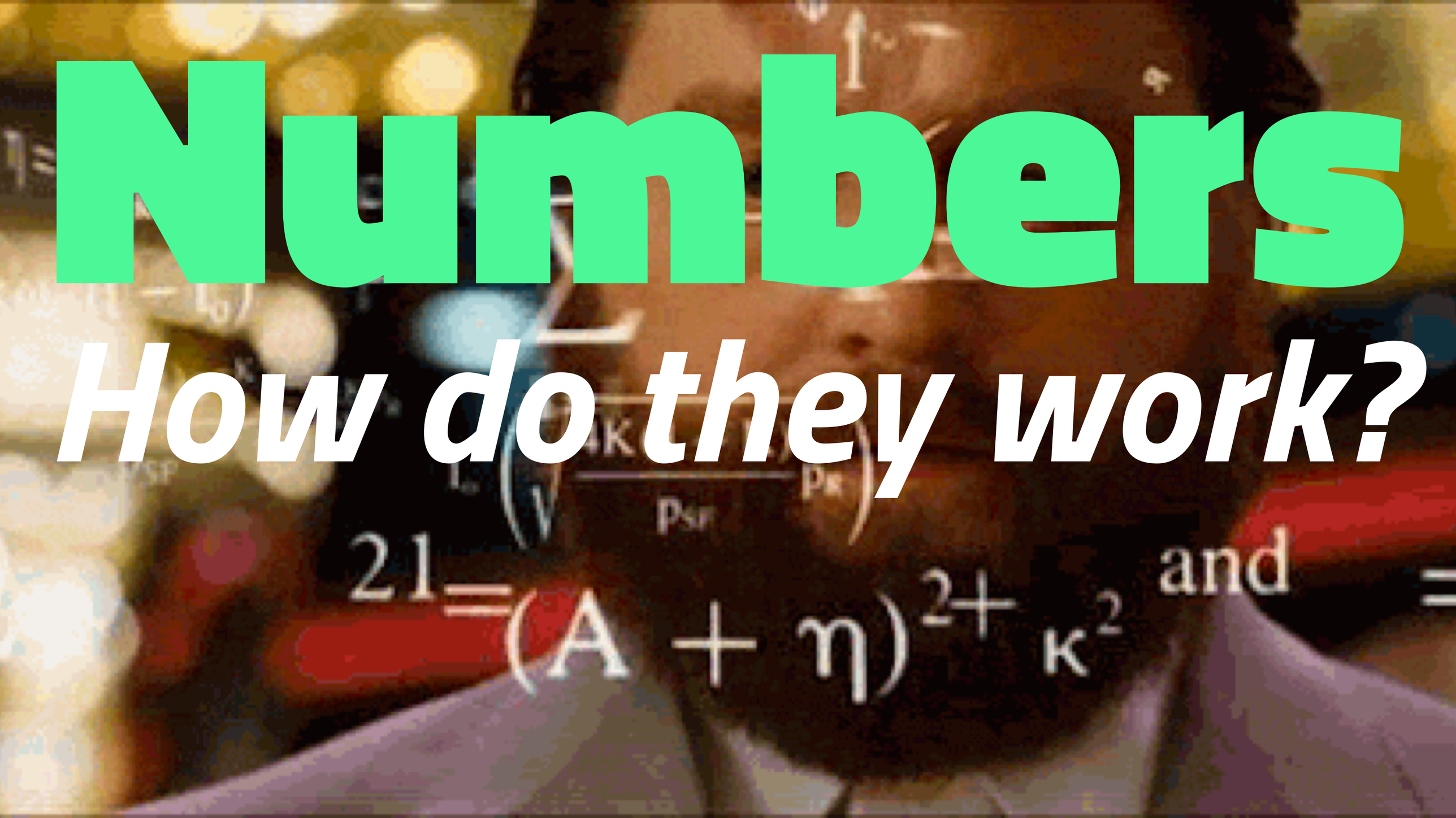
Exploring Swift's numeric types and protocols

ints, floats, and doubles — oh my!

Jesse Squires
jessesquires.com ▪ *@jesse_squires*


Numbers
How do they work?
21=(A+η)2+κ2 and

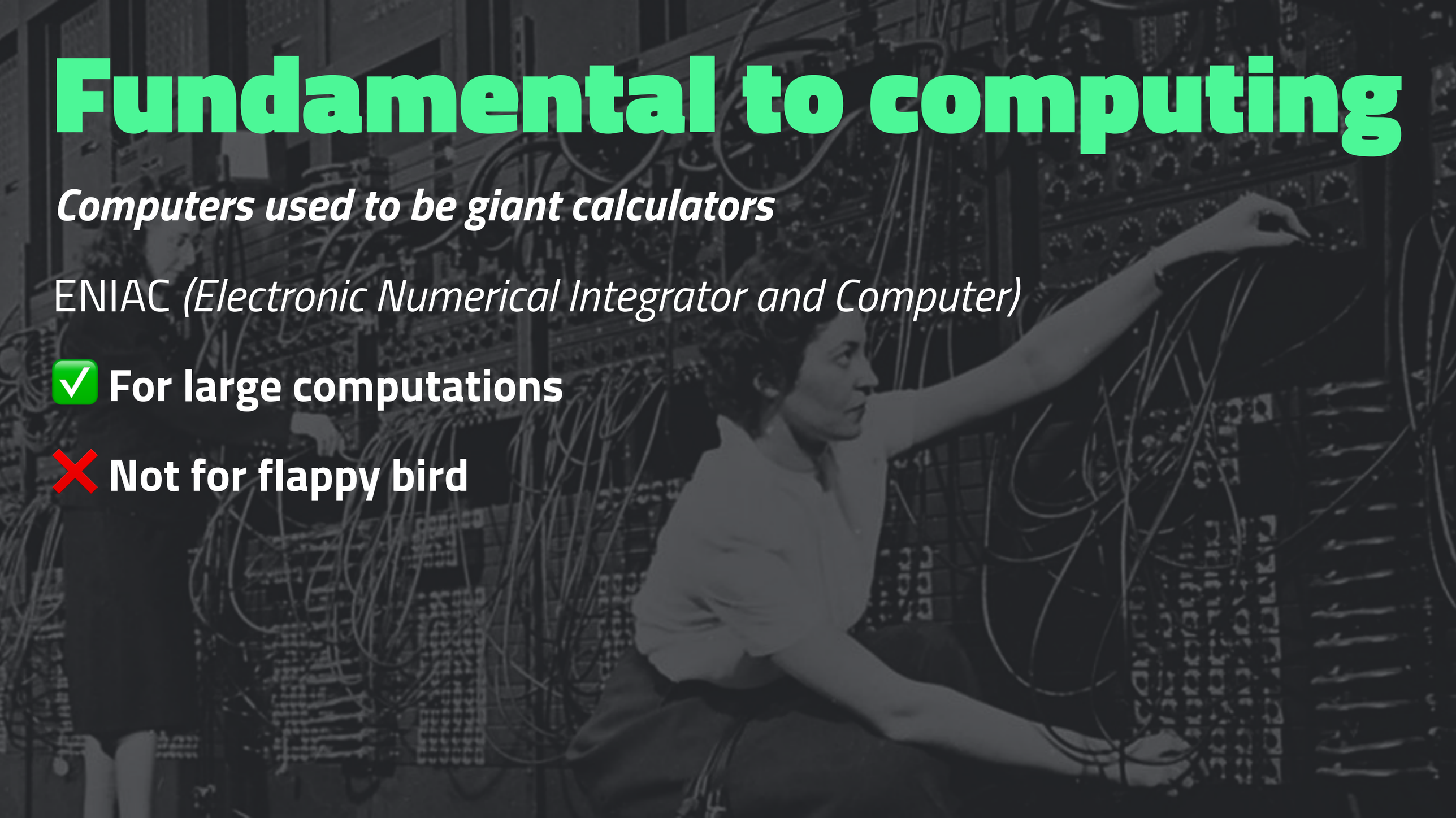
Fundamental to computing

Computers used to be giant calculators

ENIAC *(Electronic Numerical Integrator and Computer)*

✅ **For large computations**

❌ **Not for flappy bird**



Swift's numeric types

Floating-point

```
Float, Double, Float80
```

Integers

```
Int // platform native word size
Int8, Int16, Int32, Int64
```

Unsigned Integers

```
UInt // platform native word size
UInt8, UInt16, UInt32, UInt64
```

Why so many types? *(sizes)*

- Different processor architectures
- Inter-op with C

  (C functions, `char *[]` imported as `Int8` tuple)
- Inter-op with Objective-C

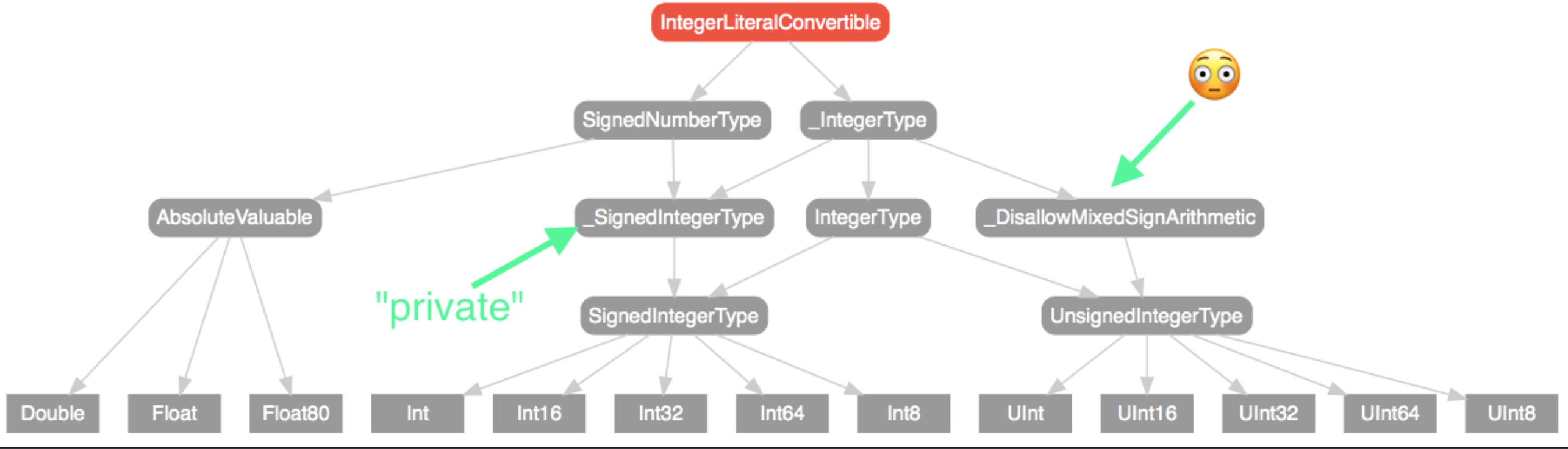
  (`BOOL` is a `typedef char`)
- SQLite / CoreData
- IoT sensors (heart rate monitor)
- Embedded systems programming

Before Swift 3 and 4

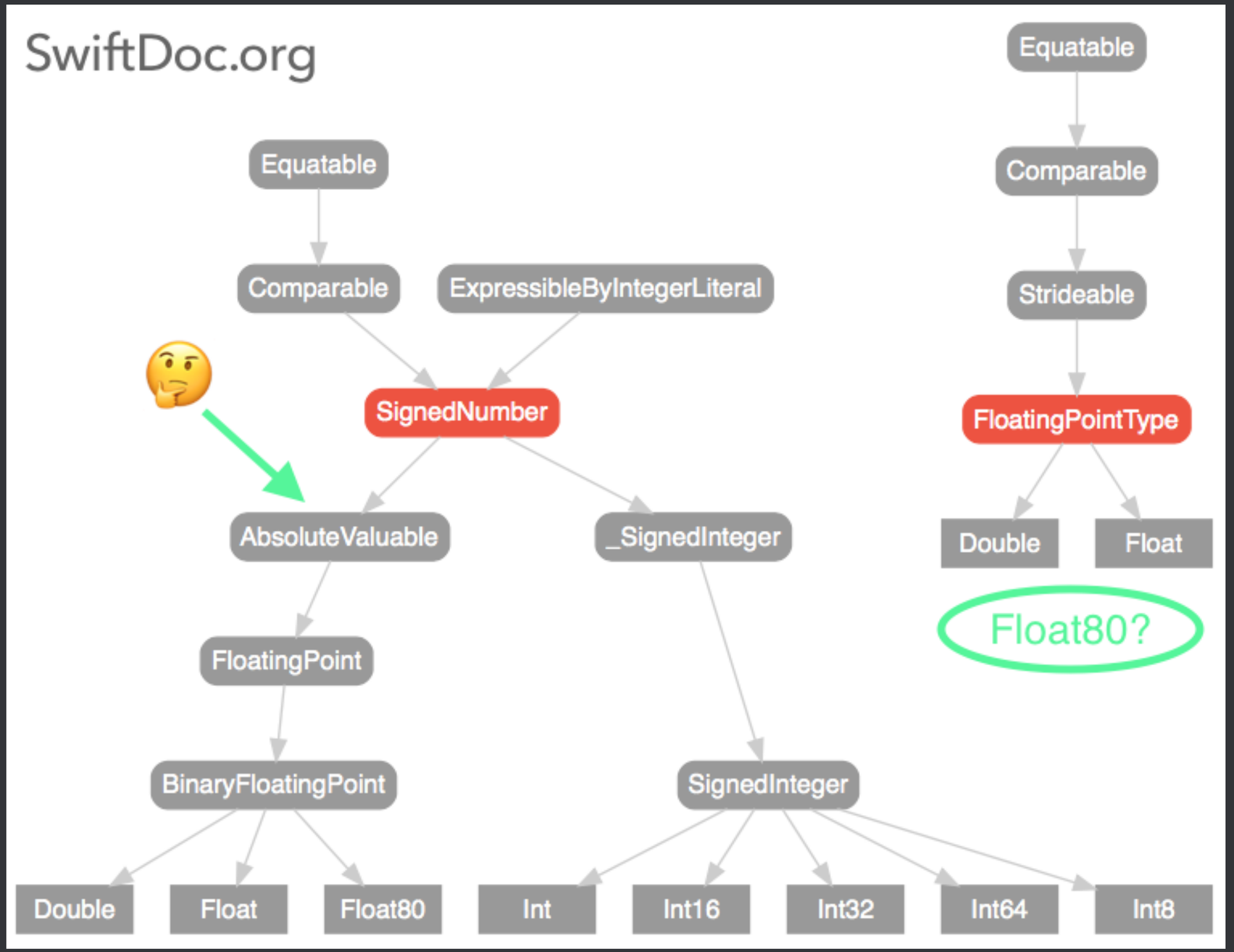
- Difficult to work with numeric types
- Difficult to extend numeric types
- **`FloatingPoint`** did not have all IEEE 754 features
- Generally rough around the edges

Swift 2

SwiftDoc.org



Swift 2


SwiftDoc.org
Equatable
Comparable
ExpressibleByIntegerLiteral
SignedNumber
AbsoluteValuable
_SignedInteger
FloatingPoint
BinaryFloatingPoint
Double
Float
Float80
SignedInteger
Int
Int16
Int32
Int64
Int8
Equatable
Comparable
Strideable
FloatingPointType
Double
Float
Float80?


THIS IS FINE.

Swift evolution

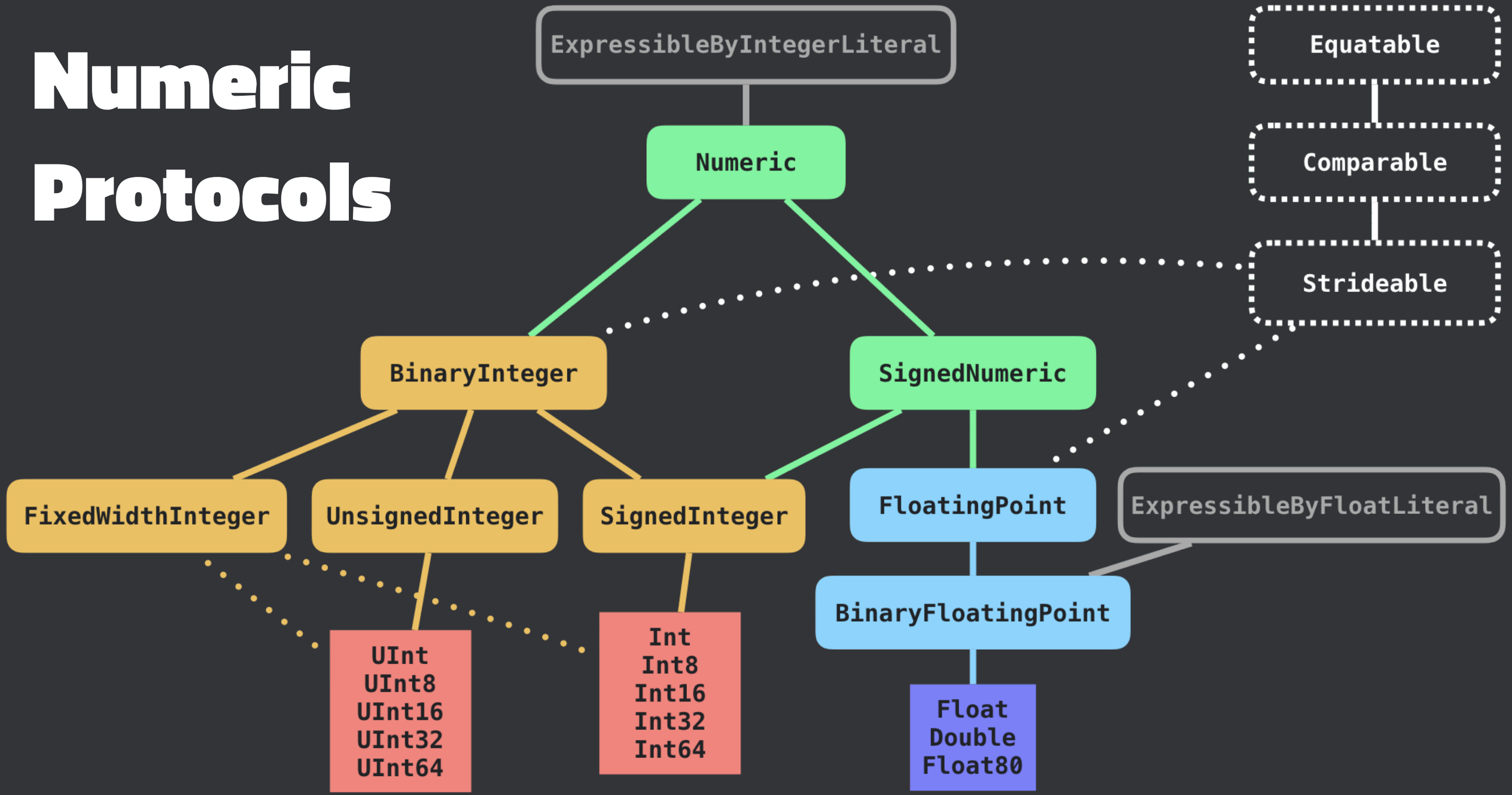
SE-0104: *Protocol-oriented integers* **(Swift 4)**

SE-0113: *Add integral rounding functions to FloatingPoint* **(Swift 3)**

SE-0067: *Enhanced Floating Point Protocols* **(Swift 3)**

- Address API shortcomings
- Refine protocol naming and contents
- Refine protocol hierarchy

Numeric Protocols



protocol Numeric

Binary arithmetic operators **+**, **-**, *****

```
extension Sequence where Element: Numeric {
    func sum() -> Element {
        return reduce(0, +)
    }
}

let sum = [1, 2, 3, 4, 5].sum() // 15
```

protocol SignedNumeric

Types that can represent both positive and negative values (**not `UInt`**)

```
var i = 5; i.negate() // -5

extension Sequence where Element: SignedNumeric & Comparable {
    func filterNegatives() -> [Element] {
        return filter { $0 > 0 }
    }
}

let allPositive = [1, 2, 3, -4, -5].filterNegatives() // [1, 2, 3]
```

protocol BinaryInteger

Basis for all the integer types provided by the standard library

Arithmetic, bitwise, and bit shifting operators **/**, **<<**, **&**, etc

```
// Convert between integer types
let x = Int16(exactly: 500) // Optional(500)
let y = Int8(exactly: 500) // nil

// Truncating - make 'q' to fit in 8 bits
let q: Int16 = 850 // 0b00000011_01010010
let r = Int8(truncatingIfNeeded: q) // 82, 0b01010010

// Compare across types
Int(-42) < Int8(4)        // true
UInt(1_000) < Int16(250) // false
```

protocol FixedWidthInteger

Endianness, type bounds, bit width

```
let x = Int16(127) // 127
x.littleEndian     // 127,   0b00000000_01111111
x.bigEndian        // 32512, 0b01111111_00000000
x.byteSwapped      // 32512, 0b01111111_00000000
```

```
Int16.bitWidth // 16
Int16.min      // -32768
Int16.max      // 32768
```

```swift
extension FixedWidthInteger {
    var binaryString: String {
        var result: [String] = []
        for i in 0..<(Self.bitWidth / 8) {
            let byte = UInt8(truncatingIfNeeded: self >> (i * 8))
            let byteString = String(byte, radix: 2)
            let padding = String(repeating: "0",
                                 count: 8 - byteString.count)
            result.append(padding + byteString)
        }
        return "0b" + result.reversed().joined(separator: "_")
    }
}

let x = Int16(4323)
x.binaryString // 0b00010000_11100011
```

protocol FloatingPoint

Represents fractional numbers, *IEEE 754 specification*

```
func hypotenuse<T: FloatingPoint>(_ a: T, _ b: T) -> T {
    return (a * a + b * b).squareRoot()
}

let (dx, dy) = (3.0, 4.0)
let dist = hypotenuse(dx, dy) // 5.0
```

protocol FloatingPoint

Provides common constants

Precision is that of the concrete type!

```
static var leastNormalMagnitude: Self     // FLT_MIN or DBL_MIN

static var greatestFiniteMagnitude: Self // FLT_MAX or DBL_MAX

static var pi: Self // 🍰 😋
```

`protocol BinaryFloatingPoint`

Specific radix-2 (binary) floating-point type

In the future, there could be a **`DecimalFloatingPoint`** protocol for decimal types (radix-10)

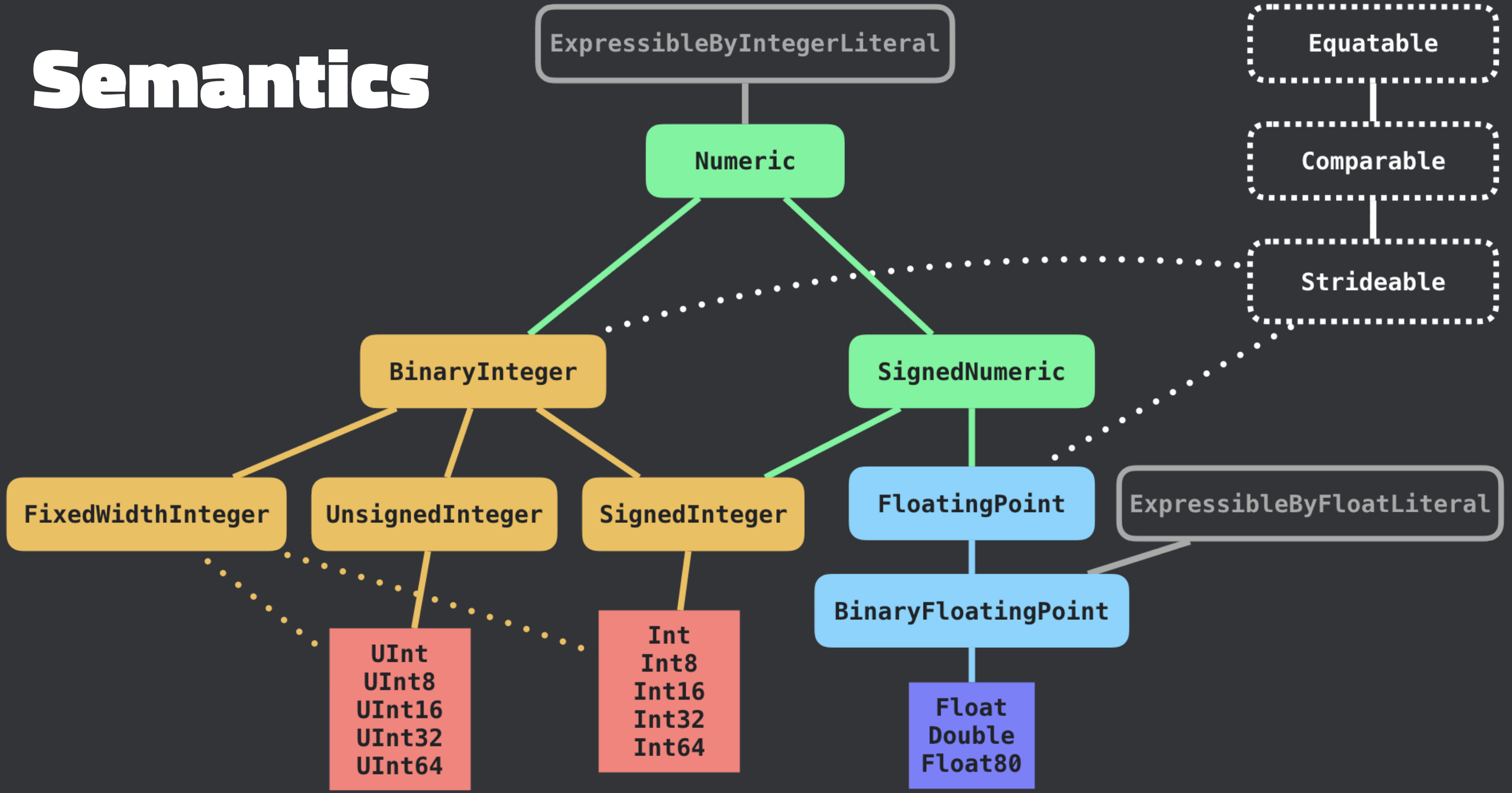
You could create your own!

(radix-8, **`OctalFloatingPoint`** protocol)

Protocols are not just *bags of syntax*[*]
Protocols have *semantics*

[*] *Protocols are more than Bags of Syntax*, Ole Begemann

Semantics


ExpressibleByIntegerLiteral
Numeric
Equatable
Comparable
Strideable
BinaryInteger
SignedNumeric
FixedWidthInteger
UnsignedInteger
SignedInteger
FloatingPoint
ExpressibleByFloatLiteral
BinaryFloatingPoint
UInt
UInt8
UInt16
UInt32
UInt64
Int
Int8
Int16
Int32
Int64
Float
Double
Float80

"Protocol-oriented" numerics

But we still need to work with *concrete types*

`Float`, `Double`, `Float80`

`Int`, `Int8`, `Int16`, `Int32`, `Int64`

`UInt`, `UInt8`, `UInt16`, `UInt32`, `UInt64`

Mixing numeric types: 😭

```
// ⚠️ Binary operator '+' cannot be applied to
// operands of type 'Double' and 'Int'
let x = 42
let y = 3.14 + x

// ⚠️ Binary operator '+' cannot be applied to
// operands of type 'Float' and 'Double'
let f = Float(1.0) + Double(2.0)

// ✅ works
let z = 3.14 + 42
```

Type inference: ☺️

```
// Binary operator '+' cannot be applied to
// operands of type 'Double' and 'Int'
let x = 42
let y = 3.14 + x

// Binary operator '+' cannot be applied to
// operands of type 'Float' and 'Double'
let f = Float(1.0) + Double(2.0)

// 42 inferred as 'Double', ExpressibleByIntegerLiteral
let z = 3.14 + 42
```

Previous example:

```
extension Sequence where Element: SignedNumeric & Comparable {
    func filterNegatives() -> [Element] {
        return filter { $0 > 0 }
    }
}

// mixing types
let allPositive = [UInt(1), 2.5, 3, Int8(-4), -5].filterNegatives()

// ⚠️ error: type of expression is ambiguous without more context
```

Previous example:

```
func hypotenuse<T: FloatingPoint>(_ a: T, _ b: T) -> T {
    return (a * a + b * b).squareRoot()
}

// mixing types
let (dx, dy) = (Double(3.0), Float(4.0))
let dist = hypotenuse(dx, dy)

// ⚠️ error: cannot convert value of type
// 'Float' to expected argument type 'Double'
```

sad.

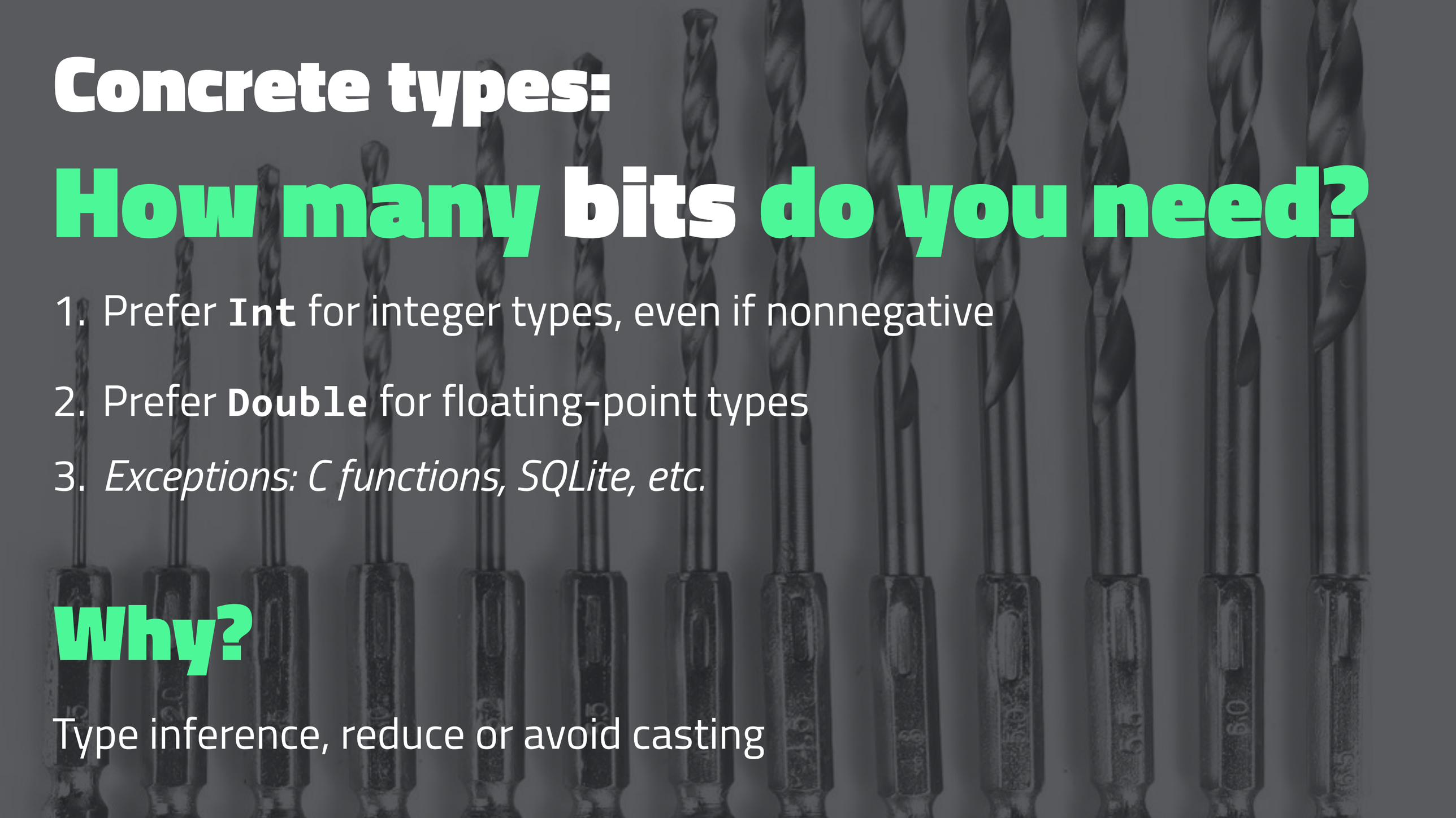
But, this

```
func hypotenuse<T: FloatingPoint>(_ a: T, _ b: T) -> T
```

Is better than this

```
func hypotenuse(_ a: Float, _ b: Float) -> Float
func hypotenuse(_ a: Double, _ b: Double) -> Double
func hypotenuse(_ a: Float80, _ b: Float80) -> Float80
func hypotenuse(_ a: CGFloat, _ b: CGFloat) -> CGFloat
```

Less sad. 



Concrete types:

How many bits do you need?

1. Prefer **`Int`** for integer types, even if nonnegative
2. Prefer **`Double`** for floating-point types
3. *Exceptions: C functions, SQLite, etc.*

Why?

Type inference, reduce or avoid casting


Making our raw
calculation
code more
expressive

Example: drawing line graphs 

```
let p1 = Point(x1, y1)
let p2 = Point(x2, y2)
let slope = p1.slopeTo(p2)
```

Need to check if the slope is:

- undefined (vertical line)
- zero (horizontal line)
- positive
- negative

Extensions for our specific domain

```
extension FloatingPoint {
    var isUndefined: Bool { return isNaN }
}

extension SignedNumeric where Self: Comparable {
    var isPositive: Bool { return self > 0 }

    var isNegative: Bool { return self < 0 }
}
```

Example: drawing line graphs 

```
if slope.isZero {

} else if slope.isUndefined {

} else if slope.isPositive {

} else if slope.isNegative {

}
```

This code reads like a sentence.

small tweaks make a

BIG

difference in readability

Like most types in the Standard Library, the numeric types are `structs`

Primitive values with *value semantics*, but also "object-oriented"

Let's go one more level down

— Greg Heo

How are they implemented?

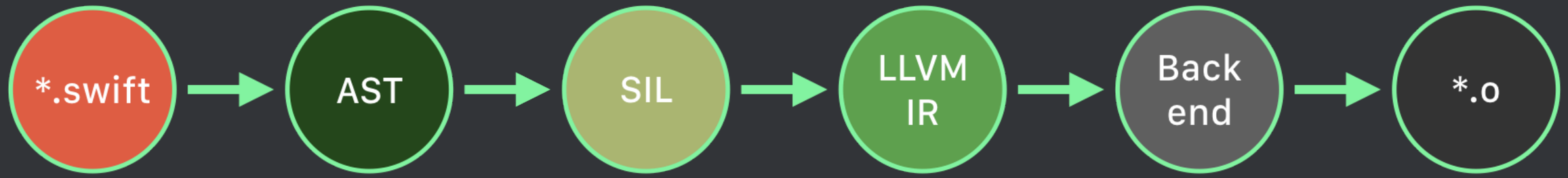
github.com/apple/swift/stdlib/public/core/

- Structs with private **`_value`** property (**`Builtin`** type)
- Conform to **`ExpressibleBy*Literal`**

Swift compiler

LLVM architecture

A brief overview



```
struct Int64 {
    var _value: Builtin.Int64

    init(_builtinIntegerLiteral x: _MaxBuiltinIntegerType) {
        _value = Builtin.s_to_s_checked_trunc_Int2048_Int64(x).0
    }
}

struct UInt8 {
    var _value: Builtin.Int8

    init(_builtinIntegerLiteral x: _MaxBuiltinIntegerType) {
        _value = Builtin.s_to_u_checked_trunc_Int2048_Int8(x).0
    }
}
```

```
struct Float {
    var _value: Builtin.FPIEEE32

    init(_bits v: Builtin.FPIEEE32) {
        self._value = v
    }

    init(_builtinIntegerLiteral value: Builtin.Int2048){
        self = Float(_bits: Builtin.itofp_with_overflow_Int2048_FPIEEE32(value))
    }

    init(_builtinFloatLiteral value: Builtin.FPIEEE64) {
        self = Float(_bits: Builtin.fptrunc_FPIEEE64_FPIEEE32(value))
    }
}
```

Constructing **`Int64`** from a **`Double`**

```
struct Int64 {

    init(_ source: Double) {
        _precondition(source.isFinite,
        "Double value cannot be converted to Int64 because it is either infinite or NaN")

        _precondition(source > -9223372036854777856.0,
        "Double value cannot be converted to Int64 because the result would be less than Int64.min")

        _precondition(source < 9223372036854775808.0,
        "Double value cannot be converted to Int64 because the result would be greater than Int64.max")

        self._value = Builtin.fptosi_FPIEEE64_Int64(source._value)
    }
}
```

Preventing underflow / overflow!

Swift is a memory-safe language

Swift guarantees 💪

- Type safety
- Boundaries of numeric types
- Traps overflow / underflow behavior and reports an error

❌ fatal errors

```
// ⚠️ fatal error: Not enough bits to represent a signed value
let i = Int8(128)

// ⚠️ fatal error: Negative value is not representable
let i = UInt(-1)

// ⚠️ fatal error: Double value cannot be converted
// to Int because the result would be greater than Int.max
let i = Int(Double.greatestFiniteMagnitude)
```

🤔 not fatal error

```
// ⚠️ inf
let f = Float32(Float80.greatestFiniteMagnitude)

// f == Float32.infinity
```

Quiz 

What is the value of sum?

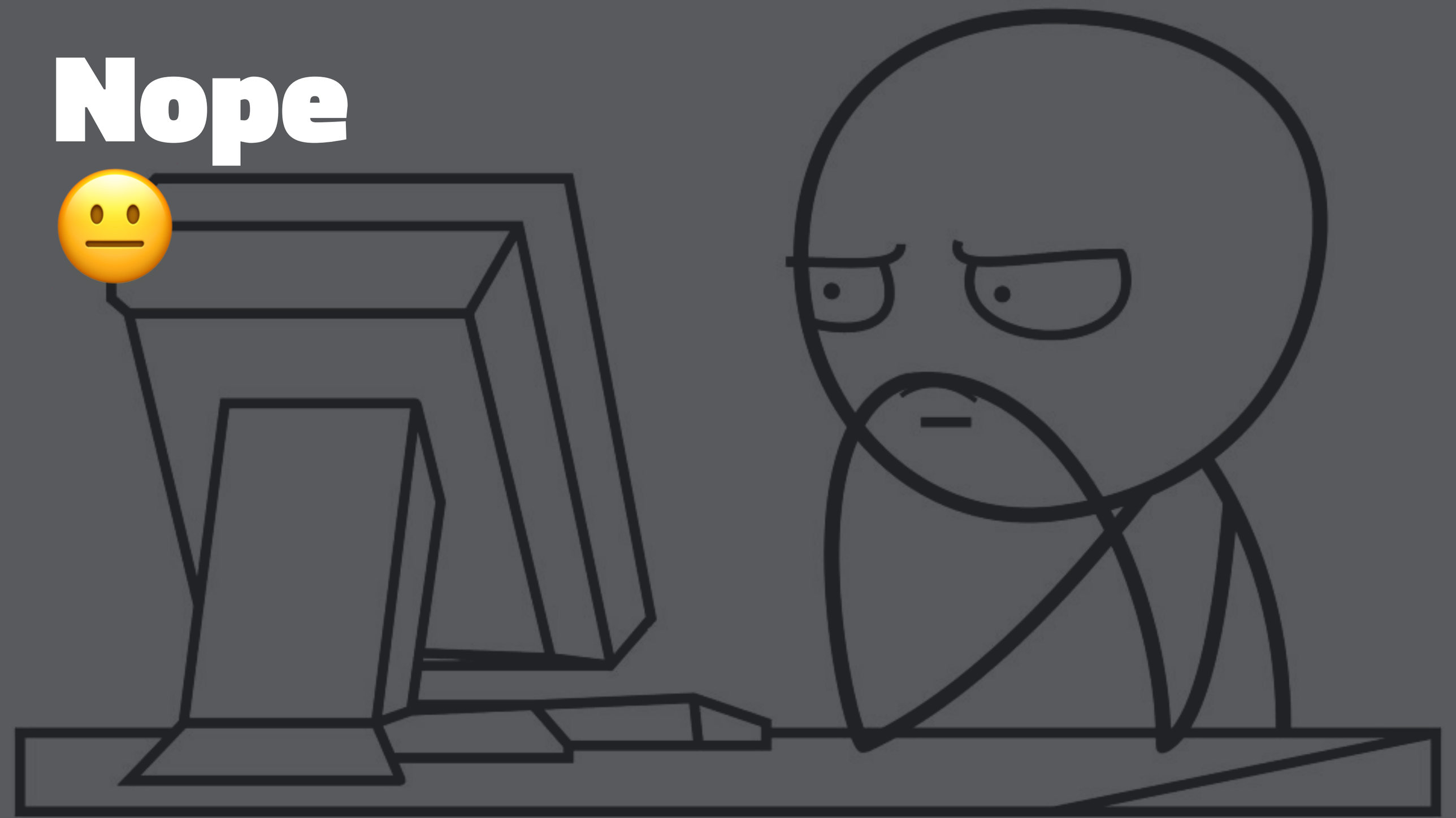
```
// Add 0.1 ten times
let f = Float(0.1)
var sum = Float(0.0)
for _ in 0..<10 {
    sum += f
}
```

Quiz 

What is the value of sum?

1.0?

Nope



Quiz 

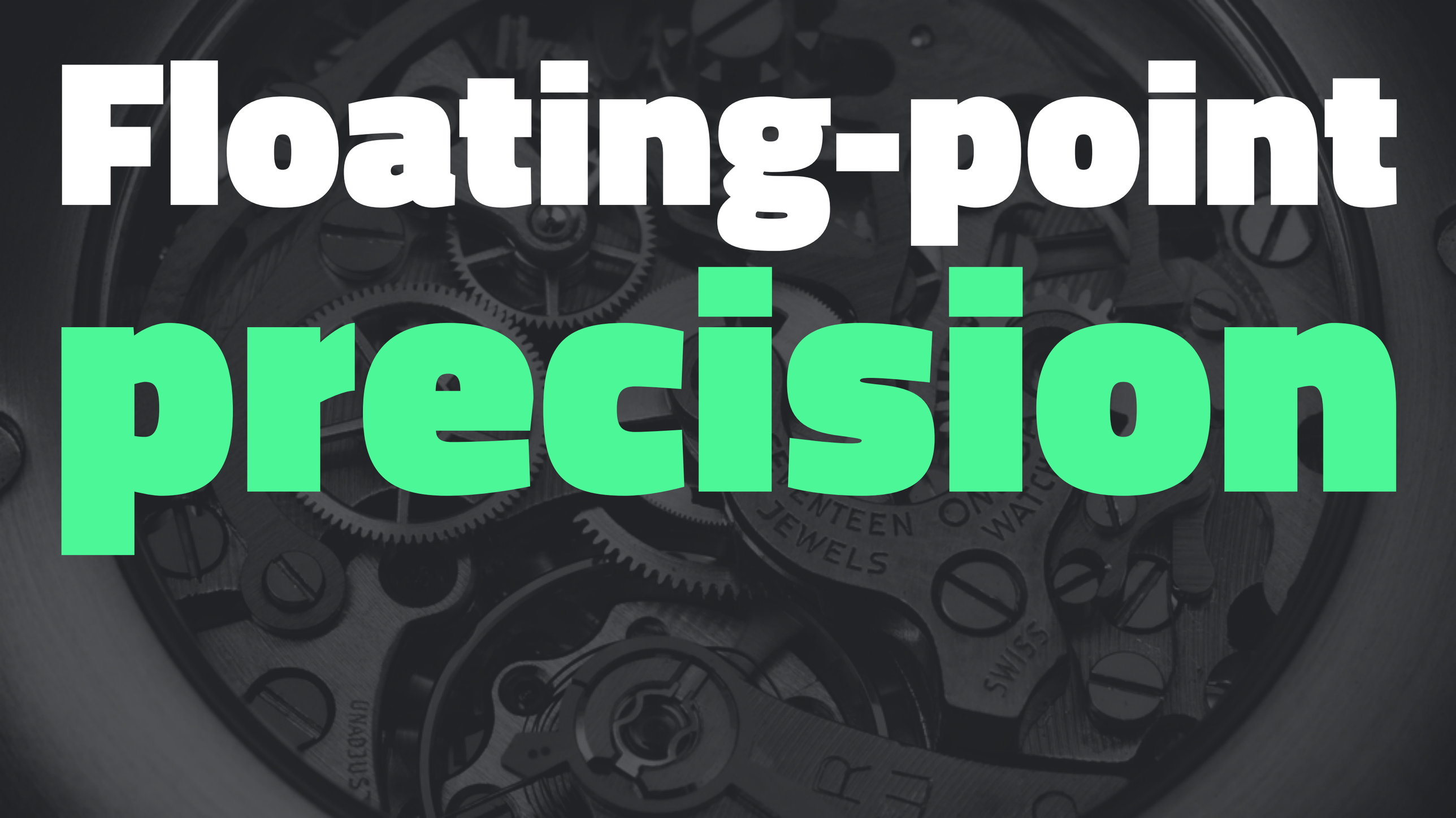
What is the value of sum?

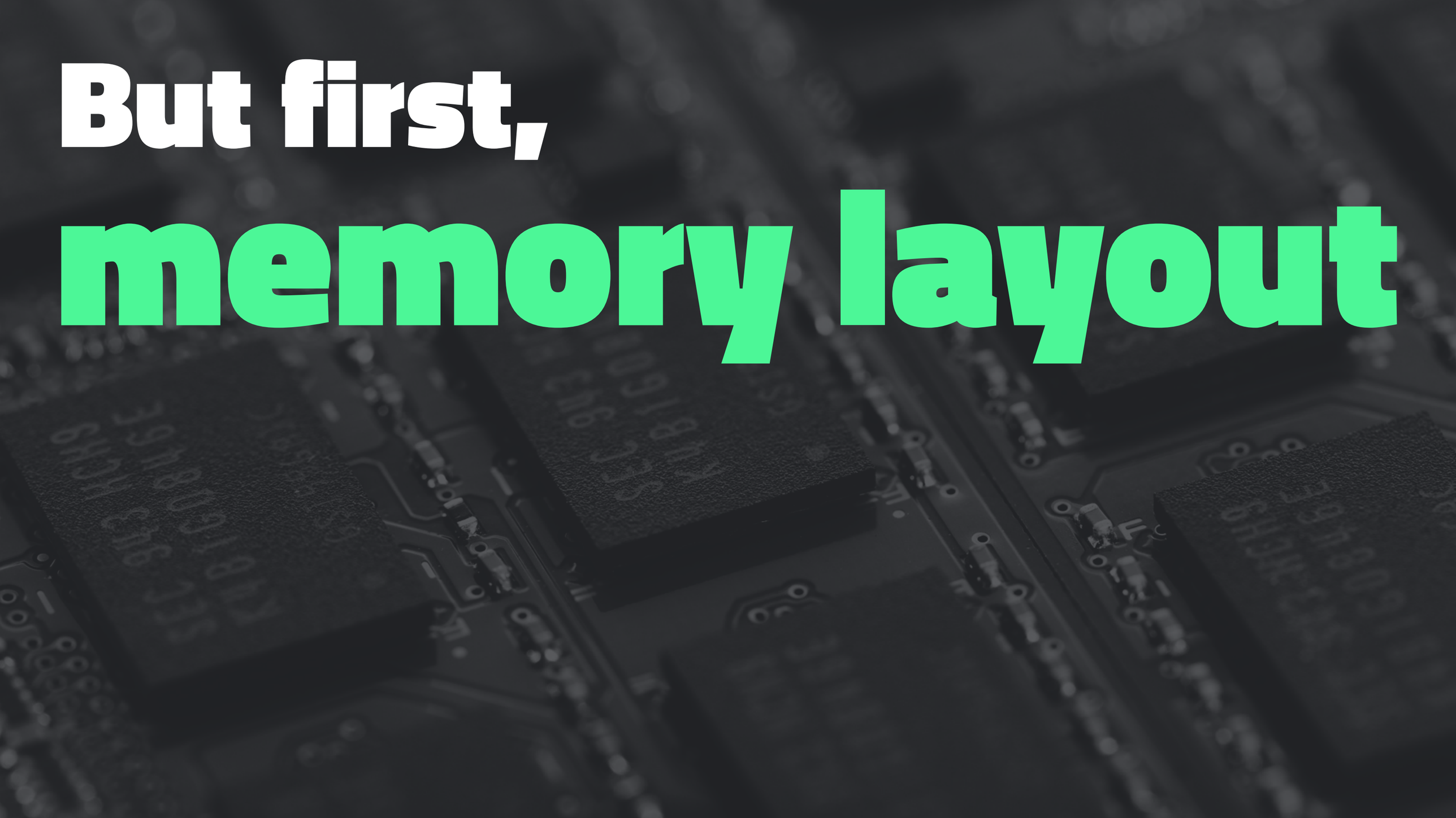
1.000000119209289550781250

Floating-point math is not exact!

Let's go one more level down *again*

— Greg Heo


Floating-point
precision


But first,
memory layout

Integer representation

`Int32`

32 bits

Integers: just bits**

Int32

0b0001_0100_0101_0110_1001_0101_0101_1111

341_218_655

** Signed integers are typically represented in two's complement, but that's an implementation detail.

Floating-point representation

4 elements:

- **Sign:** negative or positive
- **Radix (or Base):** 2 for binary, 10 for decimal, ...
- **Significand:** series of digits of the base
  The number of digits == precision
- **Exponent:** represents the offset of the significand (biased)

```
value = significand * radix ^ exponent
```

```
protocol FloatingPoint {
    var sign: FloatingPointSign { get }

    static var radix: Int { get }

    var significand: Self { get }

    var exponent: Self.Exponent { get }
}

// Float, Double, Float80
```

Floating-point: not "just bits"

IEEE 754 single-precision binary floating-point format
`Float32`

sign bit

8-bit exponent (biased by 127)

23-bit significand

Floating-point representation

`Float.pi`

🍰

significand * radix^{exponent}

Float32

sign	exponent	significand
(+)	2^1	1.57075
	128	4787798
	0b1000_0000	0b100_1001_0000_1110_0101_0110

$1.57075 * 2^1 = 3.1415$

```
let pi = 3.1415
pi.sign           // plus
pi.exponent       // 1
pi.significand  // 1.57075

// 1.57075 * 2.0^1 = 3.1415
Float(pi.significand) * powf(Float(Float.radix), Float(pi.exponent))
```

sign	exponent	significand
(+)	2^1	1.57075
	128	4787798
	0b1000_0000	0b100_1001_0000_1110_0101_0110

```
protocol BinaryFloatingPoint {
    var exponentBitPattern: Self.RawExponent { get }

    var significandBitPattern: Self.RawSignificand { get }

    static var exponentBitCount: Int { get }

    static var significandBitCount: Int { get }
}

// Float, Double, Float80
```

```
pi.sign                          // plus
pi.exponentBitPattern            // 128
pi.significandBitPattern         // 4787798

Float.exponentBitCount           // 8 bits
Float.significandBitCount        // 23 bits
```

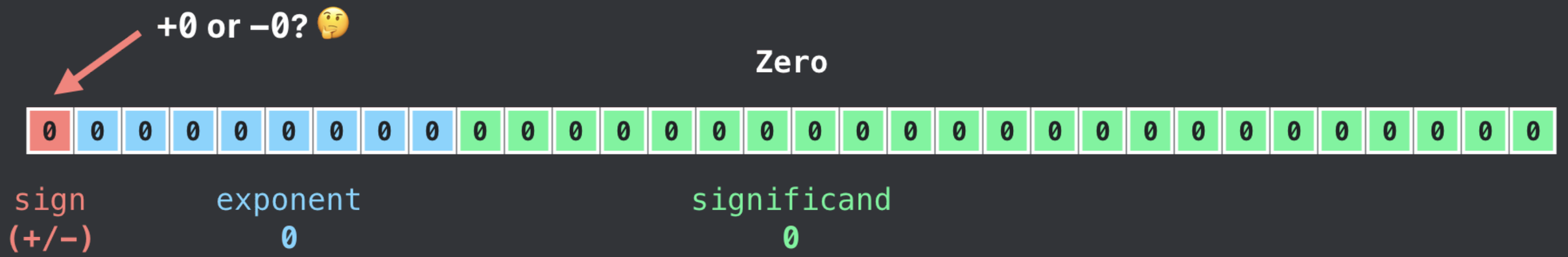
sign	exponent	significand
(+)	2^1	1.57075
	128	4787798
	0b1000_0000	0b100_1001_0000_1110_0101_0110

Interesting properties of floating-point layout

Positive and negative zero

Implementation

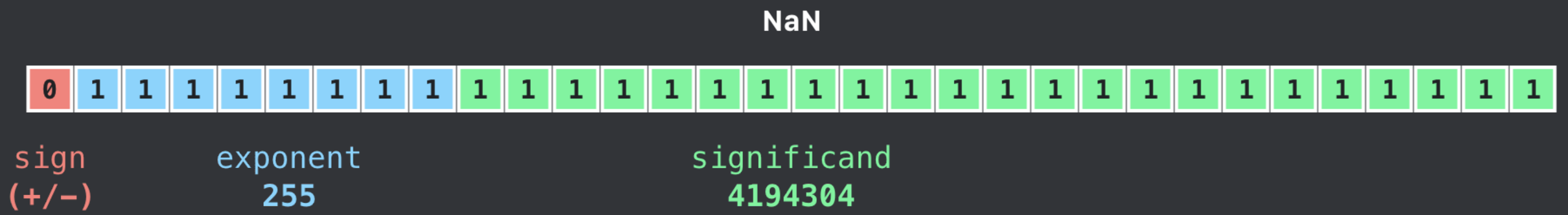
```
// FloatingPointTypes.swift
public var isZero: Bool {
    return exponentBitPattern == 0 && significandBitPattern == 0
}
```



NaN != NaN

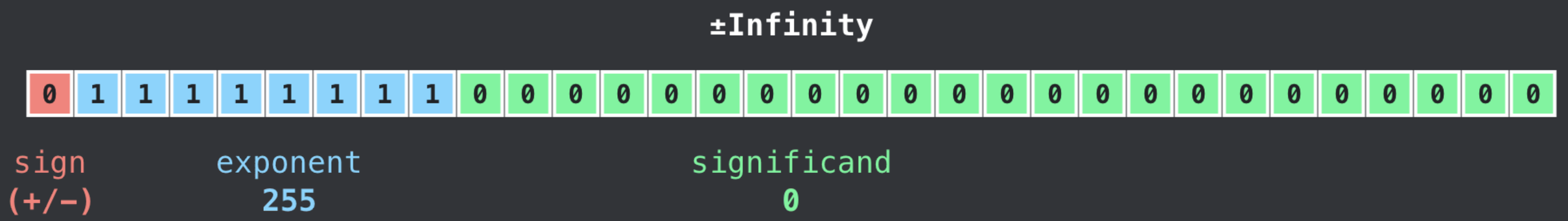
Implementation

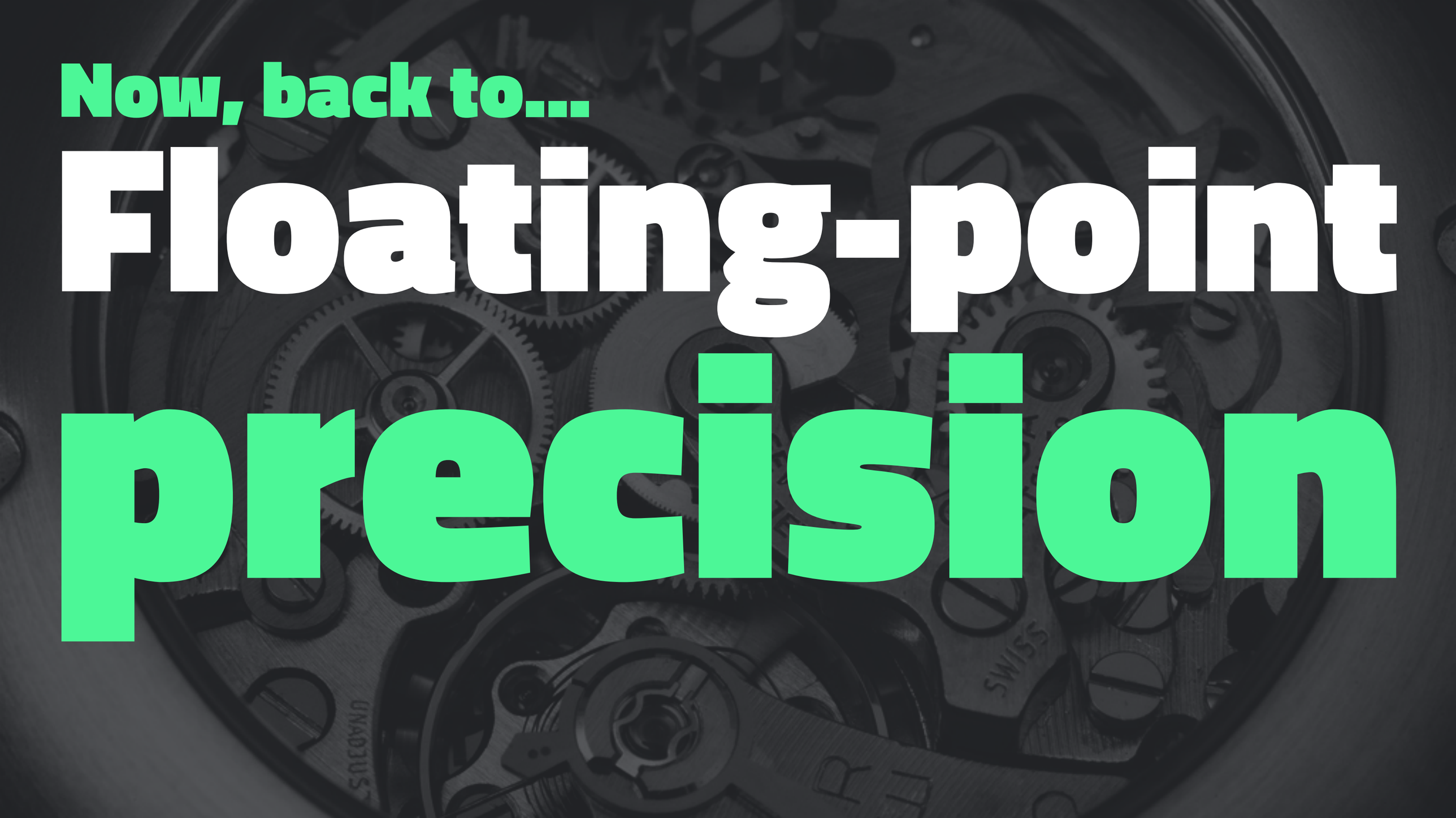
```
// FloatingPointTypes.swift
public var isNaN:  Bool {
    // isFinite == exponentBitPattern < Float._infinityExponent
    return !isFinite && significandBitPattern != 0
}
```

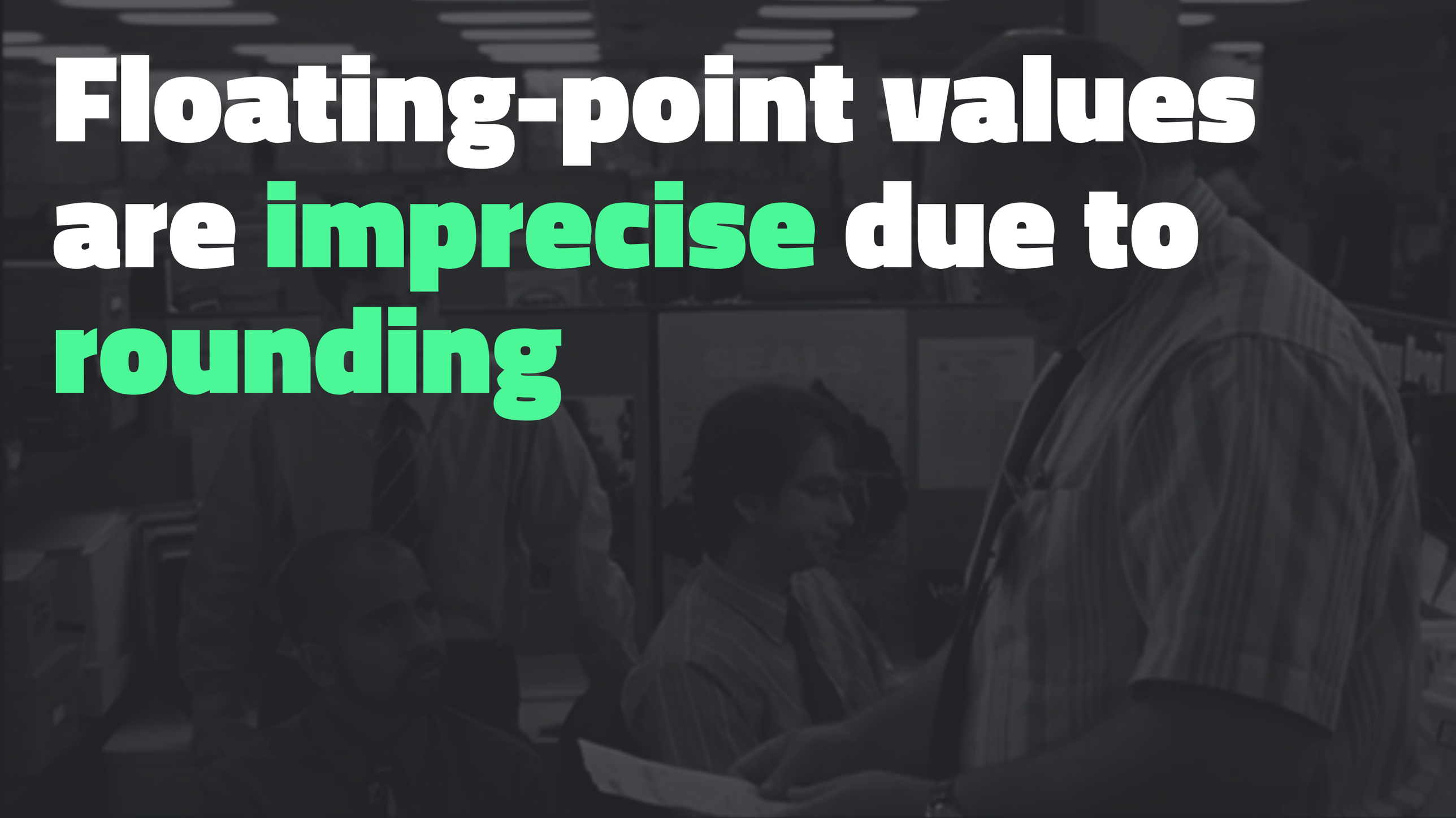


± Infinity

```
static var infinity: Float {
    // FloatingPointTypes.swift
    return Float(sign: .plus,
                 exponentBitPattern: _infinityExponent,
                 significandBitPattern: 0)
}
```




Now, back to...
Floating-point
precision



Floating-point values are imprecise due to rounding

How do we measure rounding error?

```
// Swift 4
// ⚠️ 'FLT_EPSILON' is deprecated:
// Please use 'Float.ulpOfOne' or '.ulpOfOne'.
FLT_EPSILON

protocol FloatingPoint {
    static var ulpOfOne: Self { get }
}
```

.ulpOfOne? 

Documentation

`ulpOfOne`

The unit in the last place of 1.0.

Wat

Documentation

Discussion

The positive difference between `1.0` and the next greater representable number. The `ulpOfOne` constant corresponds to the C macros `FLT_EPSILON`, `DBL_EPSILON`, and others with a similar purpose.

Machine epsilson: ISO C Standard

```
protocol FloatingPoint
```

```
Float.ulpOfOne
// FLT_EPSILON
// 1.192093e-07, or
// 0.00000011920928955078125
```

```
Double.ulpOfOne
// DBL_EPSILON
// 2.220446049250313e-16, or
// 0.0000000000000002220446049250313080847263336181640625
```

But there's also `.ulp` 🤔

Not static like **`.ulpOfOne`**!

```
protocol FloatingPoint {
    var ulp: Self { get }
}


1.0.ulp  // 🤔
3.456.ulp  // 🤔
```

ulp

Unit in the Last Place

Unit of Least Precision

It measures the distance from a value to the next representable value.

For most numbers x, this is the difference between x and the next greater (in magnitude) representable number.

Next representable `Int`

First, let's consider integers

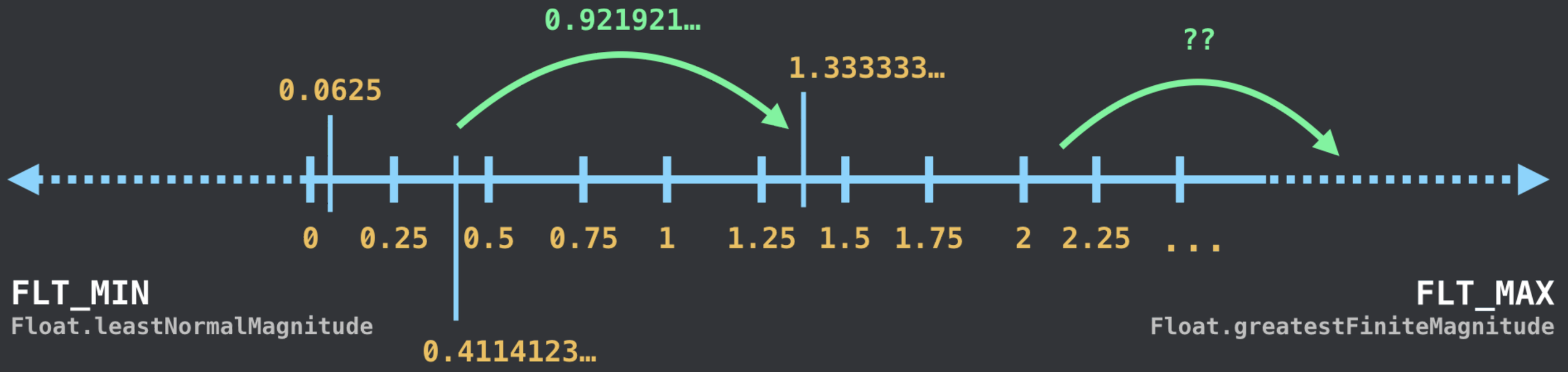
Integers are exact

We don't need any notion of "ulp"

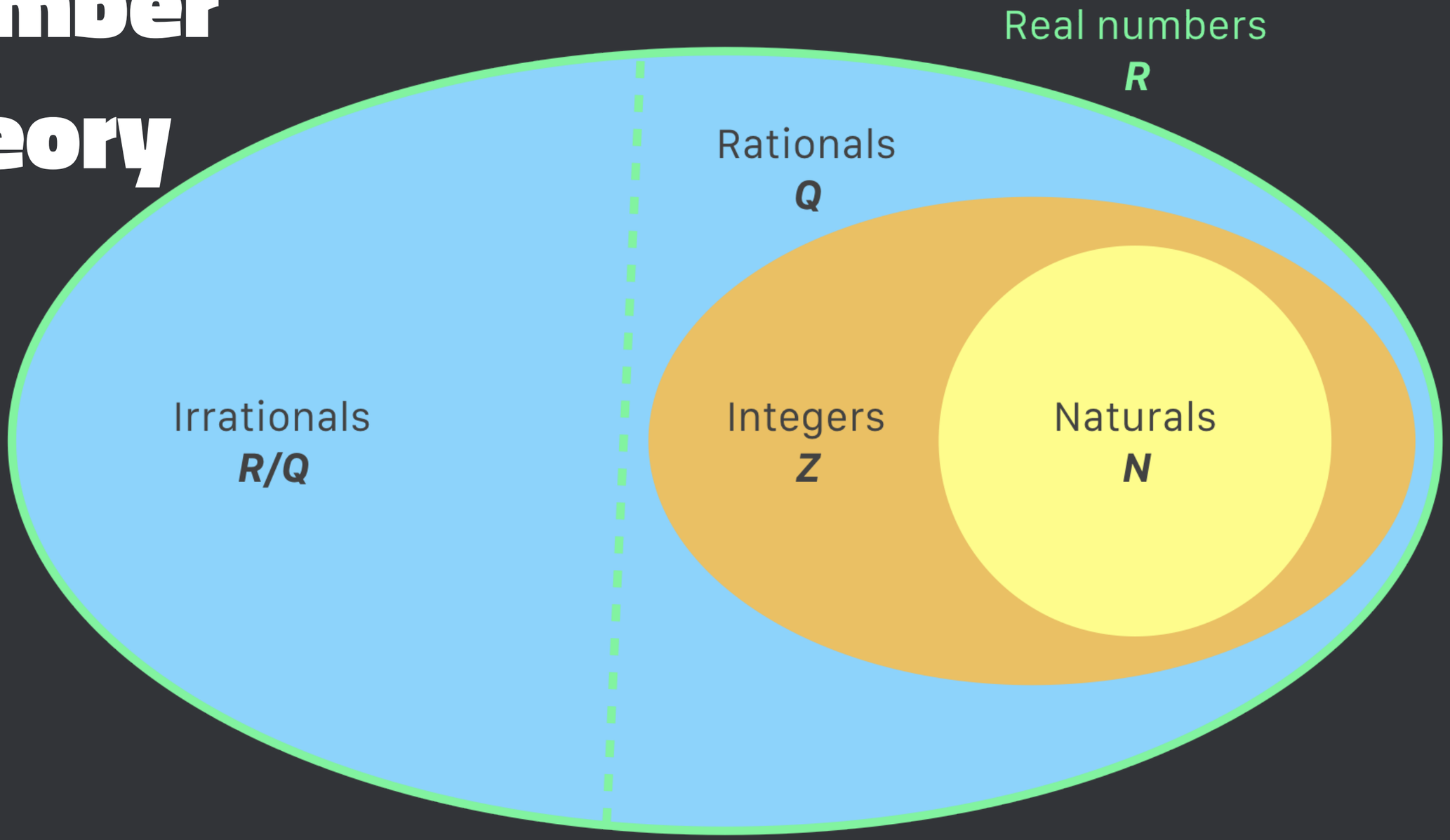
Floats, not so much

Difficult to represent in bits! *Not exact!*

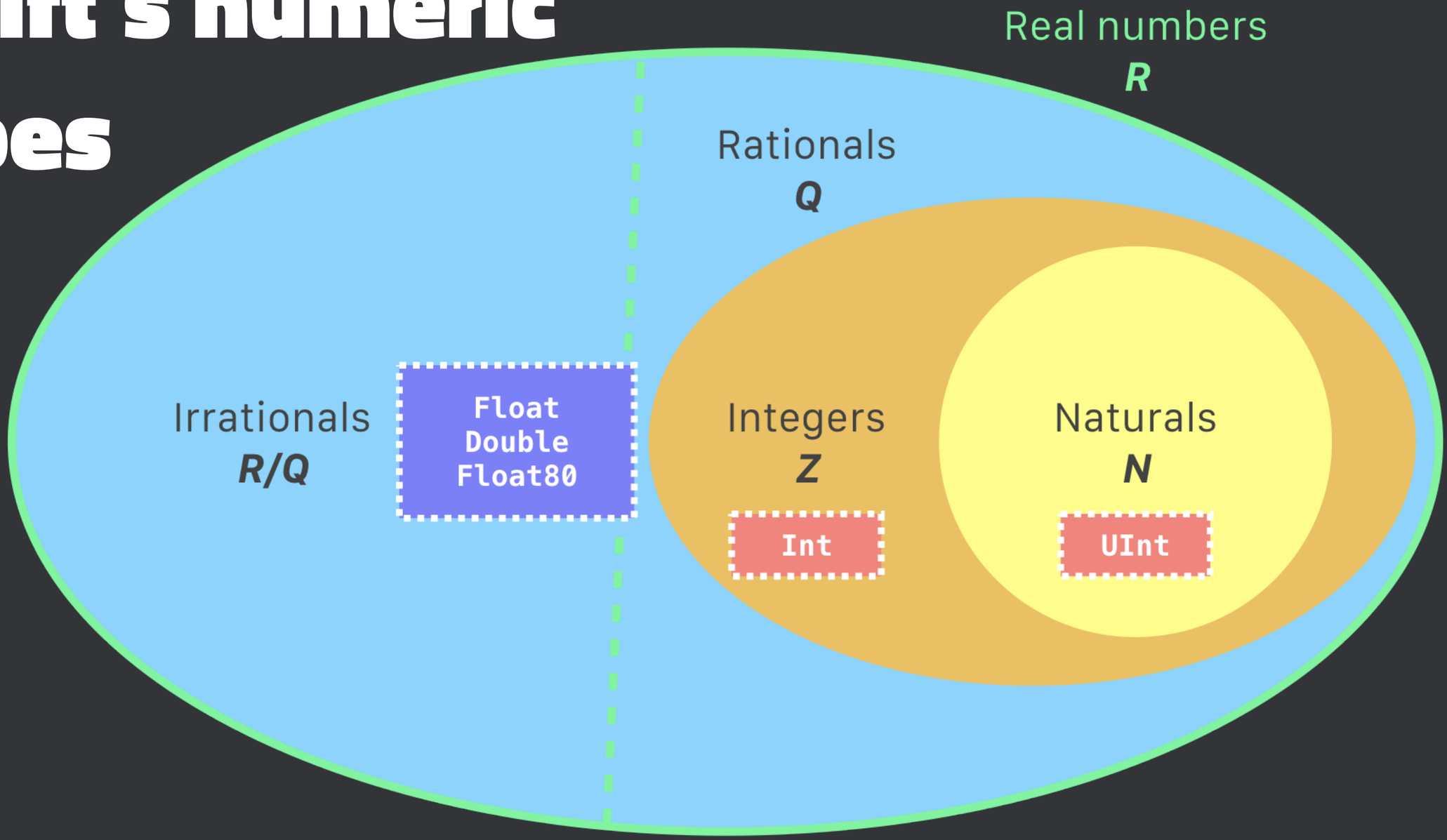
Next representable `Float`



Number Theory

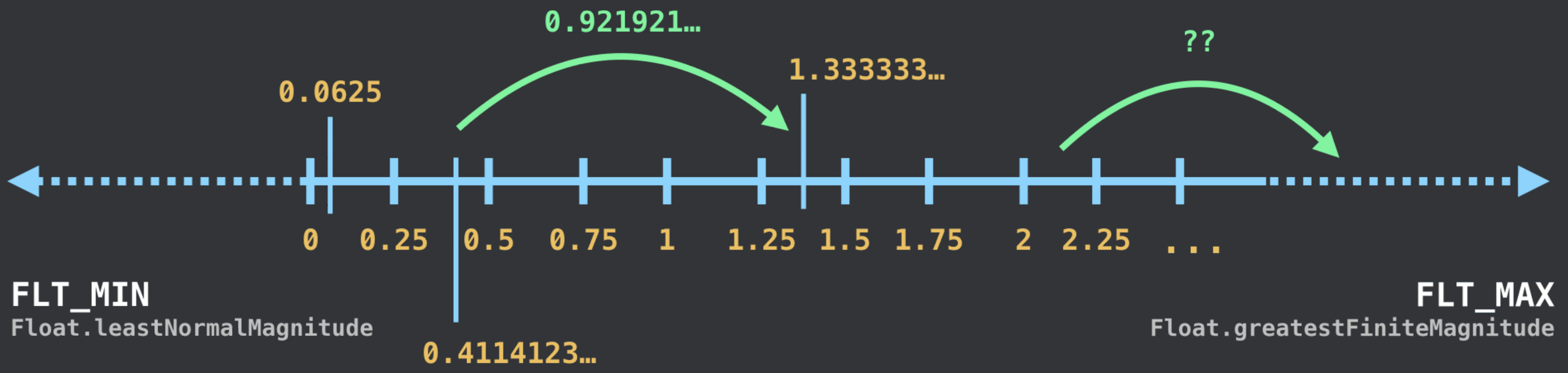


Swift's numeric types



Infinite number of values between any two floating-point values

In mathematics, but not in computing



More numbers between 0 and 1 than the entire set of integers 😱

$R/Q + Q > Z$

But we only have 32 bits! 😱

(or 64 bits)

We have to *round* because

not all values can be represented.

Thus, we need `ulp`.

(also, silicon chips are obviously finite)

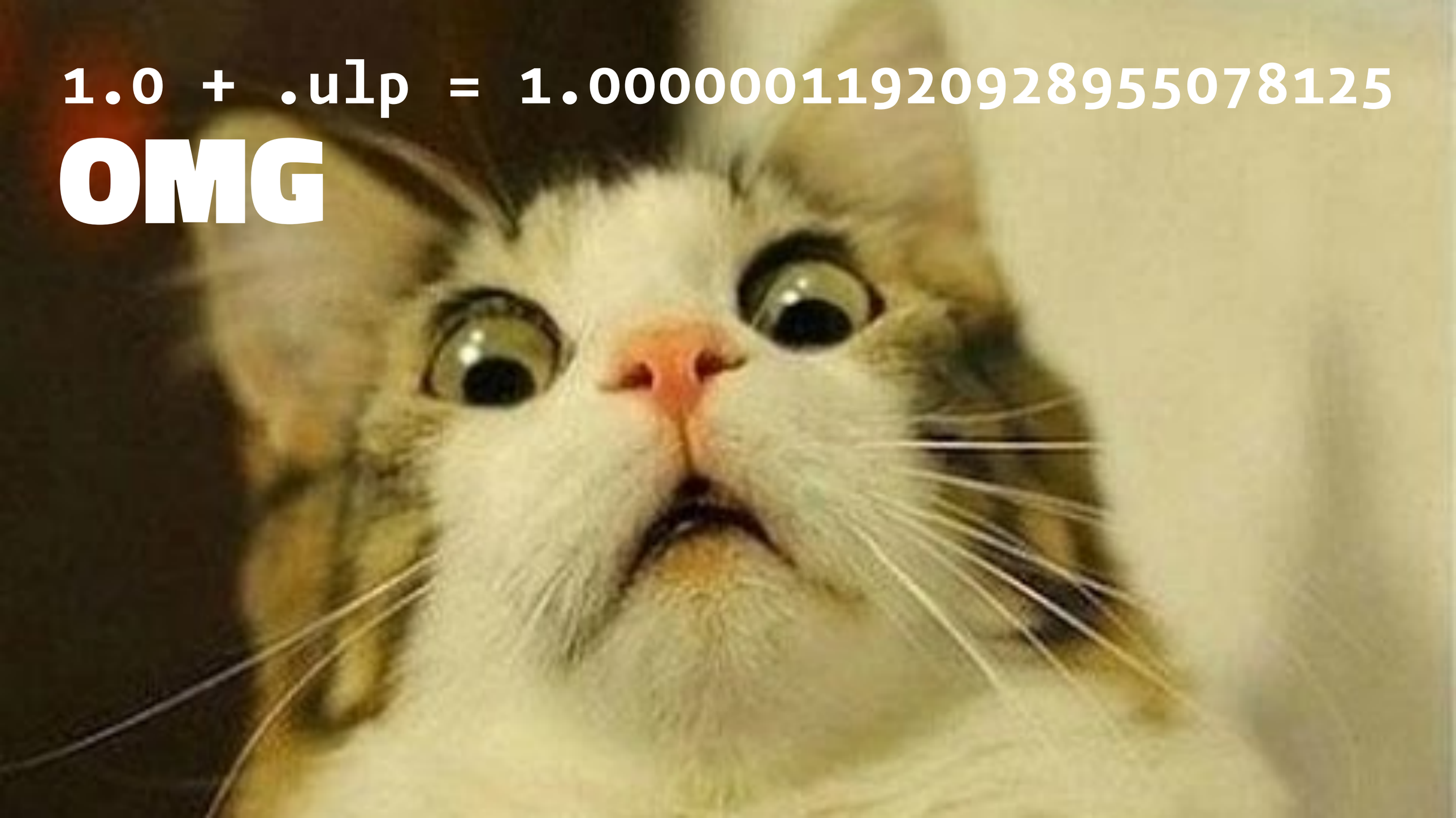
Back to that Quiz 🤓

```
let f = Float(0.1)
var sum = Float(0.0)
for _ in 0..<10 {
    sum += f
}
// sum == ?
```

1.000000119209289550781250

```
Float(1.0).ulp
0.00000011920928955078125
sum
1.00000011920928955078125
```

the ulp of one


1.0 + .ulp = 1.00000011920928955078125
OMG

Defining `ulp`

`epsilon * radix^exp`

The distance from a value to the next representable value.

```
let value = Float(3.1415)
let computedUlp = Float.ulpOfOne * powf(Float(Float.radix), Float(value.exponent))

value       // 3.1414999961853027343750
computedUlp // 0.00000023841857910156250
value.ulp   // 0.00000023841857910156250
```

Next representable value: .nextUp

```
protocol FloatingPoint {
    var nextUp: Self { get }
}

let value = Float(1.0)
value.ulp                  // 0.00000011920928955078125
value + value.ulp          // 1.00000011920928955078125
value.nextUp               // 1.00000011920928955078125
```

Precision varies

The precision of a floating-point value is proportional to its magnitude. The larger a value, the less precise.

```
let f1 = Float(1.0)
f1.ulp   // 0.00000011920928955078125

let f2 = Float(1_000_000_000.0)
f2.ulp   // 64.0
```

Comparing for equality: the big problem

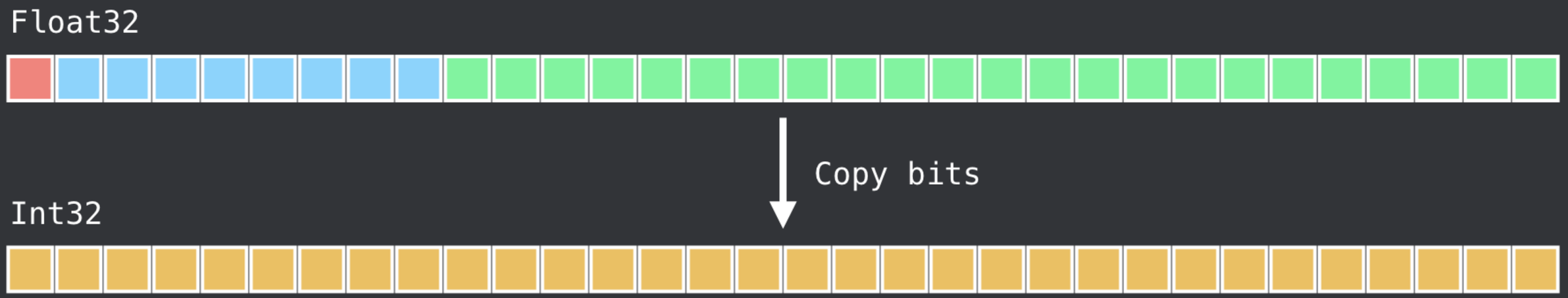
No silver bullet! 😭

- Comparing against zero, use absolute epsilon, like **`0.001`**
- ‼️ Never use **`.ulpOfOne`** (**`FLT_EPSILON`**) as tolerance
- Comparing against non-zero, use relative ULPs

Computing relative ULPs

Adjacent floats have integer representations that are adjacent.

Subtracting the integer representations gives us the number of ULPs between floats.



Computing relative ULPs

```
extension Float {
    var asInt32: Int32 {
        return Int32(bitPattern: self.bitPattern)
    }
}
```

NOTE: *This is not perfect.*
Some edge cases to handle (e.g., negatives, which are two's complement)

Comparing relative ULPs

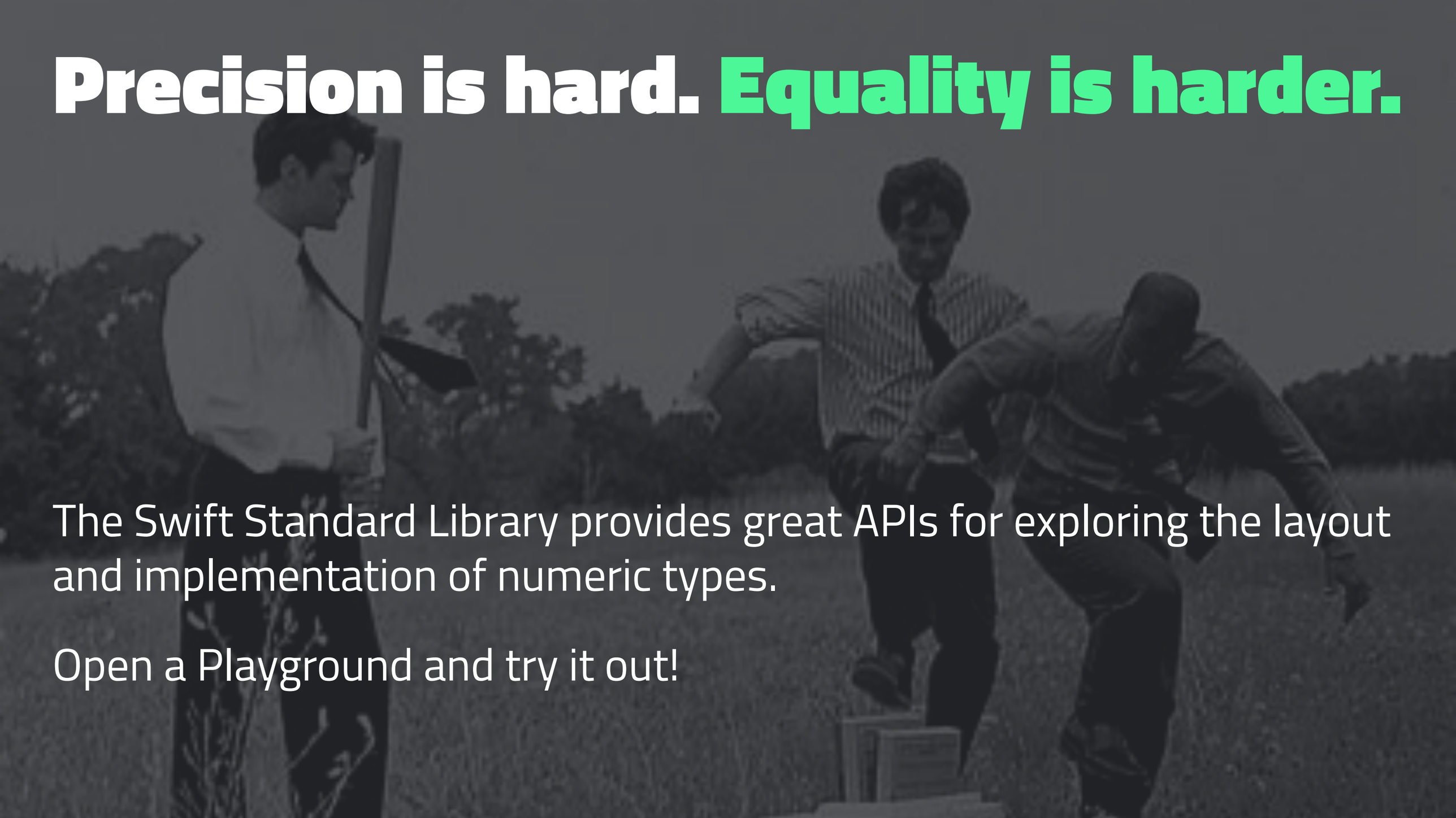
```
let f1 = Float(1_000_000.0)
let f2 = f1 + (f1.ulp * 5) // 1_000_000.31250

//  1232348160 - 1232348165
abs(f1.asInt32 - f2.asInt32) // 5 ULPs away
```

Comparing relative ULPs

- If **zero**, floats are exact same binary representation
- If **one**, floats are as close as possible without being equal
- If **more than one**, floats (potentially) differ by orders of magnitude

```
If <= 1 ulp, consider them equal
```

Precision is hard. Equality is harder.



The Swift Standard Library provides great APIs for exploring the layout and implementation of numeric types.

Open a Playground and try it out!

References & Further reading

The rabbit hole goes much deeper!

- *Comparing Floating Point Numbers, 2012 Edition*, Bruce Dawson
- *Floating Point Demystified, Part 1*, Josh Haberman
- *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, David Goldberg
- *Floating Point Visually Explained*, Fabien Sanglard
- *Lecture Notes on the Status of IEEE 754*, Prof. W. Kahan, UC Berkeley
- IEEE-754 Floating Point Converter

Thanks!

Jesse Squires

jessesquires.com ▪ @jesse_squires

Swift Weekly Brief:

swiftweekly.github.io ▪ @swiftlybrief