Swift & it's Type System

Seminar on Type Theory
University of Helsinki

Richard Topchii, 7th of May 2020

Richard Topchii

- Apple apps developer
- Learned Objective-C in 2013
- Using Swift since 2014
- Author of multiple libraries in Swift

Swift Programming language

- General-purpose, multi-paradigm

Swift Programming language

- General-purpose, multi-paradigm
- “Objective-C without C”

Swift Programming language

- General-purpose, multi-paradigm
- “Objective-C without C”
- Released in 2014
- Open-sourced in 2015

Swift Programming language

- General-purpose, multi-paradigm
- “Objective-C without C”
- Released in 2014
- Open-sourced in 2015
- 11th place in TIOBE programming languages index
- Surpassed Objective-C in popularity in 2018~2020

WWDC 2014

Objective-C
without the C

Swift Adoption

Officially Supported

iOS, iPadOS, watchOS, tvOS, macOS

1,4b active devices

900m iPhones

Ubuntu

Swift Adoption

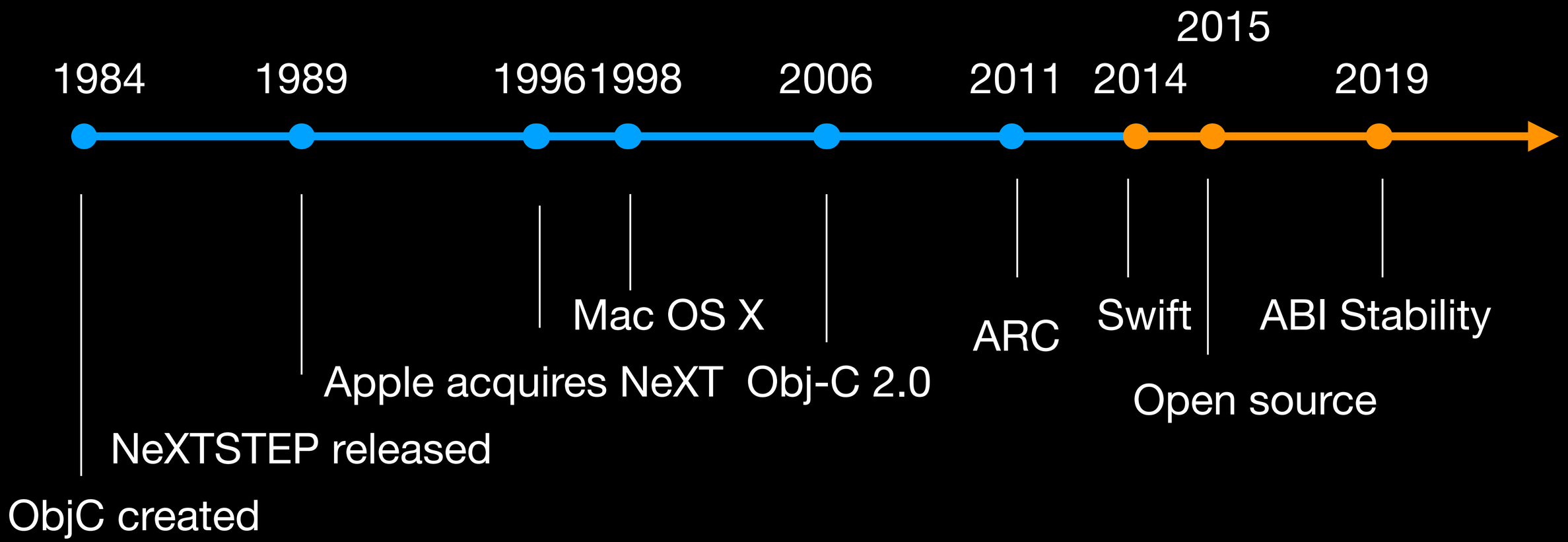
Unofficially supported

Supported by TensorFlow,
not Swift community

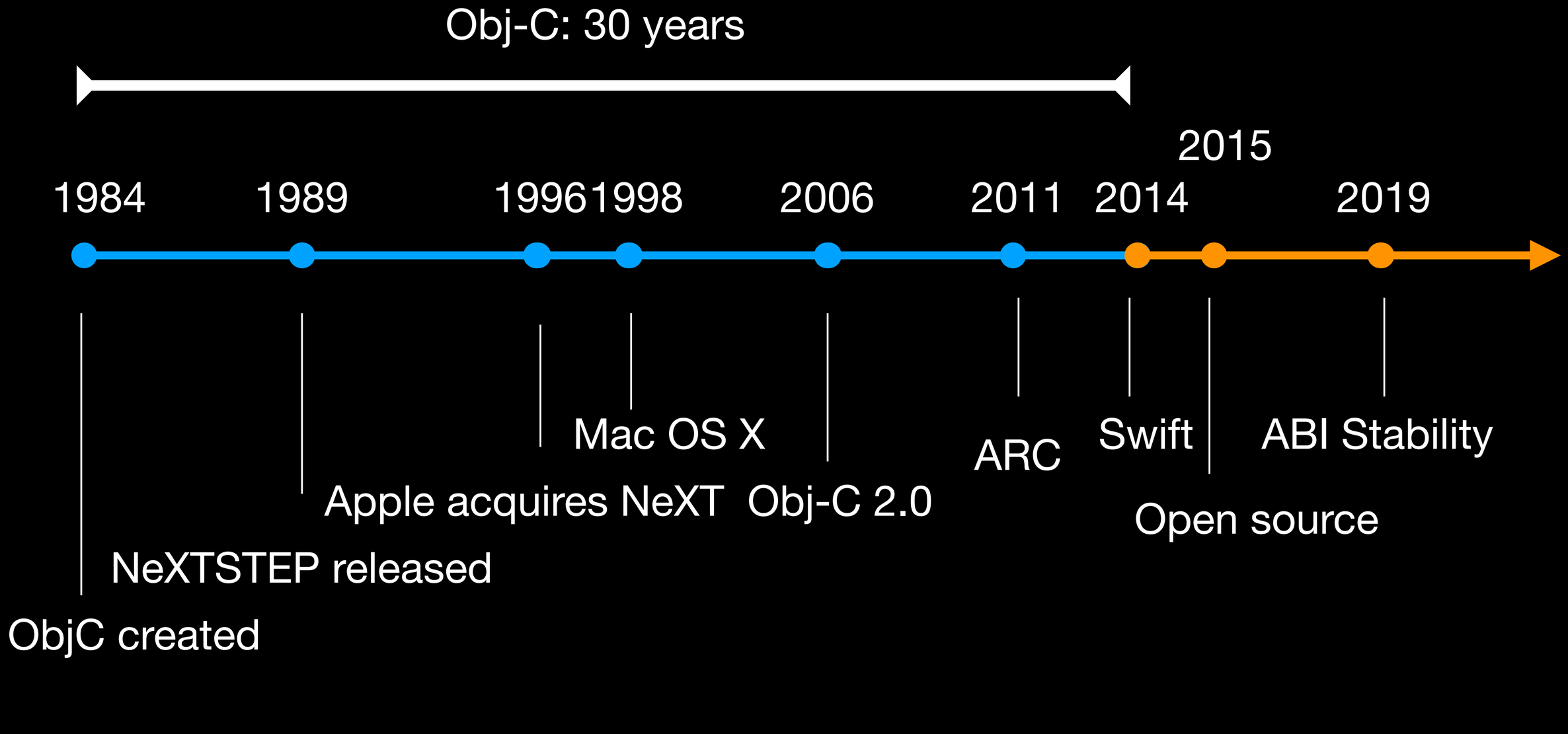
Swift Programming language

- General-purpose, multi-paradigm
- “Objective-C without C”
- Released in 2014
- Open-sourced in 2015
- 11th place in TIOBE programming languages index
- Surpassed Objective-C in popularity in 2018~2020

The need for Swift



The need for Swift



Objective-C at a glance

- Strict superset of C
- Object-Oriented, uses messages
- Weak typing (duck typing)
- Compiler checks that the method is defined for a type
- Doesn’t check that the type annotated is a correct one

Objective-C at a glance

Problems

- Global namespace for all classes
- Sending message to **nil** doesn’t cause a crash
- “Optional” interface: object might not handle a message -> crash
- Syntax differs from most of the mainstream languages
- Functions are not first-class citizens

Objective-C

```
if (self.delegate != nil) {
  if ([self.delegate respondsToSelector:
       @selector(tableView:didSelectRowAtIndexPath:)]) {
    [self.delegate tableView:tableView
     didSelectRowAtIndexPath:indexPath];
  }
}
```

Objective-C

```
if (self.delegate != nil) {
  if ([self.delegate respondsToSelector:
       @selector(tableView:didSelectRowAtIndexPath:)]) {
    [self.delegate tableView:tableView
     didSelectRowAtIndexPath:indexPath];
  }
}
```

Swift

```
delegate?.tableView?(tableView, didSelectRowAt: indexPath)
```

Example

```
NSMutableArray *mutableArray = [NSMutableArray new];
NSString *element1 = @"String";
NSString *element2 = @"String2";

[mutableArray addObject:element1];
[mutableArray addObject:element2];

NSString *string1 = [mutableArray objectAtIndex:0];
NSNumber *NoError = [mutableArray objectAtIndex:1];

[mutableArray enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger idx, BOOL * _Nonnull stop) {
  NSLog(@"%@", obj);
}];

NSLog(@"%@", string1);
NSLog(@"%@", NoError);
NSLog(@"%@", [NoError stringValue]);
```

Example

```
NSMutableArray *mutableArray = [NSMutableArray new];
NSString *element1 = @"String";
NSString *element2 = @"String2";

[mutableArray addObject:element1];
[mutableArray addObject:element2];

NSString *string1 = [mutableArray objectAtIndex:0];
NSNumber *NoError = [mutableArray objectAtIndex:1];

[mutableArray enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger idx, BOOL * _Nonnull stop) {
  NSLog(@"%@", obj);
}];

NSLog(@"%@", string1);
NSLog(@"%@", NoError);
NSLog(@"%@", [NoError stringValue]);
```

Example

```
NSMutableArray *mutableArray = [NSMutableArray new];
NSString *element1 = @"String";
NSString *element2 = @"String2";

[mutableArray addObject:element1];
[mutableArray addObject:element2];

NSString *string1 = [mutableArray objectAtIndex:0];
NSNumber *NoError = [mutableArray objectAtIndex:1];

[mutableArray enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger idx, BOOL * _Nonnull stop) {
  NSLog(@"%@", obj);
}];

NSLog(@"%@", string1);
NSLog(@"%@", NoError);
NSLog(@"%@", [NoError stringValue]);
```

✅

Crash

```
Thread 1: Exception: "-[__NSCFConstantString stringValue]: unrecognized selector sent to instance 0x101c6e080"
```

Swift Type System

Swift Type System

Named types

Compound types

Swift Type System

Named types

- Have a name
- Defined in the Standard Library
- Classes, structs, protocols…

Compound types

- Don’t have a name
- Part of the Swift language
- Only Functions and Tuples

Compound Types

Tuple

```
let user = ("Richard", 32)
var user2 = (name: "Richard", age: 22)
user2 = user
```

```
func user() -> (name: String, age: Int) {
  return (name: "Richard", age: 22)
}
```

Compound Types

Functions

```
// Simplest function
func noArgs() -> Void {}
// No need to type "->" if it doesn't return anything
func noArgs2() {}

// Multiple arguments, single return type
func createPoint(x: Int, y: Int) -> CGPoint {
  return CGPoint(x: x, y: y)
}

// Single argument, returns a tuple
func valuesFromPoint(p: CGPoint) -> (x: CGFloat, y: CGFloat) {
  return (x: p.x, y: p.y)
}

let point = createPoint(x: 1, y: 5)
valuesFromPoint(p: point)
```

Compound Types
Functions

```
func printInteger(_ value: Int) {
  print(value)
}

func executeOnInteger(_ value: Int, fn: (Int) -> Void) {
  fn(value)
}

executeOnInteger(4, fn: printInteger(_:))
```

```
4
```

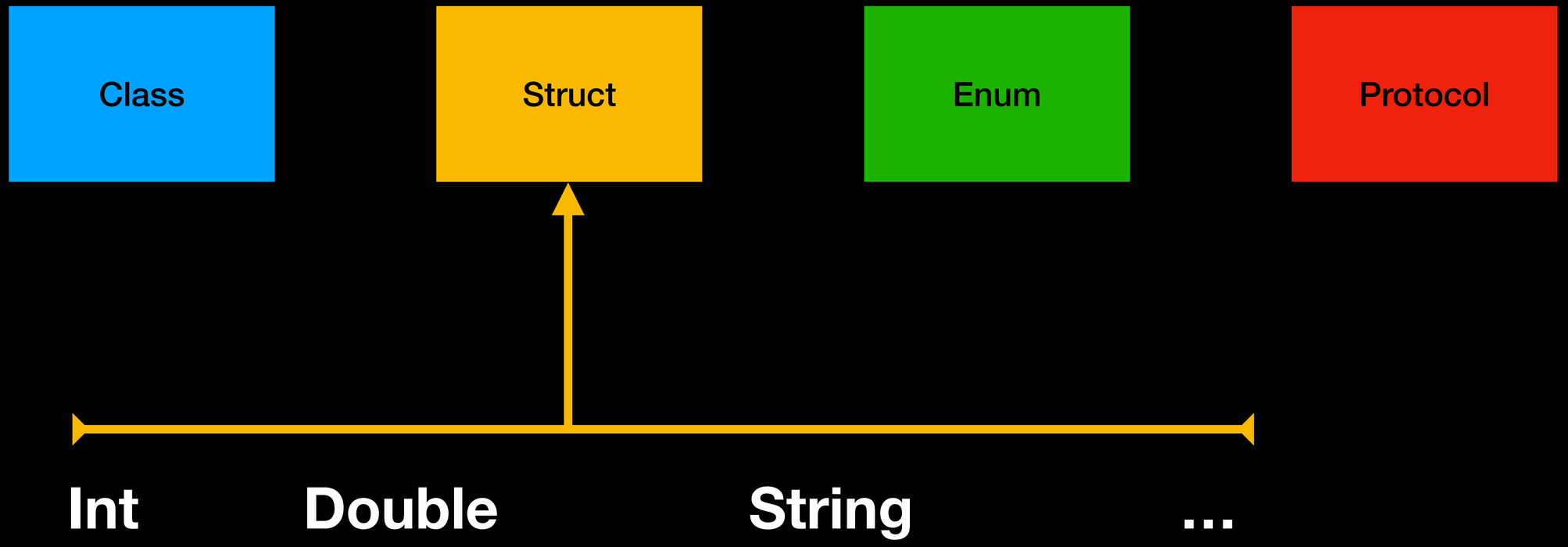
Named Types

Named Types

Named Types

Int Double String …

Named Types



Classes

Classes

```
final class MyClass2 {
  let constant: Int
  var variable: String
  lazy var lazilyInstantiated = "String"

  init() {
  }
}
```

Classes

```
final class MyClass2 {
  let constant: Int
  var variable: String
  lazy var lazilyInstantiated = "String"

  init() {
  }
}
```

Return from initializer without initializing all stored properties

Classes

```
final class MyClass {
  let constant: Int
  var variable: String
  lazy var lazilyInstantiated = "String"

  init() {
    constant = 1
    variable = "abc"
  }
}
```

Classes

```
final class MyClass {
  let constant: Int
  var variable: String
  lazy var lazilyInstantiated = "String"

  init() {
    constant = 1
    variable = "abc"
  }
}
```

No root class

Structs

Structs

```
struct Customer {
  let name: String
  let age: Int
}

// Initializer provided by default
let newCustomer = Customer(name: "Richard", age: 123)
```

Structs

```
struct Customer {
  let name: String
  let age: Int

  init(name: String, age: Int) {
    self.name = name
    self.age = age
  }
}
```

Automatically generated by the compiler

Class

- Has an identity
- Always passed by reference
- Supports inheritance

```
// Creating an instance of `MyClass`
let instance = MyClass()
// `sameInstance` refers to `instance`
let sameInstance = instance
```

Struct

- No identity
- “Value” semantics

```
let newCustomer = Customer(name: "Richard",
                           age: 123)
// `newCustomer` has been copied
let anotherCustomer = newCustomer
```

Enums

Enums

```
enum Direction {
  case left
  case right
}
```

Enums

```
enum Direction: String {
  case left
  case right
}

let direction = Direction.left
print(direction) // "left"
```

Enums

```
enum Direction: String {
  case left
  case right
}

let direction = Direction.left
print(direction) // "left"
```

Raw Type

Enums

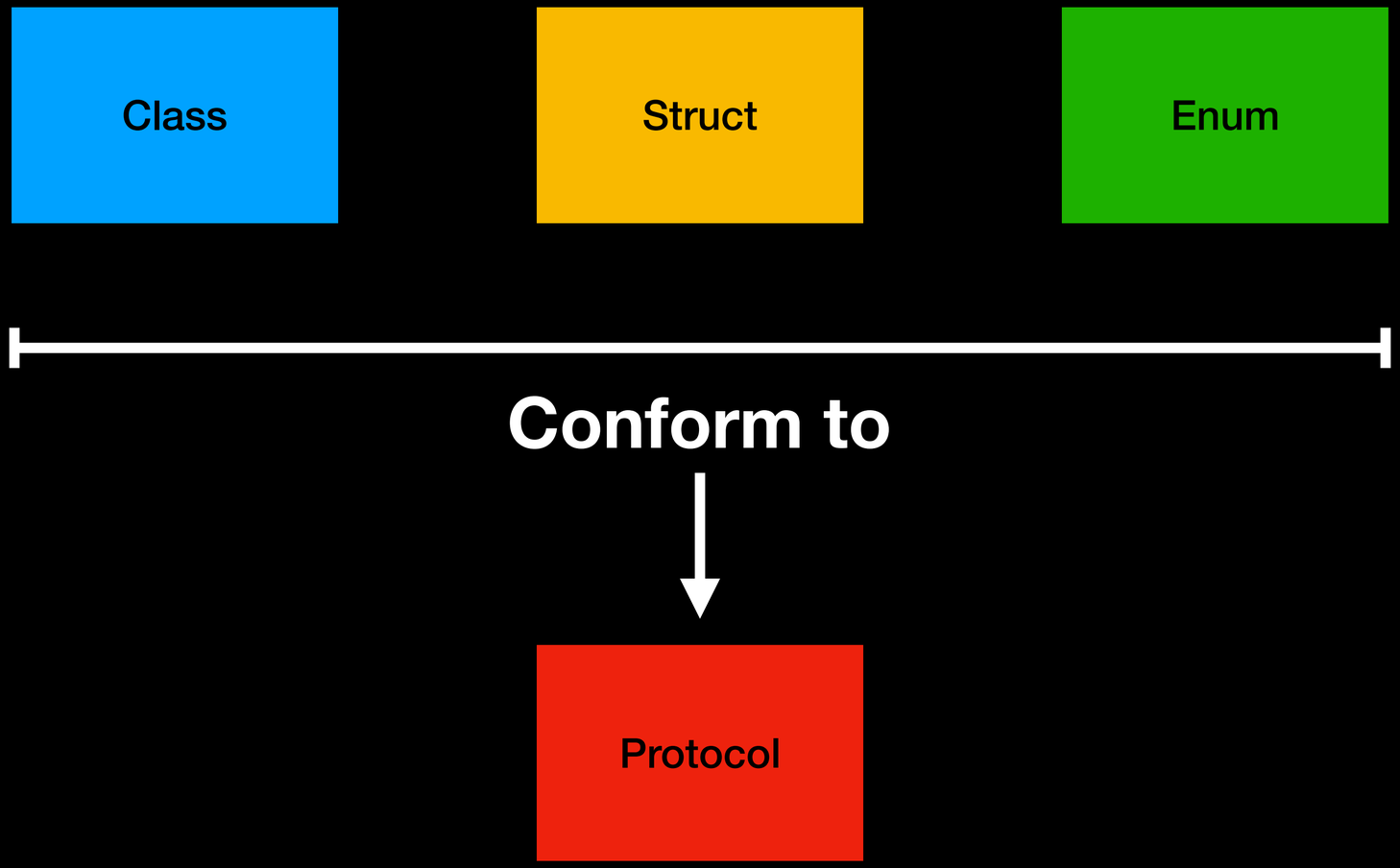
```
enum Direction: String {
  case left = "LeftDirection"
  case right = "RightDirection"
}
```

Protocols

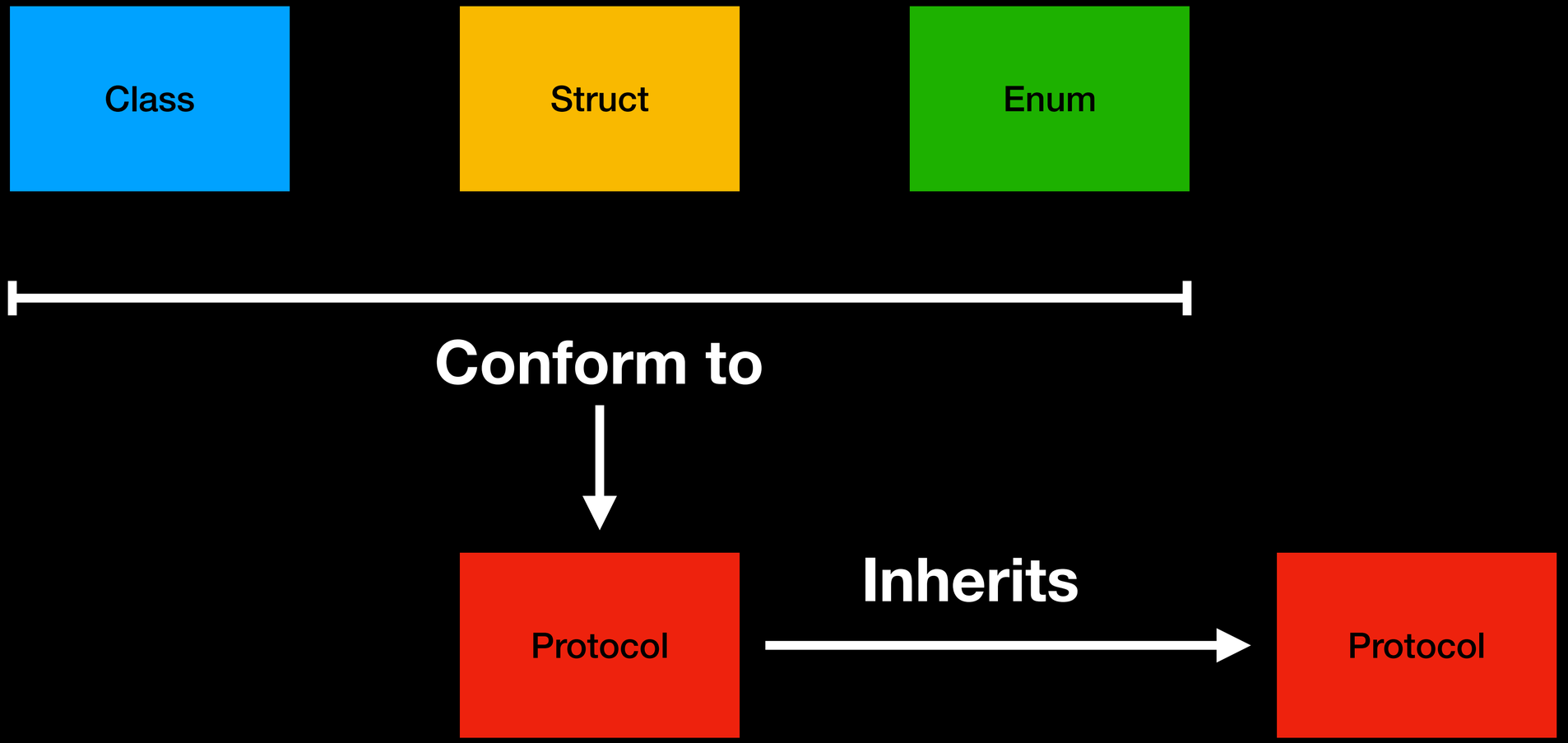
Protocols

- A set of requirements imposed on a Type
- Variable, initializer and method requirements
- Can inherit another Protocol(s)
- Can have a “default” implementation
- Protocols can be composed
- Similar to “Type classes” in Haskell

Protocols



Protocols



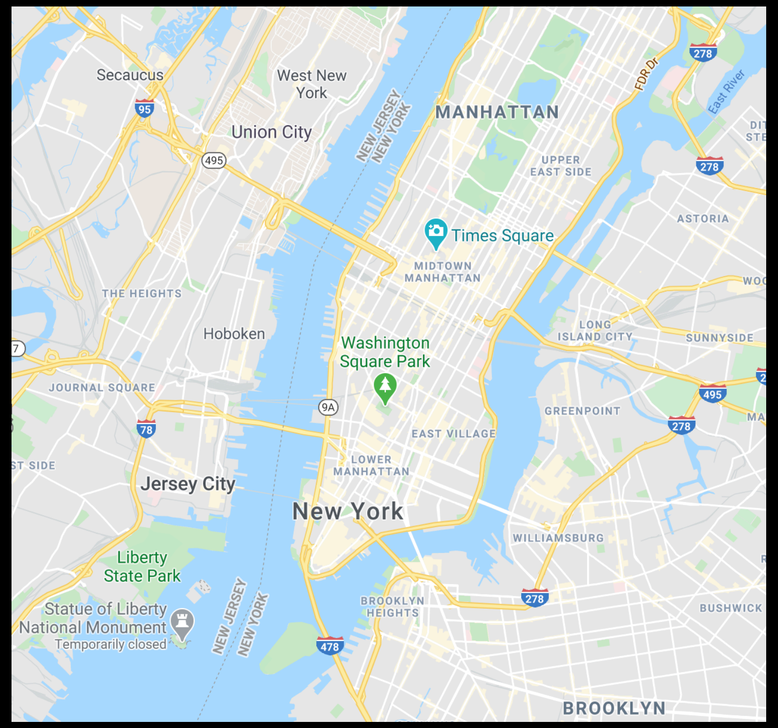
Protocols
Composition

Swift:
A Protocol-Oriented language

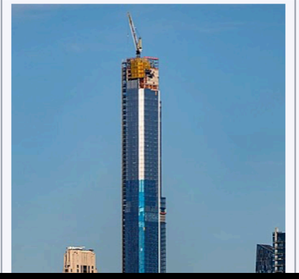
Protocol-oriented programming

An analogy: different points of view of the same object

Photo



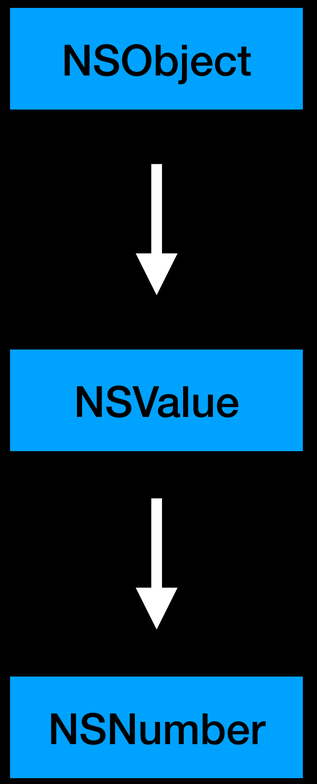
Map

Rank	Name	Image	Height ft (m)
1	One World Trade Center		1,776 (541)
2	Central Park Tower*		1,550 (472)

List of tallest buildings

Swift Standard Library

Objective-C



Protocols

- ExpressibleByFloatLiteral
- ExpressibleByIntegerLiteral
- ExpressibleByBooleanLiteral

Swift

Int

Protocols

- Hashable
- Decodable
- Encodable
- SIMDScalar
- Hashable

Equatable

```
public protocol Equatable {

    /// Returns a Boolean value indicating whether two values are equal.
    ///
    /// Equality is the inverse of inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is `false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to compare.
    static func == (lhs: Self, rhs: Self) -> Bool
}
```

Equatable

```
public protocol Equatable {

    /// Returns a Boolean value indicating whether two values are equal.
    ///
    /// Equality is the inverse of inequality. For any values `a` and `b`,
    /// `a == b` implies that `a != b` is `false`.
    ///
    /// - Parameters:
    ///   - lhs: A value to compare.
    ///   - rhs: Another value to compare.
    static func == (lhs: Self, rhs: Self) -> Bool
}


extension Equatable {

    public static func != (lhs: Self, rhs: Self) -> Bool
}
```

Codable = Decodable & Encodable

```
/// A type that can encode itself to an external representation.
public protocol Encodable {
    func encode(to encoder: Encoder) throws
}

/// A type that can decode itself from an external representation.
public protocol Decodable {
    init(from decoder: Decoder) throws
}

/// A type that can convert itself into and out of an external representation.
///
/// `Codable` is a type alias for the `Encodable` and `Decodable` protocols.
/// When you use `Codable` as a type or a generic constraint, it matches
/// any type that conforms to both protocols.
public typealias Codable = Decodable & Encodable
```

Demo

Protocol-Oriented Programming

Summary

- Focus on types and relationships between them
- Other types can be “embedded” into the relationship by adopting a protocol
- Provides option for retroactive data modeling

Q&A