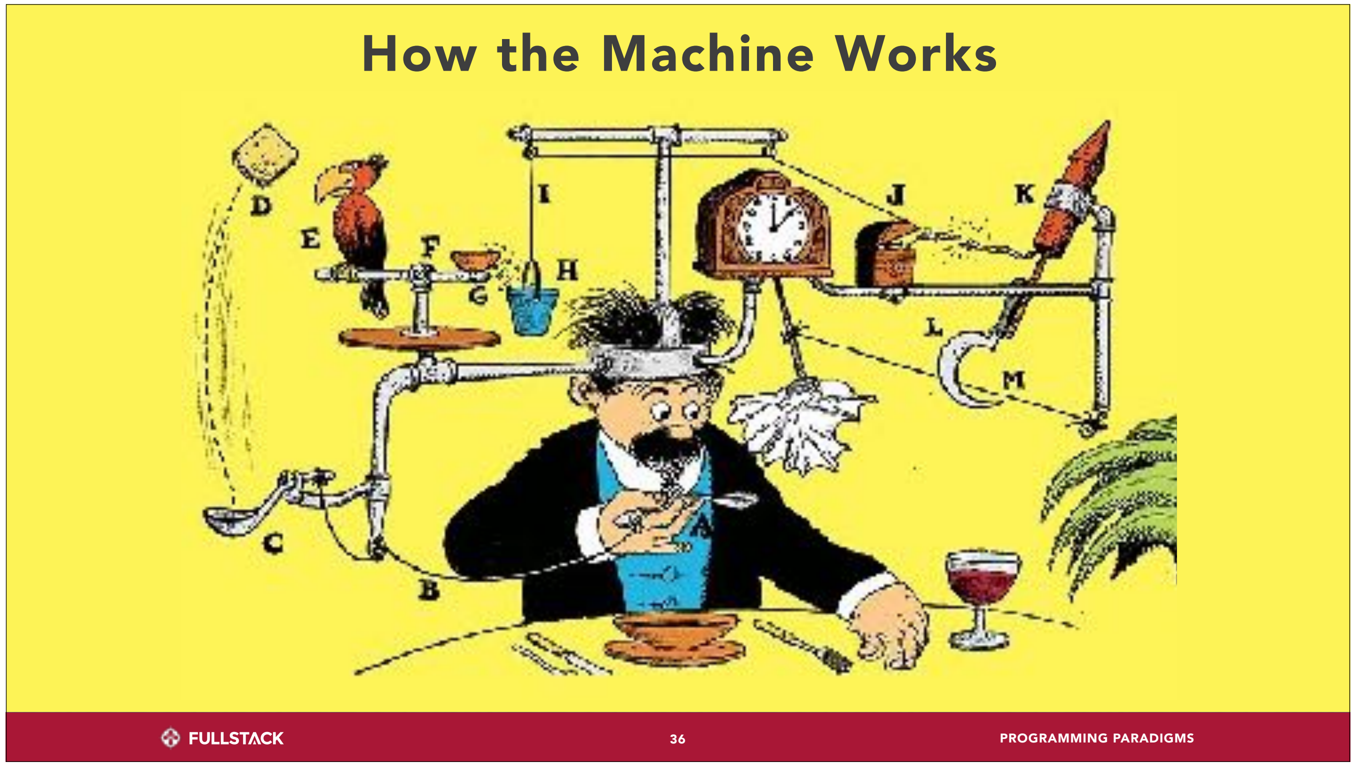
x86 Assembly

```
series:   mov ecx, 1
          xor eax, eax
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          rep ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

C

```
int sumSeries(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = sumSeries(4);
    // use res (= 6) somehow
}
```

C gets compiled to *machine code* – the actual commands for the CPU. We can more easily represent a program using textual *assembly language,* a 1-1 mapping of human-readable commands to machine code.

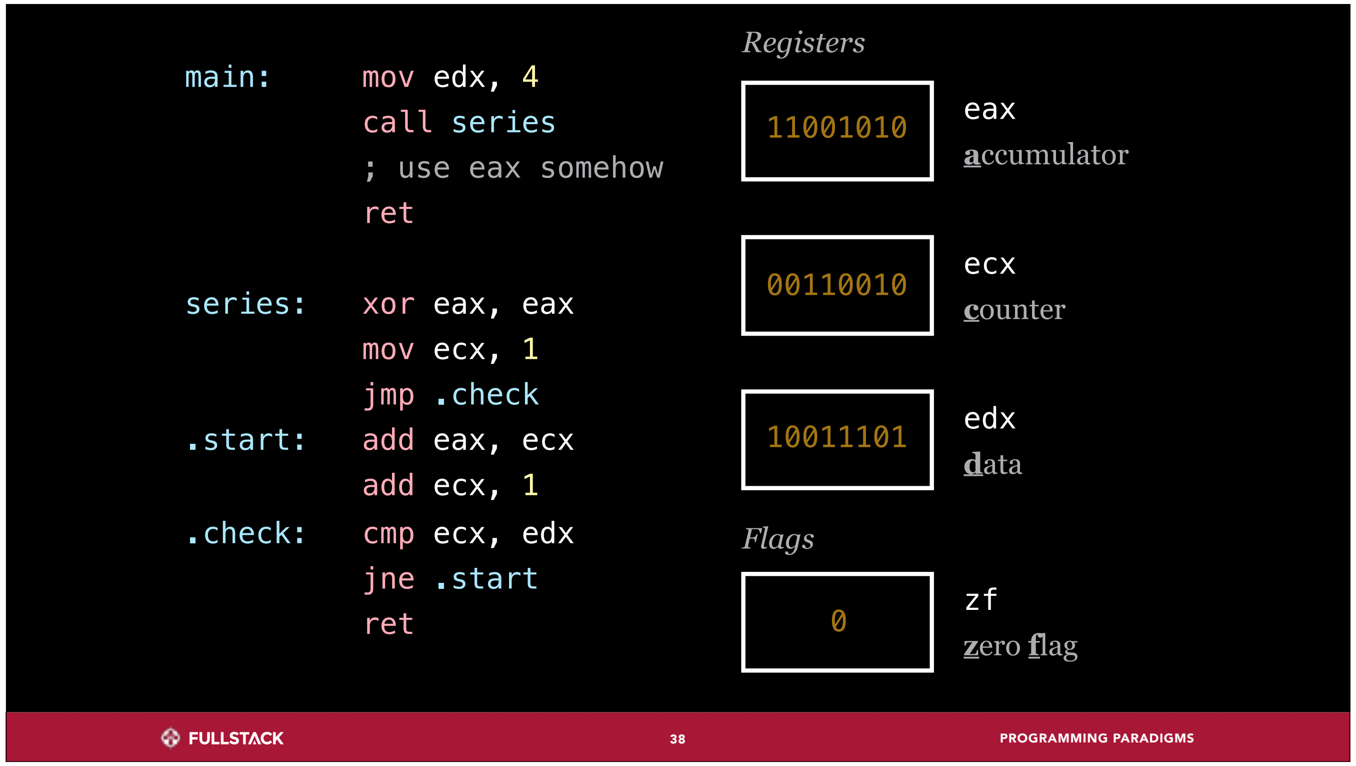


How the Machine Works

Let's see a simulated computer run this code to get a sense for what is really happening.

👩‍🎓 Reminder: this part is all *theory / context / jargon.*

You DO NOT need to learn this to succeed at Fullstack! We are introducing you to these concepts to provide additional insight.



```
main:      mov edx, 4
           call series
           ; use eax somehow
           ret

series:    xor eax, eax
           mov ecx, 1
           jmp .check
.start:    add eax, ecx
           add ecx, 1
.check:    cmp ecx, edx
           jne .start
           ret
```

Registers

11001010 eax
accumulator

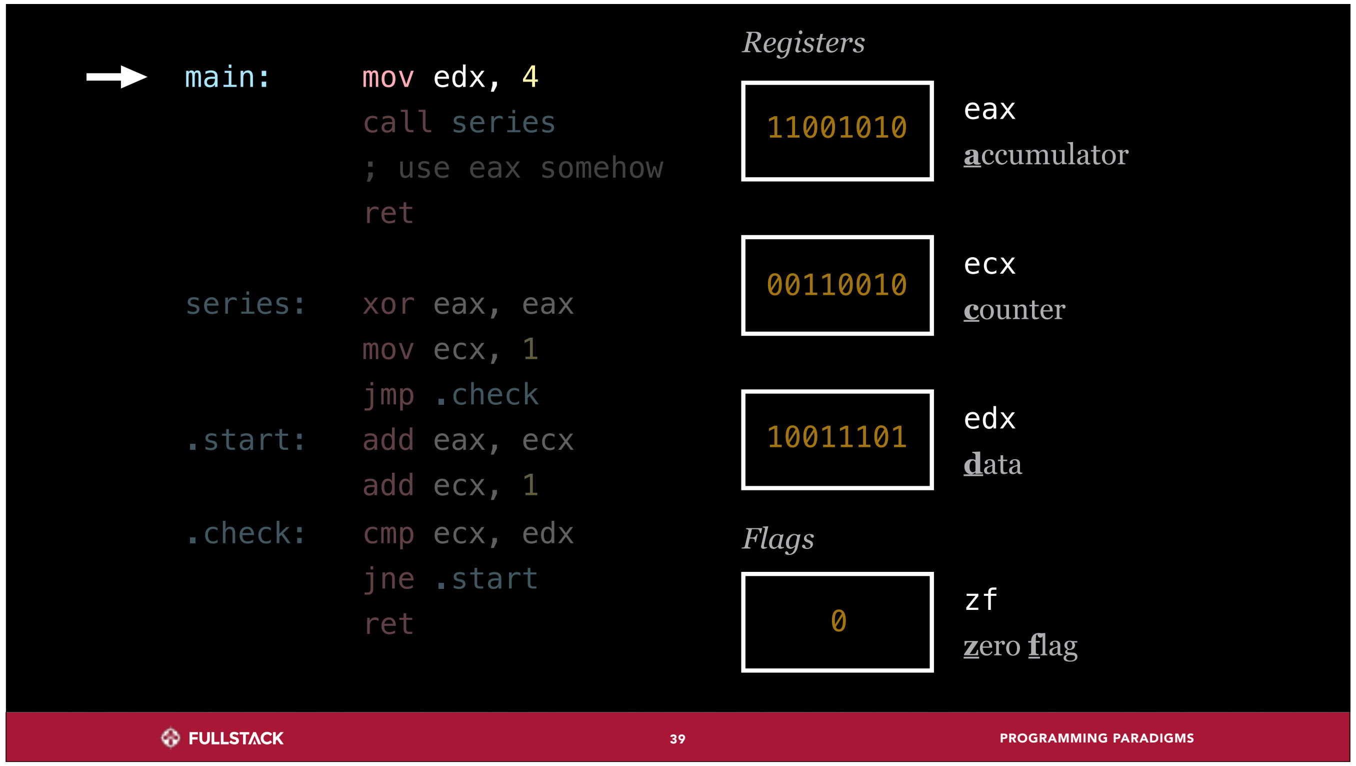
00110010 ecx
counter

10011101 edx
data

Flags

0 zf
zero **f**lag

A CPU has a limited set of named *registers,* physical buckets of data that can be used in ops. It also has a *flag* (actually one bit from a special register) for storing the result of an (in)equality check.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

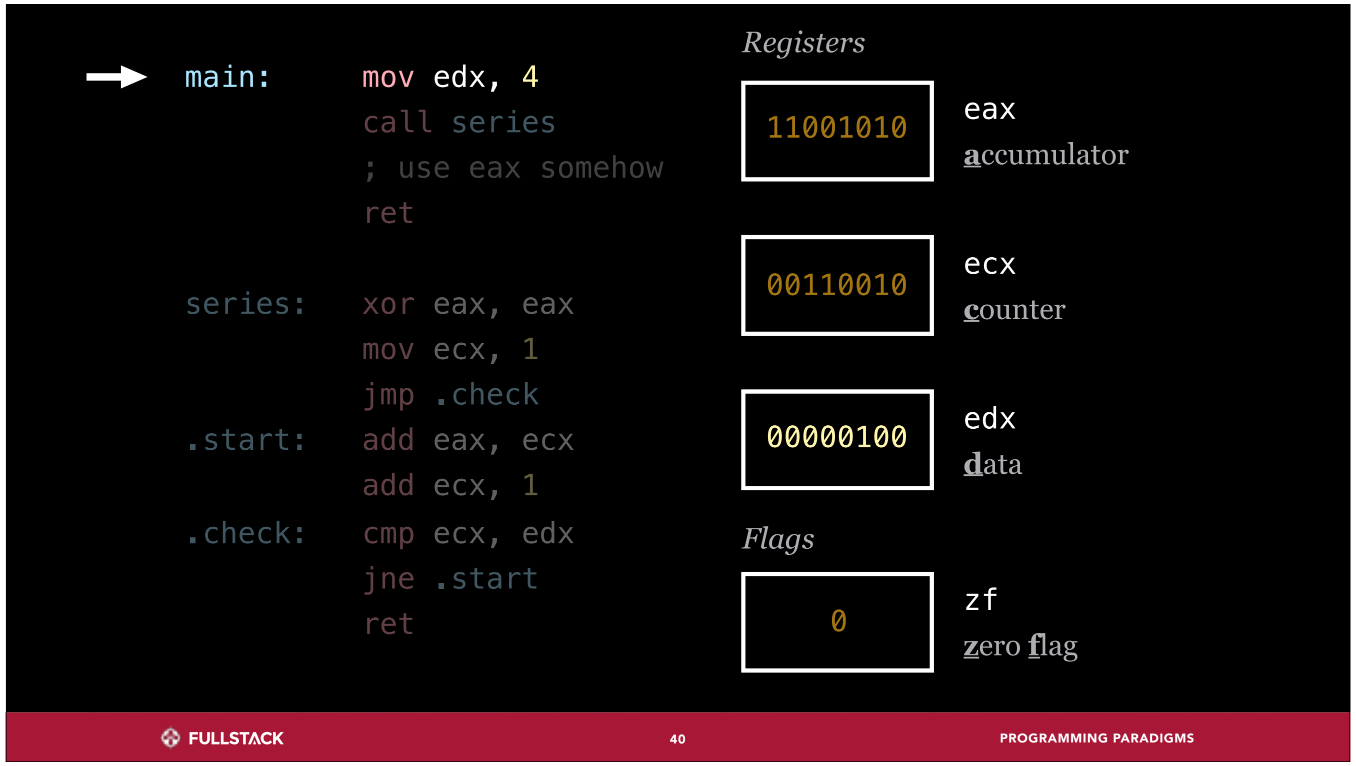
Registers

11001010	eax **a**ccumulator
00110010	ecx **c**ounter
10011101	edx **d**ata

Flags

0	zf **z**ero **f**lag

A *program counter* increments through the commands one by one. This command is "move the number 4 into register edx".



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

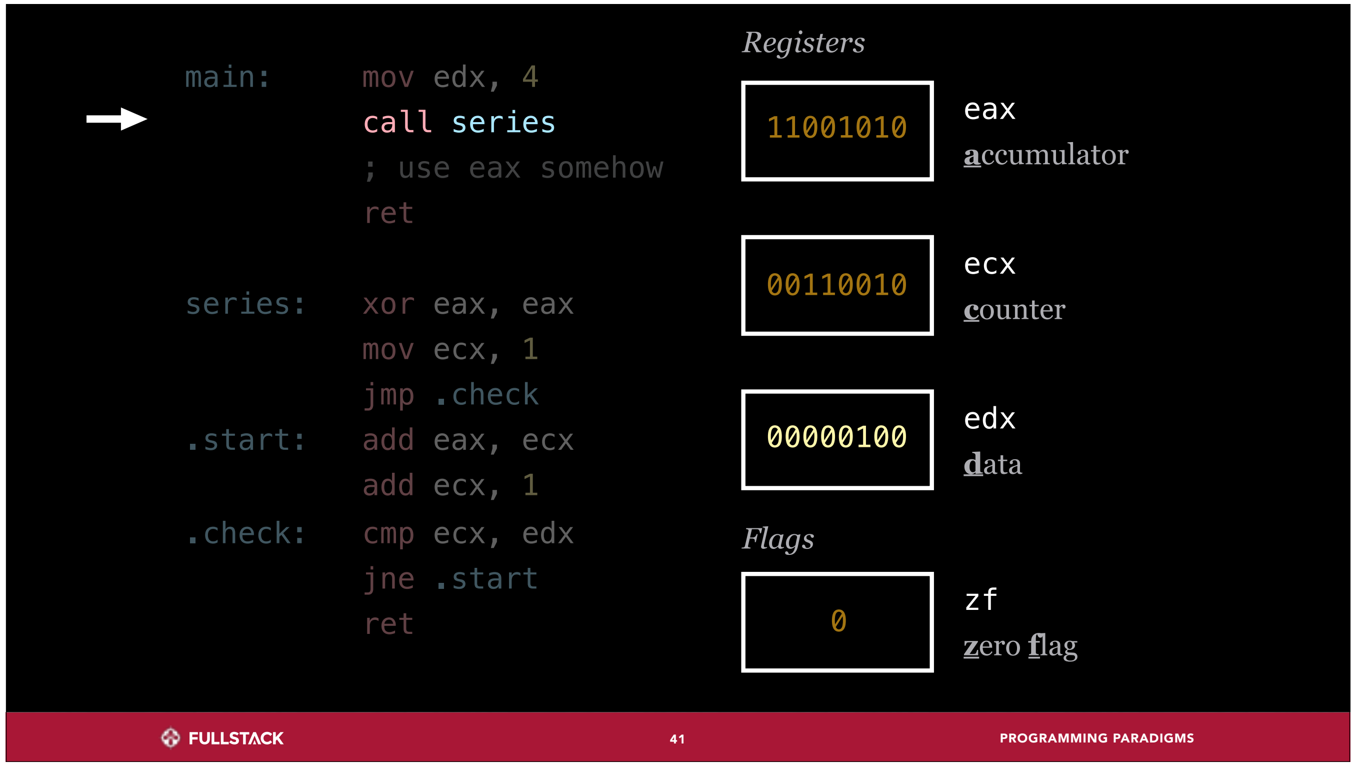
Registers

11001010	eax **a**ccumulator
00110010	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

We will use edx (the "data" register) as our function argument / loop limit.



```
main:     mov edx, 4
→         call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

Registers

11001010 eax
accumulator

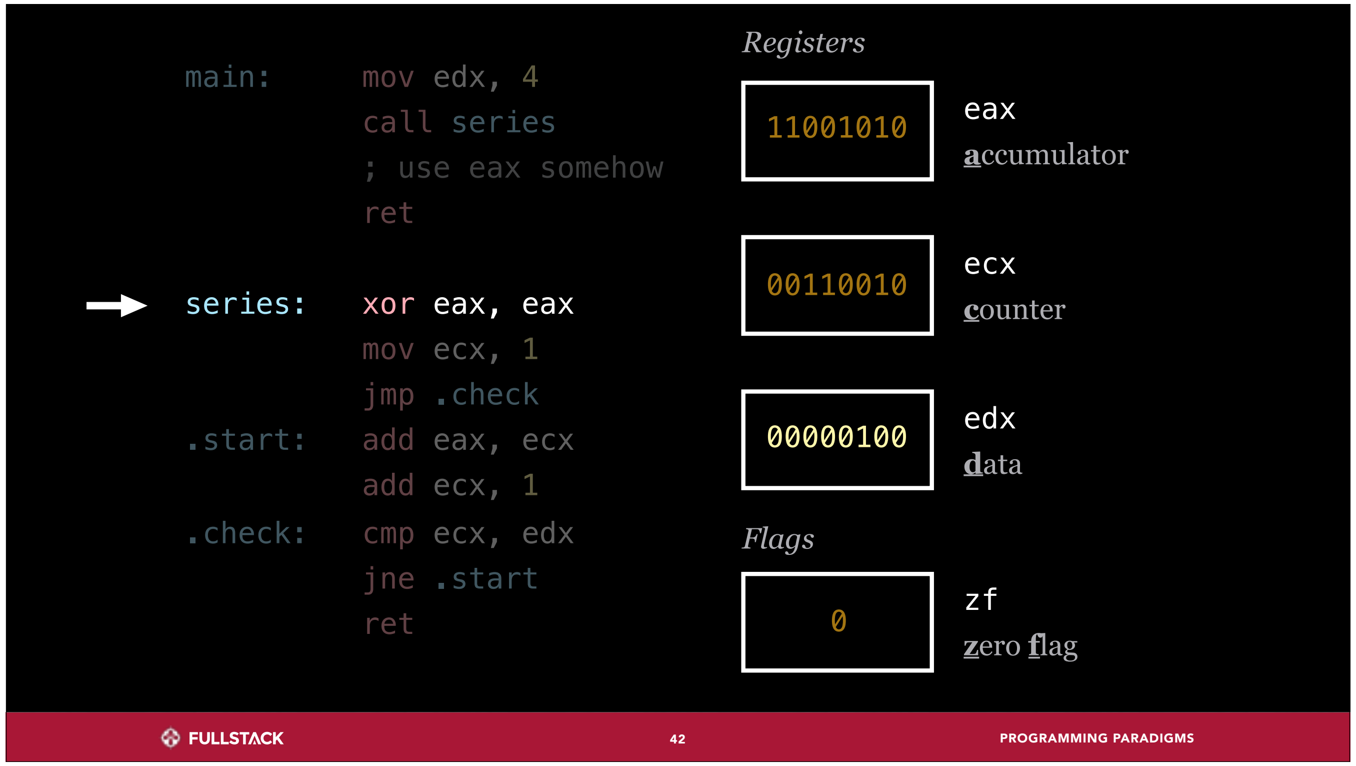
00110010 ecx
counter

00000100 edx
data

Flags

0 zf
zero **f**lag

`call` moves the pointer to a *subroutine*. Subroutines are primitive/imperative functions – they are really *procedures* used for control flow, not data.



```
         main:     mov edx, 4
                   call series
                   ; use eax somehow
                   ret

    →    series:   xor eax, eax
                   mov ecx, 1
                   jmp .check
         .start:   add eax, ecx
                   add ecx, 1
         .check:   cmp ecx, edx
                   jne .start
                   ret
```

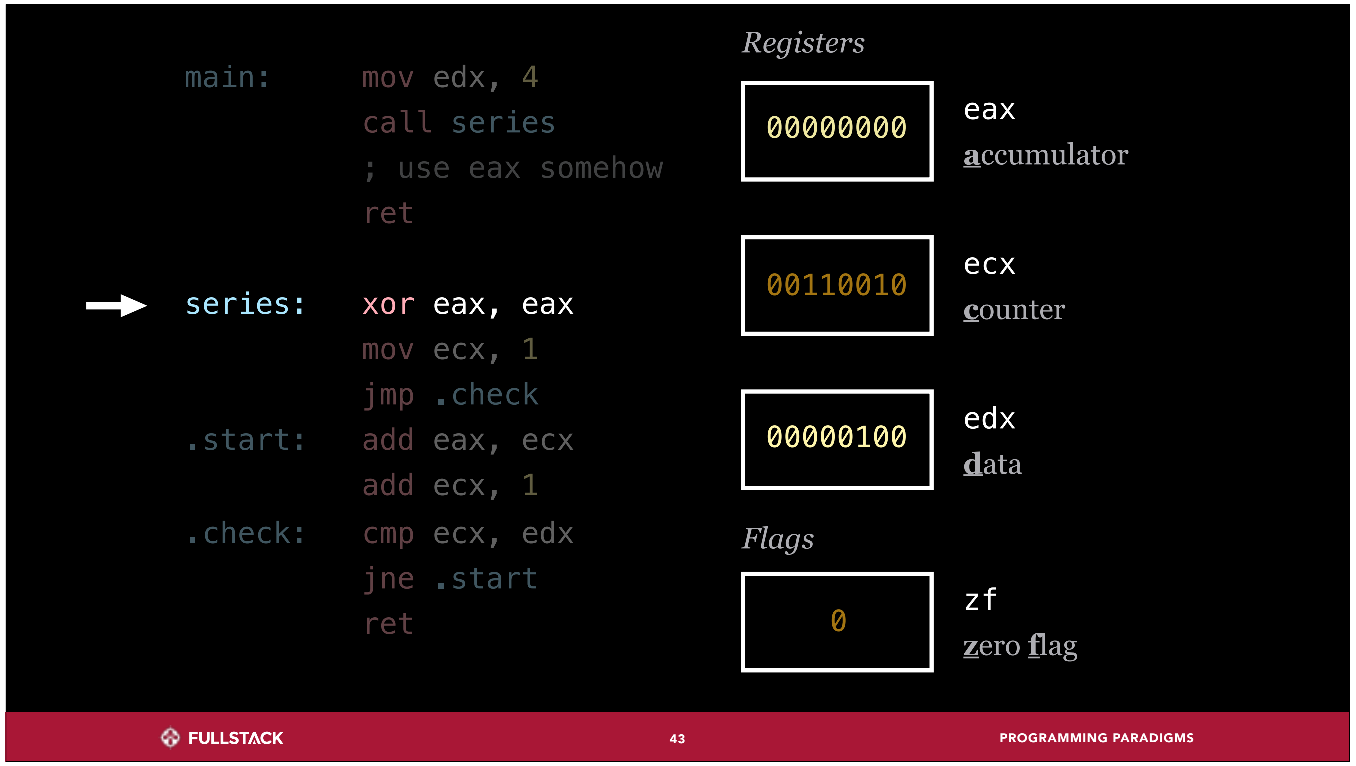
Registers

11001010	eax **a**ccumulator
00110010	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Zero out the accumulator register, which we will use as our "sum".



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

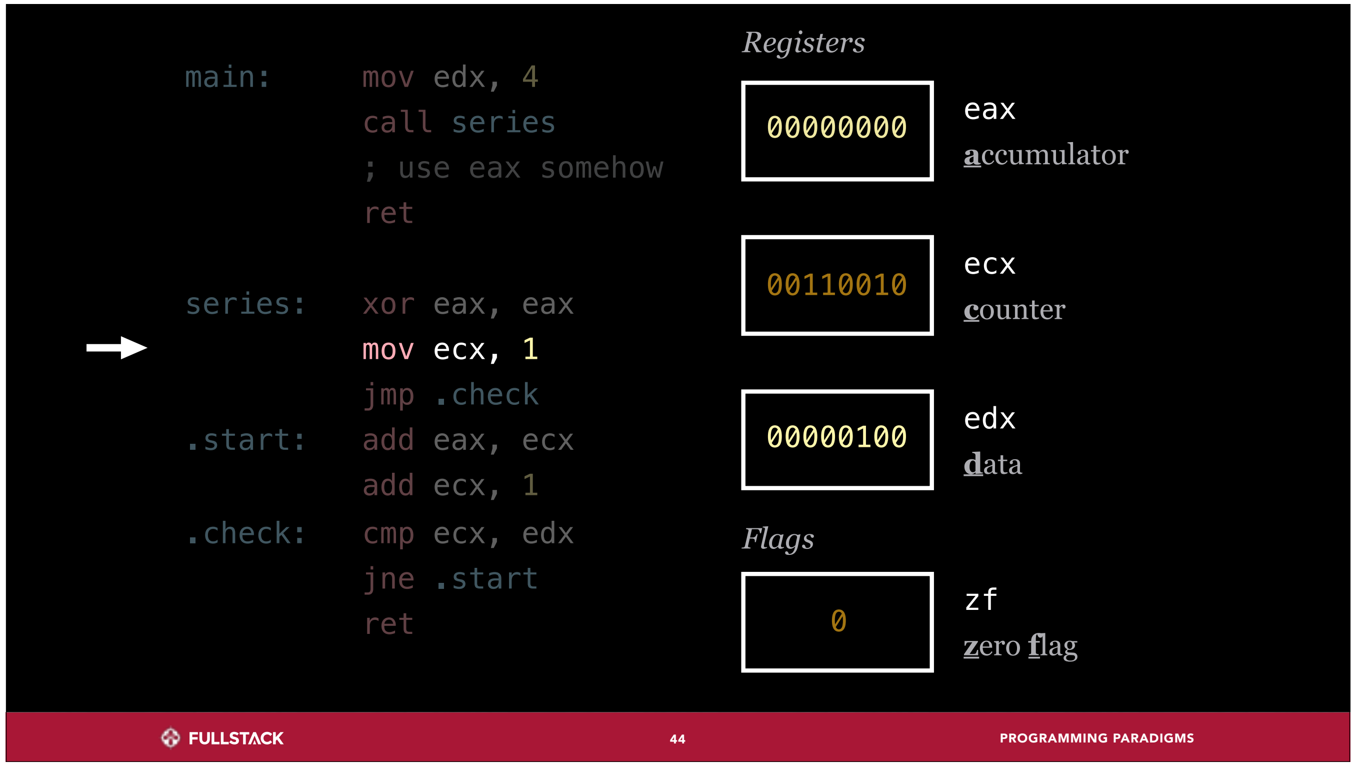
Registers

00000000	eax **a**ccumulator
00110010	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

`xor`-ing a value against itself yields 0. We could have also `mov`ed 0, but xor is faster.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

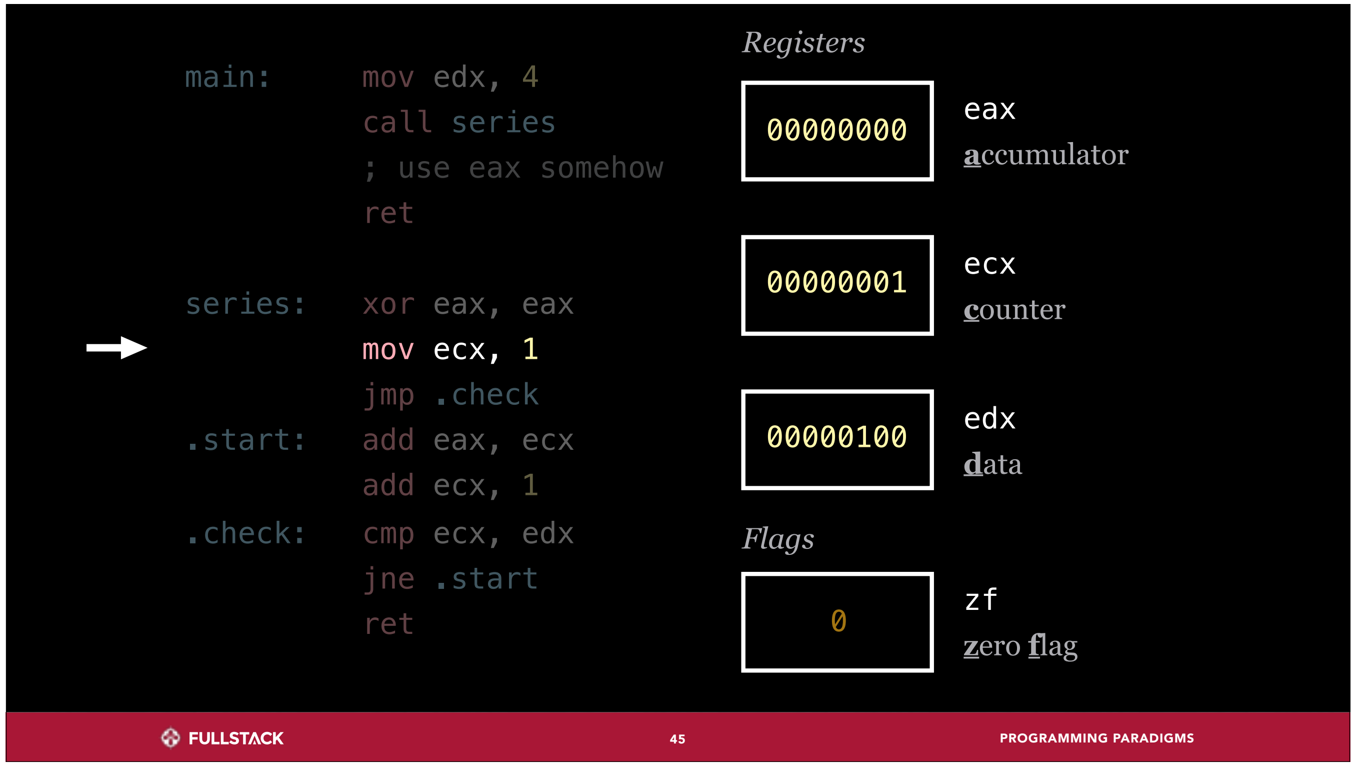
Registers

00000000	eax **a**ccumulator
00110010	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Move the number 1 into ecx (counter). Ecx will be our `i` value.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

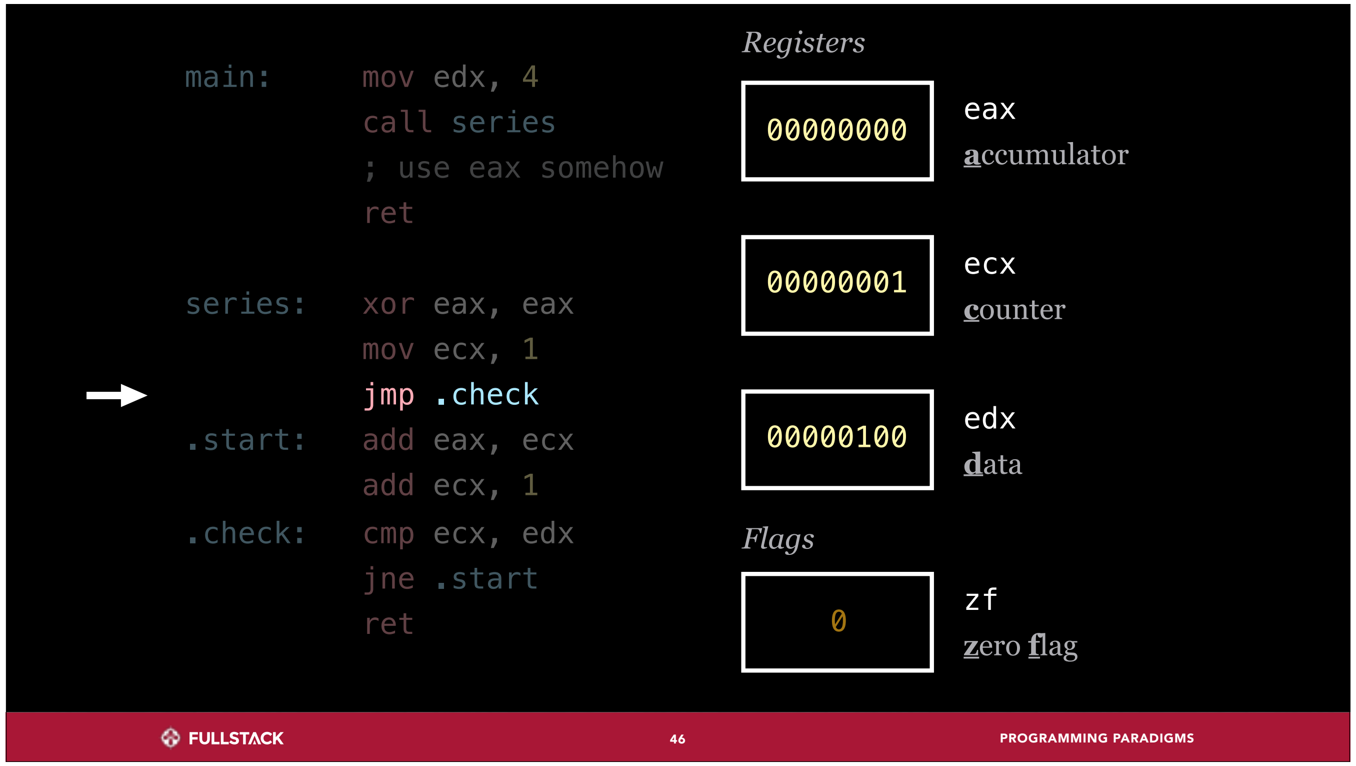
Registers

00000000	eax accumulator
00000001	ecx counter
00000100	edx data

Flags

0	zf zero flag

We initialize i to 1.



```
         main:     mov edx, 4
                   call series
                   ; use eax somehow
                   ret

         series:   xor eax, eax
                   mov ecx, 1
  →                jmp .check
         .start:   add eax, ecx
                   add ecx, 1
         .check:   cmp ecx, edx
                   jne .start
                   ret
```

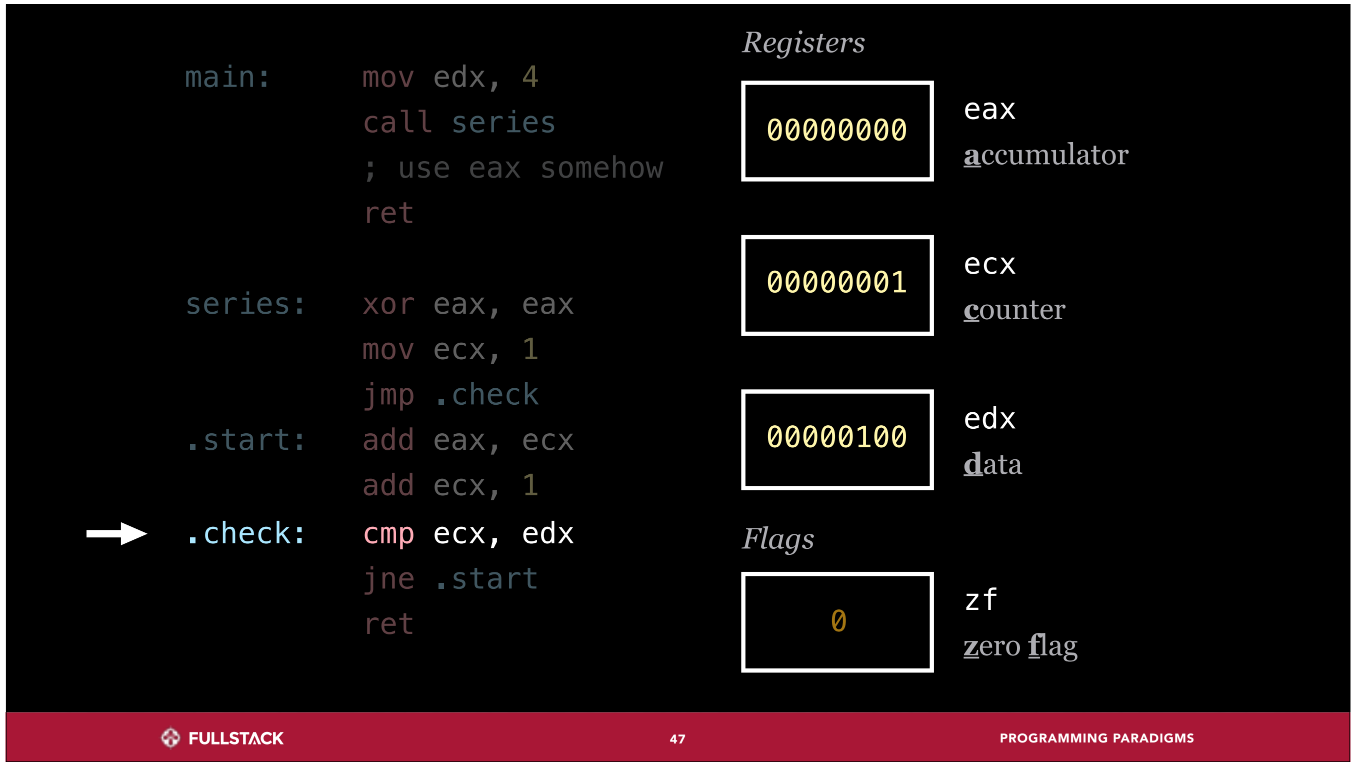
Registers

00000000	eax **a**ccumulator
00000001	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Jump commands, i.e. GOTO statements, move the program counter to a different address.



```
       main:     mov edx, 4
                 call series
                 ; use eax somehow
                 ret

       series:   xor eax, eax
                 mov ecx, 1
                 jmp .check
       .start:   add eax, ecx
                 add ecx, 1
    →  .check:   cmp ecx, edx
                 jne .start
                 ret
```

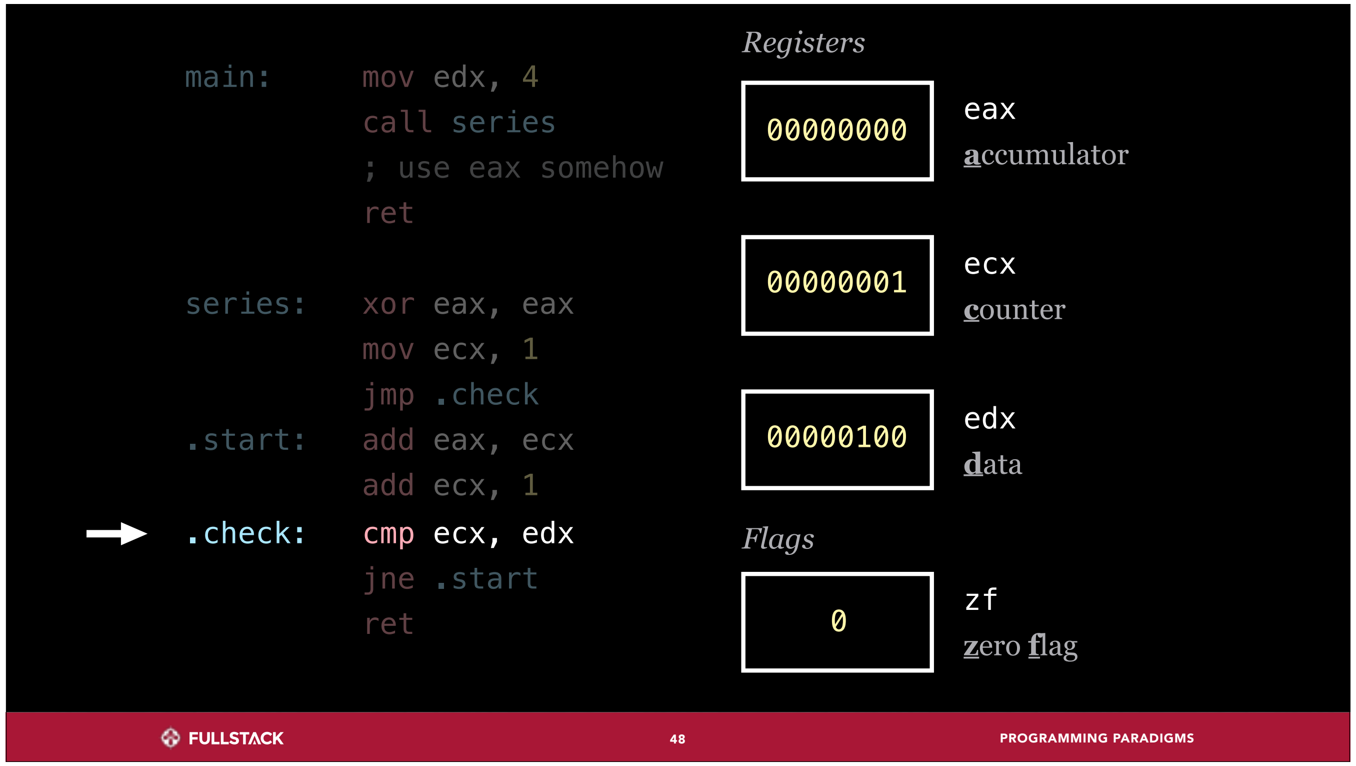
Registers

00000000	eax accumulator
00000001	ecx counter
00000100	edx data

Flags

0	zf zero flag

Here we compare (`cmp`) our counter and limit. If they are equal, the zero flag will be set to 1 (like true).



```
    main:     mov edx, 4
              call series
              ; use eax somehow
              ret

    series:   xor eax, eax
              mov ecx, 1
              jmp .check
    .start:   add eax, ecx
              add ecx, 1
→   .check:   cmp ecx, edx
              jne .start
              ret
```

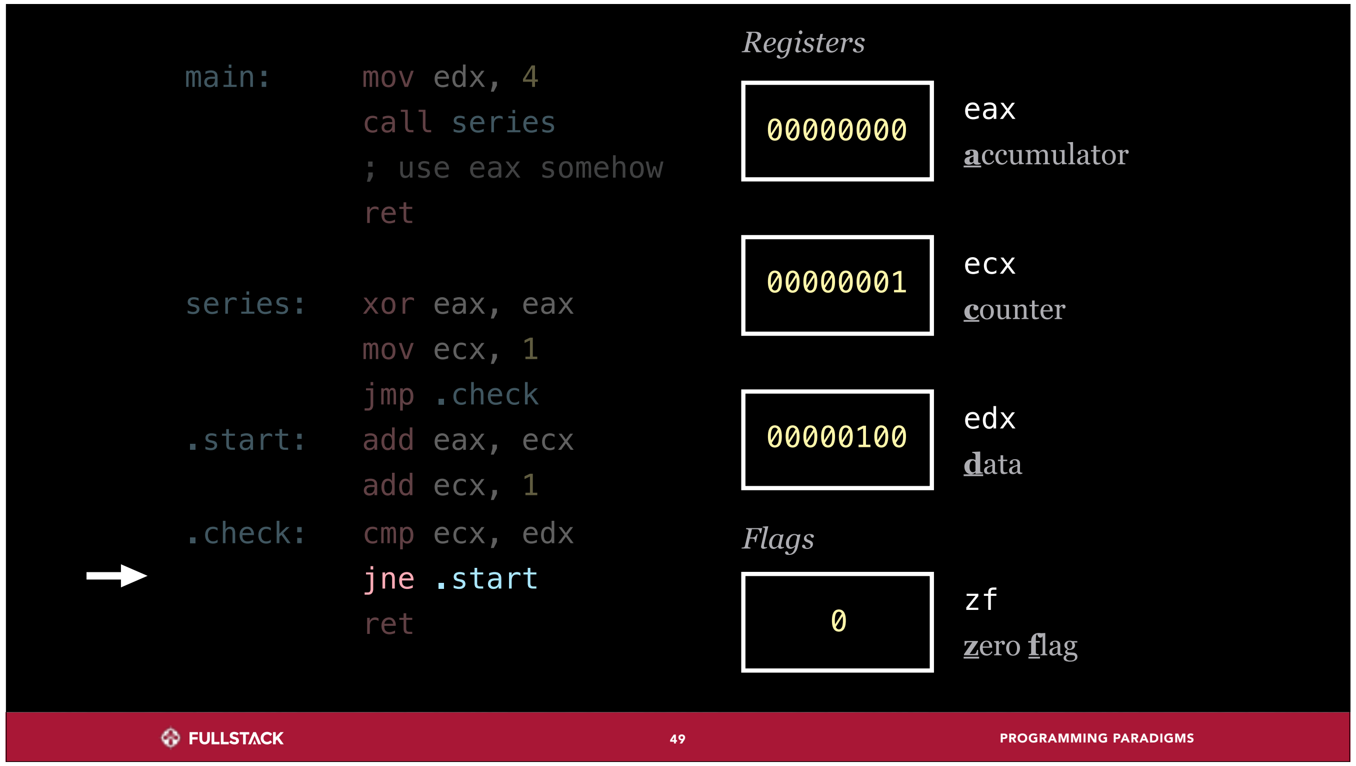
Registers

00000000	eax **a**ccumulator
00000001	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Not equal yet, so the flag is 0 (false, not equal).



```
        main:     mov edx, 4
                  call series
                  ; use eax somehow
                  ret

        series:   xor eax, eax
                  mov ecx, 1
                  jmp .check
        .start:   add eax, ecx
                  add ecx, 1
        .check:   cmp ecx, edx
  →               jne .start
                  ret
```

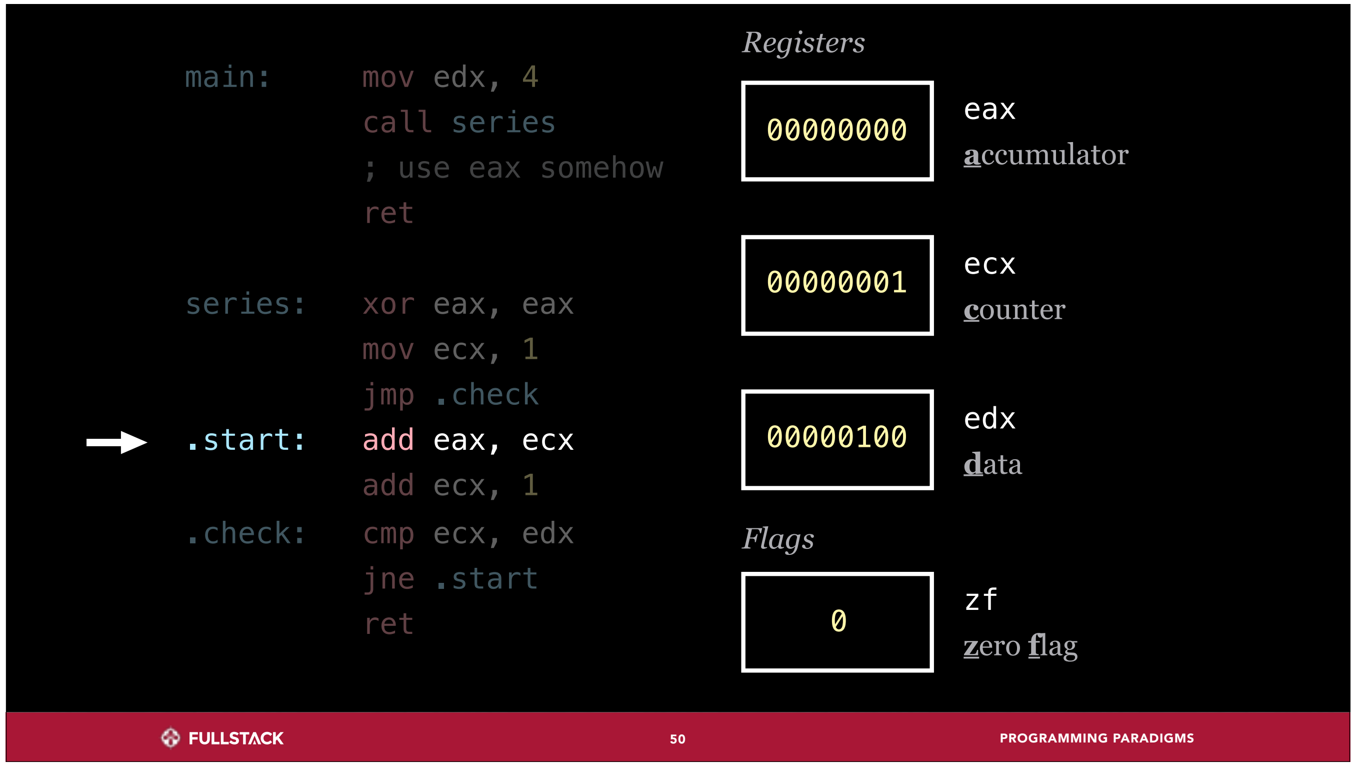
Registers

00000000	eax **a**ccumulator
00000001	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

This is a conditional jump (Jump if Not Equal). It jumps if ZF is still 0.



```
    main:     mov edx, 4
              call series
              ; use eax somehow
              ret

    series:   xor eax, eax
              mov ecx, 1
              jmp .check
→   .start:   add eax, ecx
              add ecx, 1
    .check:   cmp ecx, edx
              jne .start
              ret
```

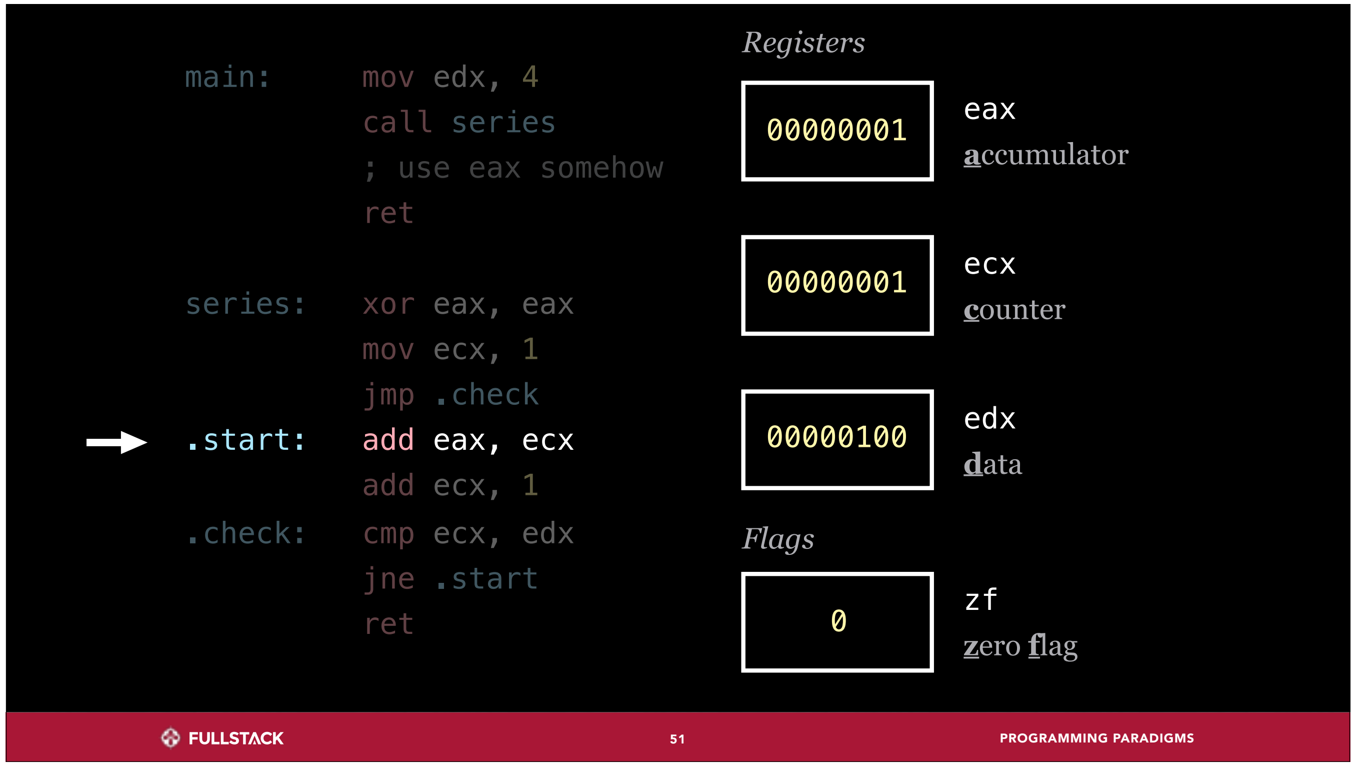
Registers

00000000	eax **a**ccumulator
00000001	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

The loop is allowed to progress! We add the counter to the sum.

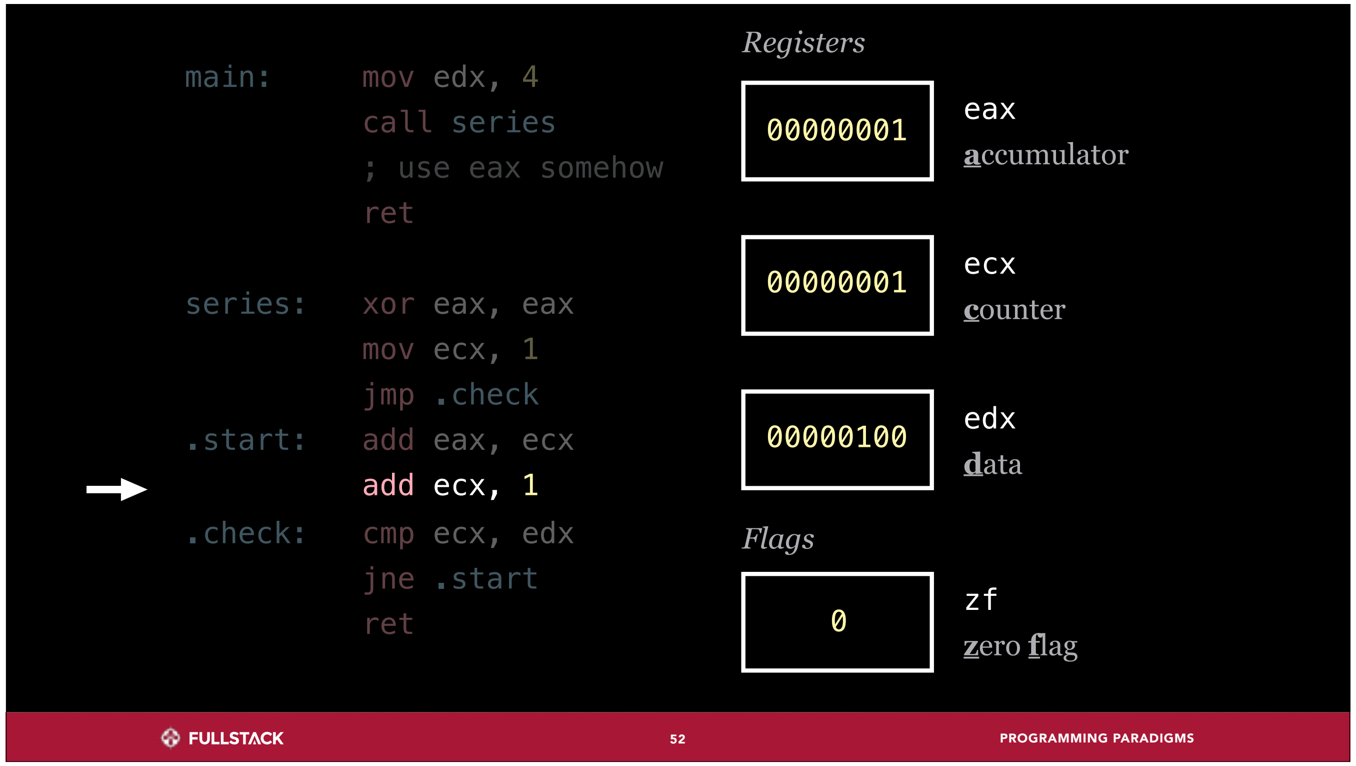


```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

Registers

Flags

Added.



```
    main:     mov edx, 4
              call series
              ; use eax somehow
              ret

    series:   xor eax, eax
              mov ecx, 1
              jmp .check
    .start:   add eax, ecx
→             add ecx, 1
    .check:   cmp ecx, edx
              jne .start
              ret
```

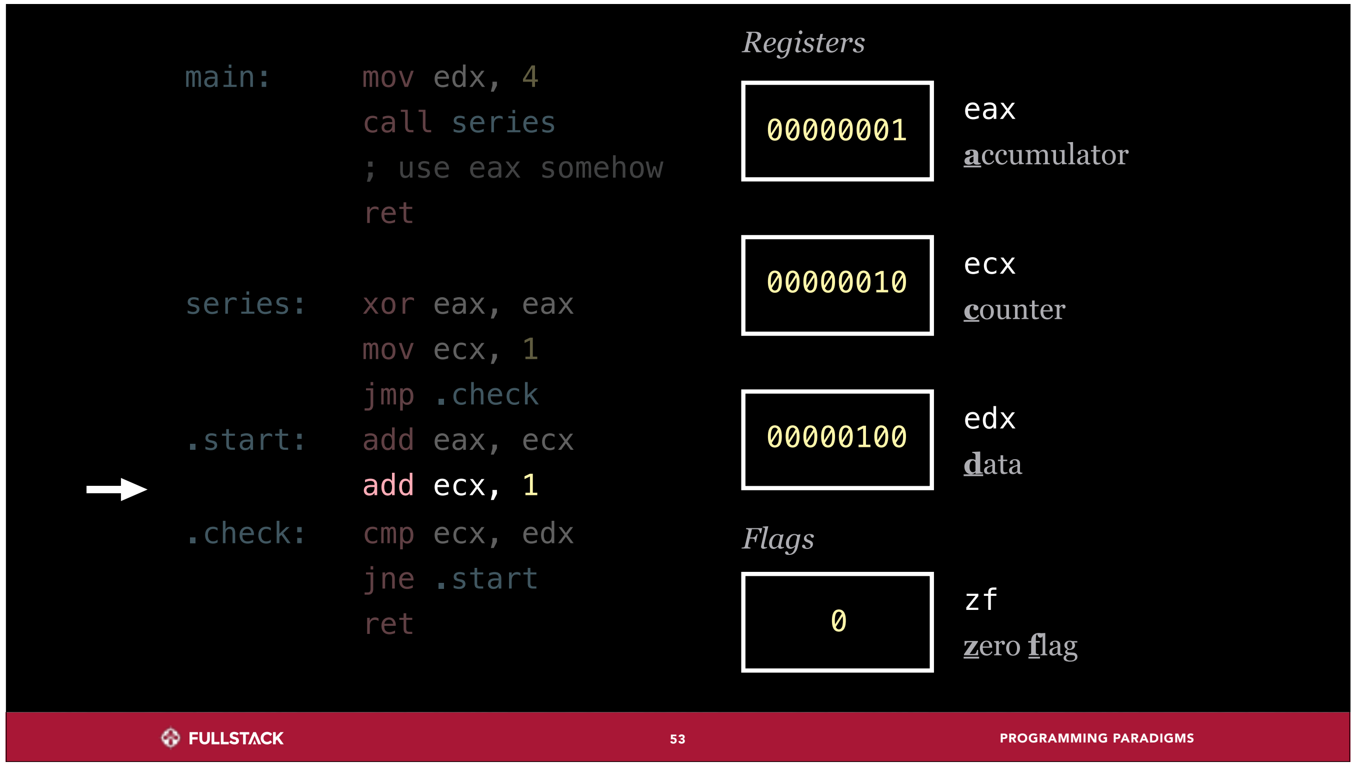
Registers

00000001	eax **a**ccumulator
00000001	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

The counter increments…



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
→         add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

Registers

`00000001` eax
accumulator

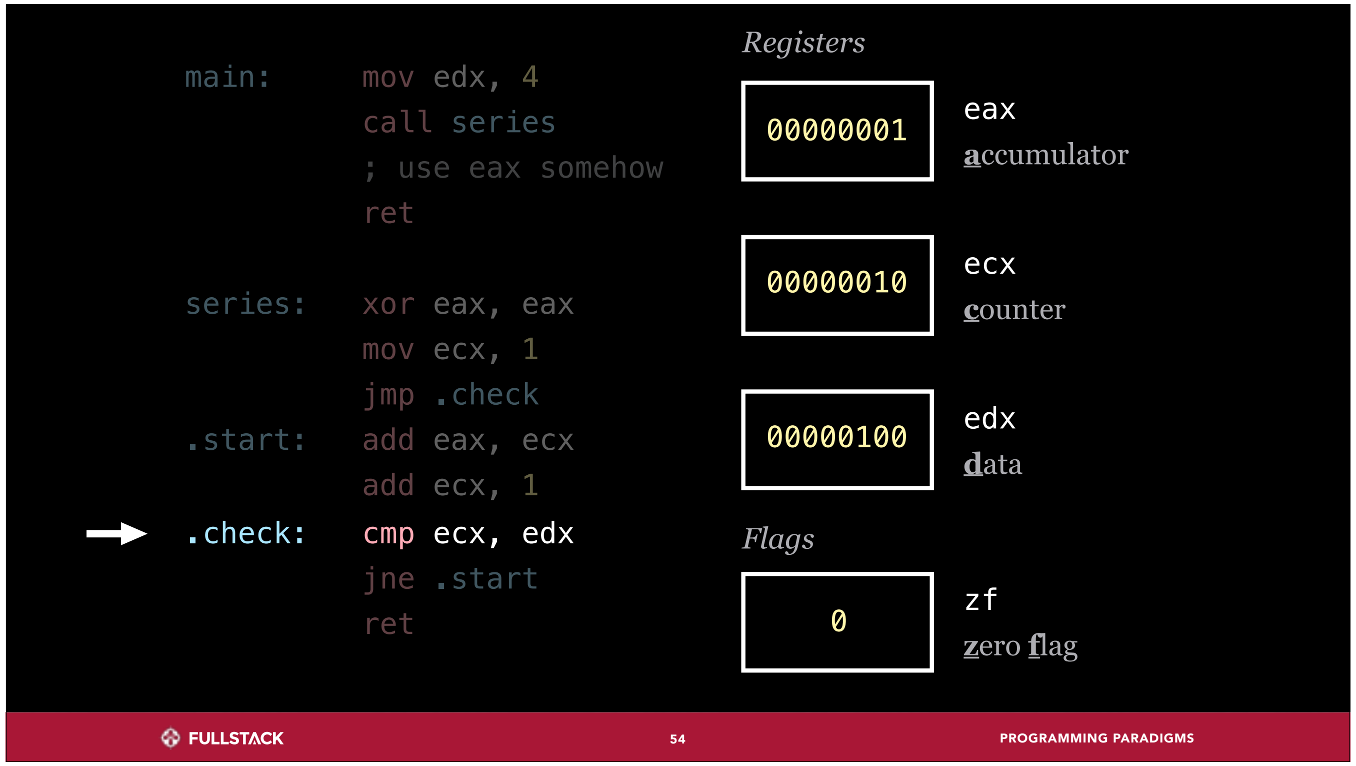
`00000010` ecx
counter

`00000100` edx
data

Flags

`0` zf
zero **f**lag

…to 2.



```
      main:     mov edx, 4
                call series
                ; use eax somehow
                ret

      series:   xor eax, eax
                mov ecx, 1
                jmp .check
      .start:   add eax, ecx
                add ecx, 1
  ➡   .check:   cmp ecx, edx
                jne .start
                ret
```

Registers

00000001 eax
accumulator

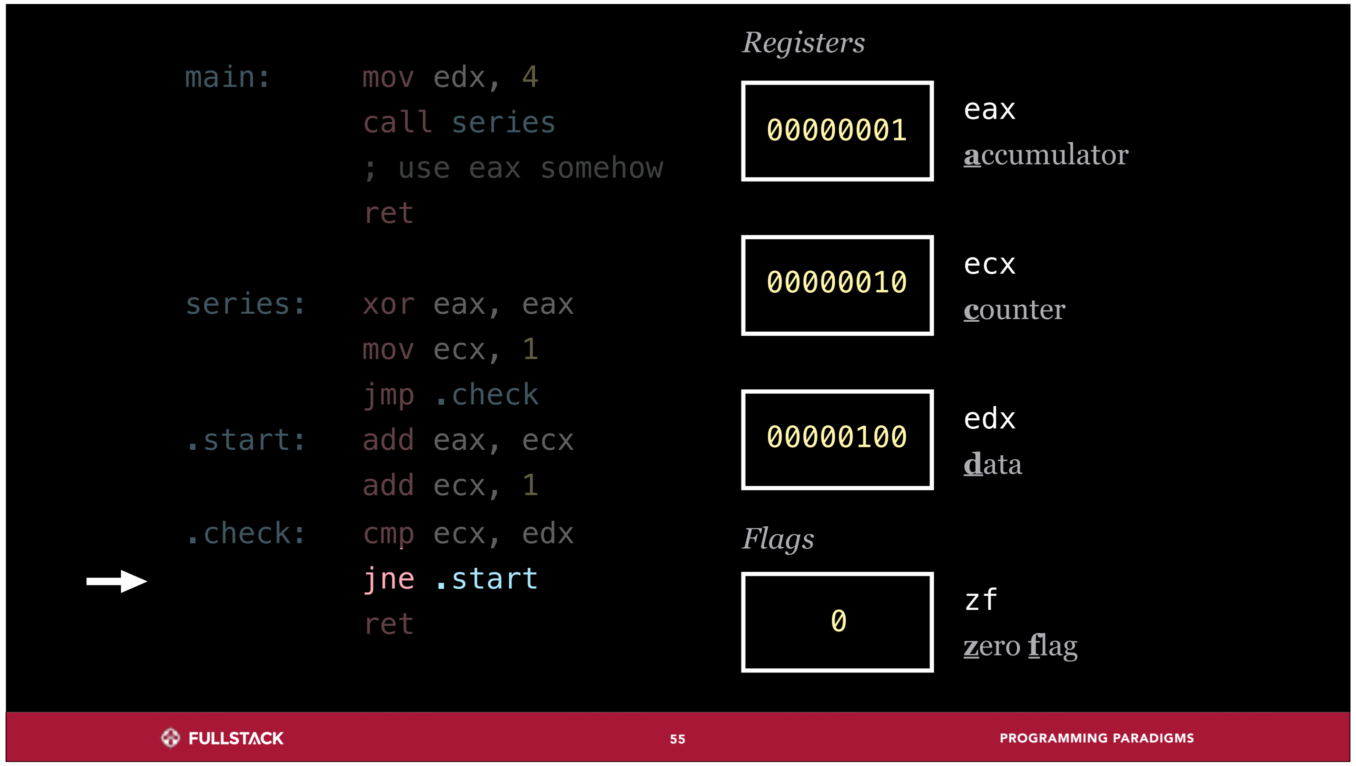
00000010 ecx
counter

00000100 edx
data

Flags

0 zf
zero **f**lag

Run the loop check again. Counter and limit are still not equal, so ZF is still 0…

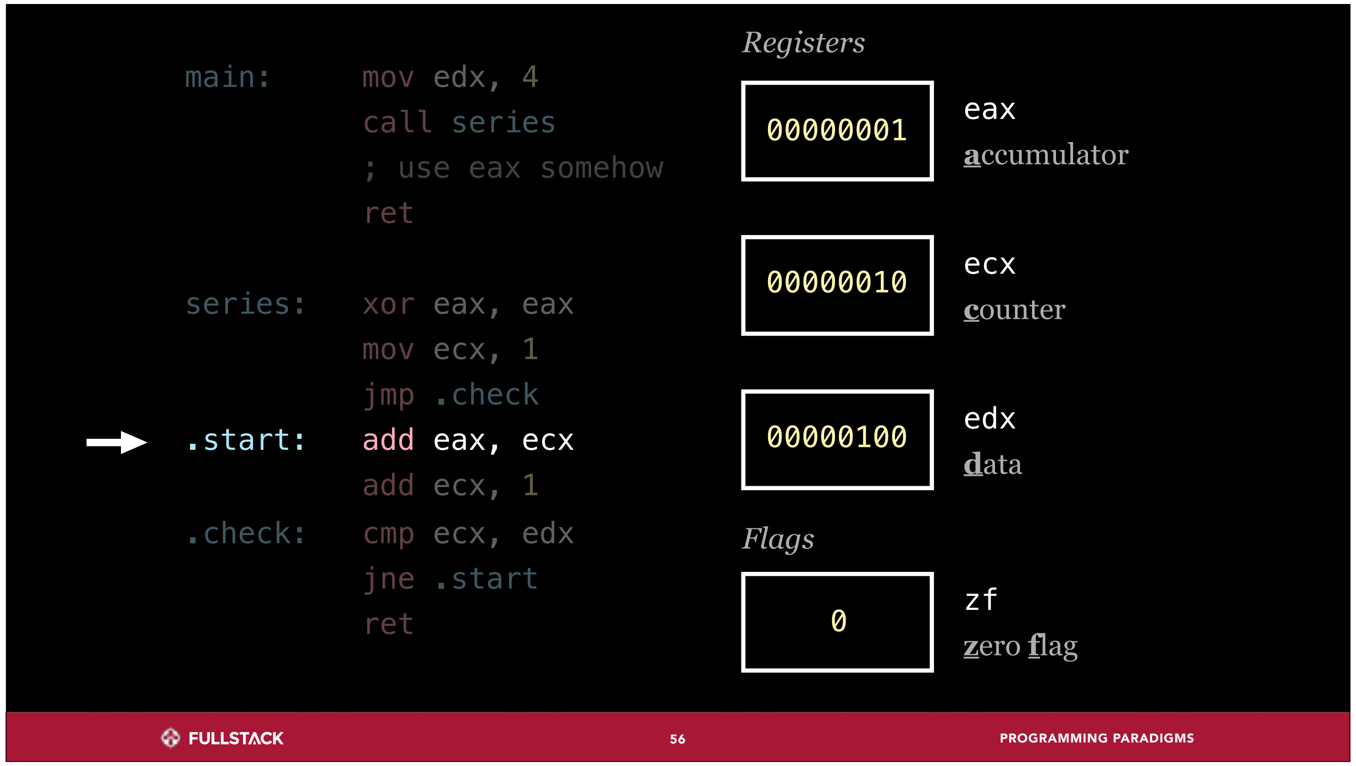


```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
→         jne .start
          ret
```

Registers

Flags

…so the jump occurs again.



```
   main:     mov edx, 4
             call series
             ; use eax somehow
             ret

   series:   xor eax, eax
             mov ecx, 1
             jmp .check
→  .start:   add eax, ecx
             add ecx, 1
   .check:   cmp ecx, edx
             jne .start
             ret
```

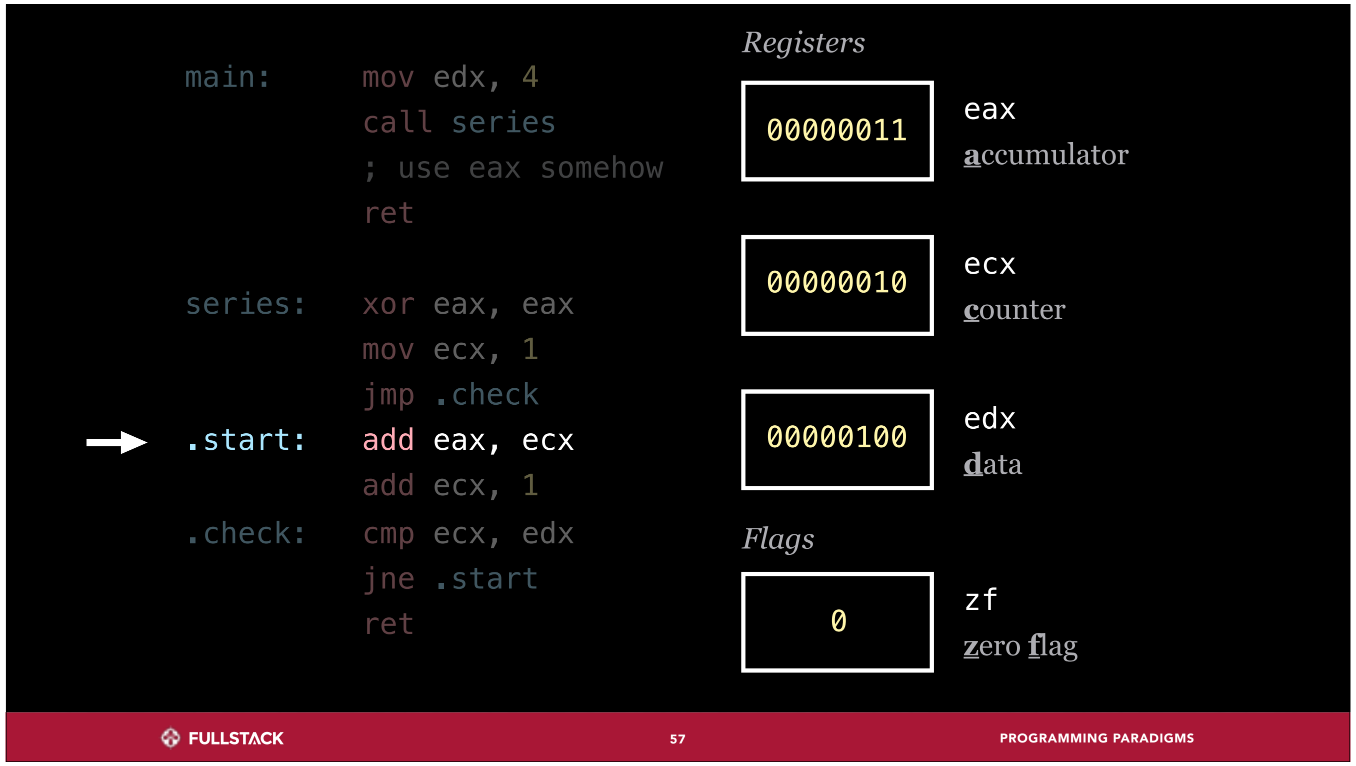
Registers

00000001	eax **a**ccumulator
00000010	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Adding i to sum…



```
    main:     mov edx, 4
              call series
              ; use eax somehow
              ret

    series:   xor eax, eax
              mov ecx, 1
              jmp .check
→   .start:   add eax, ecx
              add ecx, 1
    .check:   cmp ecx, edx
              jne .start
              ret
```

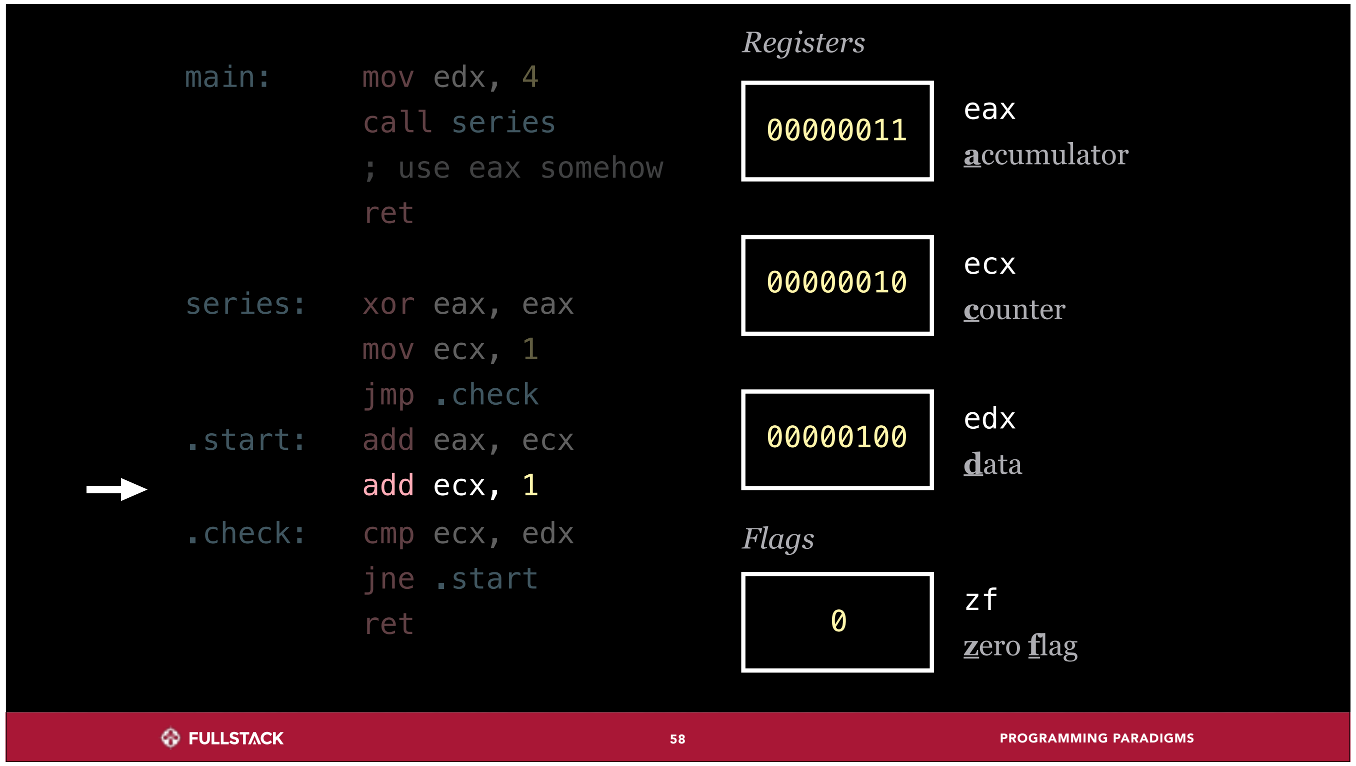
Registers

00000011	eax **a**ccumulator
00000010	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

…sum is now 3.



```
    main:     mov edx, 4
              call series
              ; use eax somehow
              ret

    series:   xor eax, eax
              mov ecx, 1
              jmp .check
    .start:   add eax, ecx
→             add ecx, 1
    .check:   cmp ecx, edx
              jne .start
              ret
```

Registers

00000011 eax
accumulator

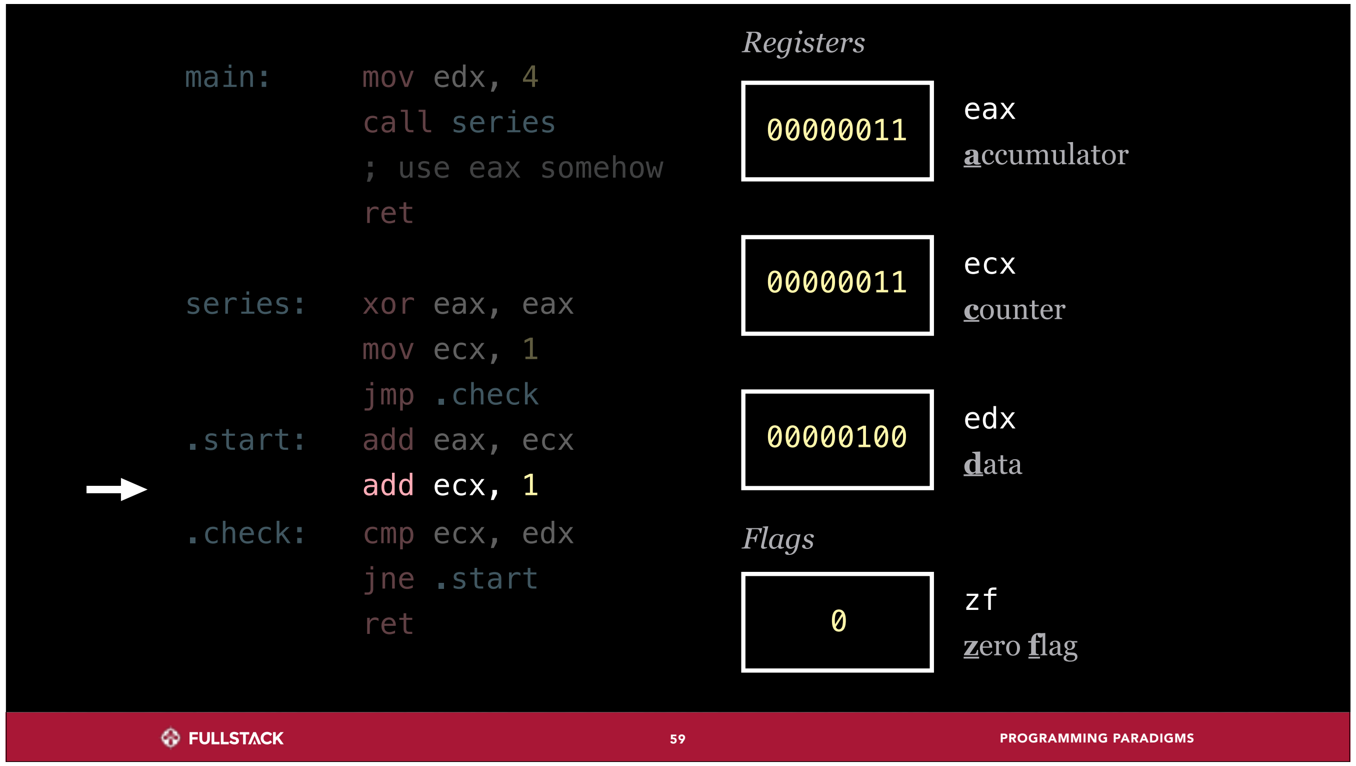
00000010 ecx
counter

00000100 edx
data

Flags

0 zf
zero **f**lag

Increment i…



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
→         add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

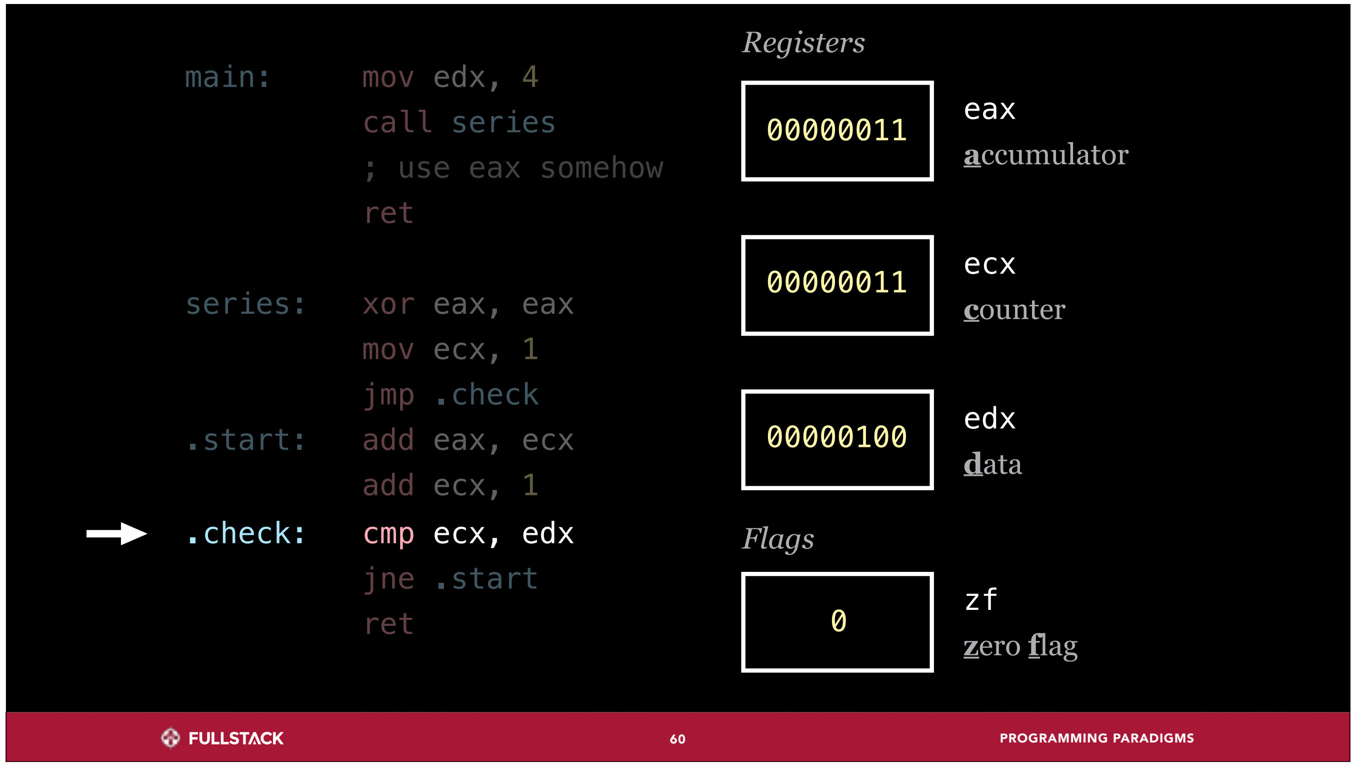
Registers

00000011	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

…to 3. There is also an `inc` command, but `add ___, 1` is faster.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

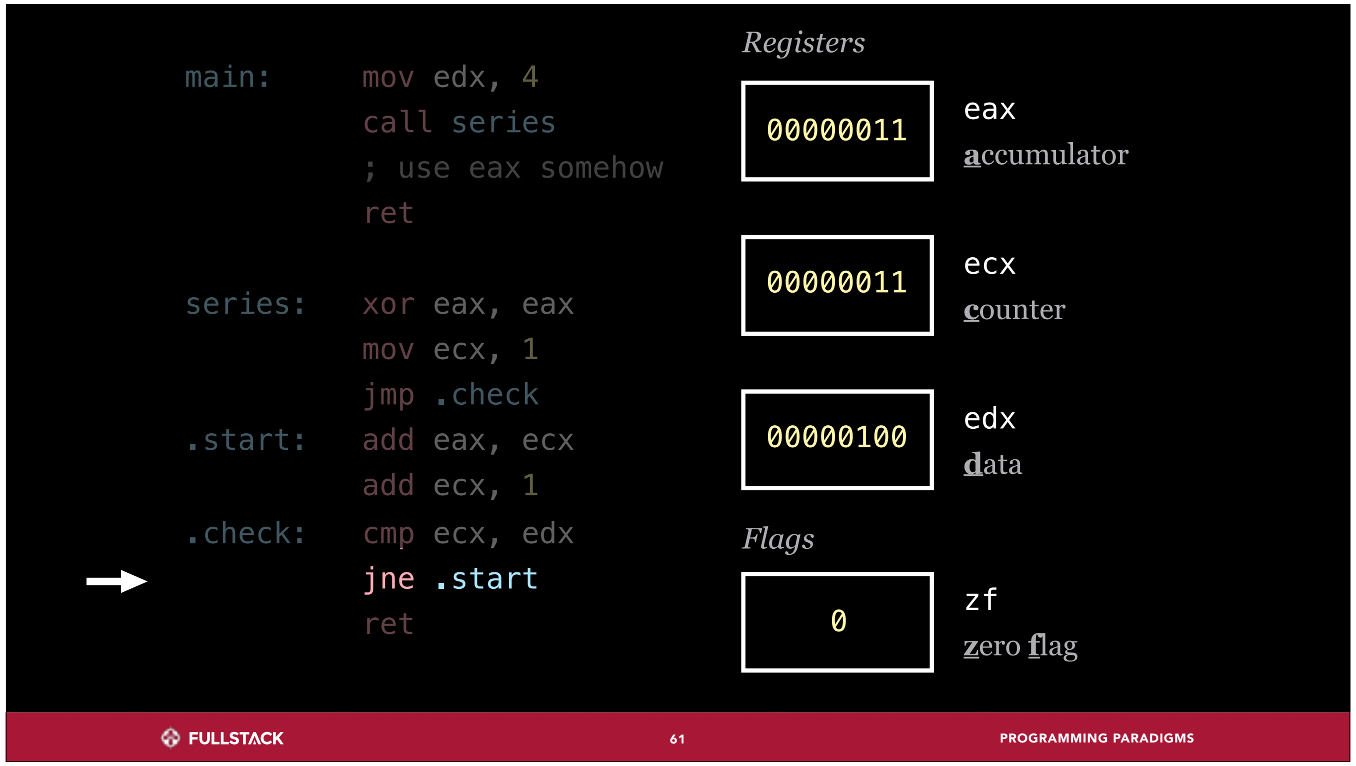
Registers

00000011	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Check if i == n. Not yet.



```
main:      mov edx, 4
           call series
           ; use eax somehow
           ret

series:    xor eax, eax
           mov ecx, 1
           jmp .check
.start:    add eax, ecx
           add ecx, 1
.check:    cmp ecx, edx
     →     jne .start
           ret
```

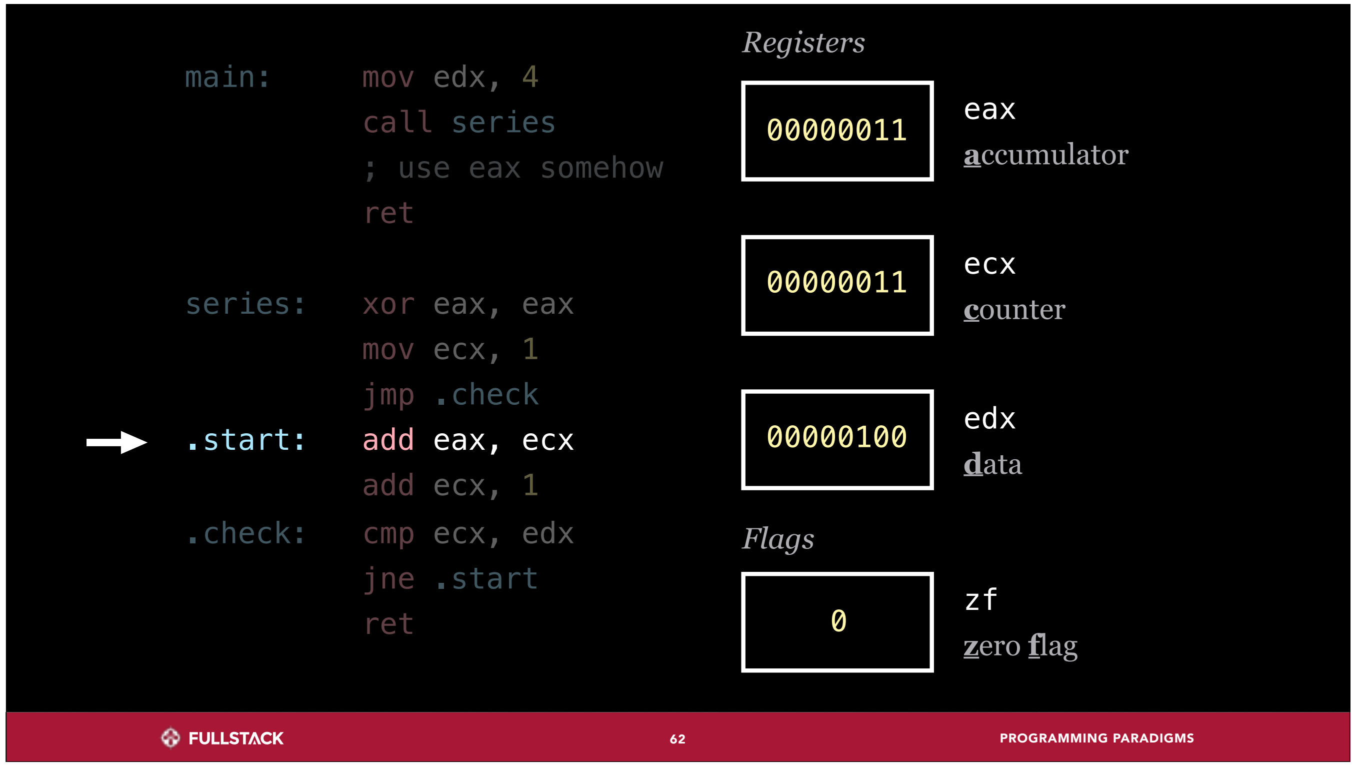
Registers

00000011	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

So jump again!



```
        main:     mov edx, 4
                  call series
                  ; use eax somehow
                  ret

        series:   xor eax, eax
                  mov ecx, 1
                  jmp .check
→       .start:   add eax, ecx
                  add ecx, 1
        .check:   cmp ecx, edx
                  jne .start
                  ret
```

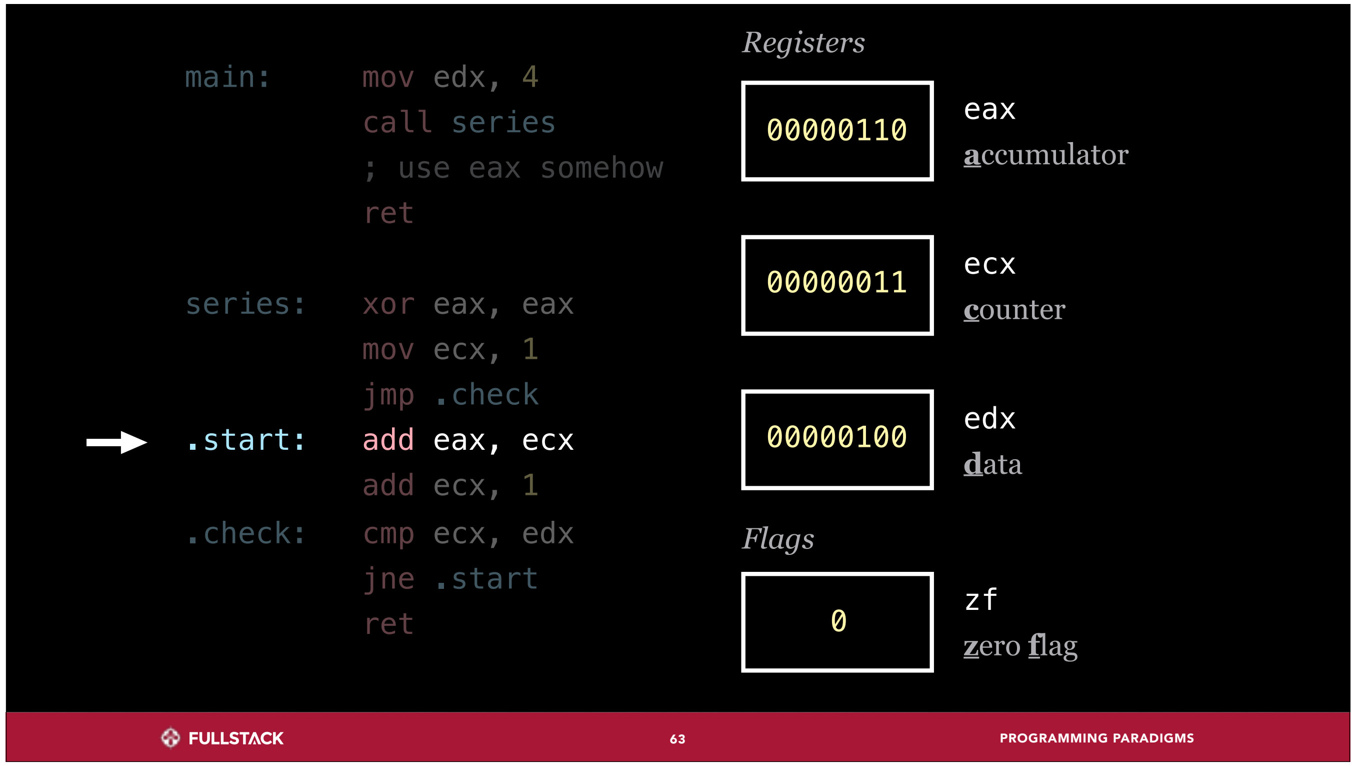
Registers

00000011	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Adding i to sum,



```
      main:     mov edx, 4
                call series
                ; use eax somehow
                ret

      series:   xor eax, eax
                mov ecx, 1
                jmp .check
 →    .start:   add eax, ecx
                add ecx, 1
      .check:   cmp ecx, edx
                jne .start
                ret
```

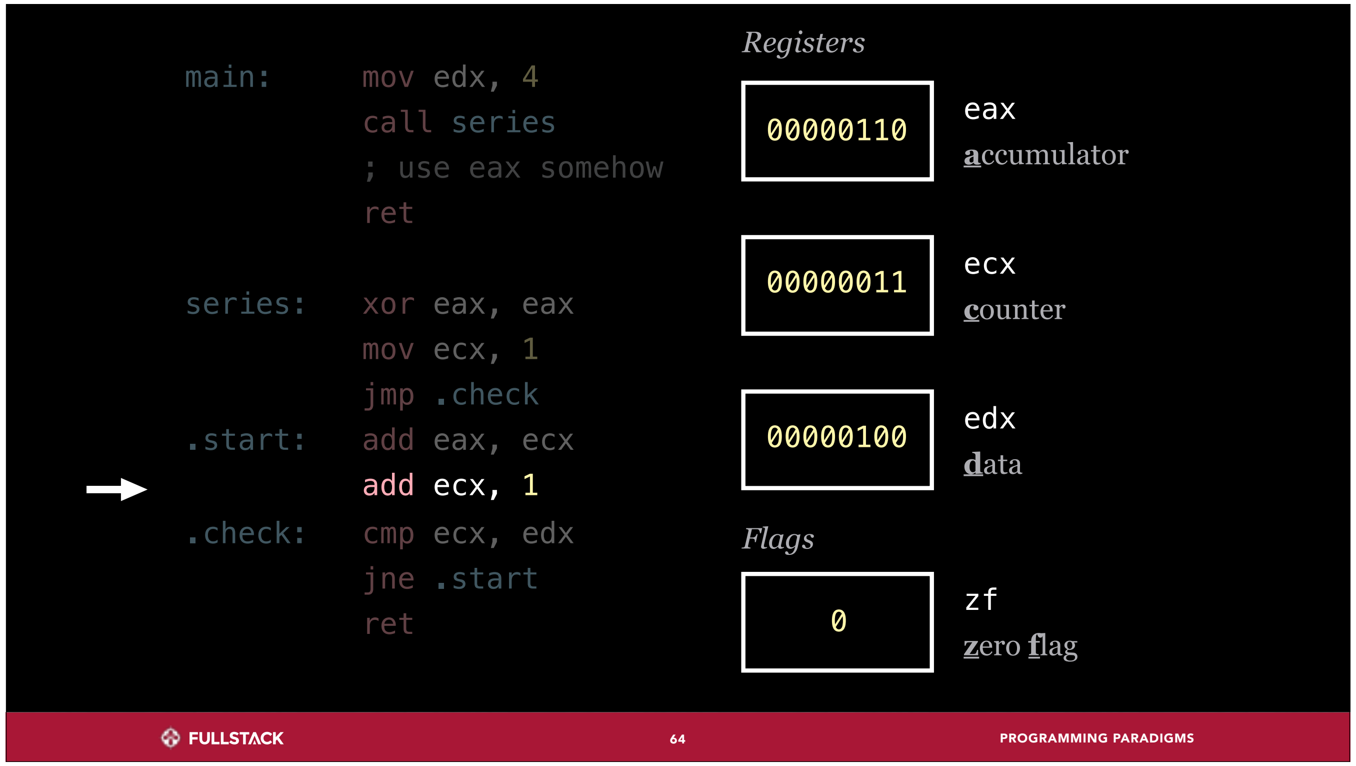
Registers

00000110	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

we now have 6 in sum.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

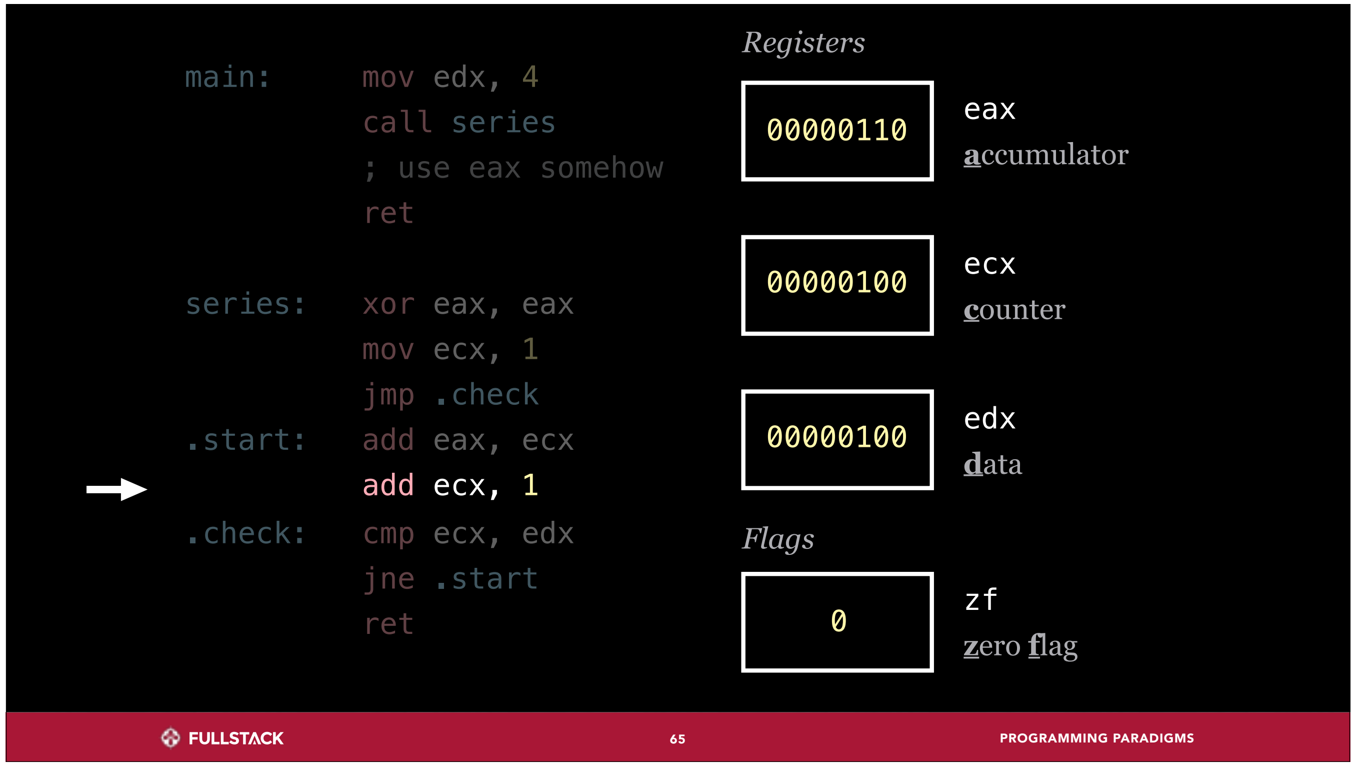
Registers

00000110	eax **a**ccumulator
00000011	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Incrementing i…



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
→         add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

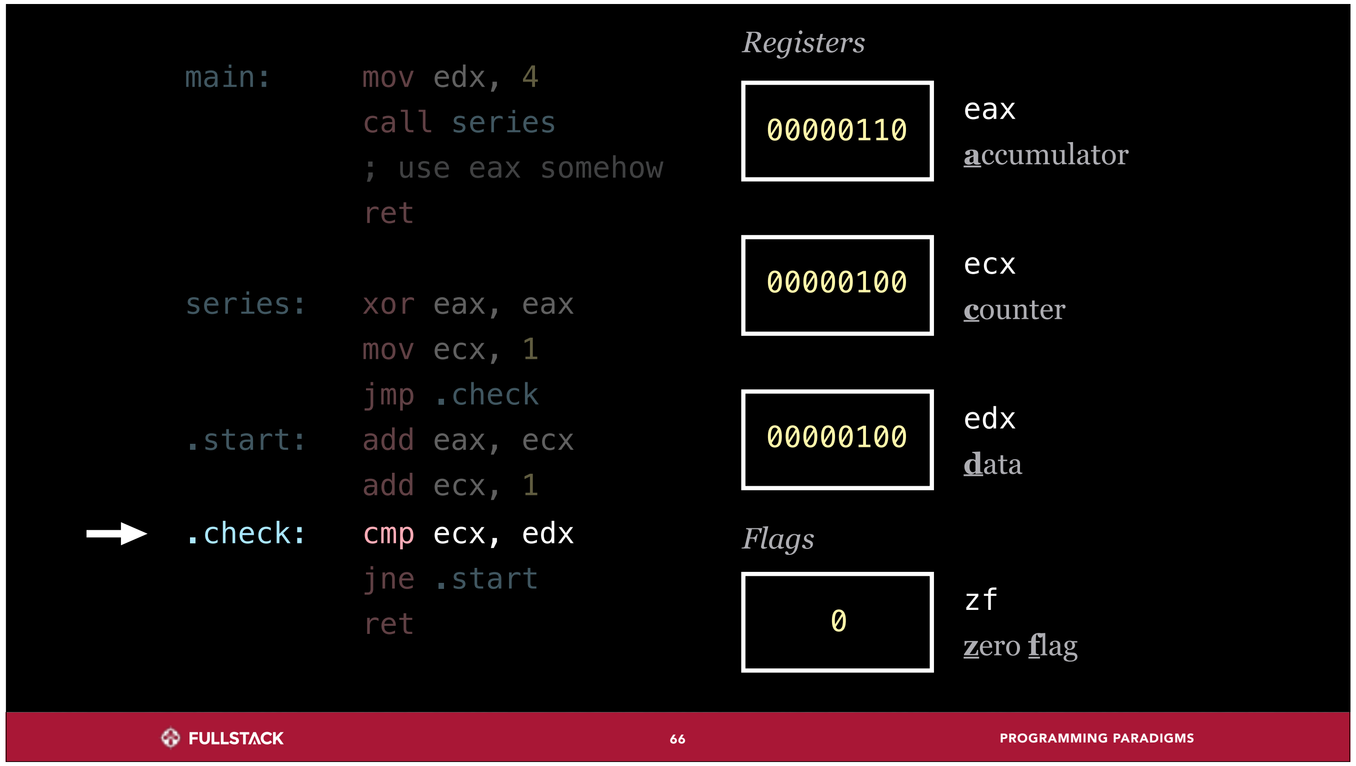
Registers

00000110	eax **a**ccumulator
00000100	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

which is now 4.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

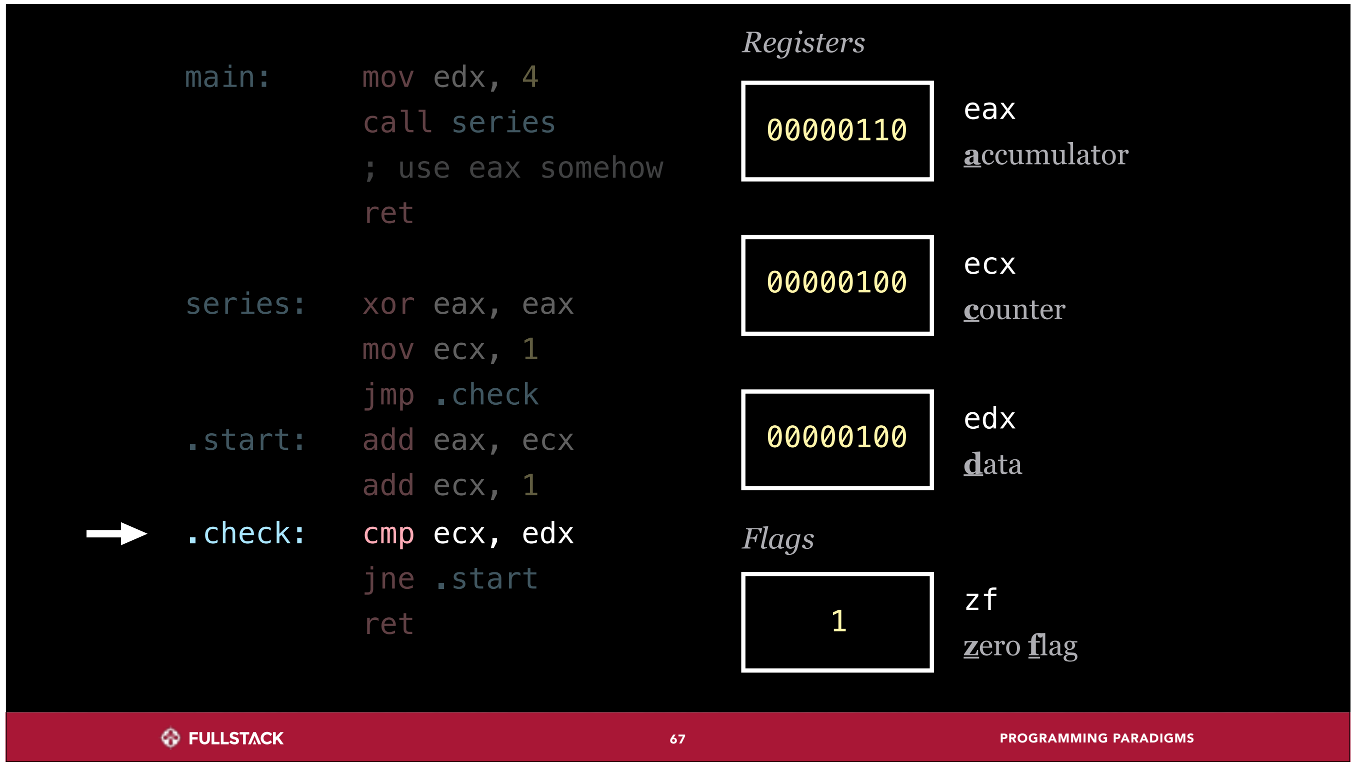
Registers

00000110	eax **a**ccumulator
00000100	ecx **c**ounter
00000100	edx **d**ata

Flags

0	zf **z**ero **f**lag

Uh oh, i == n, so…



```
     main:     mov edx, 4
               call series
               ; use eax somehow
               ret

     series:   xor eax, eax
               mov ecx, 1
               jmp .check
     .start:   add eax, ecx
               add ecx, 1
  →  .check:   cmp ecx, edx
               jne .start
               ret
```

Registers

00000110 eax
accumulator

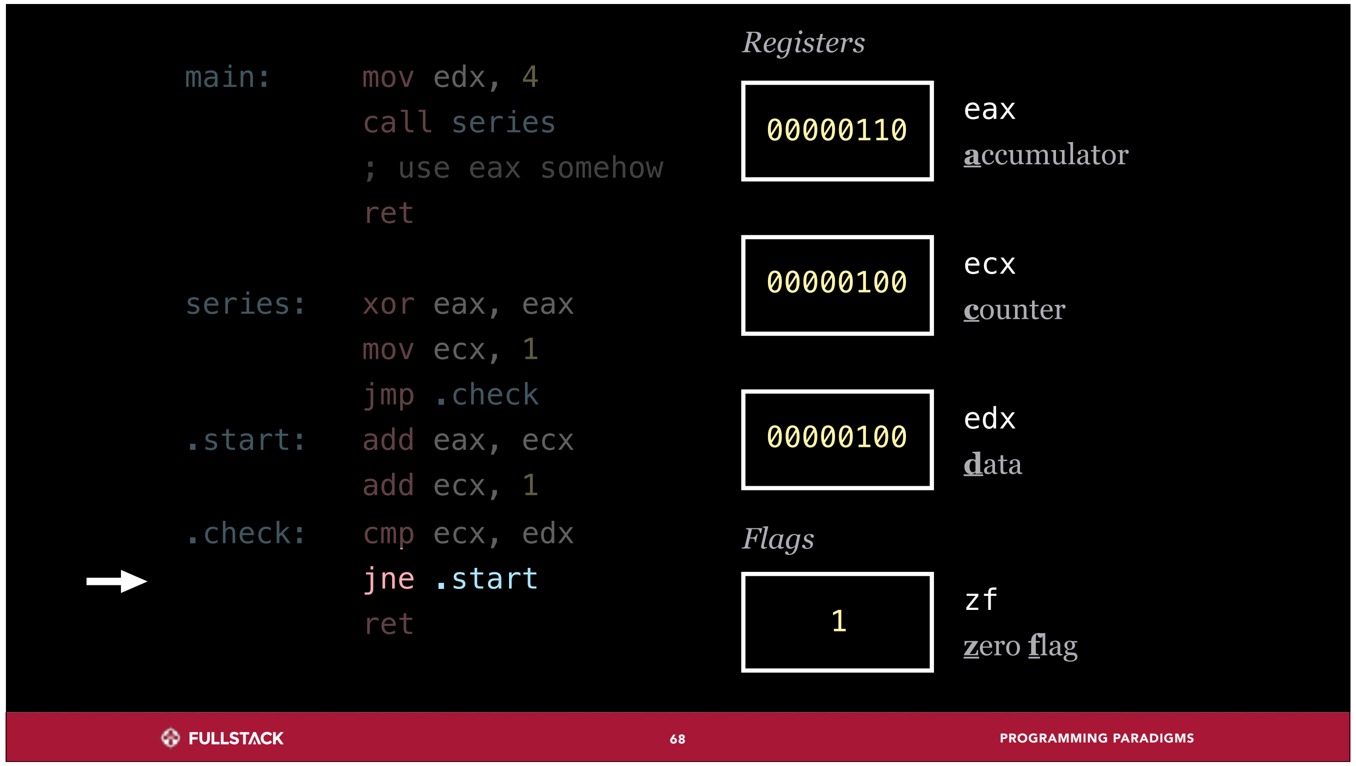
00000100 ecx
counter

00000100 edx
data

Flags

1 zf
zero flag

…the zero flag is turned on.



```
        main:     mov edx, 4
                  call series
                  ; use eax somehow
                  ret

        series:   xor eax, eax
                  mov ecx, 1
                  jmp .check
        .start:   add eax, ecx
                  add ecx, 1
        .check:   cmp ecx, edx
  →               jne .start
                  ret
```

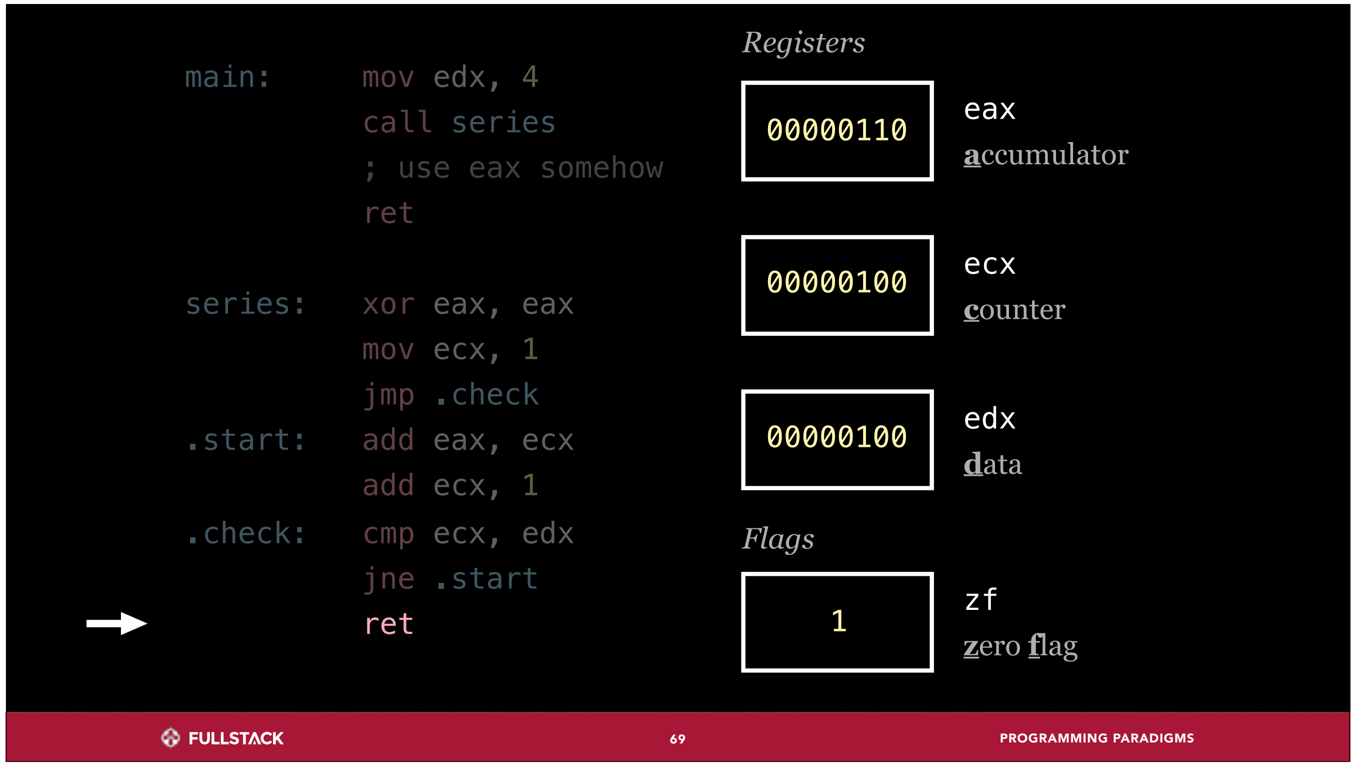
Registers

00000110	eax **a**ccumulator
00000100	ecx **c**ounter
00000100	edx **d**ata

Flags

1	zf **z**ero **f**lag

This time the jump doesn't occur!



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

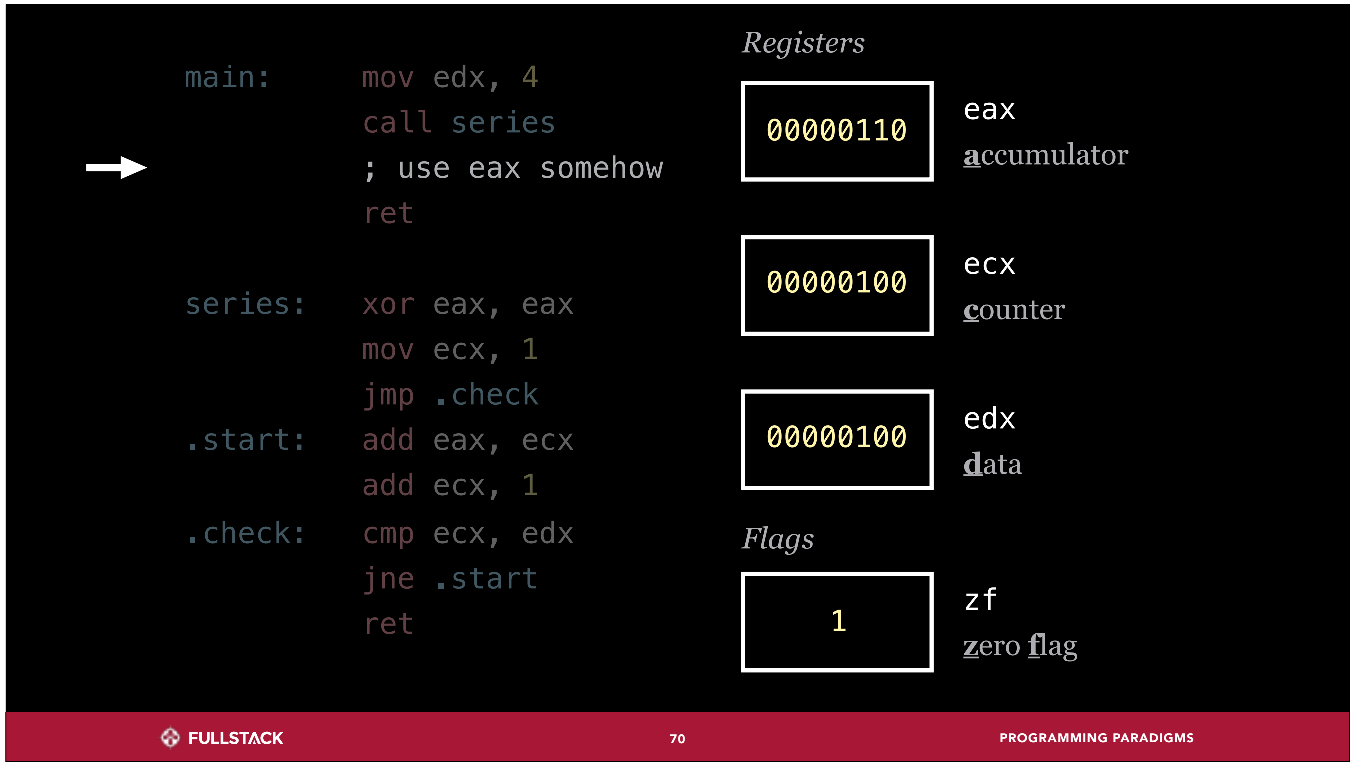
Registers

00000110	eax **a**ccumulator
00000100	ecx **c**ounter
00000100	edx **d**ata

Flags

1	zf **z**ero **f**lag

And our series function ends.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

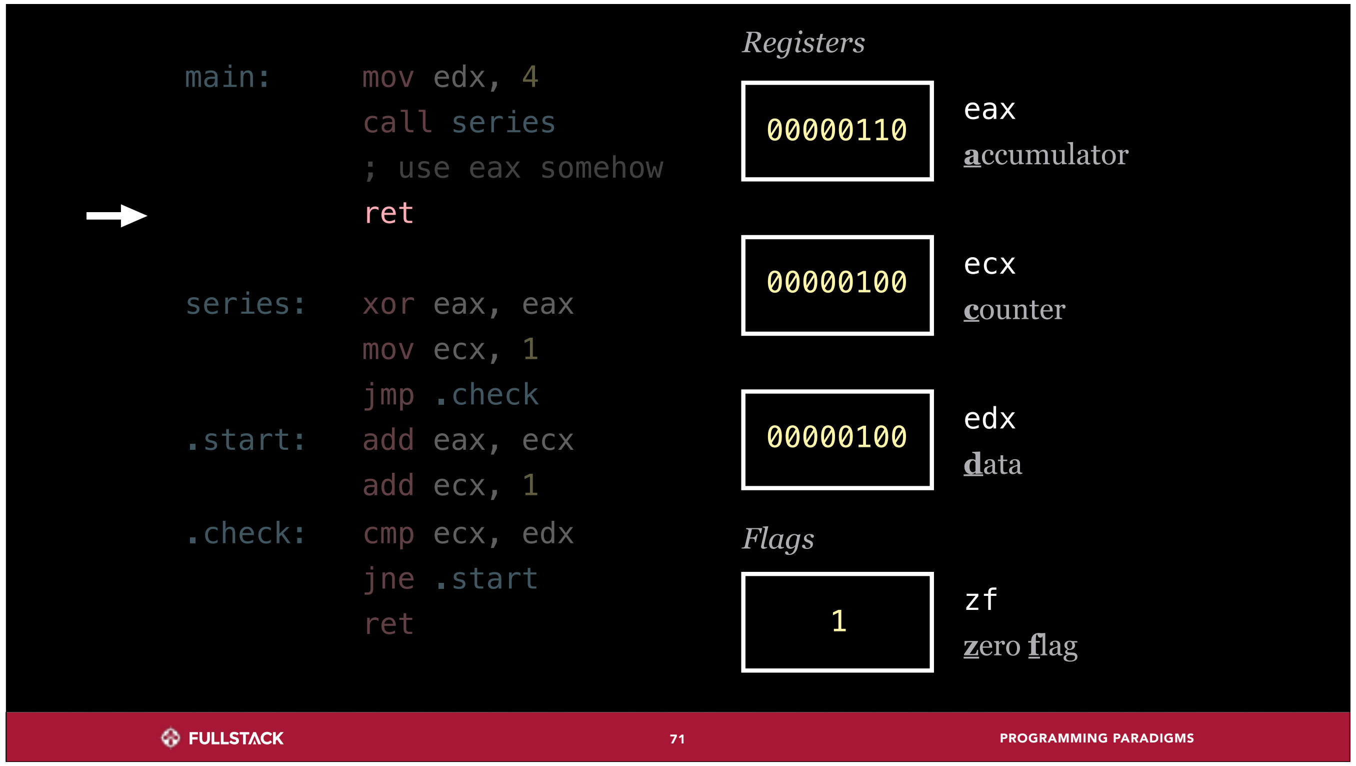
Registers

00000110	eax accumulator
00000100	ecx counter
00000100	edx data

Flags

1	zf zero flag

We set the accumulator register with the results of the function call, so our calling function can use it. Maybe it'll print it to STDOUT? Who knows.



```
main:     mov edx, 4
          call series
          ; use eax somehow
          ret

series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
```

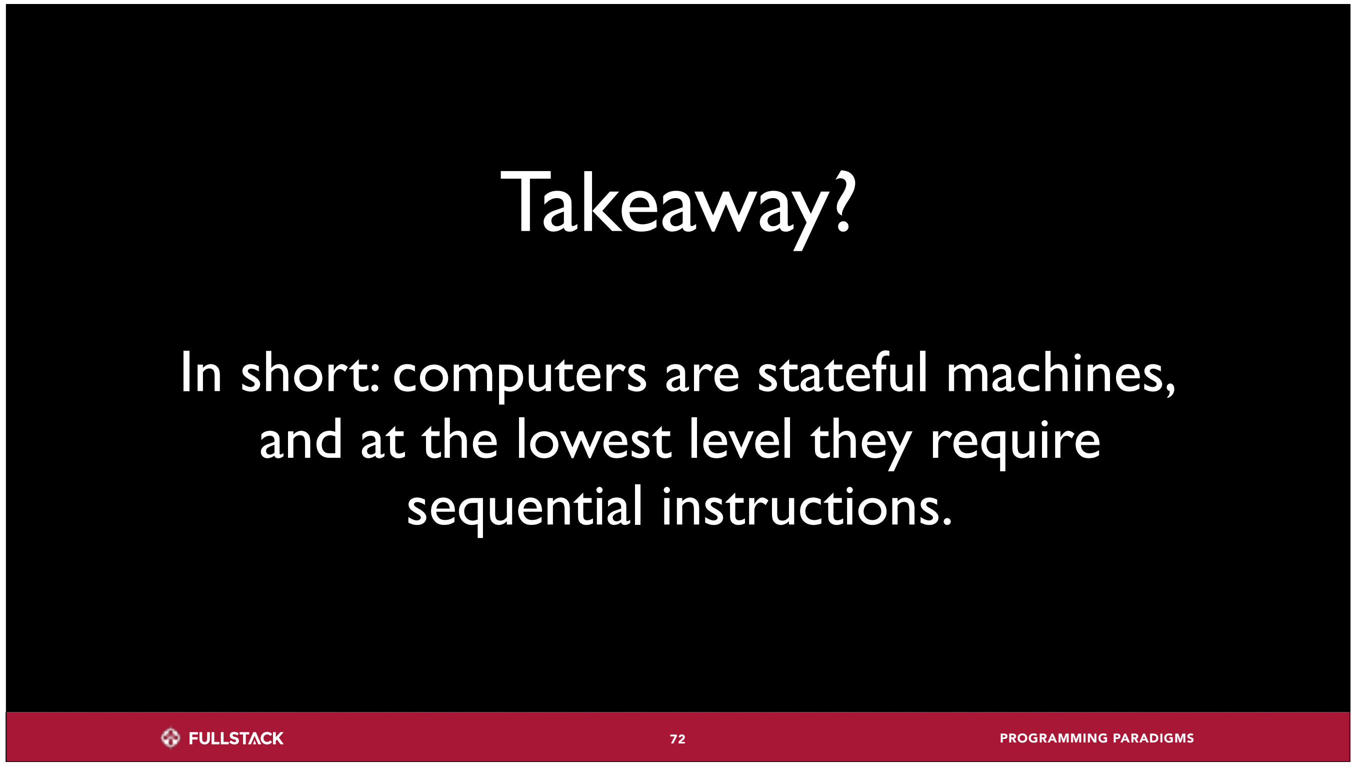
Registers

00000110	eax **a**ccumulator
00000100	ecx **c**ounter
00000100	edx **d**ata

Flags

1	zf **z**ero **f**lag

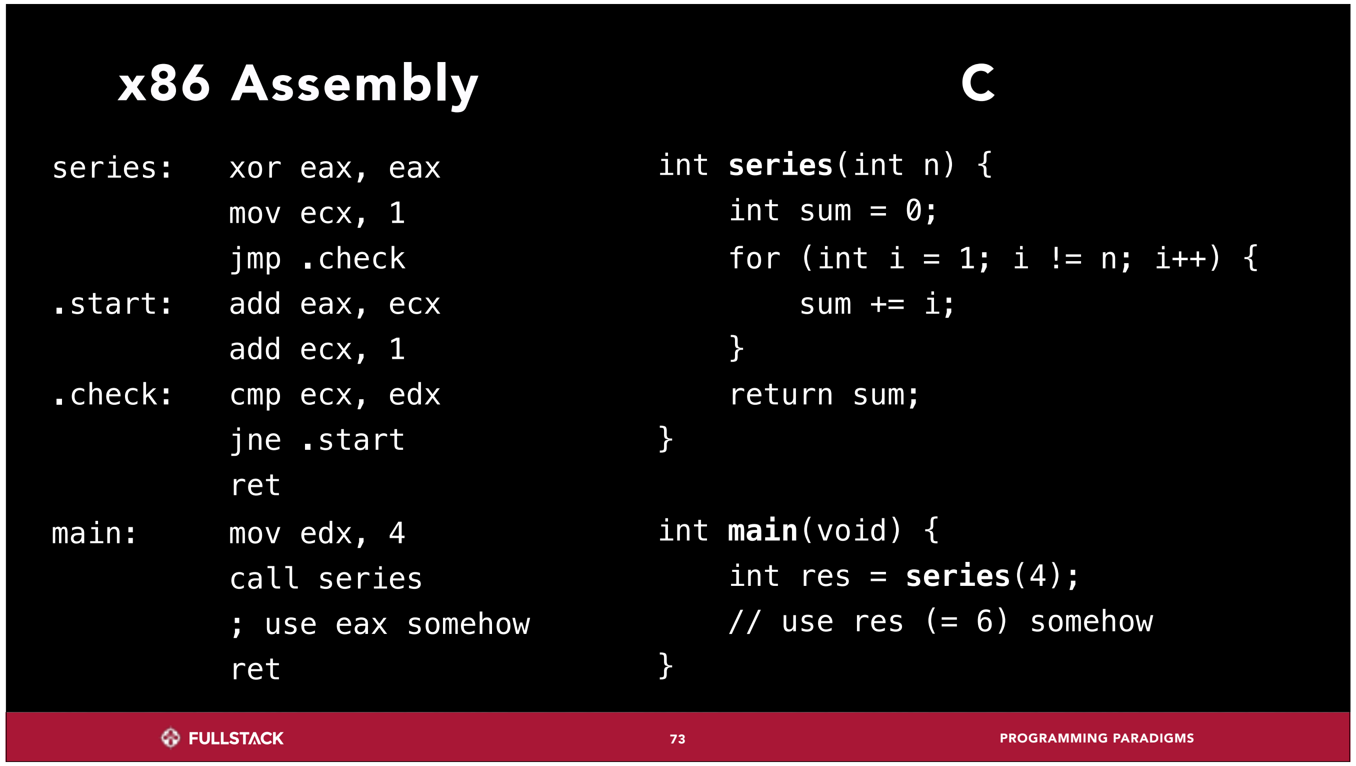
Eventually, the program terminates, yielding control to the OS.



Takeaway?

In short: computers are stateful machines, and at the lowest level they require sequential instructions.

Similarities:
- You can (and do) mutate stateful memory.
- You do things in a particular order – moving statements changes the meaning of the program.



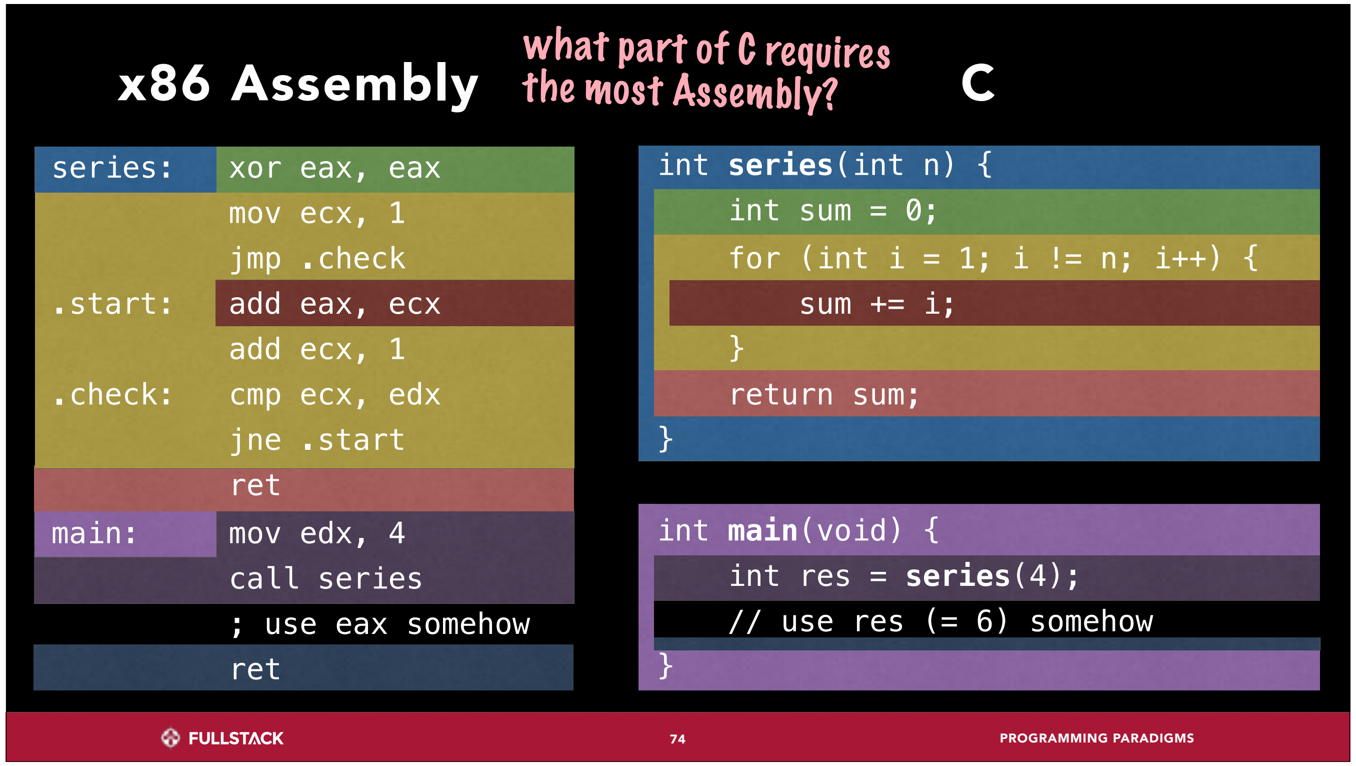
x86 Assembly

```
series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

C

```
int series(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = series(4);
    // use res (= 6) somehow
}
```

So what's the point of showing you Assembly?



x86 Assembly

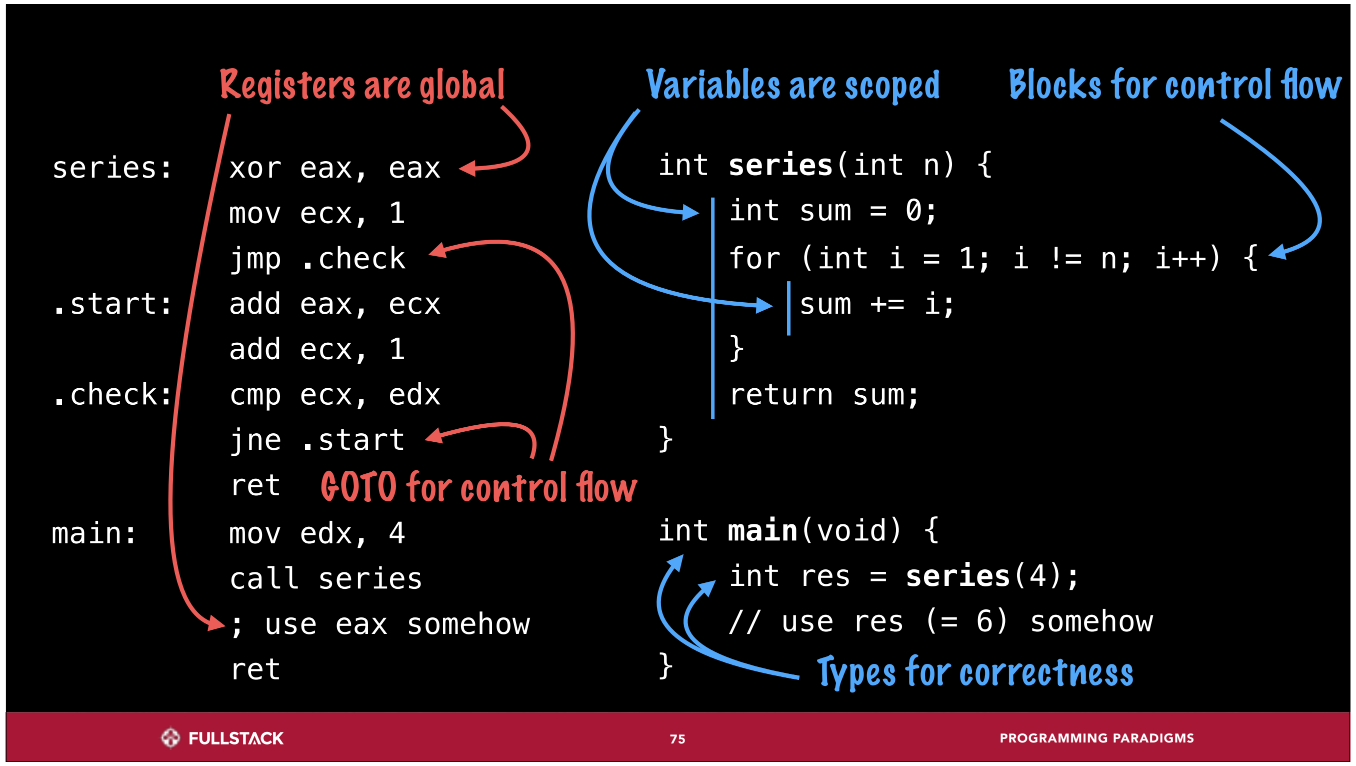
what part of C requires the most Assembly?

C

```
series:  xor eax, eax
         mov ecx, 1
         jmp .check
.start:  add eax, ecx
         add ecx, 1
.check:  cmp ecx, edx
         jne .start
         ret
main:    mov edx, 4
         call series
         ; use eax somehow
         ret
```

```c
int series(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = series(4);
    // use res (= 6) somehow
}
```

Here we color-code how each C statement can be expressed via one or more assembly statements.
* What part of C requires the most assembly?
* What does C give that assembly does not?



Registers are global

```
series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

GOTO for control flow

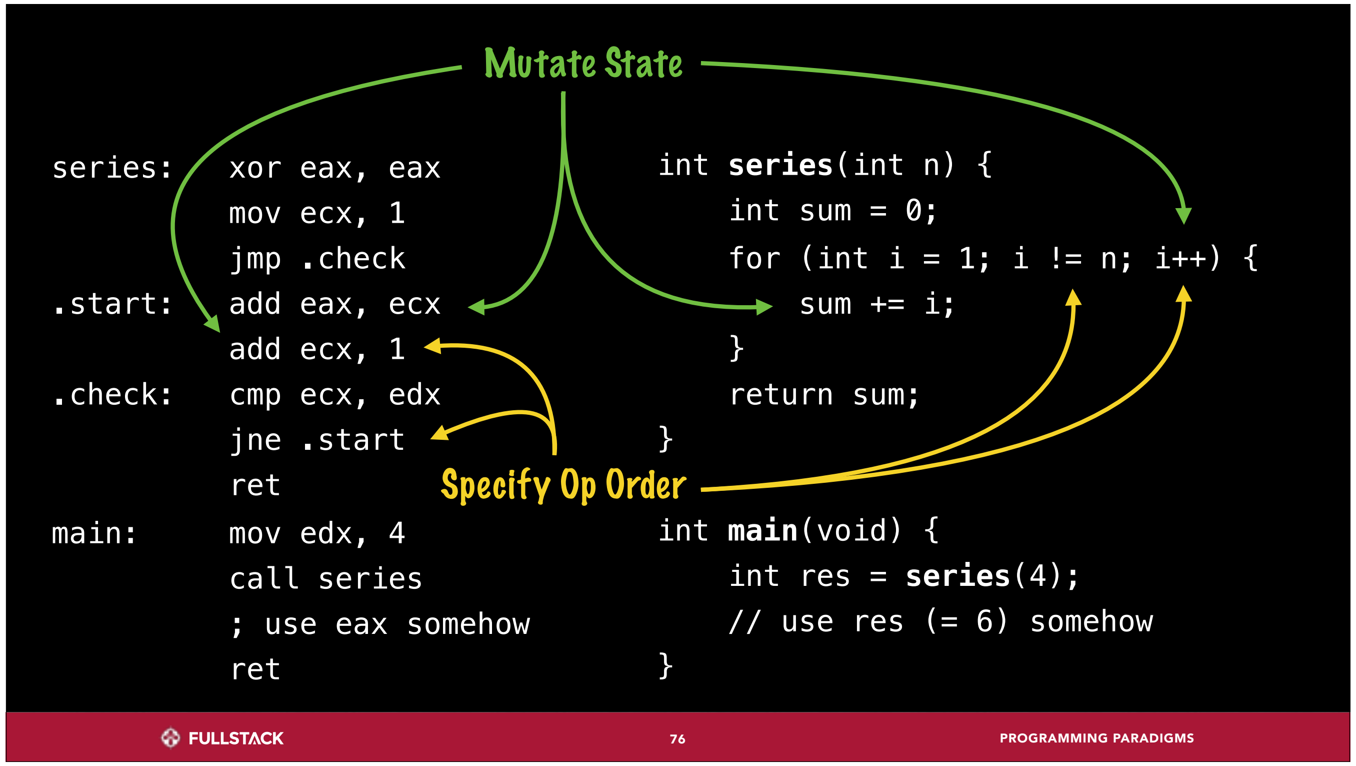
Variables are scoped

Blocks for control flow

```
int series(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}

int main(void) {
    int res = series(4);
    // use res (= 6) somehow
}
```

Types for correctness

Some differences:
- In assembly, you can access any memory at any time. C has scopes preventing out-of-scope access.
- C has blocks for easier control flow. Assembly just has jumps (GOTO, call).



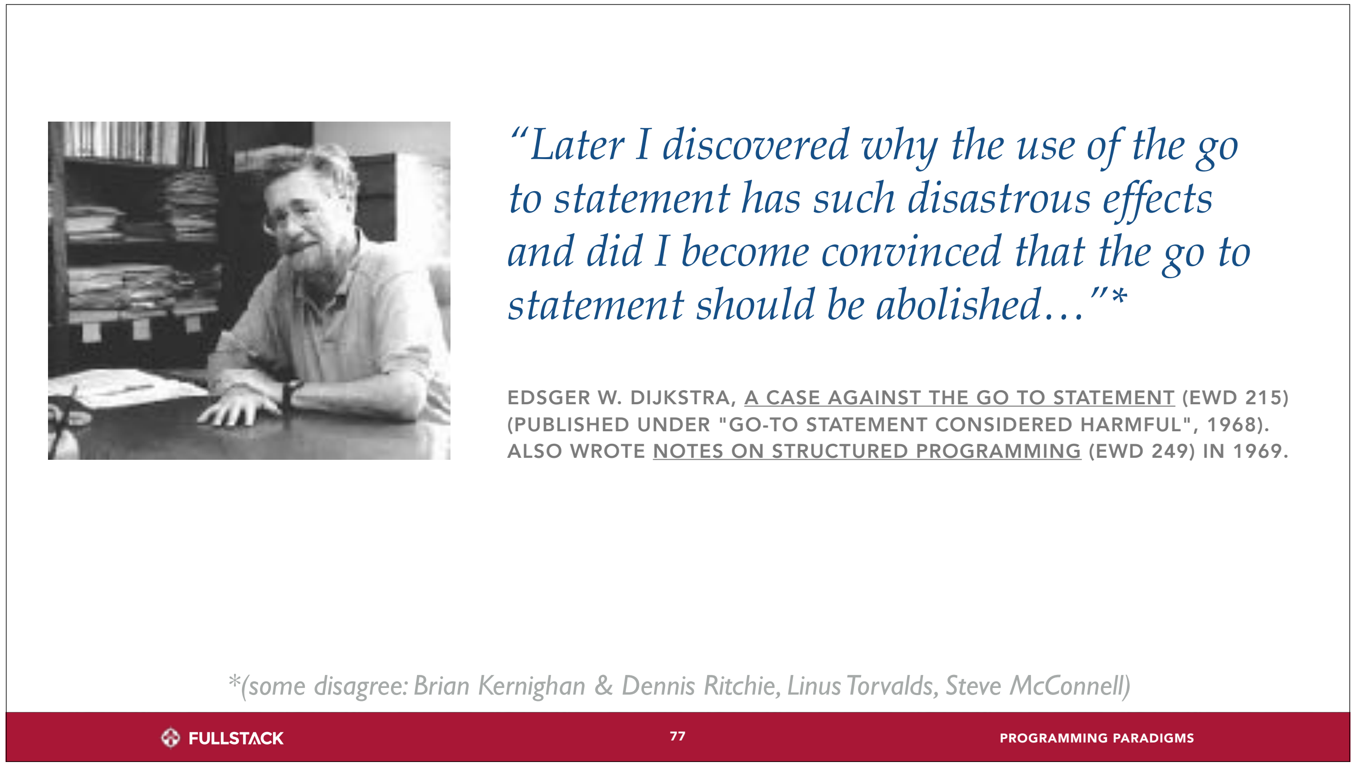
Mutate State

```
series:   xor eax, eax
          mov ecx, 1
          jmp .check
.start:   add eax, ecx
          add ecx, 1
.check:   cmp ecx, edx
          jne .start
          ret
main:     mov edx, 4
          call series
          ; use eax somehow
          ret
```

Specify Op Order

```
int series(int n) {
    int sum = 0;
    for (int i = 1; i != n; i++) {
        sum += i;
    }
    return sum;
}
int main(void) {
    int res = series(4);
    // use res (= 6) somehow
}
```

Similarities:
- You can (and do) mutate stateful memory.
- You do things in a particular order – moving statements changes the meaning of the program.



*“Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished…”**

EDSGER W. DIJKSTRA, A CASE AGAINST THE GO TO STATEMENT (EWD 215) (PUBLISHED UNDER "GO-TO STATEMENT CONSIDERED HARMFUL", 1968). ALSO WROTE NOTES ON STRUCTURED PROGRAMMING (EWD 249) IN 1969.

**(some disagree: Brian Kernighan & Dennis Ritchie, Linus Torvalds, Steve McConnell)*

Dijkstra is credited with many things in CS, including coining "structured programming". An editor published his GOTO letter as "go-to statement considered harmful", the origin of that popular formula.

"…its title, which became a cornerstone of my fame by becoming a template: we would see all sorts of articles under the title "X considered harmful" for almost any X, including one titled "Dijkstra considered harmful". But what had happened? I had submitted a paper under the title "A case against the goto statement", which, in order to speed up its publication, the editor had changed into a "letter to the Editor", and in the process he had given it a new title of his own invention! The editor was Niklaus Wirth."

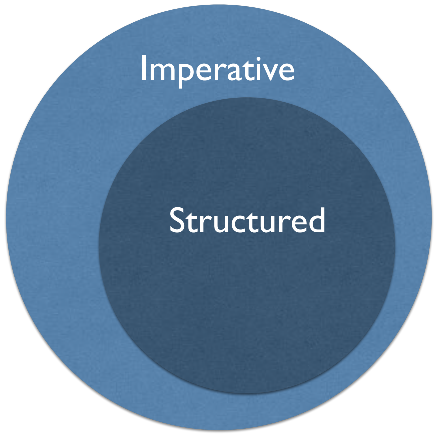
Imperative

- 🤖 Instruct the machine what to do
- ▶️ Specify exact order of operations
- 🔥 Mutate state (often direct memory)

Structured

Abstraction!

- 🗄️ Blocks for control flow
- ↔️ Branching: if/then/else, switch/case
- 🔃 Iteration: do/while/for



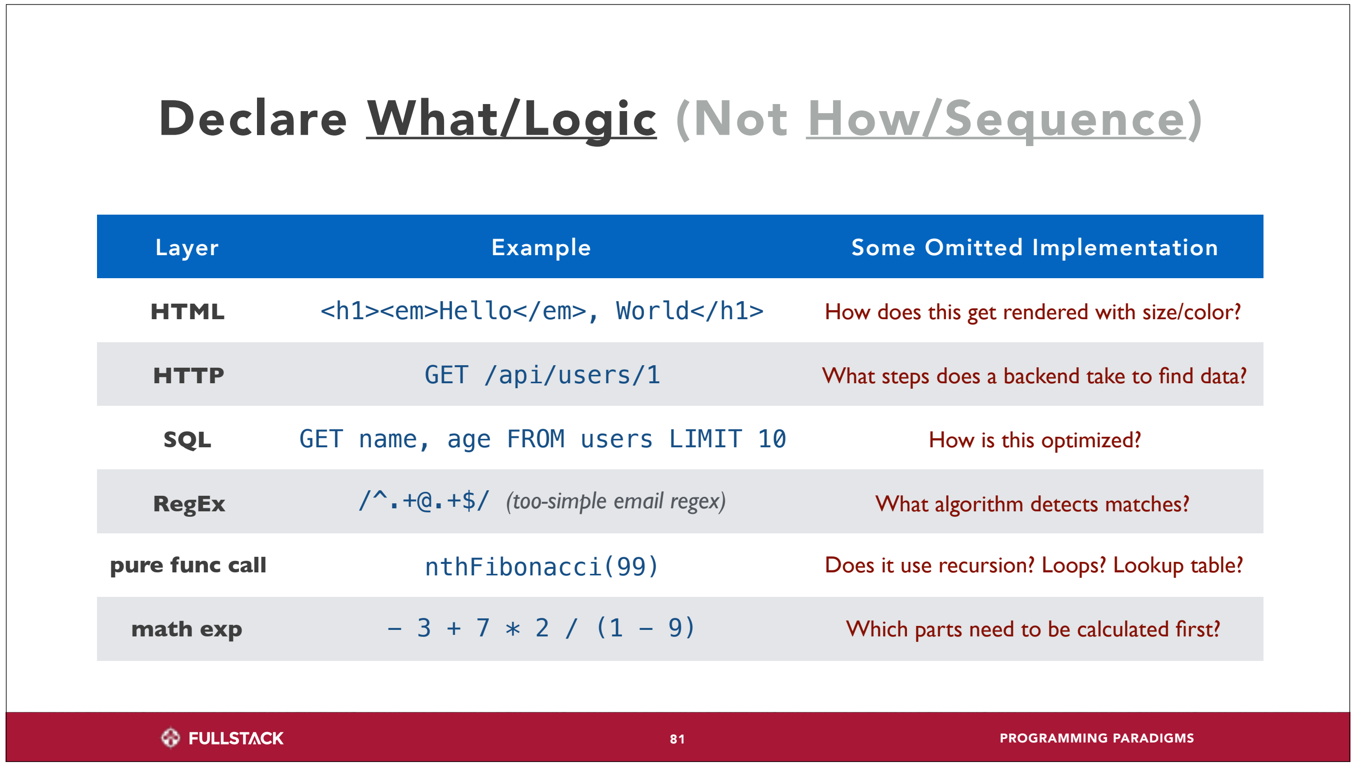
*Simplifications; many definitions vary in scope & detail

5 minute break!

Paradigm: Declarative

…and beginning to get functional, too…

We are going to take a (very quick!) look at two other paradigms: declarative & object-oriented.

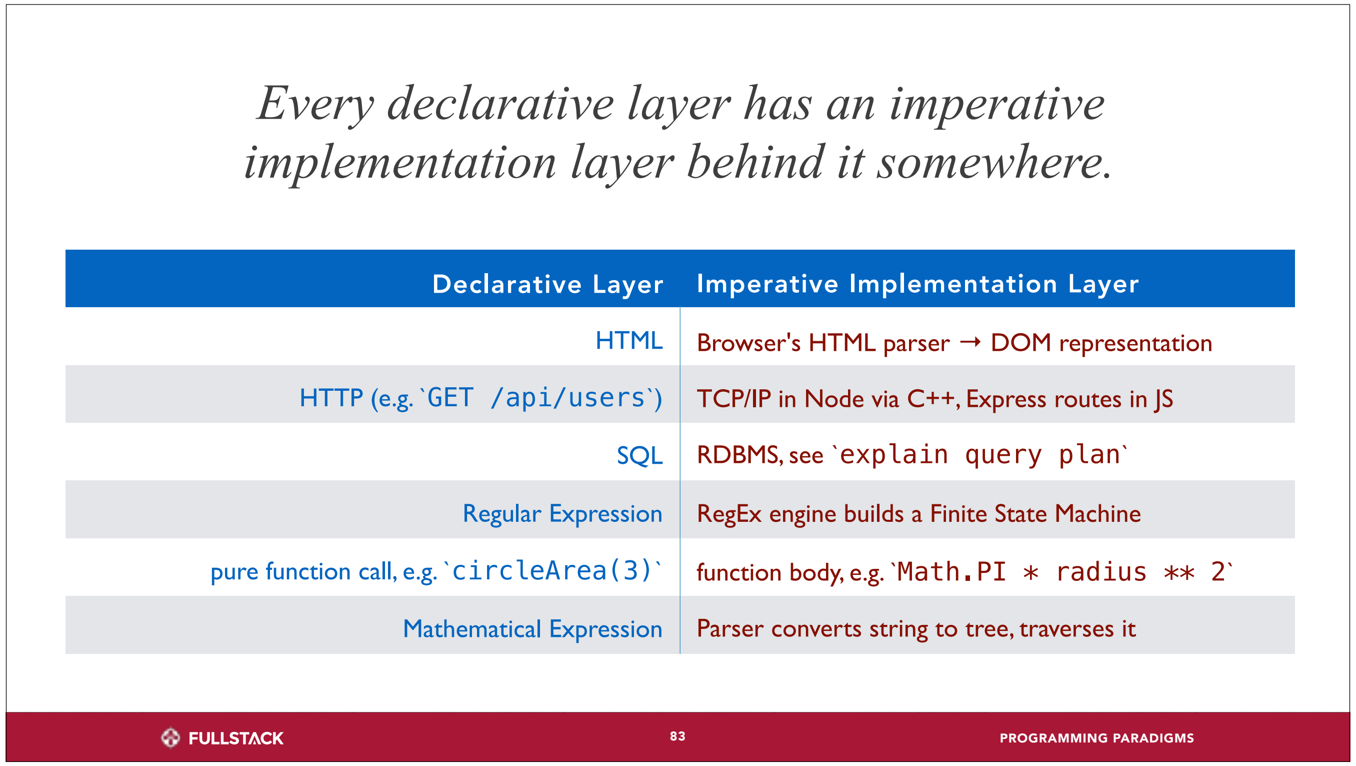


Declare What/Logic (Not How/Sequence)

Layer	Example	Some Omitted Implementation
HTML	`<h1><em>Hello</em>, World</h1>`	How does this get rendered with size/color?
HTTP	`GET /api/users/1`	What steps does a backend take to find data?
SQL	`GET name, age FROM users LIMIT 10`	How is this optimized?
RegEx	`/^.+@.+$/` *(too-simple email regex)*	What algorithm detects matches?
pure func call	`nthFibonacci(99)`	Does it use recursion? Loops? Lookup table?
math exp	`− 3 + 7 ∗ 2 / (1 − 9)`	Which parts need to be calculated first?

Declarative languages break away from instructions. Declarations specify logical relationships or desired results, not the implementation details.

Every declarative layer has an imperative implementation layer behind it somewhere.



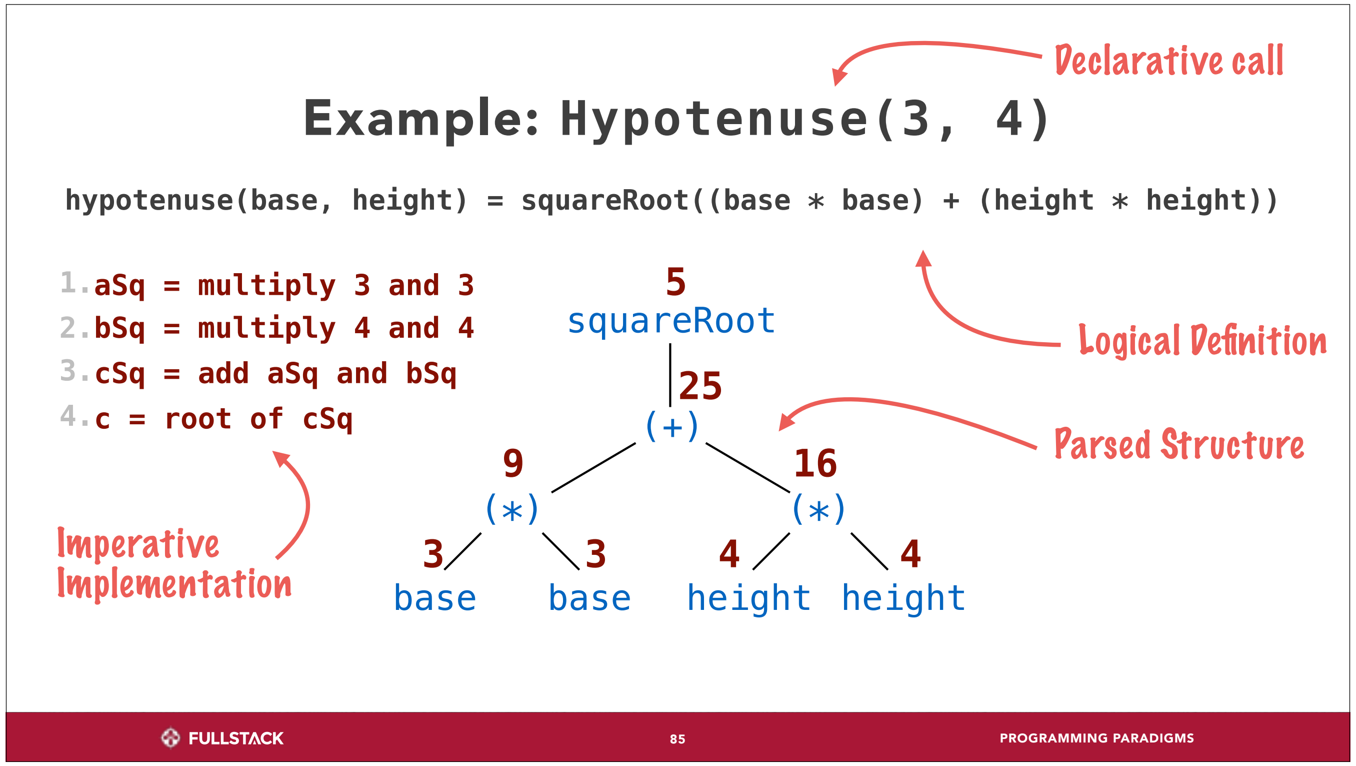
Every declarative layer has an imperative implementation layer behind it somewhere.

Declarative Layer	Imperative Implementation Layer
HTML	Browser's HTML parser → DOM representation
HTTP (e.g. `GET /api/users`)	TCP/IP in Node via C++, Express routes in JS
SQL	RDBMS, see `explain query plan`
Regular Expression	RegEx engine builds a Finite State Machine
pure function call, e.g. `circleArea(3)`	function body, e.g. `Math.PI * radius ** 2`
Mathematical Expression	Parser converts string to tree, traverses it

Human beings are much better at expressing what we want than rigorously figuring out how to make it happen. Declarative languages support that, but only because some system somewhere is capable of digesting the declaration and transforming it into a sequence of steps to perform.

Programs as Evaluations of a Tree

- System figures out order of operations for you
- No direct mutation of state, only descriptions of relationships
- How? Compiler changes description into a sequence of steps.



Declarative call

Example: Hypotenuse(3, 4)

```
hypotenuse(base, height) = squareRoot((base * base) + (height * height))
```

Logical Definition

```
aSq = multiply 3 and 3
bSq = multiply 4 and 4
cSq = add aSq and bSq
c = root of cSq
```

Imperative Implementation

Parsed Structure

[SKIPPABLE] Let's see a quick concrete example.

MORE DECLARATIVE / FUNCTIONAL ⟷ MORE IMPERATIVE / PROCEDURAL

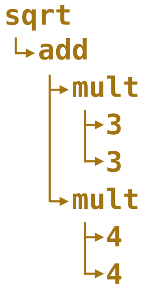
function call

hypotenuse(3, 4)

function definition

sqrt($3^2 + 4^2$)

(parsed as)



```
sqrt
 └→add
    ├→mult
    │  ├→3
    │  └→3
    └→mult
       ├→4
       └→4
```

procedural implementation

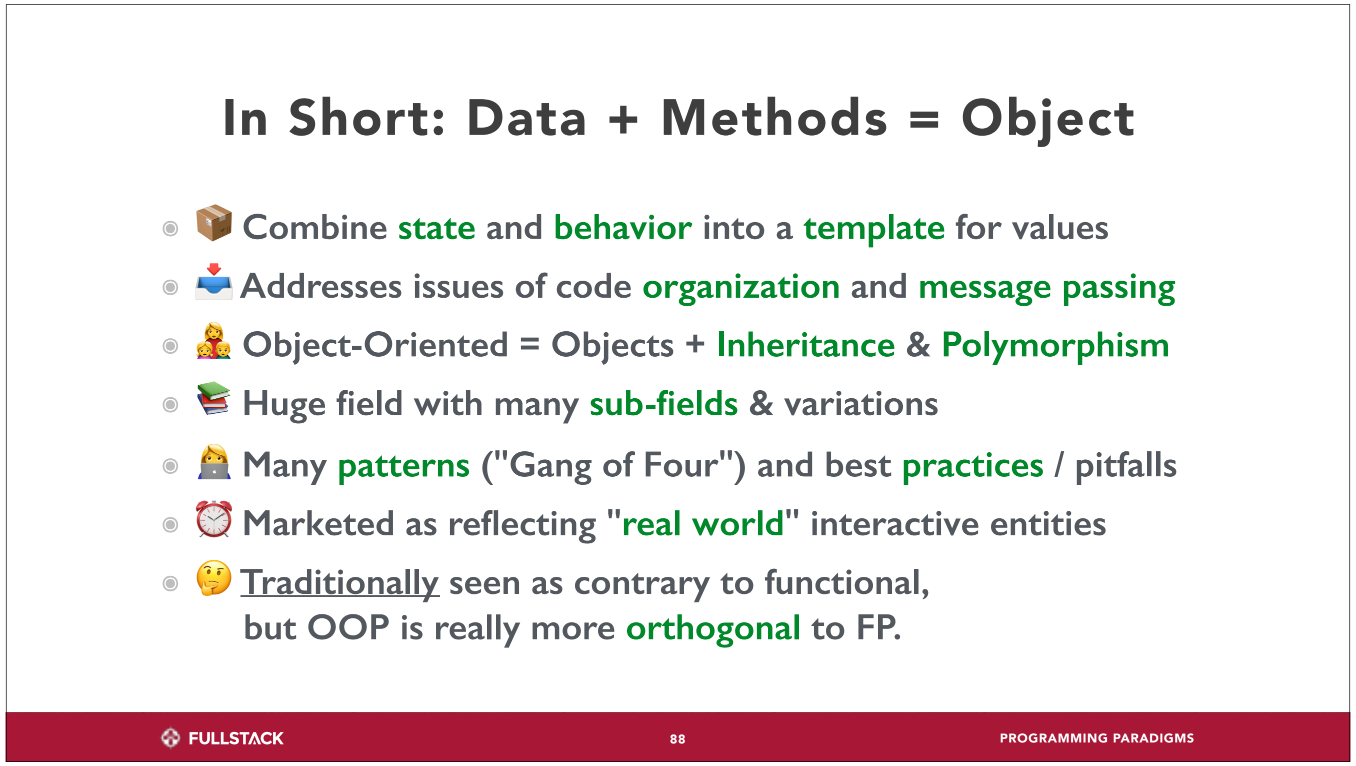
```
aSq = multiply 4 and 4
bSq = multiply 3 and 3
cSq = add aSq and bSq
c = root of cSq
```

(or)

```
bSq = multiply 3 and 3
aSq = multiply 4 and 4
cSq = add aSq and bSq
c = root of cSq
```

Paradigm: Object-Oriented

(super-duper condensed version)



In Short: Data + Methods = Object

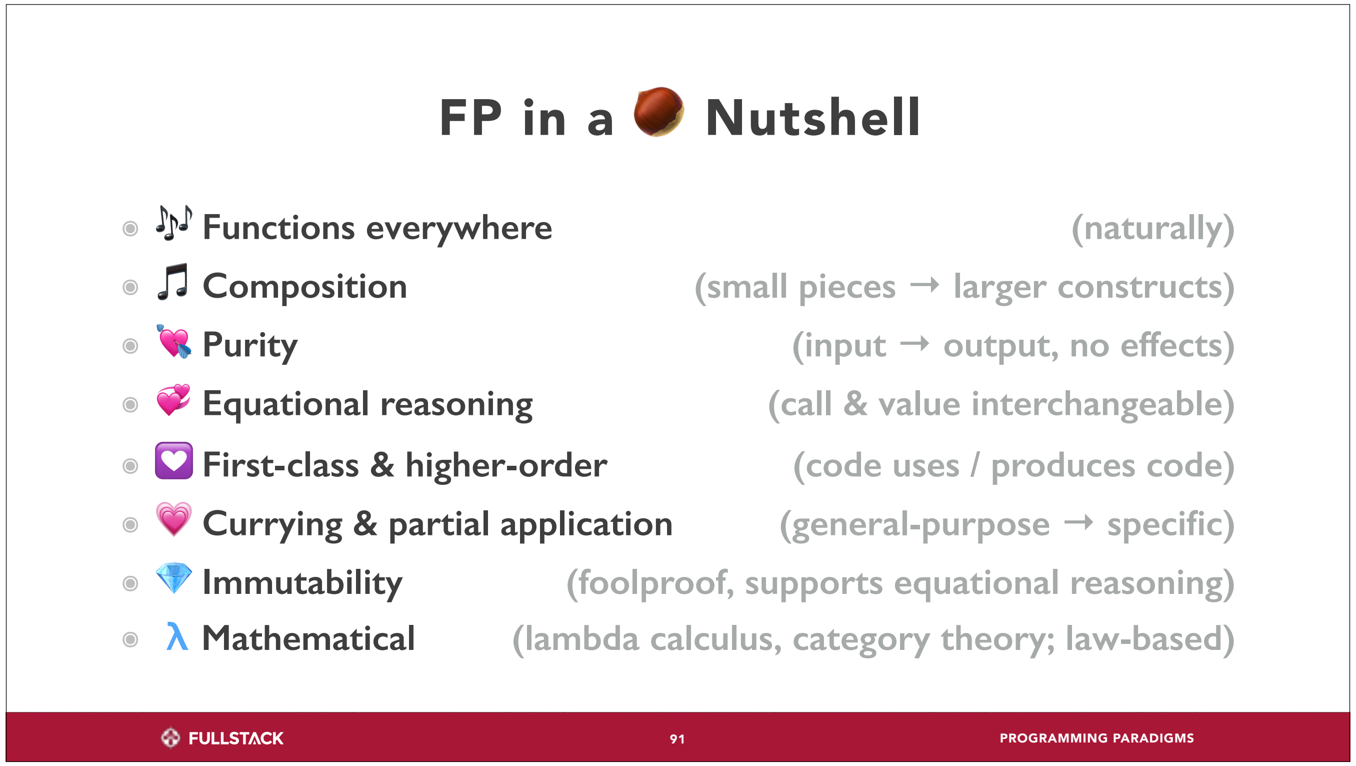
- 📦 Combine state and behavior into a template for values
- 📥 Addresses issues of code organization and message passing
- 👩‍👧‍👦 Object-Oriented = Objects + Inheritance & Polymorphism
- 📚 Huge field with many sub-fields & variations
- 👩‍💻 Many patterns ("Gang of Four") and best practices / pitfalls
- ⏰ Marketed as reflecting "real world" interactive entities
- 🤔 Traditionally seen as contrary to functional, but OOP is really more orthogonal to FP.

We are not going to focus on OOP for this lecture as it is a giant topic which won't really be necessary to appreciate or contrast against upcoming FP topics. A bigger split is FP vs. Imperative, hence why we focused on it.

What about doing this in a (drumroll) FUNCTIONAL way?

Concepts of Functional Programming

Overview, History, Theory &c.



FP in a 🌰 Nutshell

🎶 Functions everywhere	(naturally)
🎵 Composition	(small pieces → larger constructs)
💘 Purity	(input → output, no effects)
💞 Equational reasoning	(call & value interchangeable)
💟 First-class & higher-order	(code uses / produces code)
💗 Currying & partial application	(general-purpose → specific)
💎 Immutability	(foolproof, supports equational reasoning)
λ Mathematical	(lambda calculus, category theory; law-based)

We will see more on these, this is to give you a taste of what's to come.
* Composition = seamlessly combining small things into bigger things
* Purity = same output for same input + no effects
* Referential transparency = function call can be replaced with value, no change in meaning
* First-class / HoF = functions are values, and functions can take and/or return functions
* Currying / partial application = give function only some args, returns a "prebaked" function waiting for more
* Immutable = cannot alter, can only generate new versions (which may share data)
* Lambda calc = basis of FP, category theory = wellspring of applicable composition patterns

💡 Motivations 💡

Derive new code from old

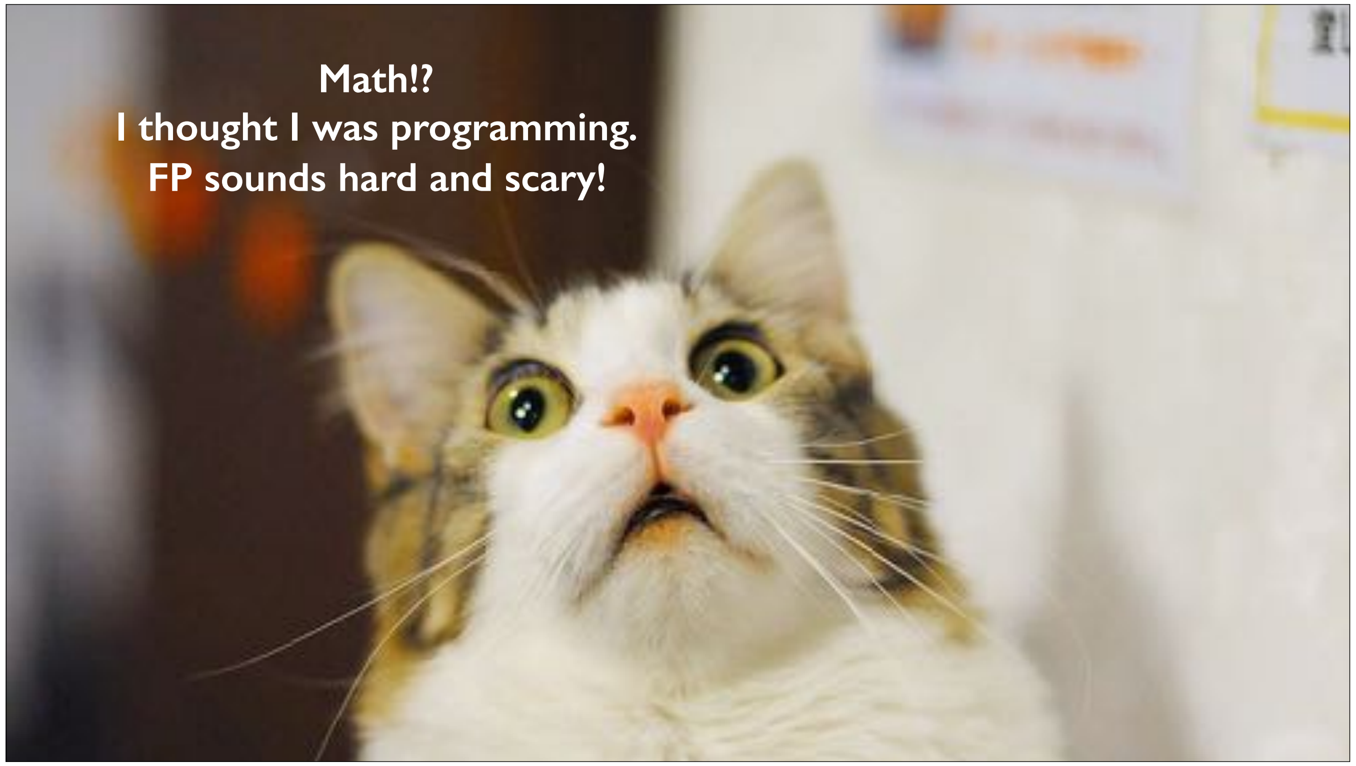
Pieces work well together

Reduced mental scope while writing

Certain classes of bug are made impossible

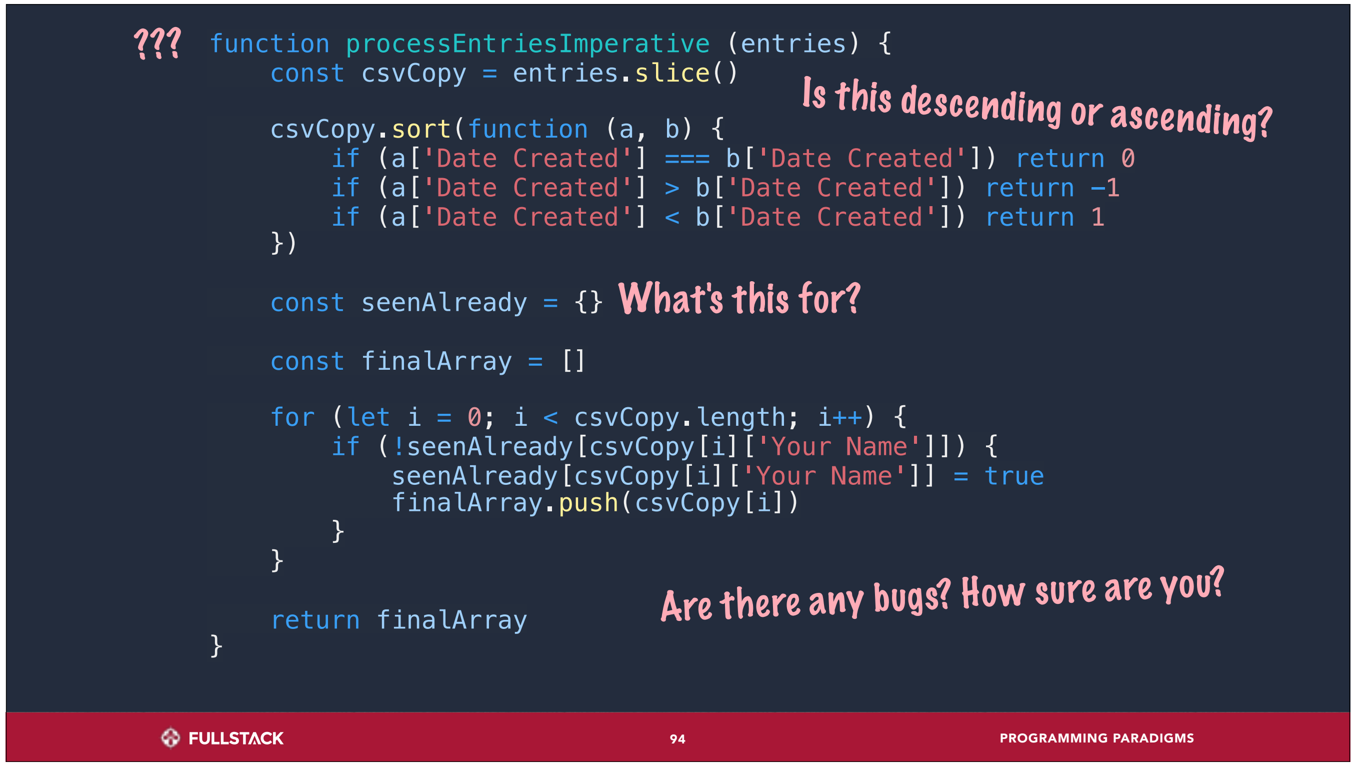
Mathematical laws are universal, unambiguous

You can't have Functional without Fun!



Math!?
I thought I was programming.
FP sounds hard and scary!

[SKIPPABLE]



???

```
function processEntriesImperative (entries) {
    const csvCopy = entries.slice()

    csvCopy.sort(function (a, b) {
        if (a['Date Created'] === b['Date Created']) return 0
        if (a['Date Created'] > b['Date Created']) return -1
        if (a['Date Created'] < b['Date Created']) return 1
    })

    const seenAlready = {}

    const finalArray = []

    for (let i = 0; i < csvCopy.length; i++) {
        if (!seenAlready[csvCopy[i]['Your Name']]) {
            seenAlready[csvCopy[i]['Your Name']] = true
            finalArray.push(csvCopy[i])
        }
    }

    return finalArray
}
```

Is this descending or ascending?

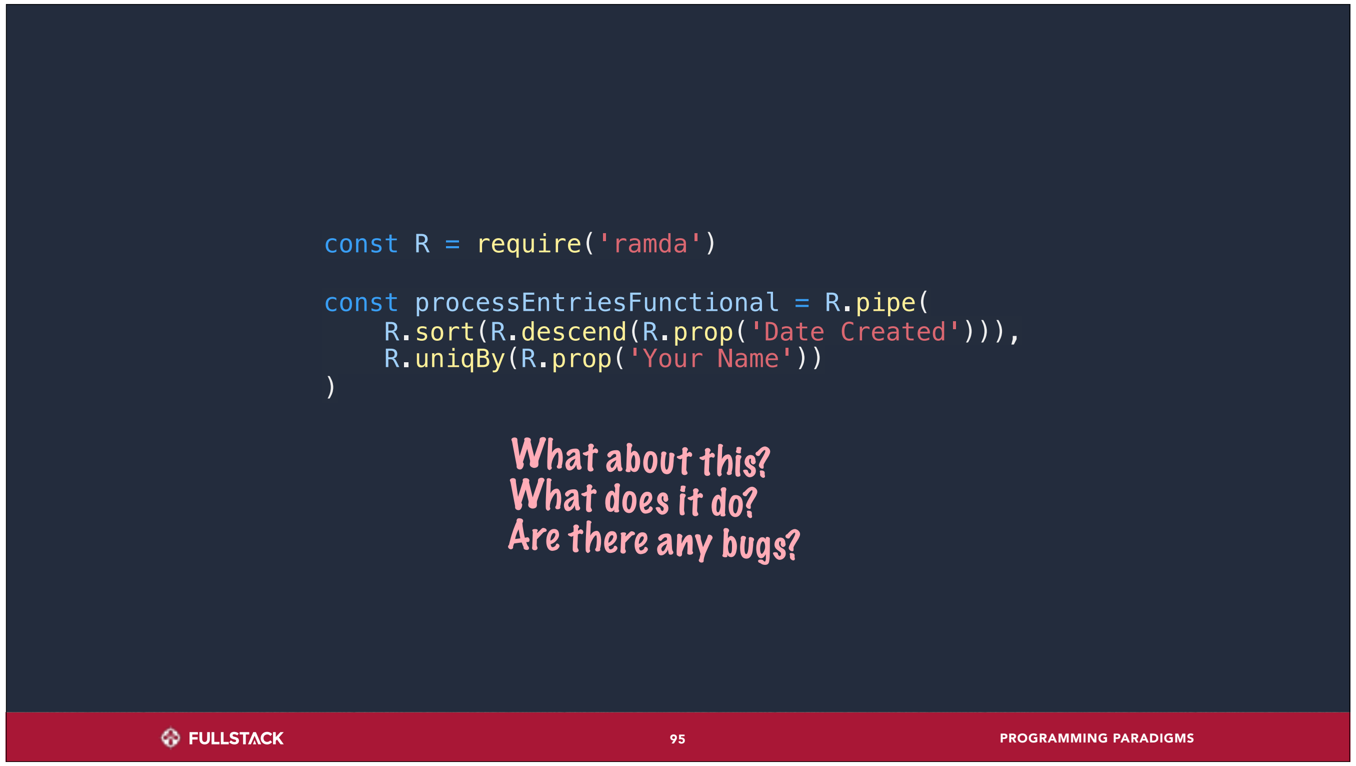
What's this for?

Are there any bugs? How sure are you?

Is this code imperative or declarative?

Are there any bugs?

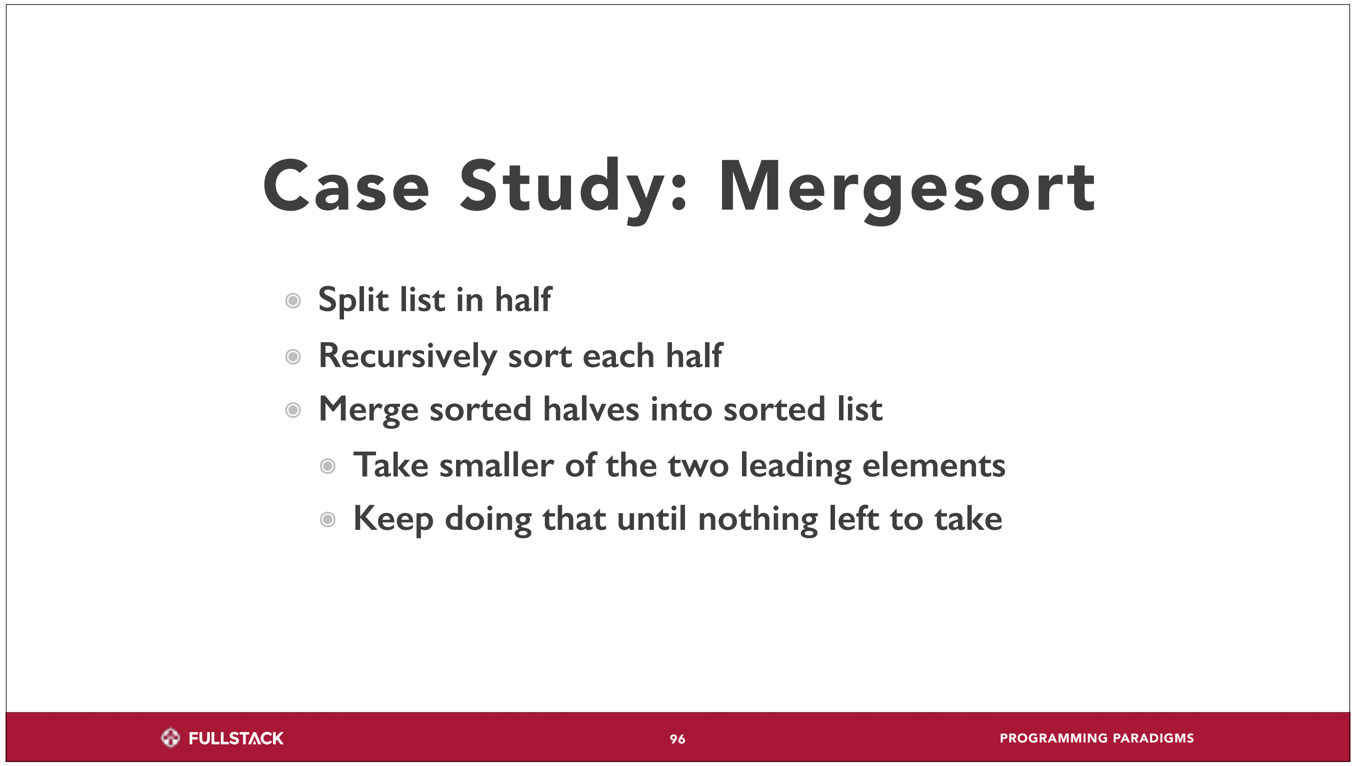
What does this code actually do? Does it sort ascending or descending? Whats' the purpose of `seenAlready`?



```
const R = require('ramda')

const processEntriesFunctional = R.pipe(
    R.sort(R.descend(R.prop('Date Created'))),
    R.uniqBy(R.prop('Your Name'))
)
```

What about this?
What does it do?
Are there any bugs?

Is this imperative or declarative?
Assuming the functions are correct, are there any bugs here? (Very little room to hide!)
What does this code actually do? How quickly/easily could you figure that out?



Case Study: Mergesort

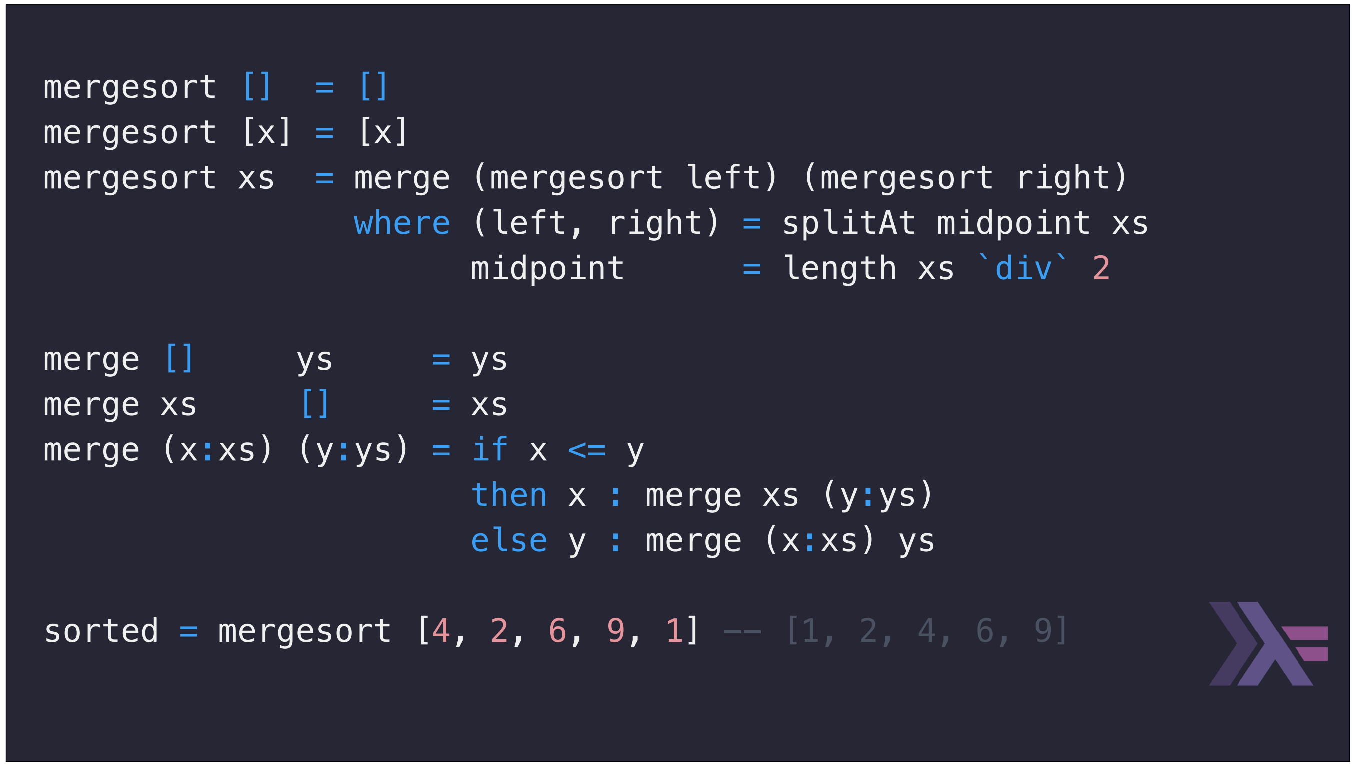
- Split list in half
- Recursively sort each half
- Merge sorted halves into sorted list
  - Take smaller of the two leading elements
  - Keep doing that until nothing left to take

Merge sort! You remember this, right?

But we are going to see merge sort in Haskell (eek!).

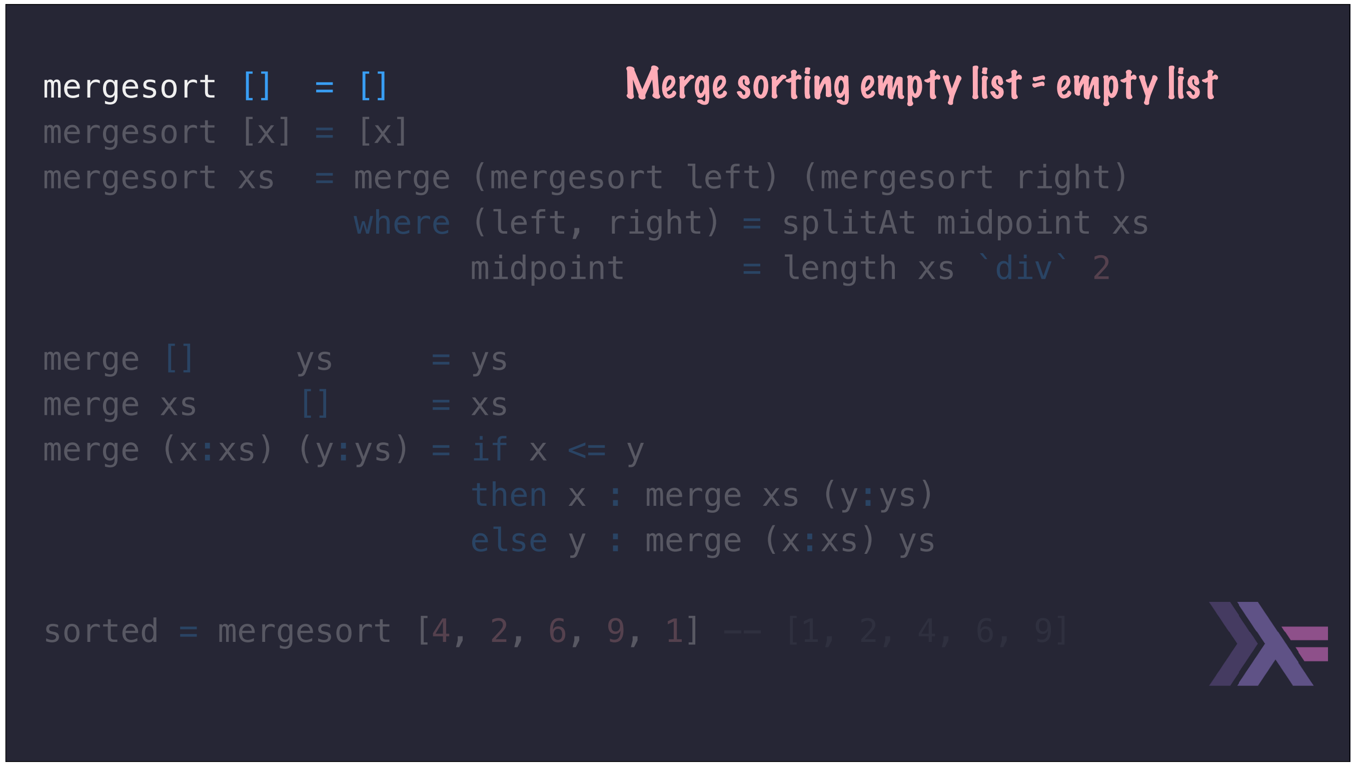
👩‍🎓 Reminder: this part is all *theory / context / jargon.*

You DO NOT need to learn this to succeed at Fullstack! We are introducing you to these concepts to provide additional insight.



```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

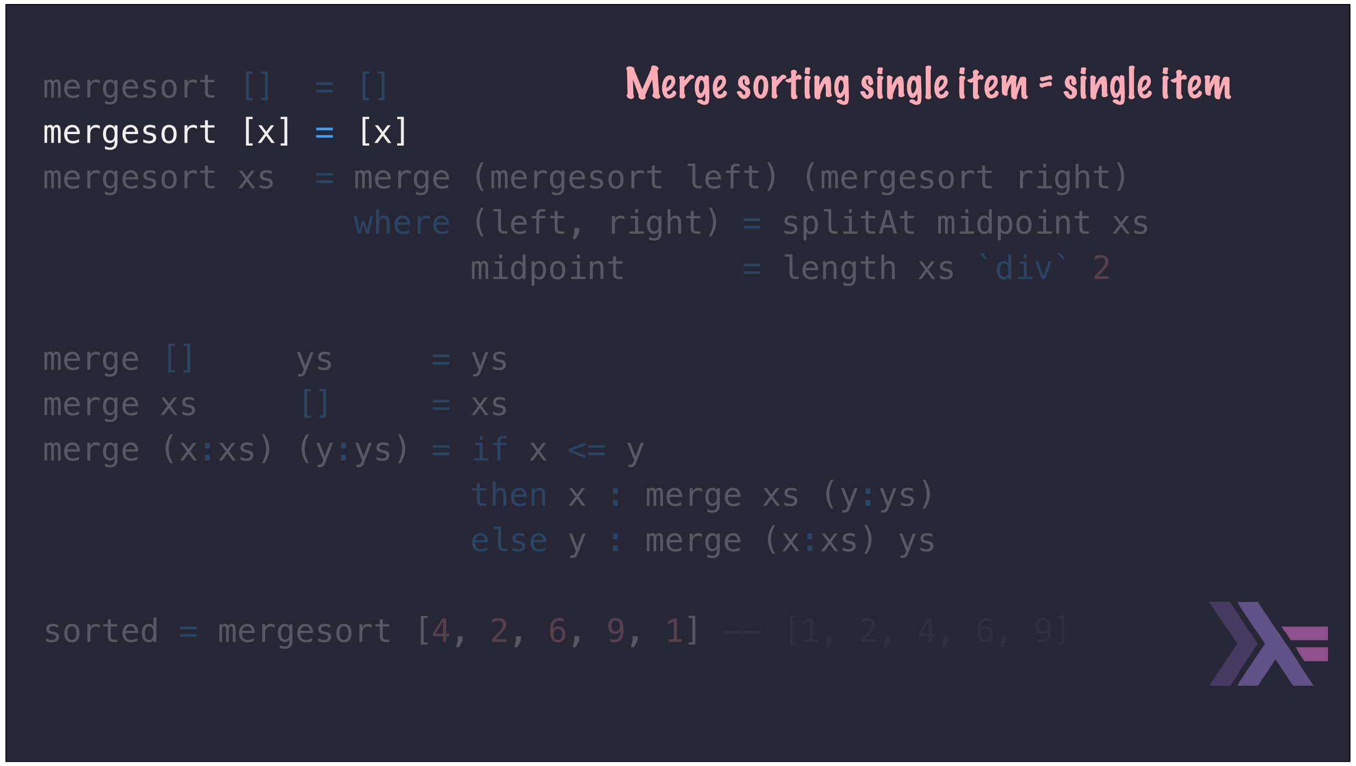
Here it is in Haskell. It's not important to fully understand Haskell syntax, but try to get a sense of the flavor.



```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sorting empty list = empty list

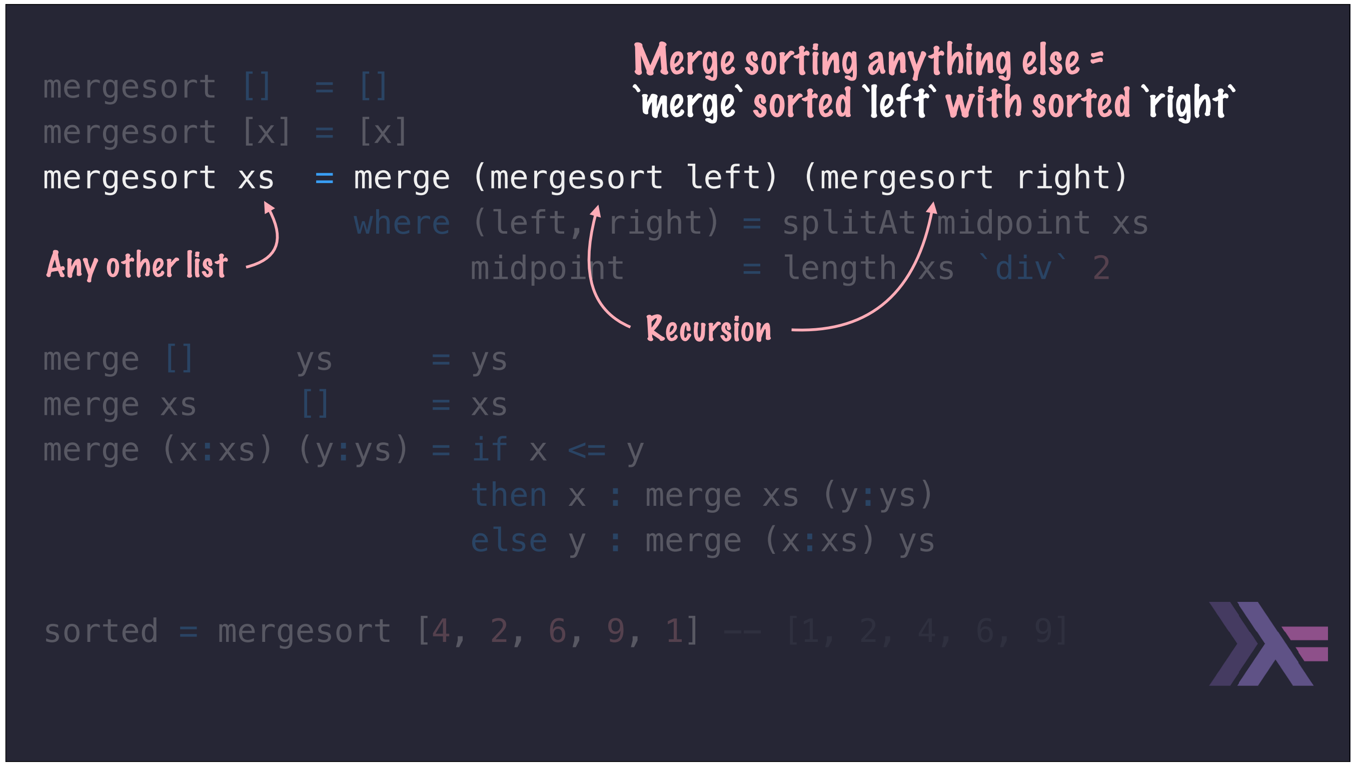
[SKIPPABLE] In Haskell, there is no difference between saying things are equal and defining a function return. It's identical, because all functions are pure. This is the definition of *referential transparency / equational reasoning* – if you see the left side, you can replace it with the right side (and vice-versa).



Merge sorting single item = single item

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

[SKIPPABLE]



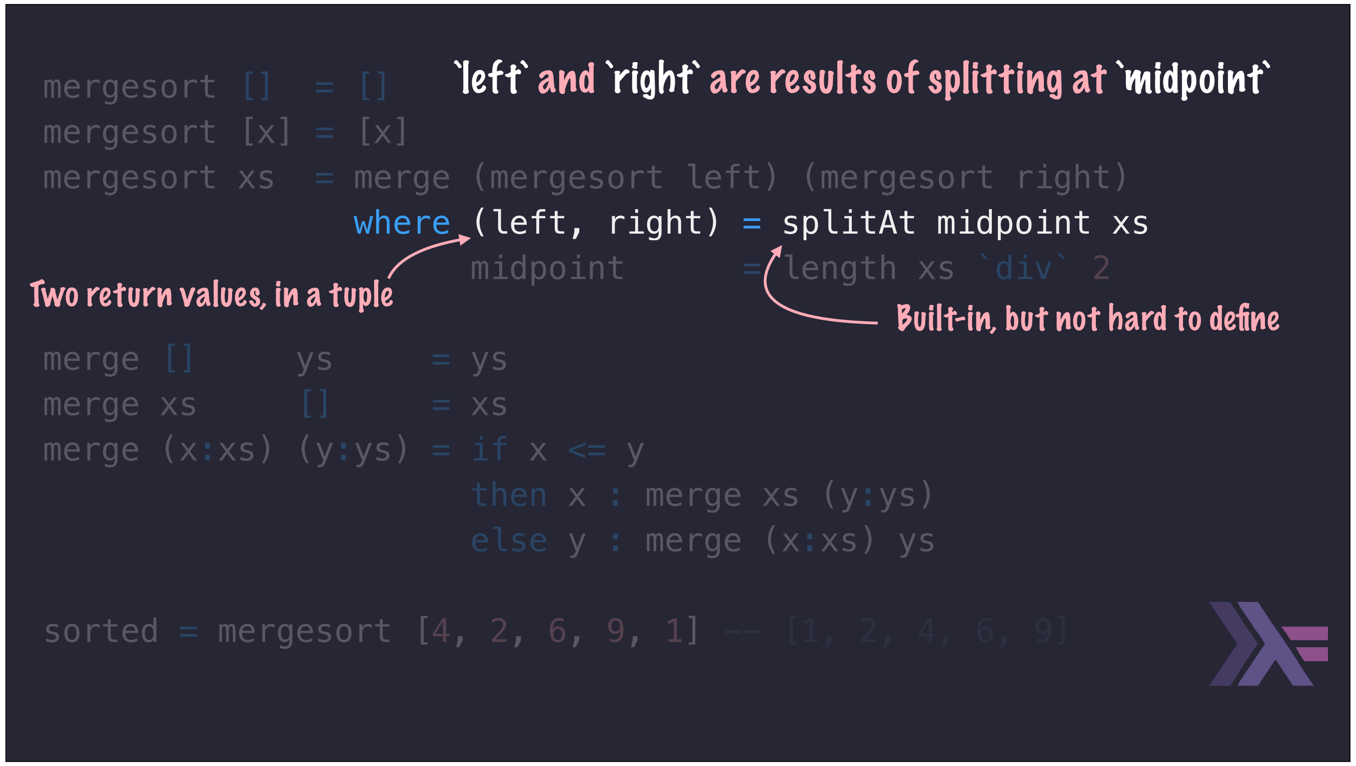
Merge sorting anything else =
`merge` sorted `left` with sorted `right`

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Any other list

Recursion

[SKIPPABLE] FP cannot mutate variables, so there are no `for` loops. All iteration is done through recursion.
In Haskell, function application is just a space. So JS `func(arg)` is Haskell `func arg`.



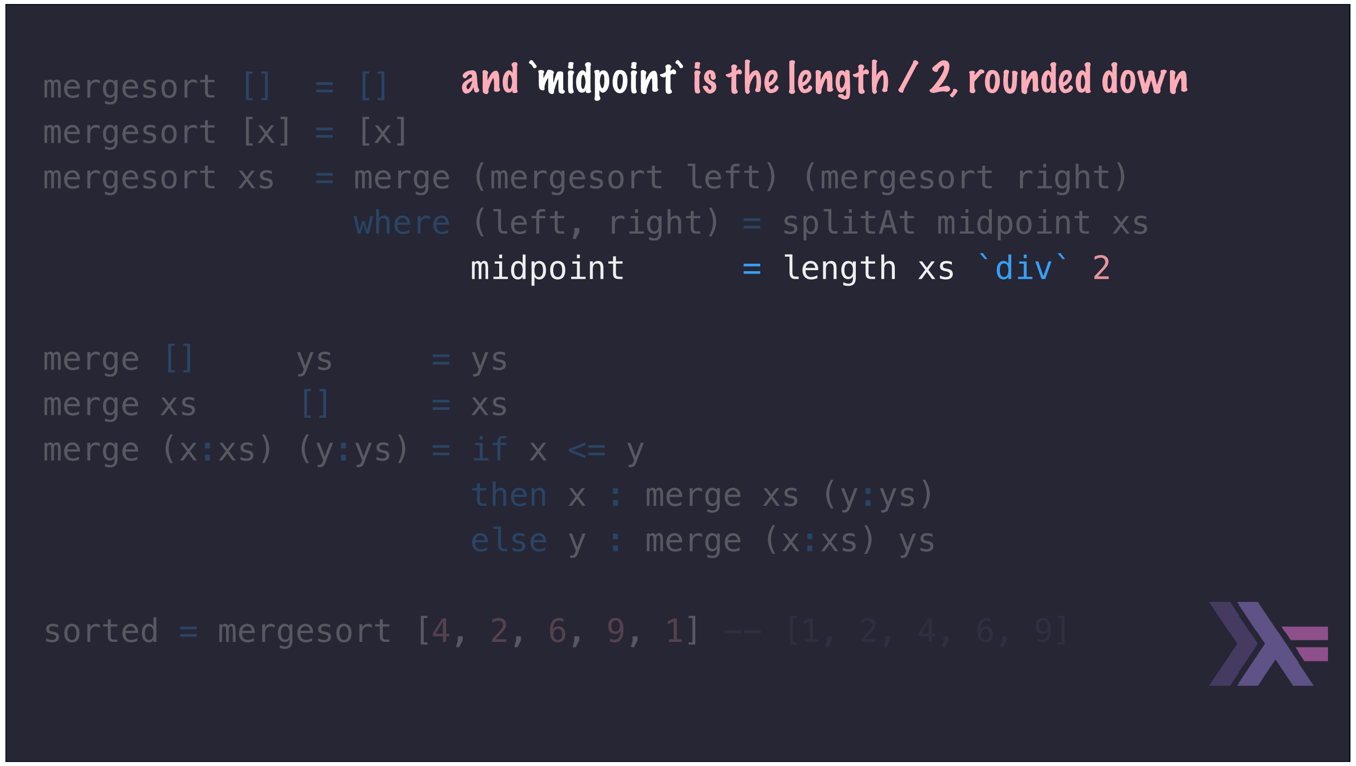
`left` and `right` are results of splitting at `midpoint`

```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Two return values, in a tuple

Built-in, but not hard to define

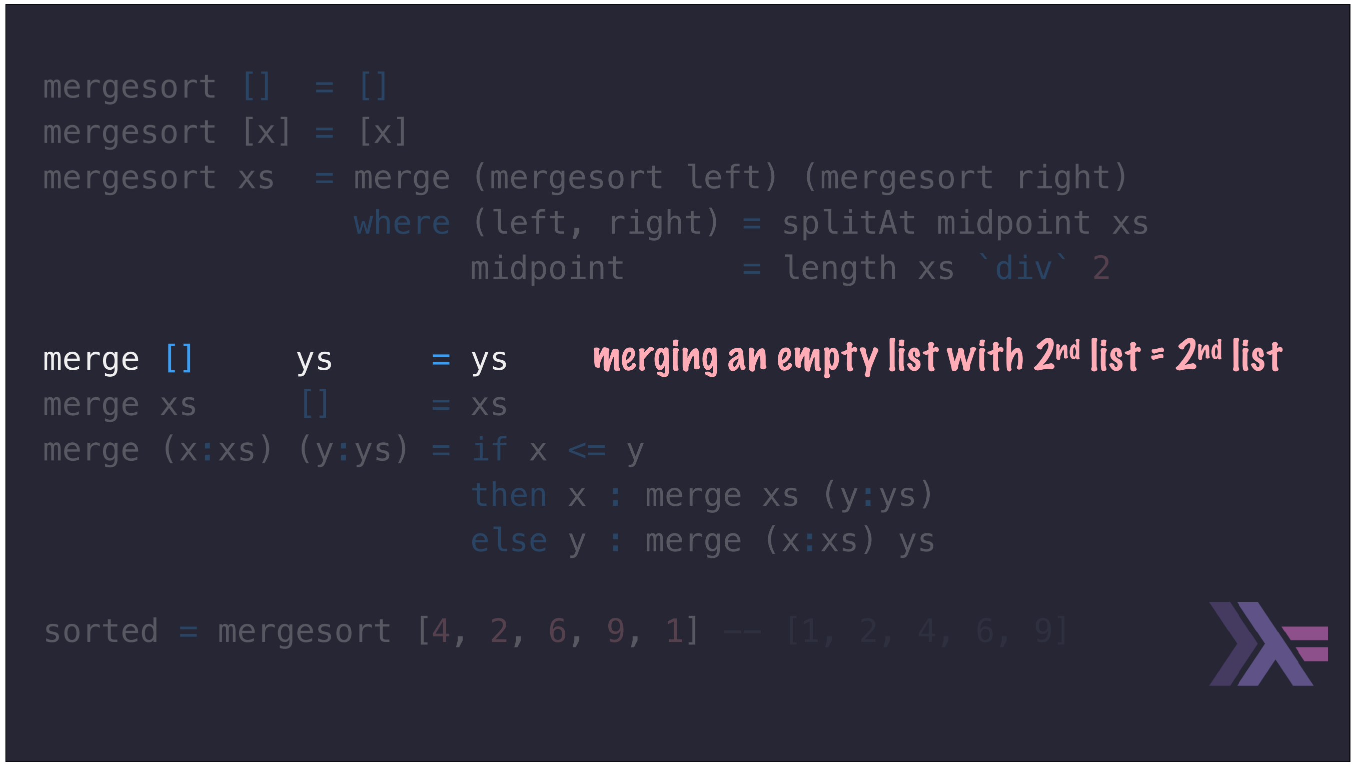
[SKIPPABLE] Haskell has first-class support for tuples. Basically think of them like lightweight arrays or objects, used for returning or passing around multiple values of different types.



and `midpoint` is the length / 2, rounded down

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

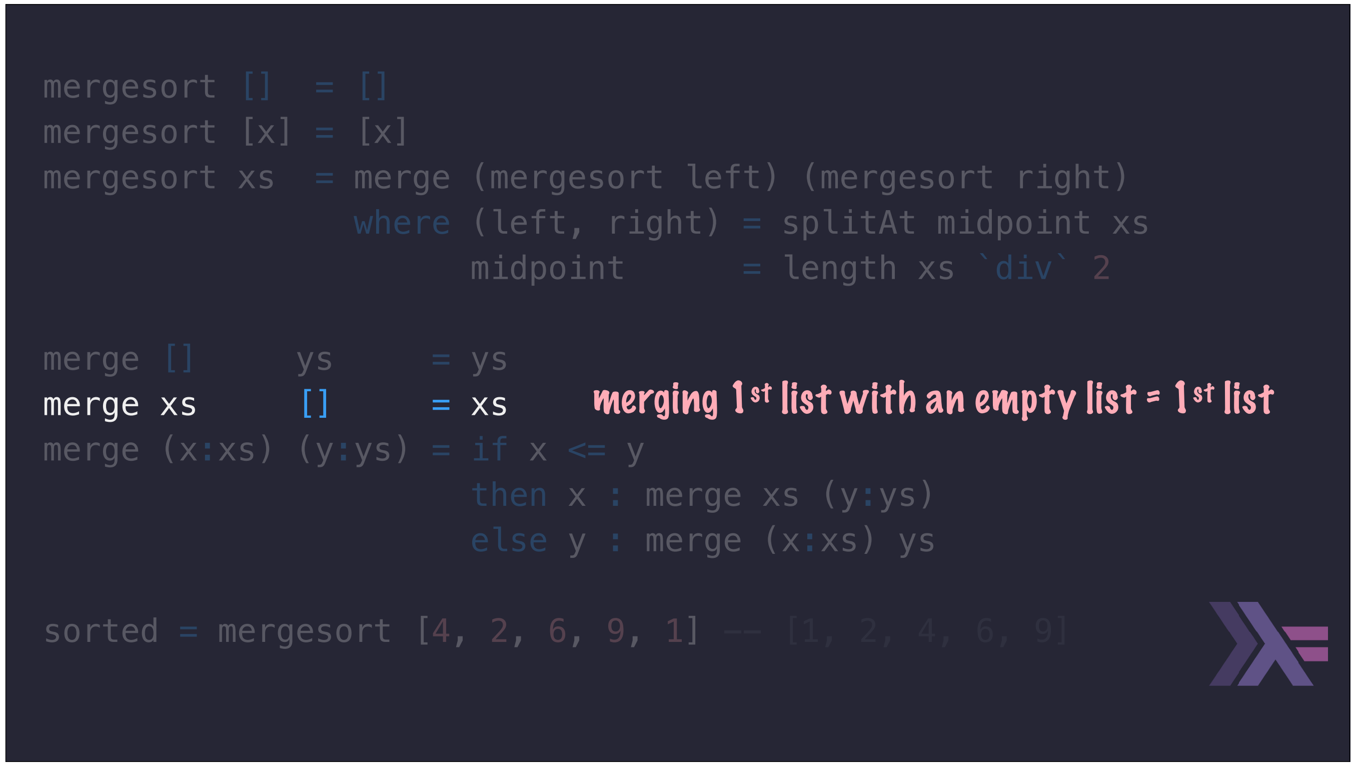
[SKIPPABLE]



```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging an empty list with 2nd list = 2nd list

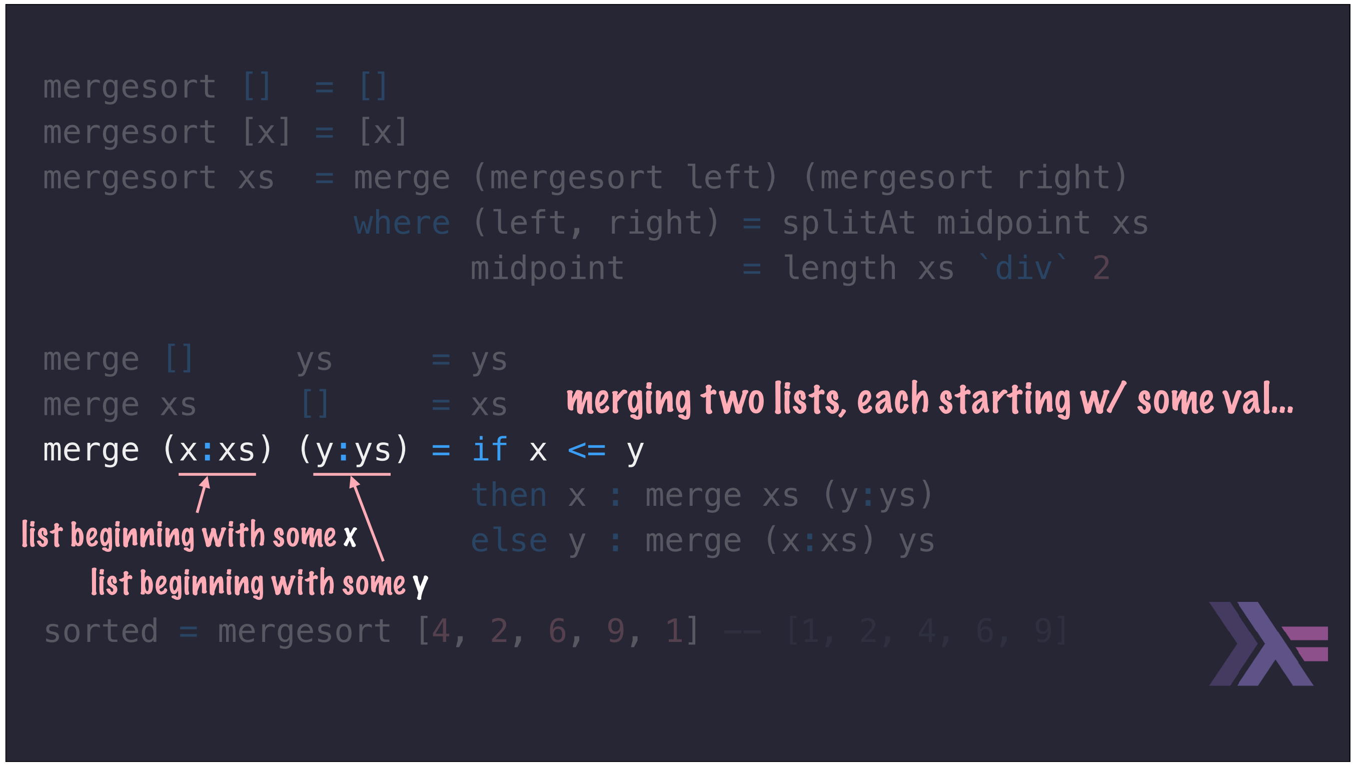
[SKIPPABLE]



```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging 1st list with an empty list = 1st list

[SKIPPABLE]



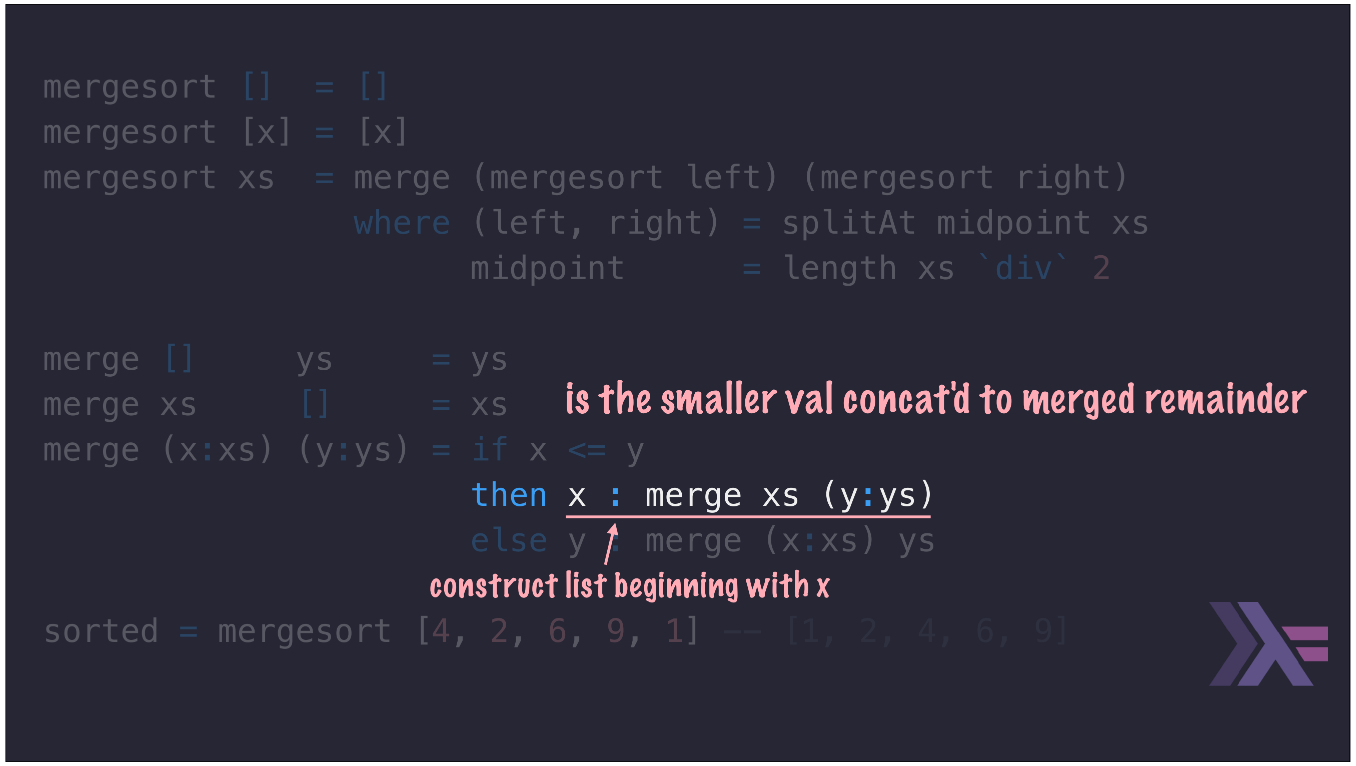
```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging two lists, each starting w/ some val...

list beginning with some x

list beginning with some y

[SKIPPABLE] Don't get too hung up on pattern matching, the list constructor (:), etc. But if students insist: (el:els) matches a list whose first element will be bound as `el` and whose following elements will be bound as `els`. So `el` will be an element from the list, and `els` will be the remaining list (possibly an empty list).

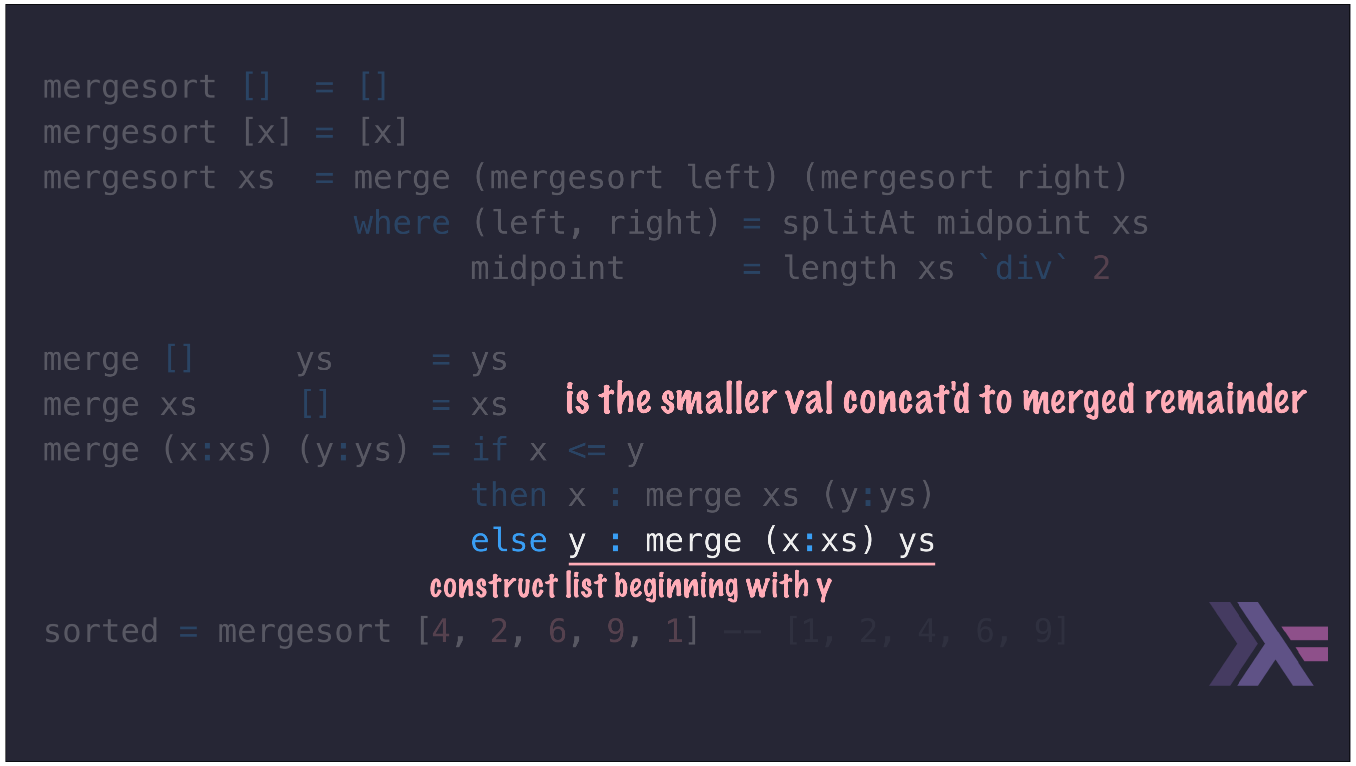


```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

is the smaller val concat'd to merged remainder

construct list beginning with x

[SKIPPABLE] Again, the syntax gets a bit more hairy here, but we are construction (with `:`) a list starting with `x` and ending with the rest of the list elements merged together.

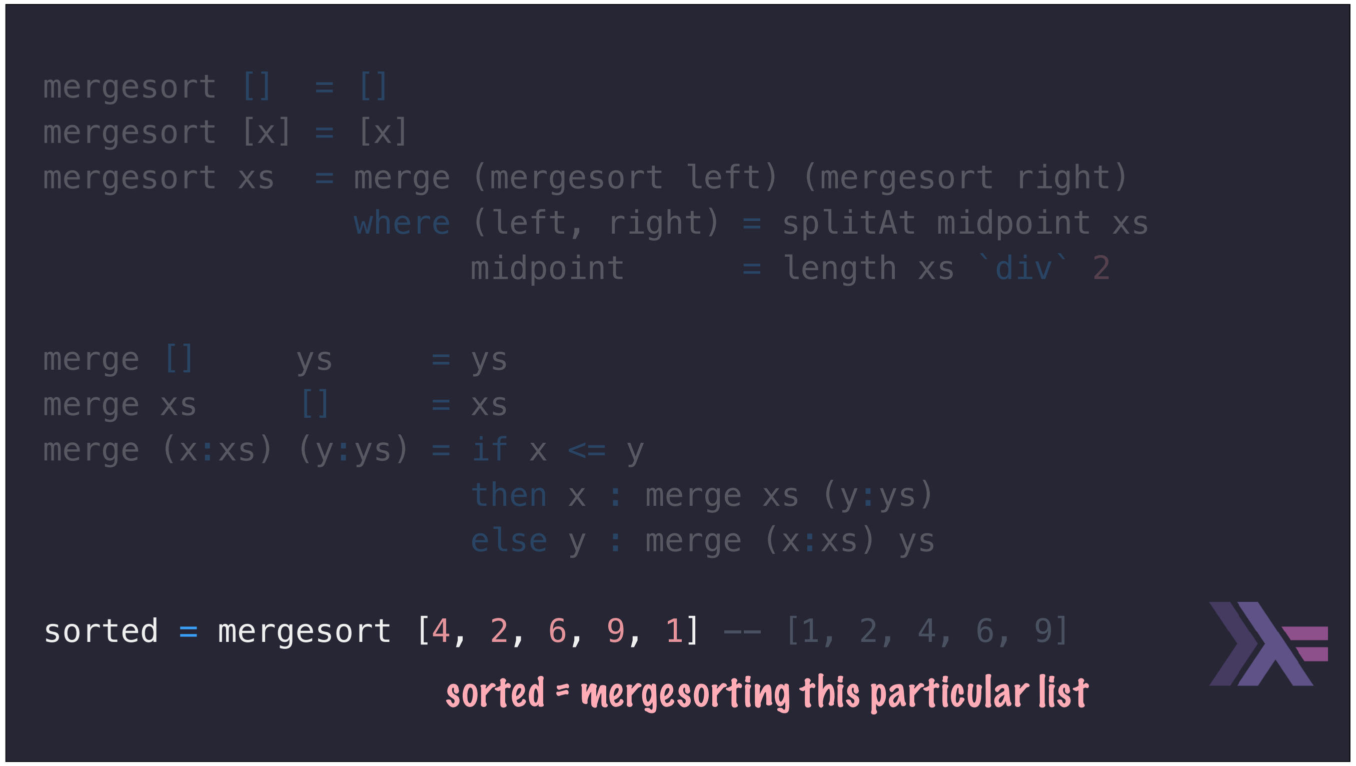


```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

is the smaller val concat'd to merged remainder

construct list beginning with y

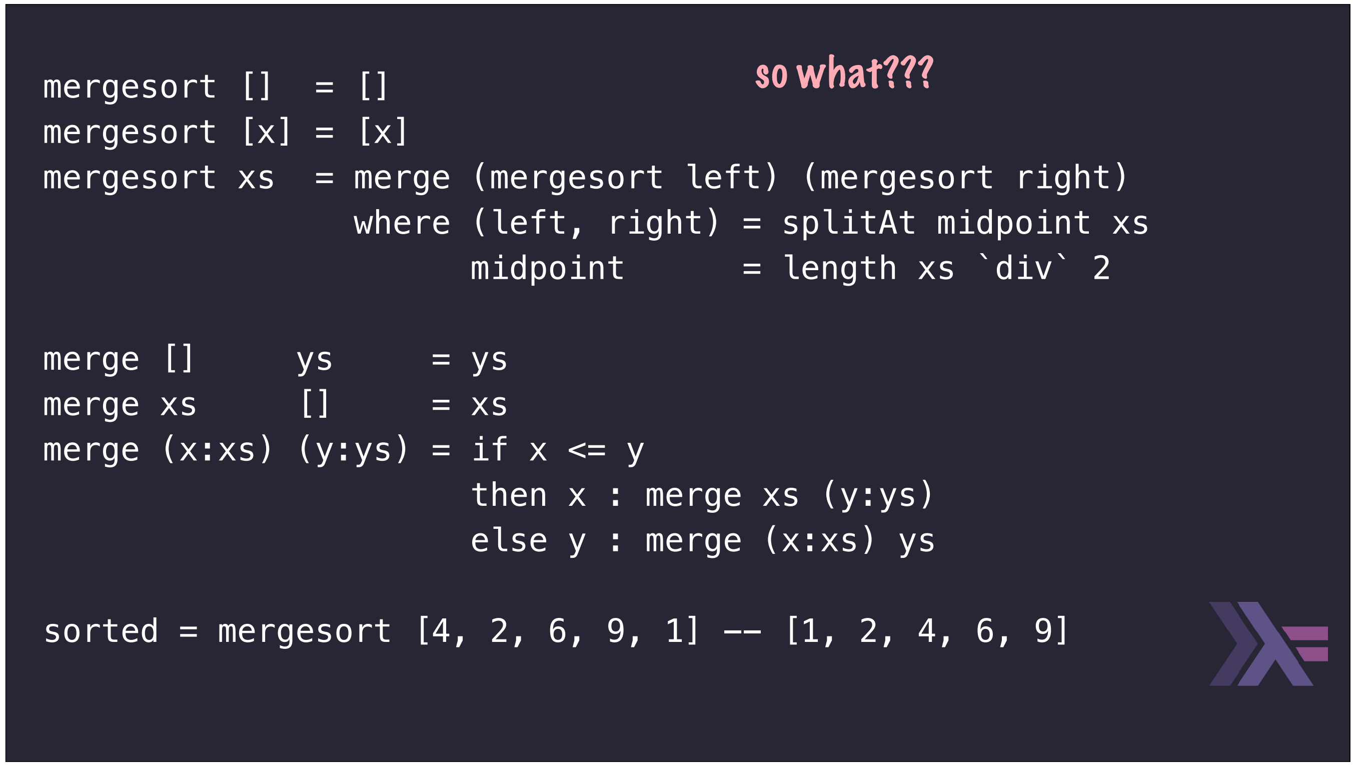
[SKIPPABLE]



```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

sorted = mergesorting this particular list

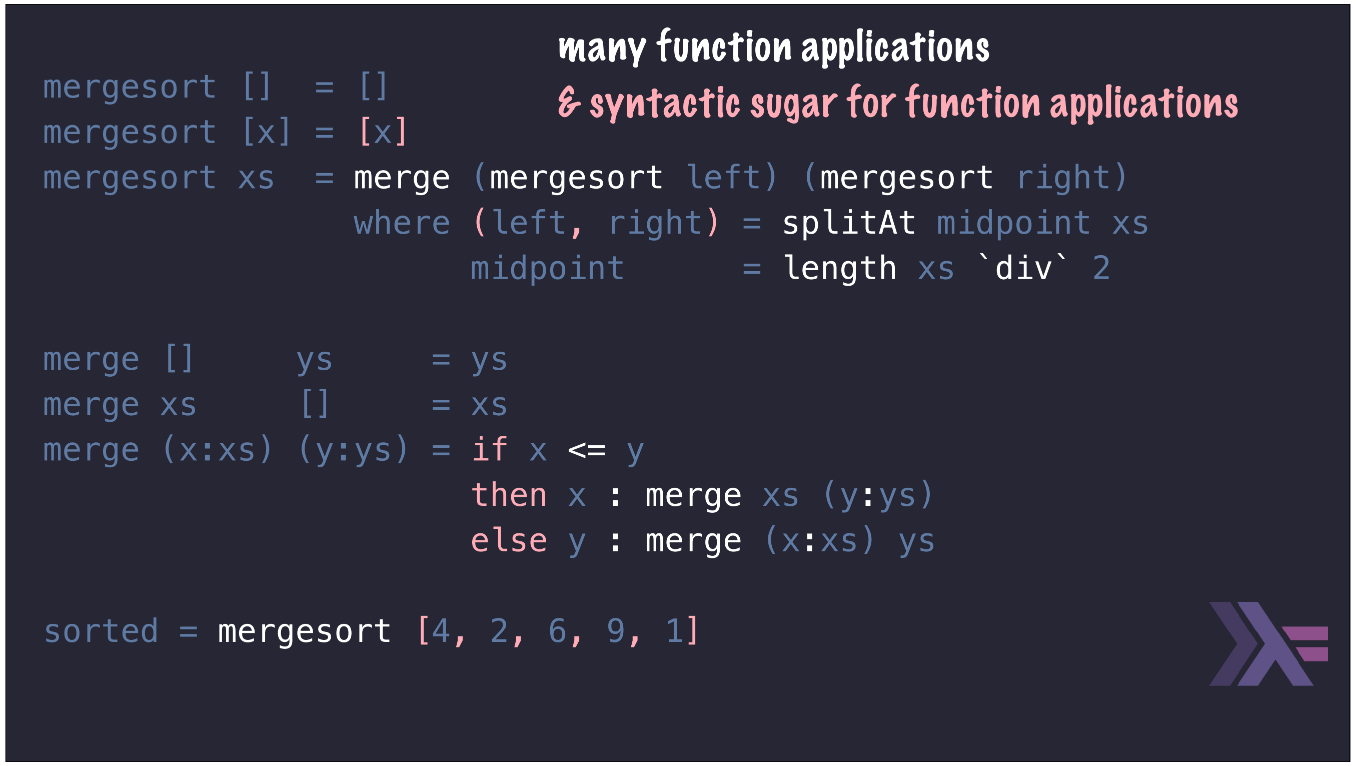
[SKIPPABLE]



so what???

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

So why show you Haskell?

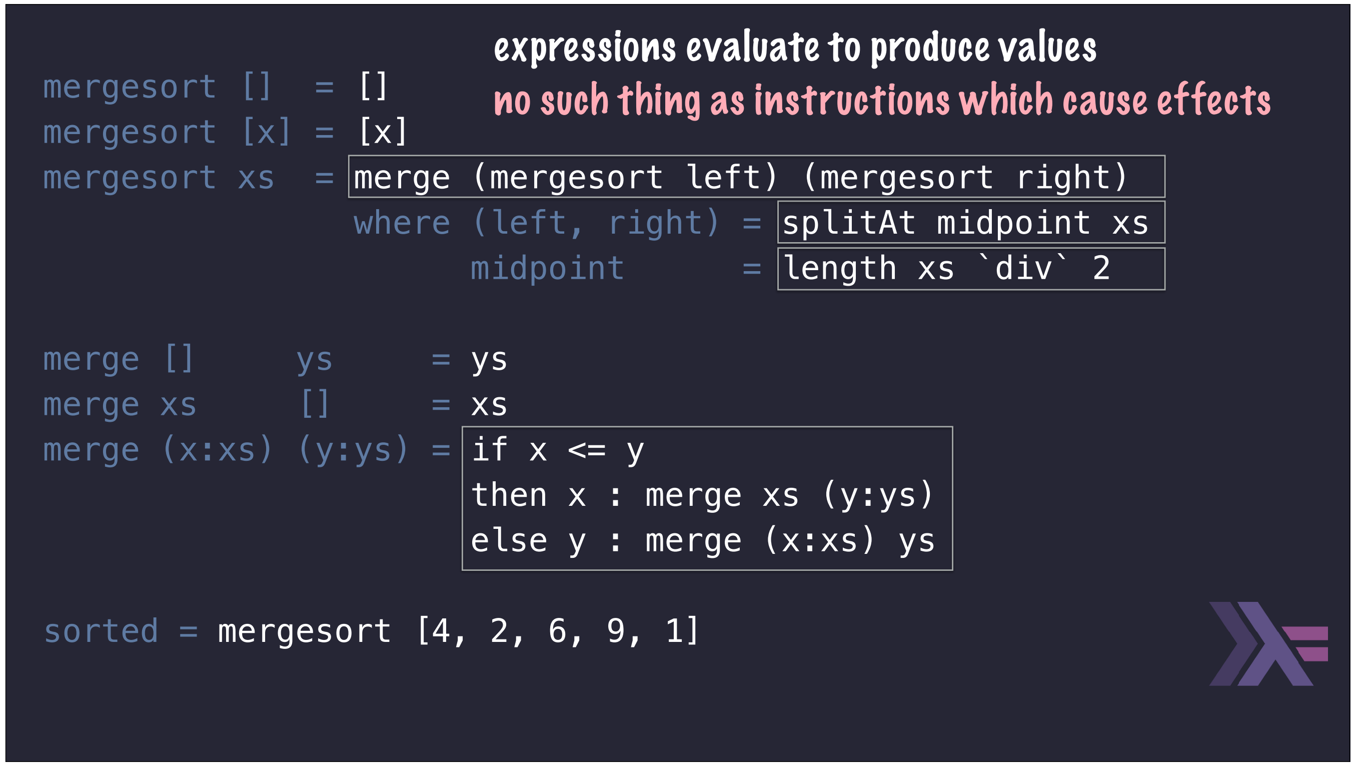


many function applications

& syntactic sugar for function applications

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

Functional programming uses **lots** of **functions** (obviously). Even operators (`==`, `<`) and data constructors (`( , )`, `[ , , ]`) are actually functions or syntactic sugar for functions.

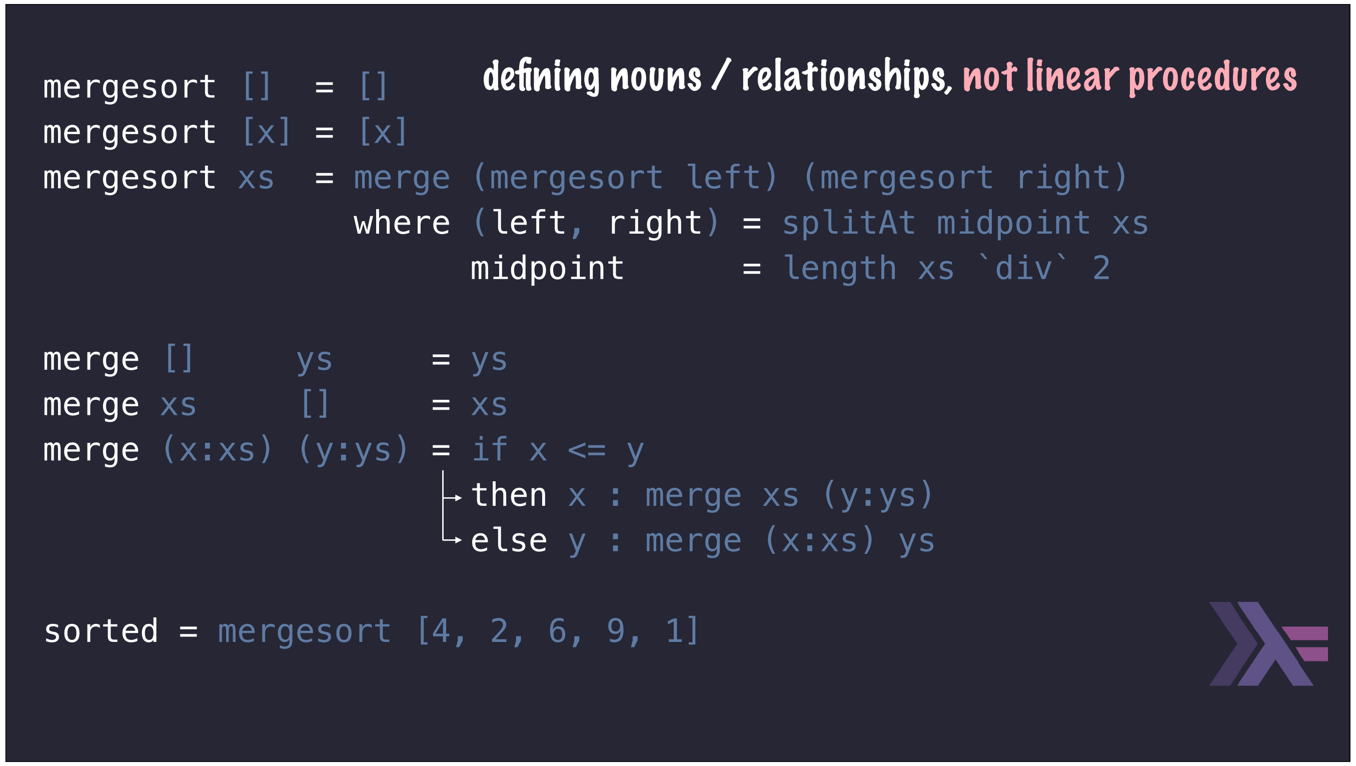


expressions evaluate to produce values

no such thing as instructions which cause effects

```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

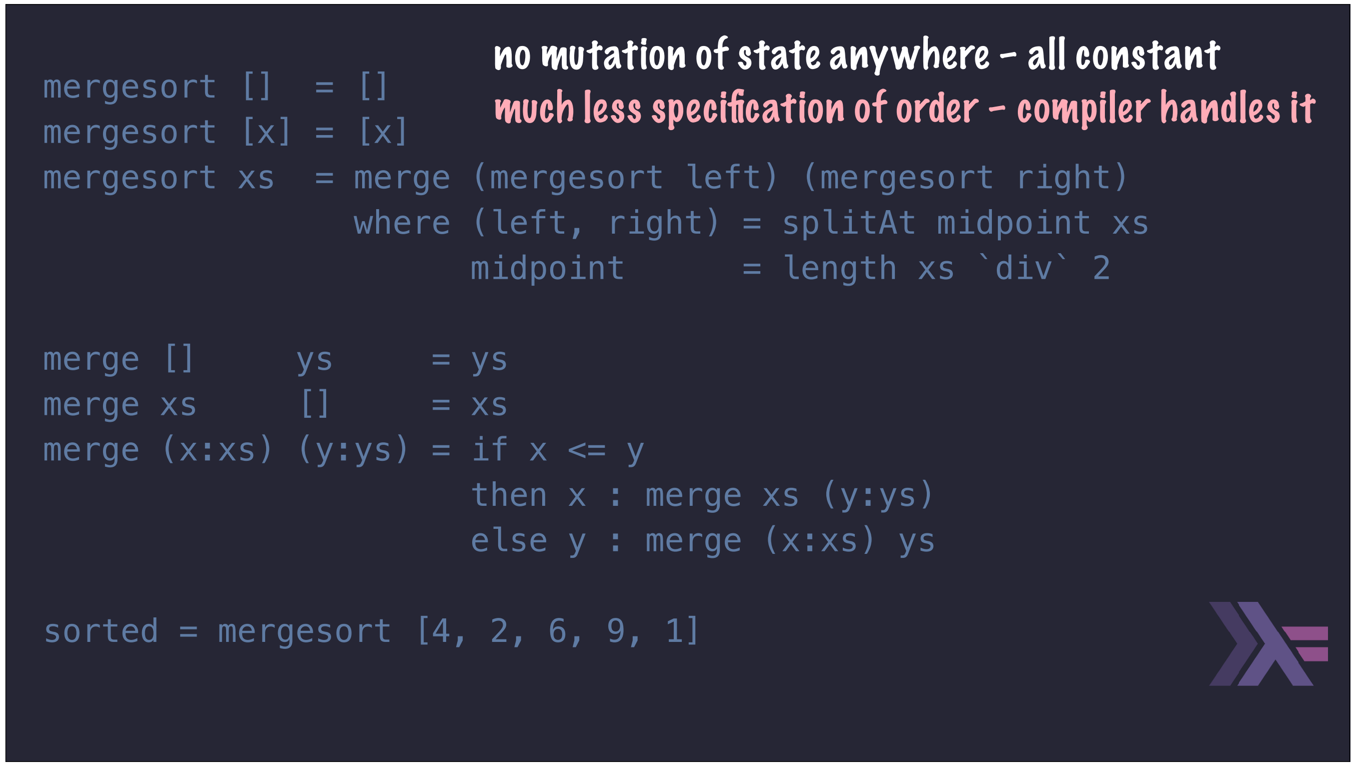
Functions **produce values**. They **do not** "do actions" or cause changes. This separates **functions** from **procedures**. Even "if-then-else" produces a value, i.e. it's a JS ternary. Expressions are built from nested sub-expressions.



defining nouns / relationships, not linear procedures

```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                    ├→then x : merge xs (y:ys)
                    └→else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

Things are specified in terms of *what they are* or *their relationships*. This means **functional** programming overlaps with **declarative** programming.

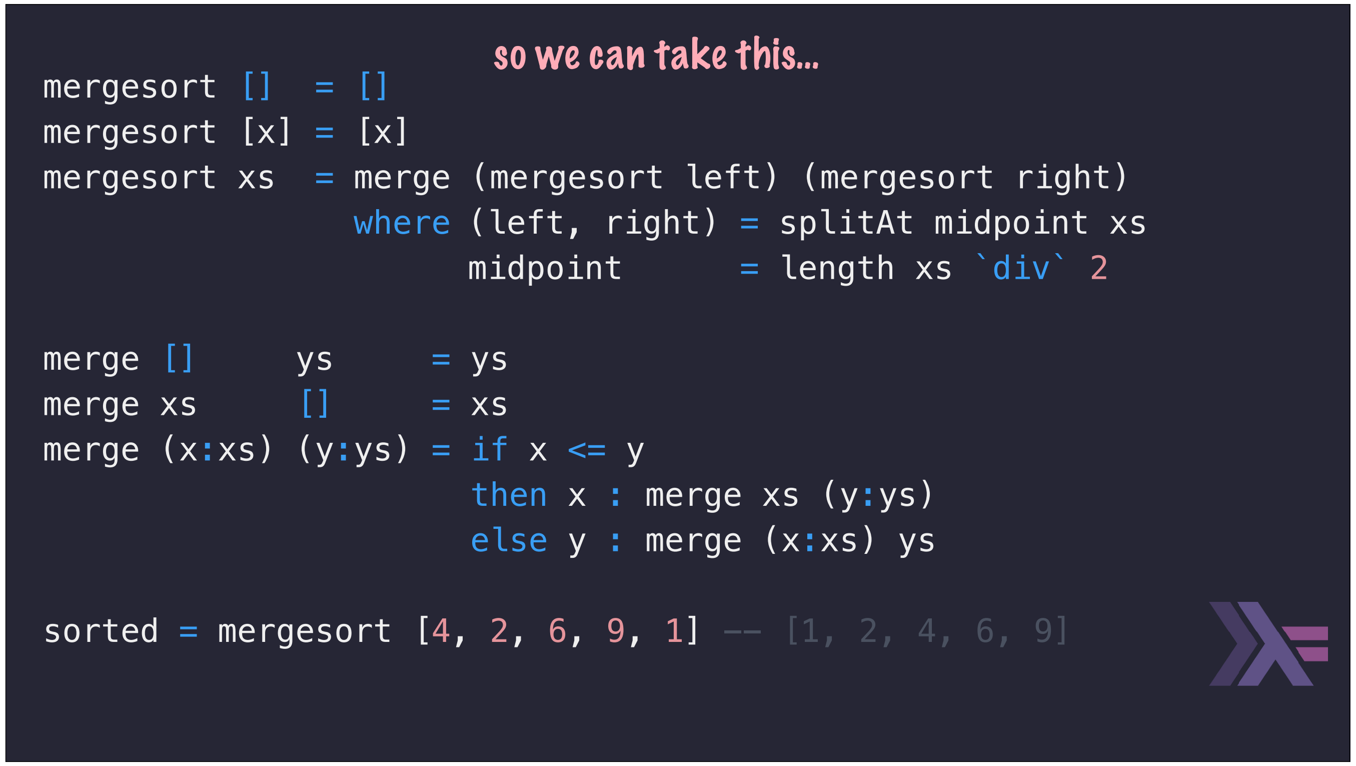


no mutation of state anywhere - all constant

much less specification of order - compiler handles it

```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

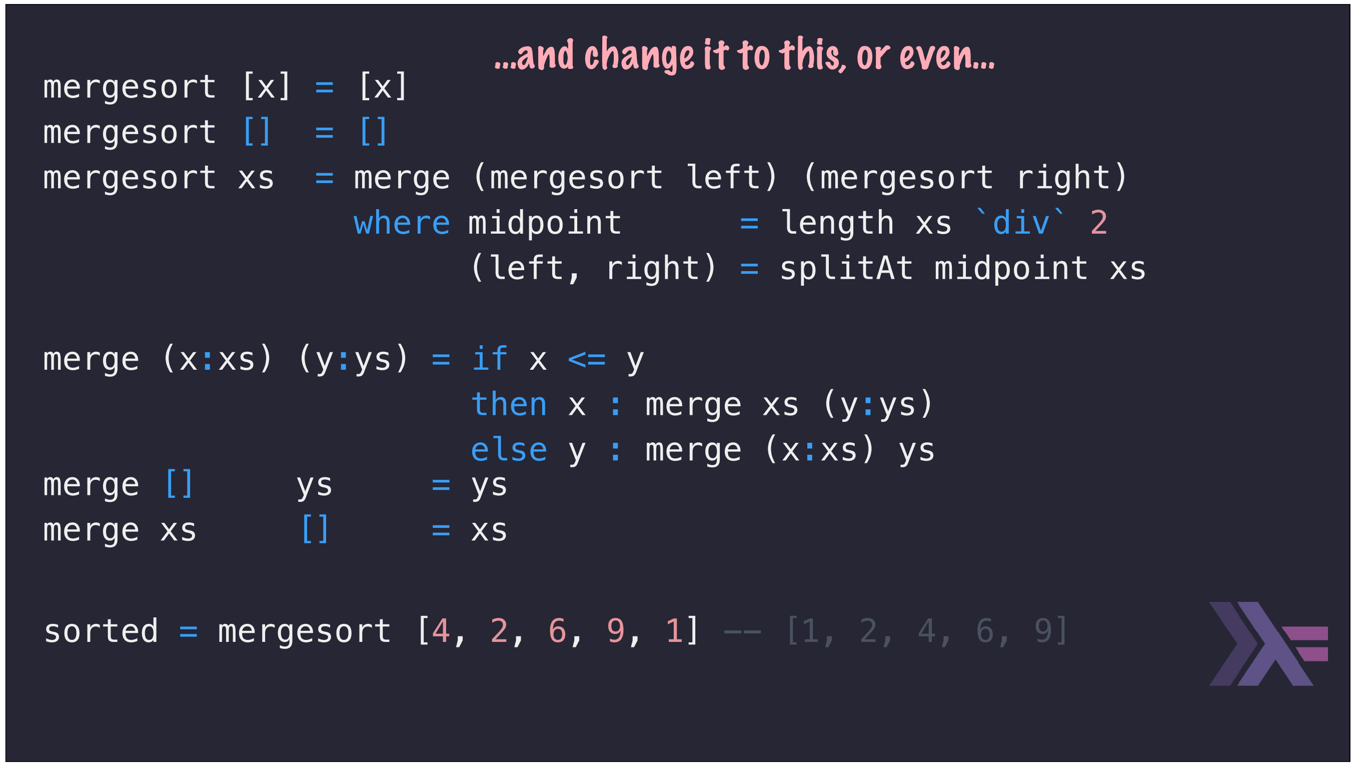
Therefore, code order doesn't matter quite as much. The compiler figures out what order to perform work in many cases. Also, in pure FP you never change a value; all data is immutable. So it doesn't matter *when* or *if* a function runs, it cannot cause a problem in your program.



so we can take this...

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `div` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

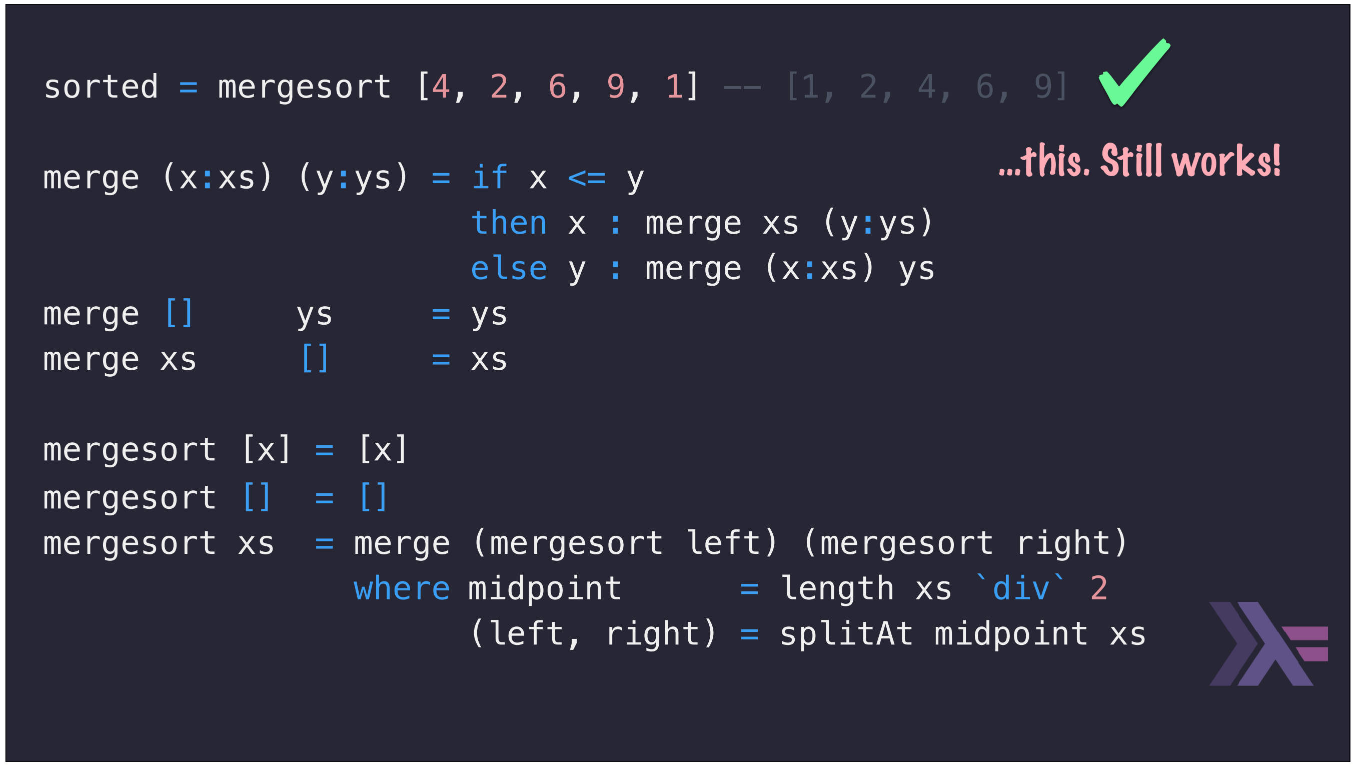
So we can modify this…



...and change it to this, or even...

```haskell
mergesort [x] = [x]
mergesort []  = []
mergesort xs  = merge (mergesort left) (mergesort right)
                where midpoint      = length xs `div` 2
                      (left, right) = splitAt midpoint xs

merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
merge []     ys     = ys
merge xs     []     = xs

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

…to this…



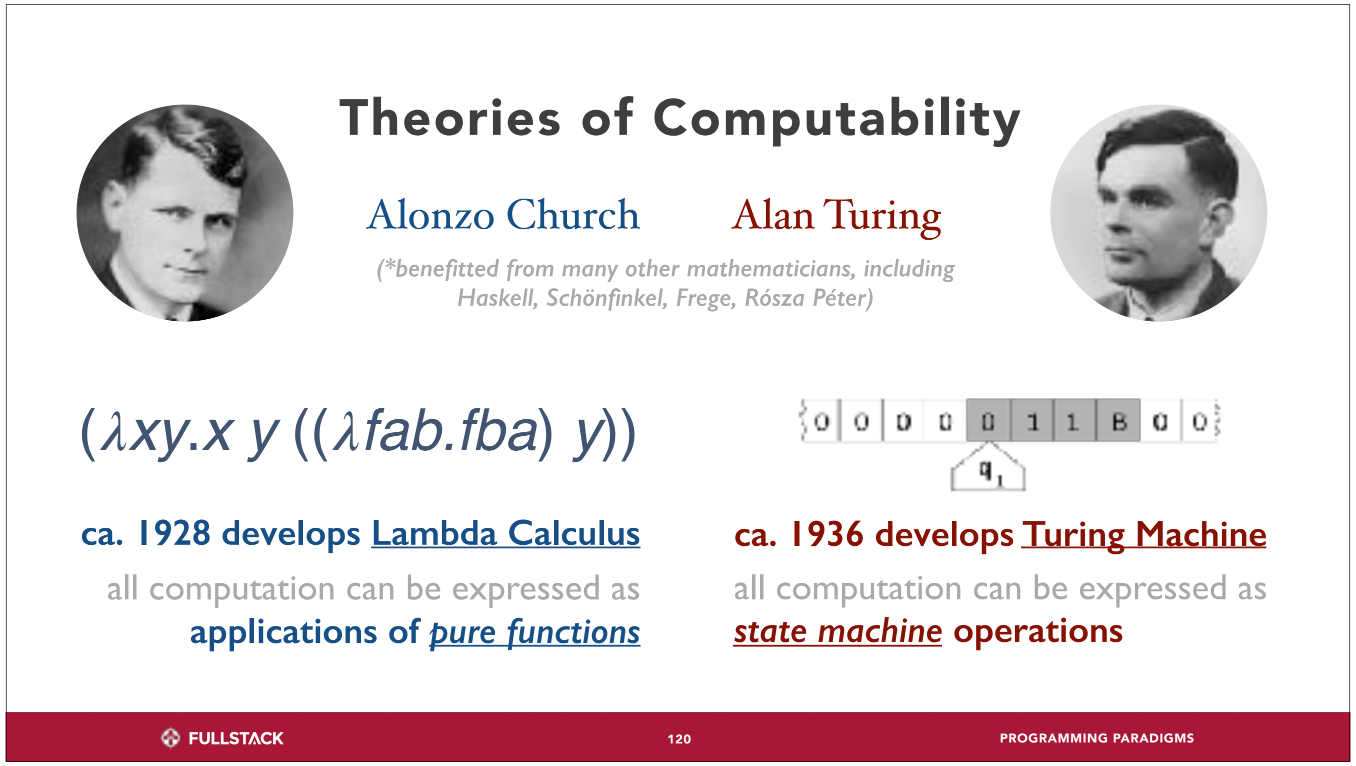
```haskell
sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]

merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
merge []     ys     = ys
merge xs     []     = xs

mergesort [x] = [x]
mergesort []  = []
mergesort xs  = merge (mergesort left) (mergesort right)
                where midpoint      = length xs `div` 2
                      (left, right) = splitAt midpoint xs
```

…this. Still works!

…and even this, and it all still works perfectly. (*Some* order still matters, e.g. arguments & pattern matching – but it is still a *lot less* order).

HISTORY λ

We will touch on some history briefly, for flavor, and to underscore some of the most important foundations of FP.



Theories of Computability

Alonzo Church

Alan Turing

(*benefitted from many other mathematicians, including Haskell, Schönfinkel, Frege, Rósza Péter)

$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y))$

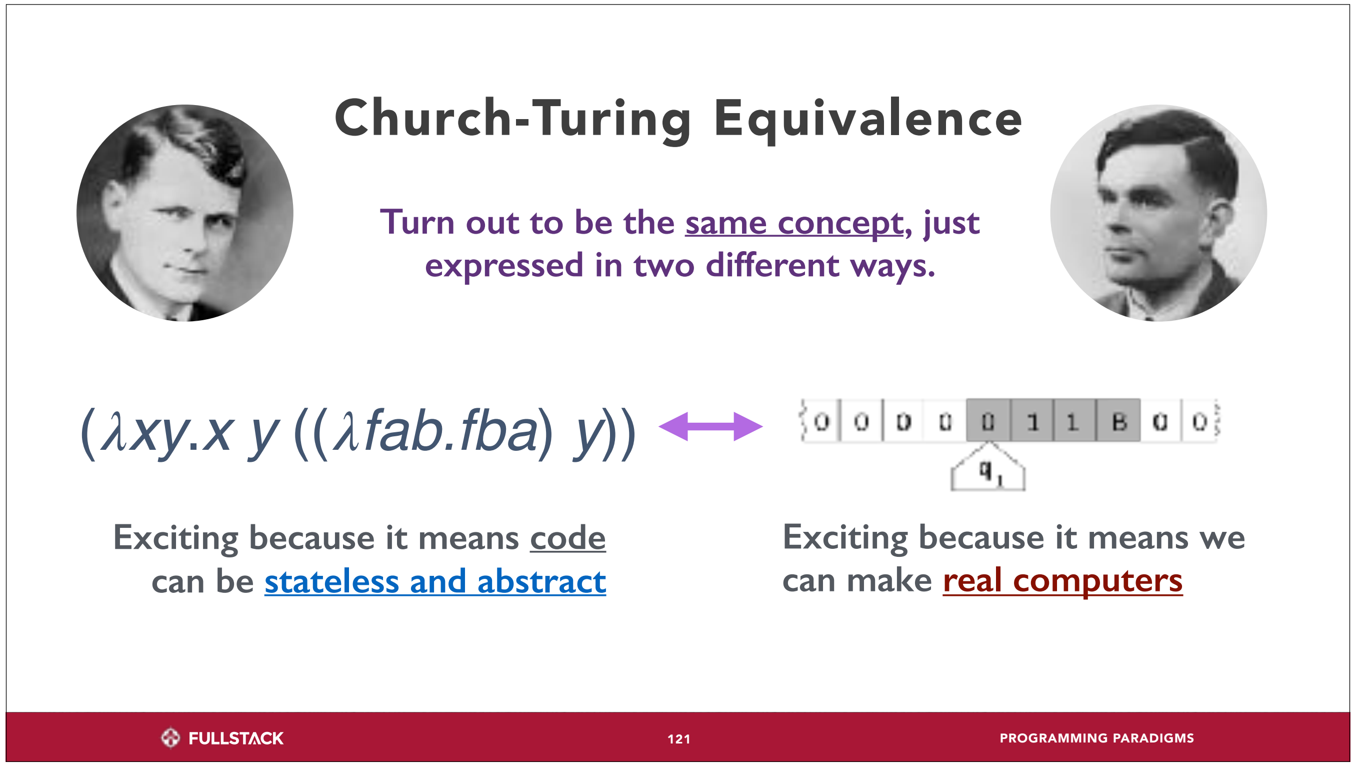
ca. 1928 develops Lambda Calculus

all computation can be expressed as **applications of *pure functions***

ca. 1936 develops Turing Machine

all computation can be expressed as ***state machine* operations**

In the 1920s/30s, mathematicians were building on earlier efforts to define the foundations of logic. One branch, computability theory, benefitted tremendously from the efforts of Alonzo Church and Alan Turing. Church developed Lambda Calculus, and Turing later published Turing Machines.



Church-Turing Equivalence

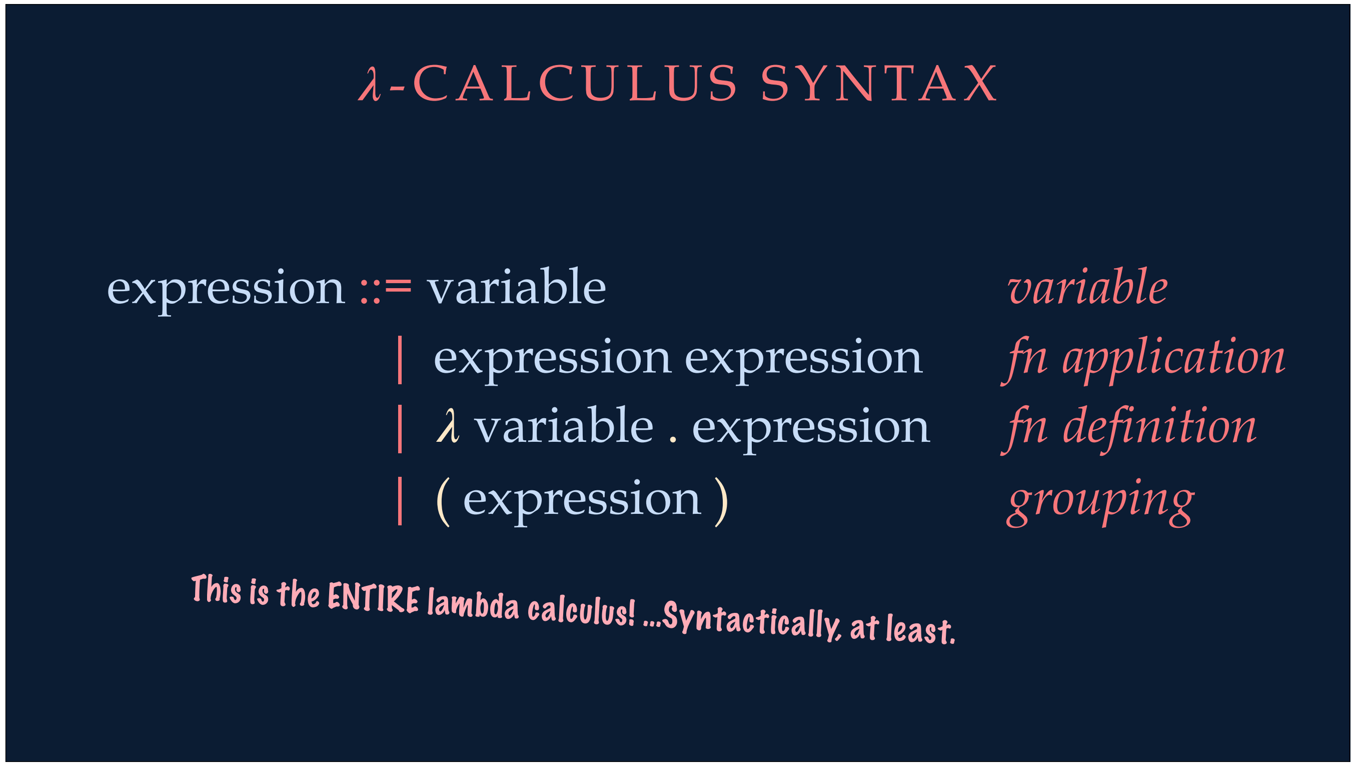
Turn out to be the same concept, just expressed in two different ways.

$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y))$ ↔

Exciting because it means code can be stateless and abstract

Exciting because it means we can make real computers

It turned out that both were identically powerful / totally equivalent systems, expressed quite differently. Turing Machines get a lot of press because they are a hypothetical machine which can compute anything; from his work, people developed real computers. LC however is exciting because it has no concept of state; it is entirely abstracted away from any notion of machine.

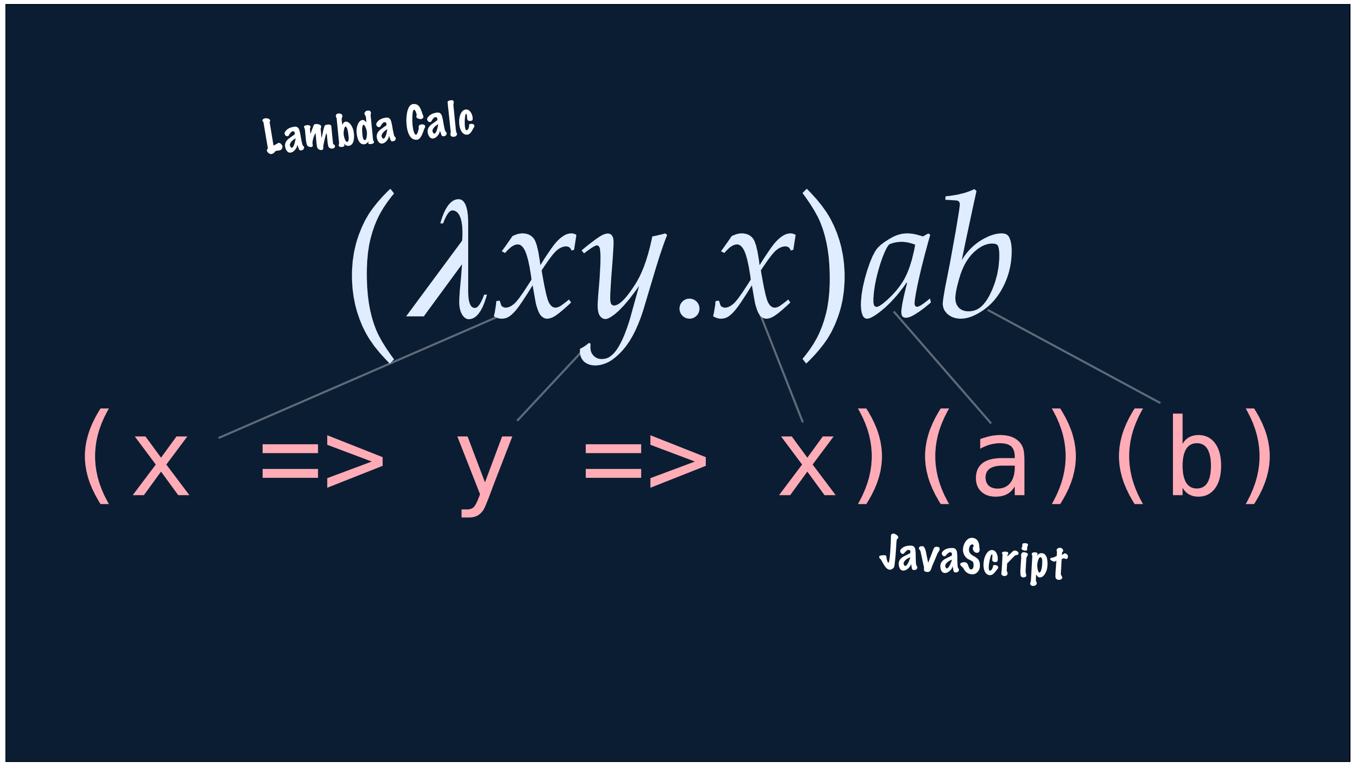


λ-CALCULUS SYNTAX

```
expression ::= variable                         variable
             |  expression expression           fn application
             |  λ variable . expression         fn definition
             |  ( expression )                  grouping
```

This is the ENTIRE lambda calculus! ...Syntactically, at least.

[SKIPPABLE] LC has been called the world's smallest programming language. Here it is, in its entirety. This alone (when you learn the rules of what you can do with these things) is capable of computing anything – including arithmetic and branching logic.



Lambda Calc

$$(\lambda xy.x)ab$$

```
(x => y => x)(a)(b)
```

JavaScript

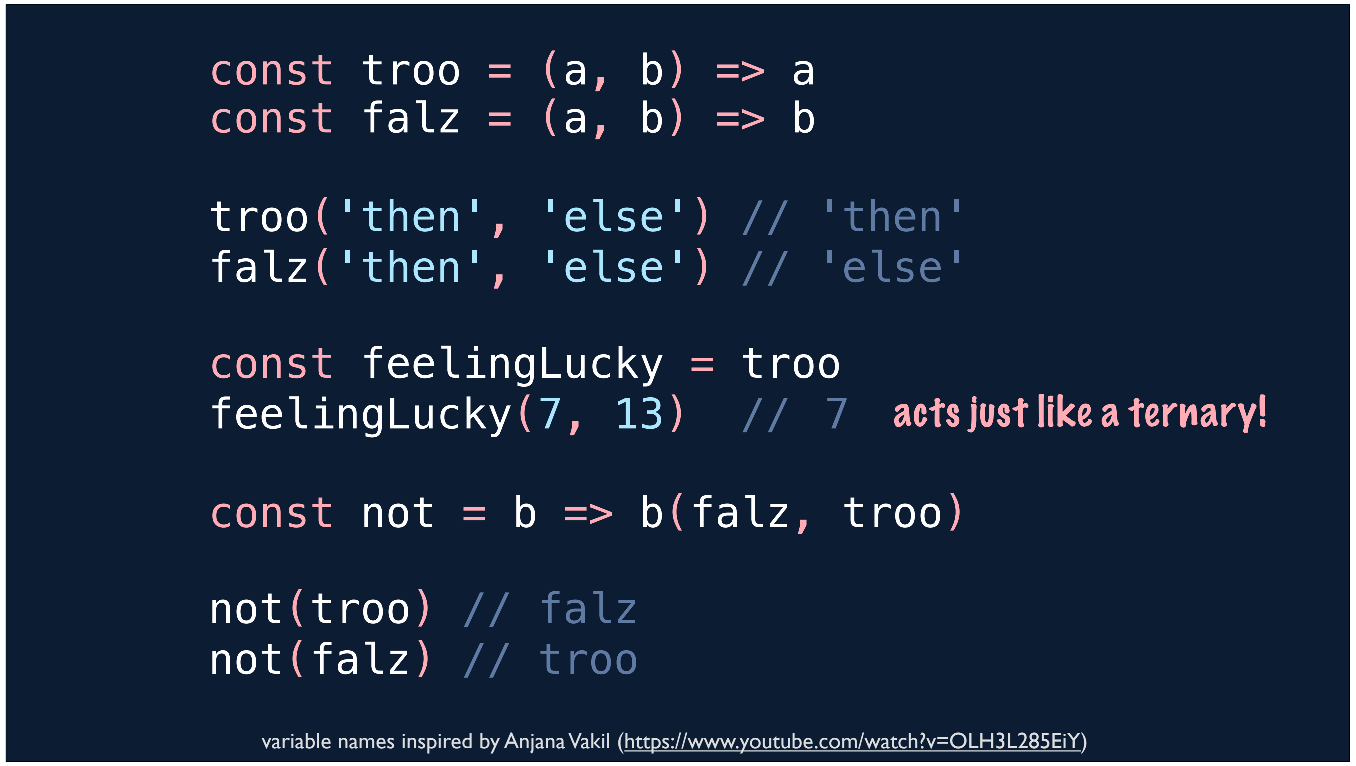
[SKIPPABLE] Lambda calculus has only a few rules. Everything in it is functions. No numbers, no booleans, no nothing. Only functions (and purely abstract variables, which might stand for functions). A "lambda" really just means a **unary, anonymous, pure, first-class function.**

EVERYTHING

CAN BE

FUNCTIONS

[SKIPPABLE] Gabriel L. says: I have a talk on this which I do sometimes (or it is also recorded on YouTube https://www.youtube.com/watch?v=3VQ382QG-y4).

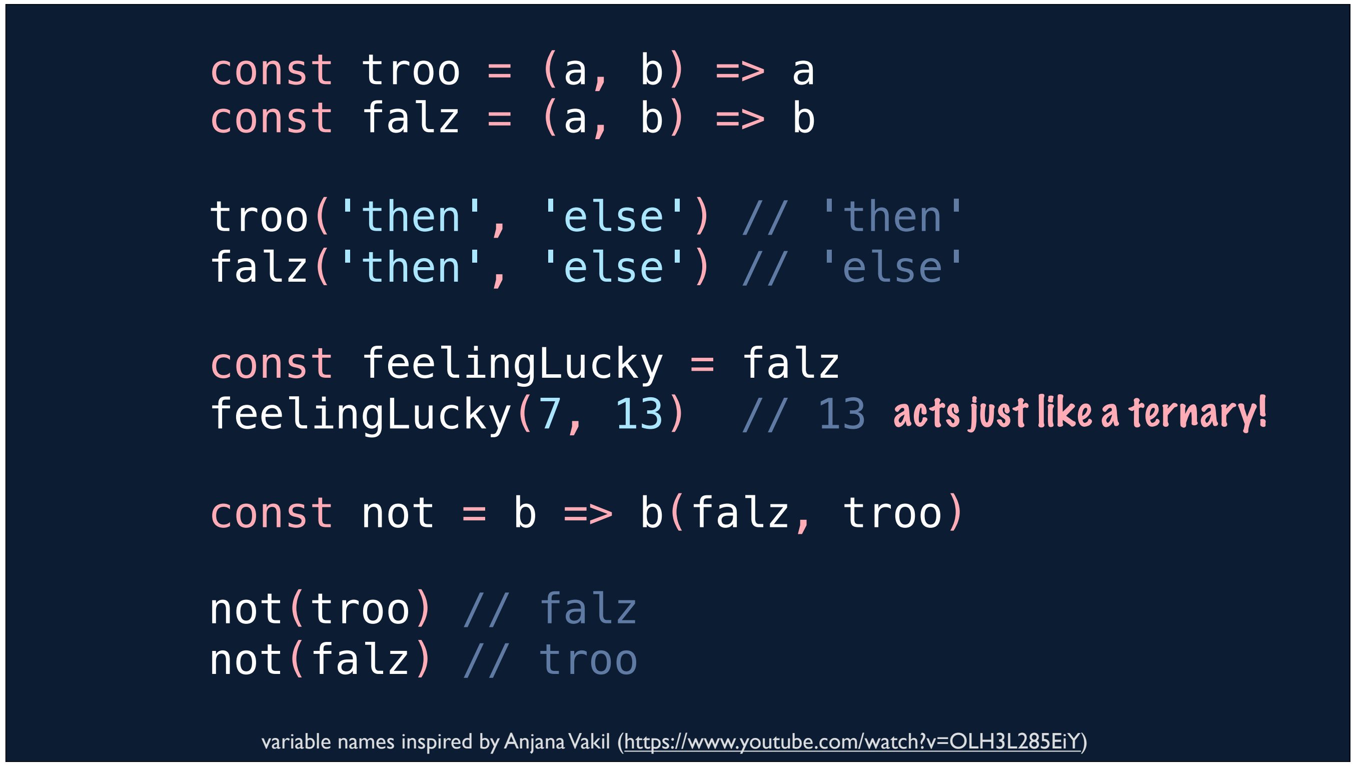


```
const troo = (a, b) => a
const falz = (a, b) => b

troo('then', 'else') // 'then'
falz('then', 'else') // 'else'

const feelingLucky = troo
feelingLucky(7, 13)  // 7

const not = b => b(falz, troo)

not(troo) // falz
not(falz) // troo
```

acts just like a ternary!

variable names inspired by Anjana Vakil (https://www.youtube.com/watch?v=OLH3L285EiY)

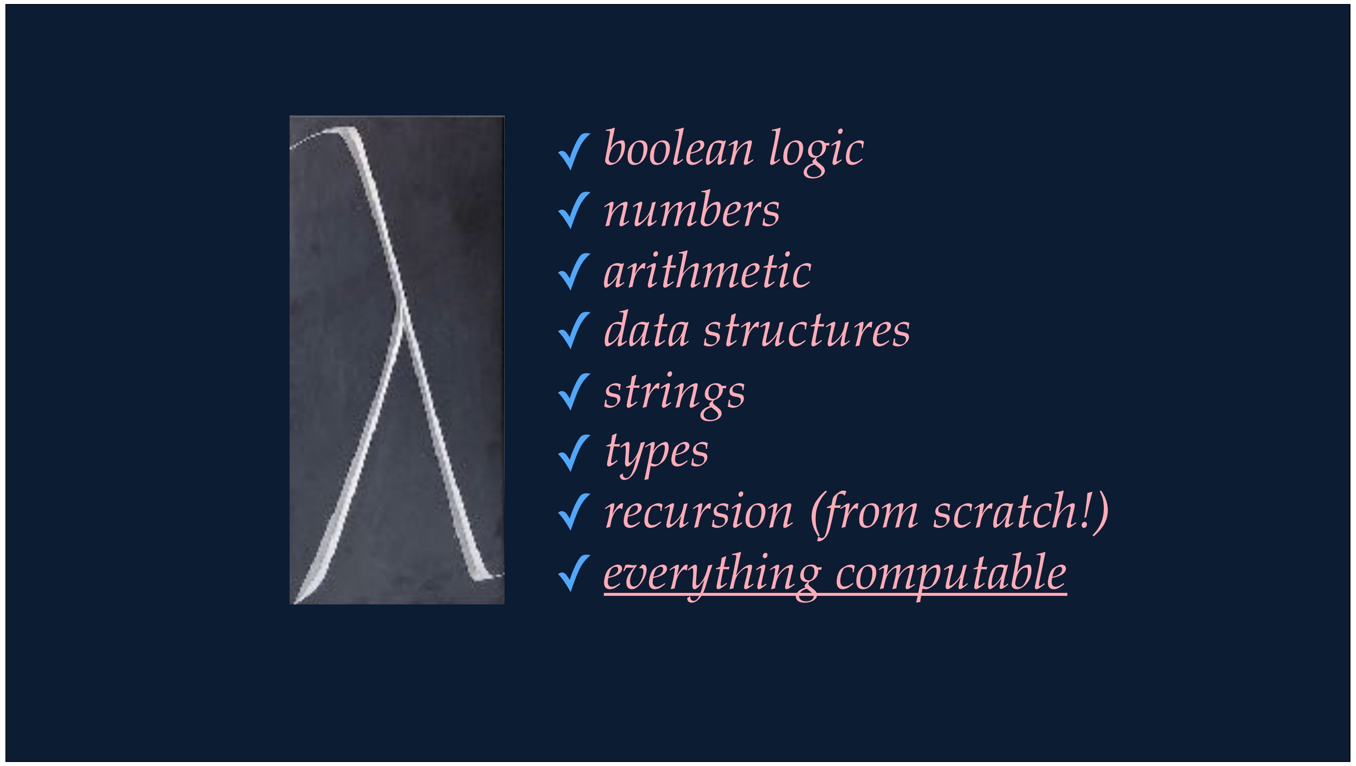
[SKIPPABLE] Small (slightly non-LC) example: create booleans & not from scratch, using only arrows. This actually does work.
https://repl.it/@glebec/booleansAsFunctions



```
const troo = (a, b) => a
const falz = (a, b) => b

troo('then', 'else') // 'then'
falz('then', 'else') // 'else'

const feelingLucky = falz
feelingLucky(7, 13)  // 13  acts just like a ternary!

const not = b => b(falz, troo)

not(troo) // falz
not(falz) // troo
```

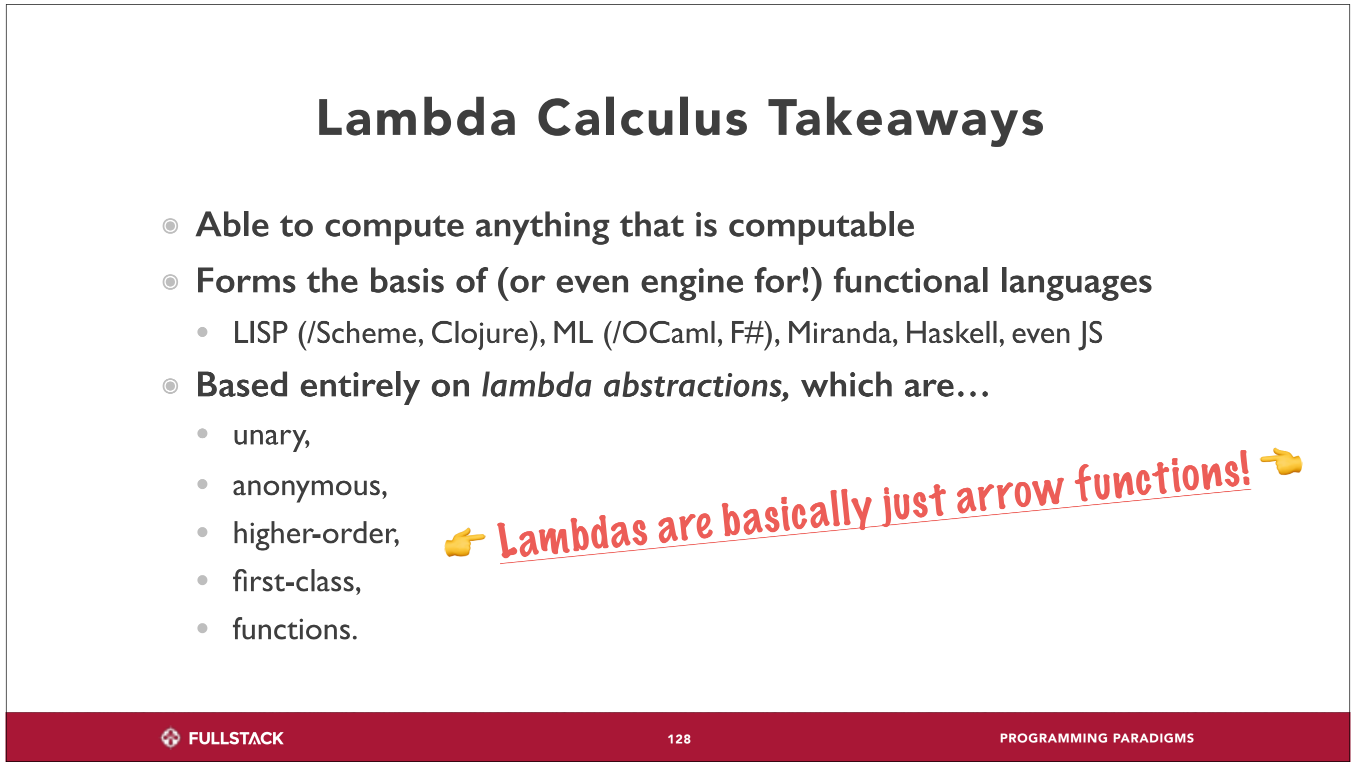
variable names inspired by Anjana Vakil (https://www.youtube.com/watch?v=OLH3L285EiY)

[SKIPPABLE] Small (slightly non-LC) example: create booleans & not from scratch, using only arrows. This actually does work.
https://repl.it/@glebec/LightpinkProudSearch



✓ *boolean logic*
✓ *numbers*
✓ *arithmetic*
✓ *data structures*
✓ *strings*
✓ *types*
✓ *recursion (from scratch!)*
✓ *everything computable*

Just as we saw with Booleans & Boolean Logic, LC can recreate (from complete scratch!) all these things. The point is, functions are incredibly capable.

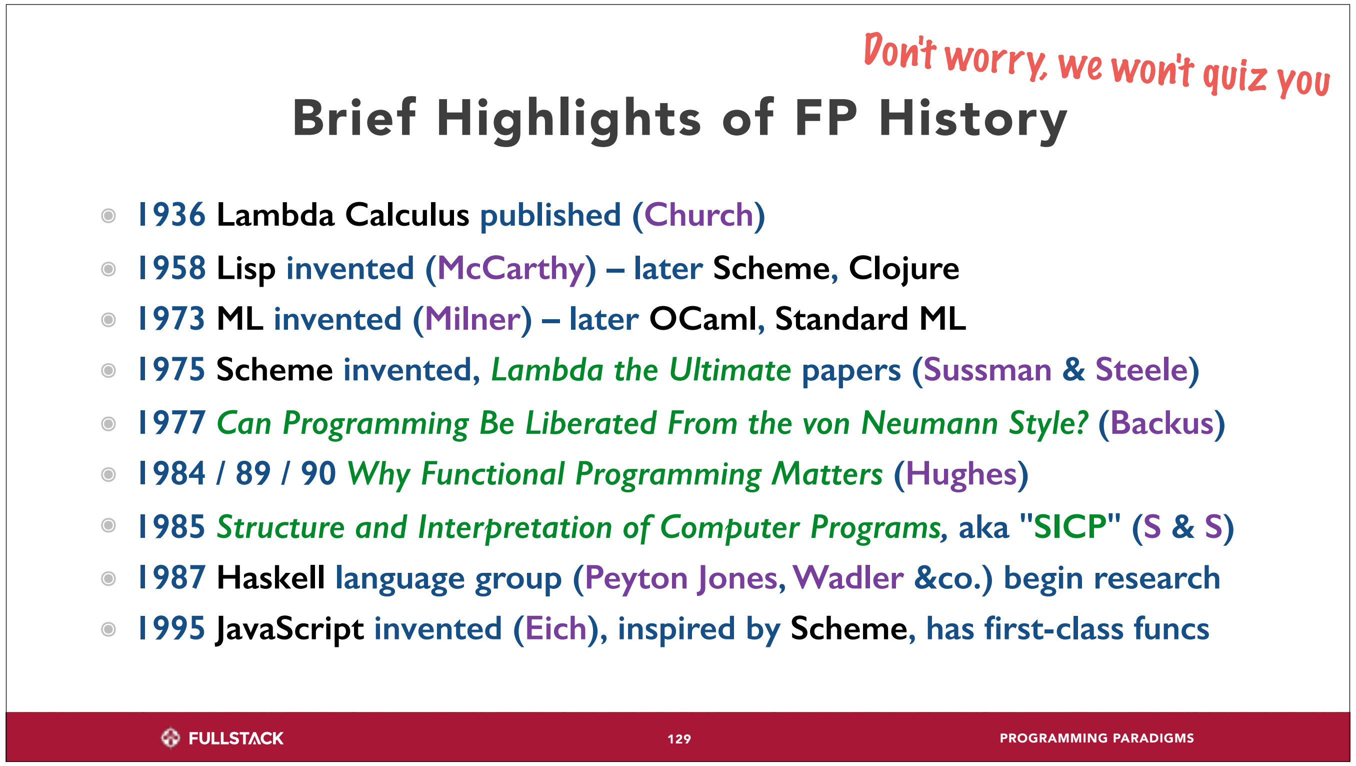


Lambda Calculus Takeaways

- **Able to compute anything that is computable**
- **Forms the basis of (or even engine for!) functional languages**
  - LISP (/Scheme, Clojure), ML (/OCaml, F#), Miranda, Haskell, even JS
- **Based entirely on *lambda abstractions,* which are…**
  - unary,
  - anonymous,
  - higher-order,
  - first-class,
  - functions.

👉 Lambdas are basically just arrow functions! 👈

When someone says "that language has lambdas", think "arrow functions" (not in terms of syntax, but by being anonymous first-class funcs).



Don't worry, we won't quiz you

Brief Highlights of FP History

- 1936 Lambda Calculus published (Church)
- 1958 Lisp invented (McCarthy) – later Scheme, Clojure
- 1973 ML invented (Milner) – later OCaml, Standard ML
- 1975 Scheme invented, *Lambda the Ultimate* papers (Sussman & Steele)
- 1977 *Can Programming Be Liberated From the von Neumann Style?* (Backus)
- 1984 / 89 / 90 *Why Functional Programming Matters* (Hughes)
- 1985 *Structure and Interpretation of Computer Programs*, aka "SICP" (S & S)
- 1987 Haskell language group (Peyton Jones, Wadler &co.) begin research
- 1995 JavaScript invented (Eich), inspired by Scheme, has first-class funcs

[SKIPPABLE] Feel free to gloss over this slide, which naturally cannot come close to capturing all the important milestones anyway.

5 minute break!