iPhone Application Programming
Lab 3: Swift Types and Custom Operator
+ A02 discussion

Nur Al-huda Hamdan
Media Computing Group
RWTH Aachen University

Winter Semester 2015/2016

http://hci.rwth-aachen.de/iphone

Learning Objectives

- Discuss A02
- Another implementation for A02
  - Concepts: swift types (class, struct, enum), extensions, operators, typealias, access control, closures, first responder, alert view
  - Introduce A03

A02 Discussion

- KVO is another mechanism from object-to-object combination, especially Model → Controller
  - The observed object should inherit from `NSObject`. The observed property should be `dynamic`
  - The observing object should declare a context variable and override `observeValueForKeyPath`. Don't forget to remove the observer in `deinint`
- Device orientation (inherited from `UIViewController`)
  - Support orientations in overridden function `supportedInterfaceOrientations`
  - React to orientation change in `willRotateToInterfaceOrientation`
  - Swap views (portrait and landscape), or swap view controllers in a navigation controller, or implement constraints manually???
- The `info.plist` and the project settings should be equivalent when configuring your app
- `let tempValues = (-80...80).map { $0 }`

Store and Retrieve Data from NSUserDefaults

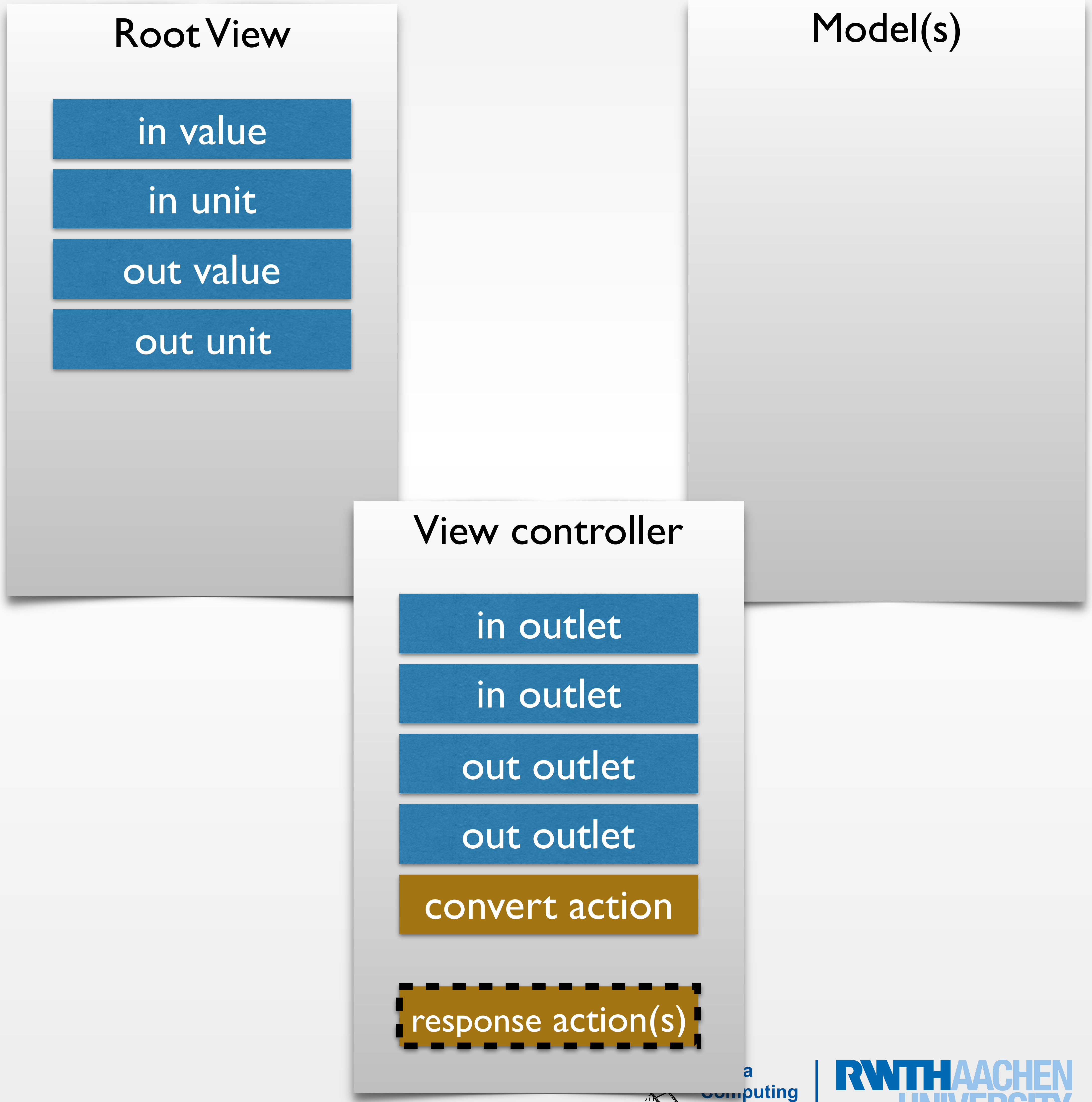
```
override func viewDidLoad() {
       super.viewDidLoad()
       let row = selectedRow()
       …
   }
 func selectedRow() -> Int {
       let selectedRow = NSUserDefaults.standardUserDefaults().objectForKey(userDefaultsLastRowKey) as? Int
       if let _ = selectedRow  {
           return selectedRow
       } else {
           …
       }
   }
 func pickerView(pickerView: UIPickerView, didSelectRow row: Int,
       inComponent component: Int) {
       …
       saveSelectedRow(row)
   }
func saveSelectedRow(row: Int) {
       let defaults = NSUserDefaults.standardUserDefaults()
       defaults.setInteger(row, forKey: userDefaultsLastRowKey)
       defaults.synchronize()
   }
```

A02 Reimplemented

- Functionality, extend to include Kelvin
- App structure: MVC
- Conversion algorithm

App Structure

- Read input: temperature value, from unit, to unit
- Convert temperature value (from unit → to unit)
- Write output: converted temperature value
- V from a design specification sheet
- C connect to V and M
- M?



Models

- We have a temperature of 2 properties: `value` and `unit` and a `convert` function
  - Class or Struct? to encapsulate `Temperature`
- We have 3 types of temperature units
  - Enum: TemperatureUnit
  - Enums can have functions that operate on their cases

Classes

- Inheritance
  - Initializers initialize all members before calling the parent initializer (2-phase init)
- Support for de-initializers
- Provide reference semantics
- Are (usually) created on the heap
- Good for shared data, large data, or as a resource handle

```
class Person {
  var firstName: String
  var lastName: String
  var available = true

  init(firstName: String, lastName: String) {
  self.firstName = firstName
  self.lastName = lastName
  }

  func marry(other: Person, takeTheirName: Bool) {
  if (takeTheirName) {
    self.lastName = other.lastName
  }
  self.available = false
  }

  func stringify() -> String {
  return firstName + " " + lastName +
         (available ? " is still available!"
                    : " is married.")
  }
}
```

Structs

- Collection of named properties
- Can have initializers and methods
- Provide value semantics
- Are (usually) created on the stack
- Can conform to protocols, can have extensions, but no inheritance
- Good for data aggregation without implicit sharing

```
struct MapPoint: Stringifyable {
  var longitude: Double
  var latitude: Double

  func rhumbDistance(other: MapPoint) ->
Double {
   let dLong = self.longitude -
other.longitude
   let dLat = self.latitude - other.latitude
   return sqrt(dLong * dLong + dLat * dLat)
  }

  func stringify() -> String {
   return "(\(longitude); \(latitude))"
  }
}
```

Structs vs. Classes

- **Structs**
  - short lived objects
  - objects that are created often
  - model objects
  - data capsules (represent only their values)

- **Classes**
  - long lived objects
  - controller and view objects
  - class hierarchies
  - objects in the true sense (representing some identity)

If unsure, try a struct first; you can change it later

Models

- ☑ We have a temperature of 2 properties: `value` and `unit` and a `convert` function
  - ☑ Class or Struct? `Temperature`
- We have 3 types of temperature units
  - Enum `TemperatureUnit`
  - Enums can have functions that operate on their cases

Enumerations

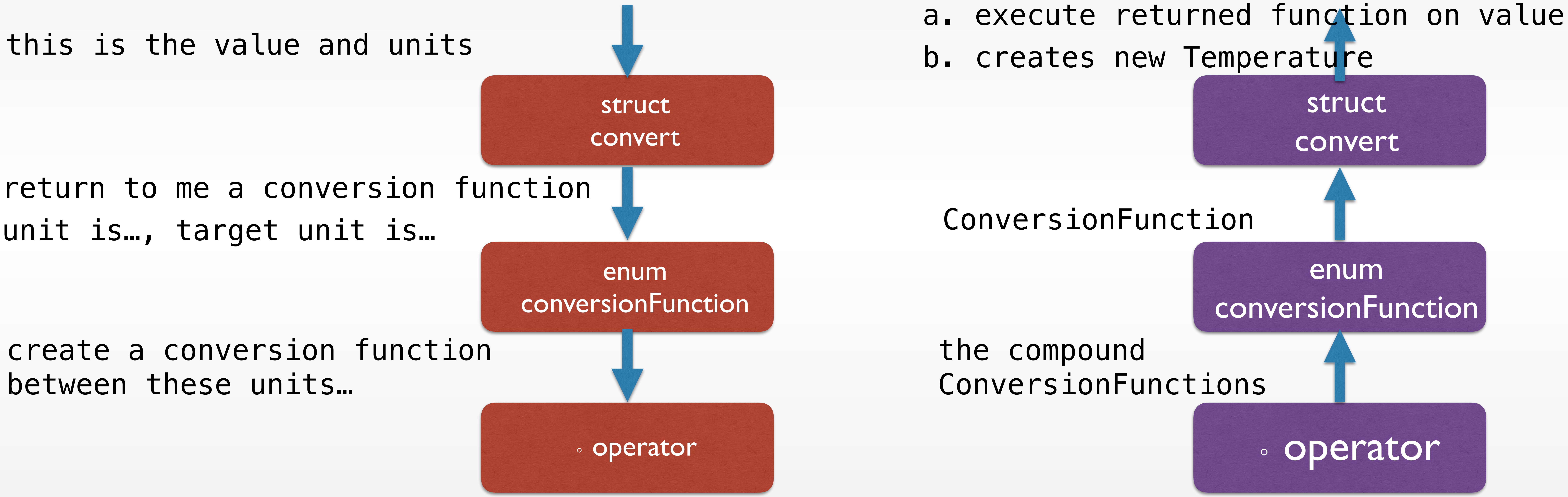
- Represent a finite number of states
- There are two distinct types of enumerations in Swift
  - Raw value enumerations
    - Similar to Java or C enumerations
  - Associated value enumerations
    - Similar to tagged unions (e.g. in Haskell)

Raw Value Enumerations

- Much more powerful than C enumerations
  - Can have methods and initializers, can have extensions and can conform to protocols
- More flexible than Java enumerations
  - Can be defined over other underlying types (String, Character, all numeric types)

```swift
enum TrainClass: String, Stringifyable {
  case S = "S-Bahn"
  case RB = "Regionalbahn"
  case RE = "Regional-Express"
  case IC = "Intercity"
  case ICE = "Intercity Express"
  static let allCases = [S, RB, RE, IC, ICE]

  func onTime() -> Bool {
   if self == .S || self == .ICE {
     return true
   }
   return false
  }

 func stringify() -> String {
   return self.rawValue
  }
}
```

Conversion Algorithm



this is the value and units

struct convert

return to me a conversion function

unit is…, target unit is…

enum conversionFunction

create a conversion function between these units…

∘ operator

a. execute returned function on value

b. creates new Temperature

struct convert

ConversionFunction

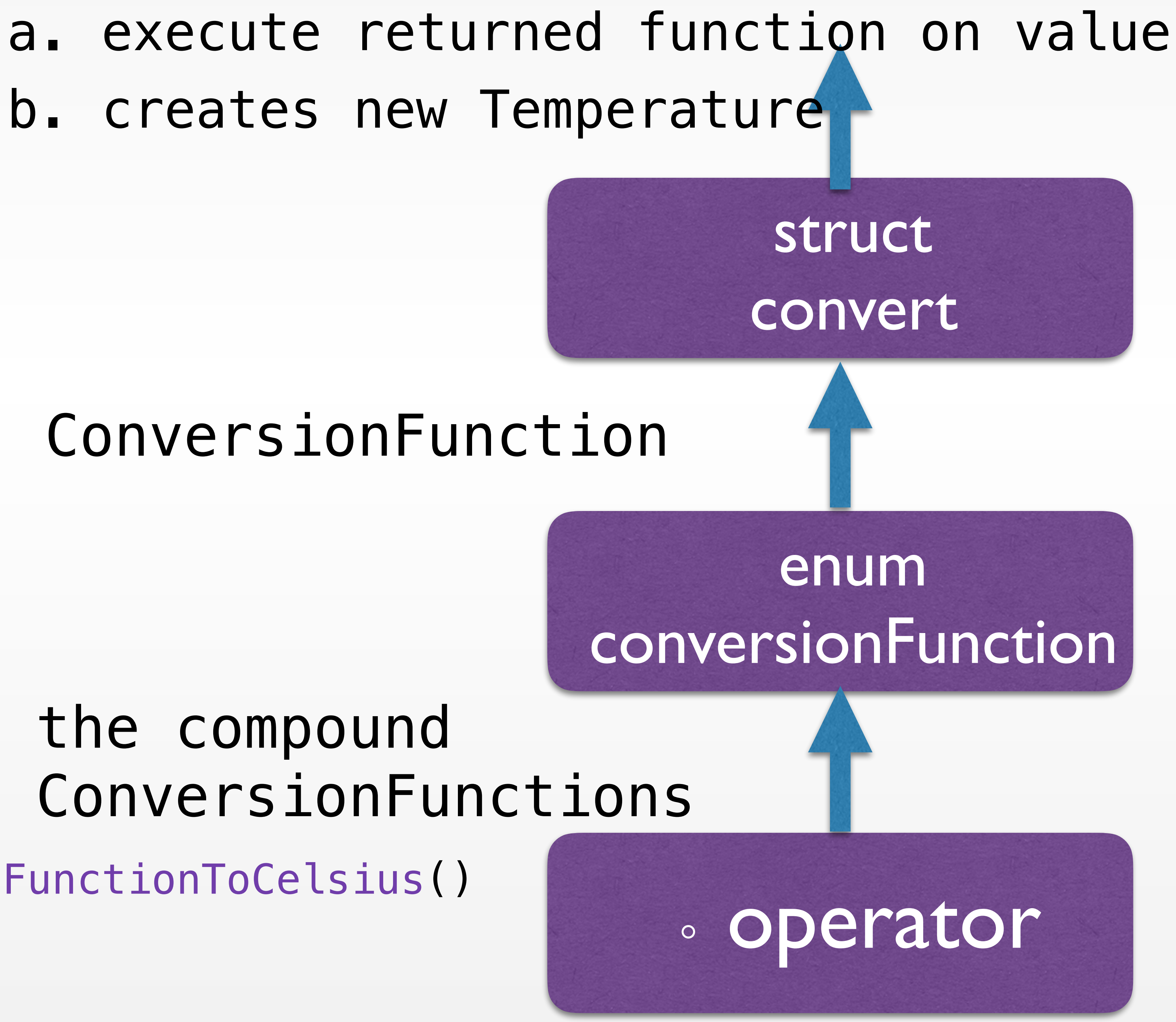
enum conversionFunction

the compound ConversionFunctions

∘ operator

Conversion Algorithm

- We want a function that takes temperature value, converts it to celsius, and then from celsius to target unit
    - `typealias ConversionFunction = (Double) -> Double`
- In the enum `TemperatureUnit`
    - `someCase.conversionFunction`, which calls
    - `someCase.conversionFunctionToCelsius`
    - `someCase.conversionFunctionFromCelsius`
- We will make `conversionFunction` call a custom operator
    - `targetUnitCase.conversionFunctionFromCelsius() ∘ currentUnitCase.conversionFunctionToCelsius()`
    - returns a compound function (type `ConversionFunction`) `conversionFunctionFromCelsius(conversionFunctionToCelsius(value))`



a. execute returned function on value

b. creates new Temperature

struct convert

ConversionFunction

enum conversionFunction

the compound ConversionFunctions

∘ operator

Conversion Algorithm

```
struct Temperature {
   let value : Double
   let unit : TemperatureUnit
   func convert(toTargetUnit targetUnit: TemperatureUnit) -> Temperature
  {unit.conversionFunction…..}
}
```

```
typealias ConversionFunction = (Double) -> Double
enum TemperatureUnit : String {
   case celsius = "°C"
   case fahrenheit = "°F"
   case kelvin = “°K”
    private func conversionFunctionFromCelsius() -> ConversionFunction {…}
   private func conversionFunctionToCelsius() -> ConversionFunction {…}
   func conversionFunction(toUnit targetUnit: TemperatureUnit) -> ConversionFunction {…}
}
```

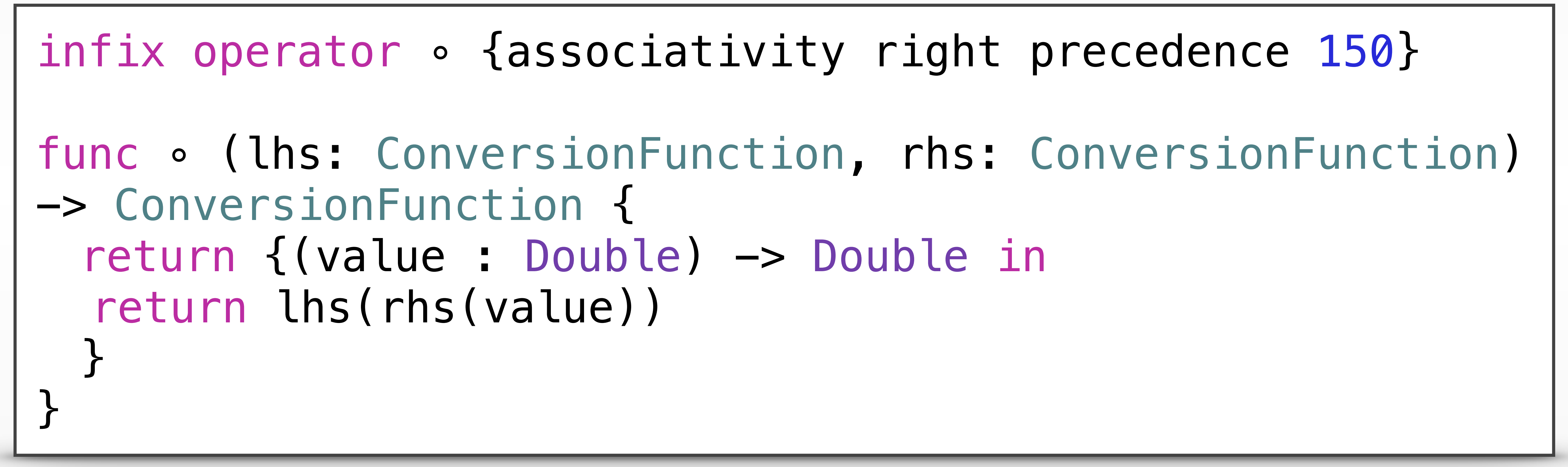
```
infix operator ∘ {associativity right precedence 150}

func ∘ (lhs: ConversionFunction, rhs: ConversionFunction) -> ConversionFunction {
   return {(value : Double) -> Double in
     return lhs(rhs(value))
   }
}
```

Access Control

- `private` entities are available only from within the source file where they are defined
- `internal` entities are available to the entire module that includes the definition (e.g. an app or framework target) ← the default case
- `public` entities are intended for use as API, and can be accessed by any file that imports the module, e.g. as a framework used in several of your projects
- Apply to classes, structures, and enumerations, properties, methods, initializers, and subscripts
- Global constants, variables, functions, and protocols can be restricted to a certain context

Custom Operators

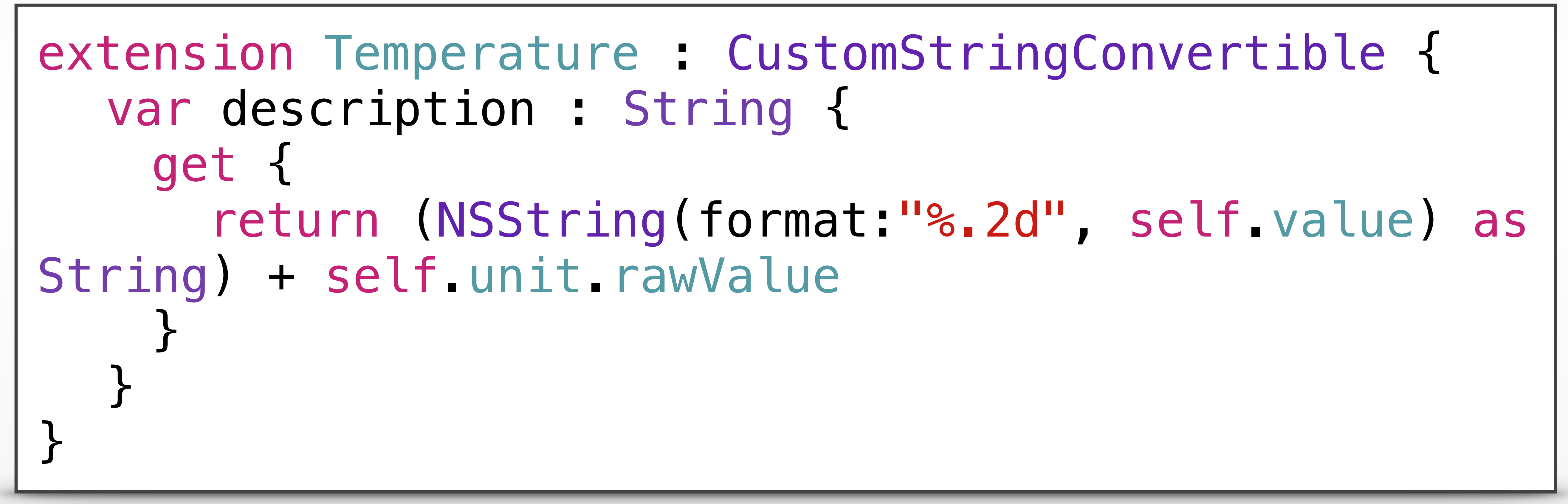
- Operators can be declared at global scope
- Can have prefix, infix or postfix modifiers
- Infix operators have associativity and precedence values
- Operators are implemented as functions at global scope
- Be very conservative when overloading operators!



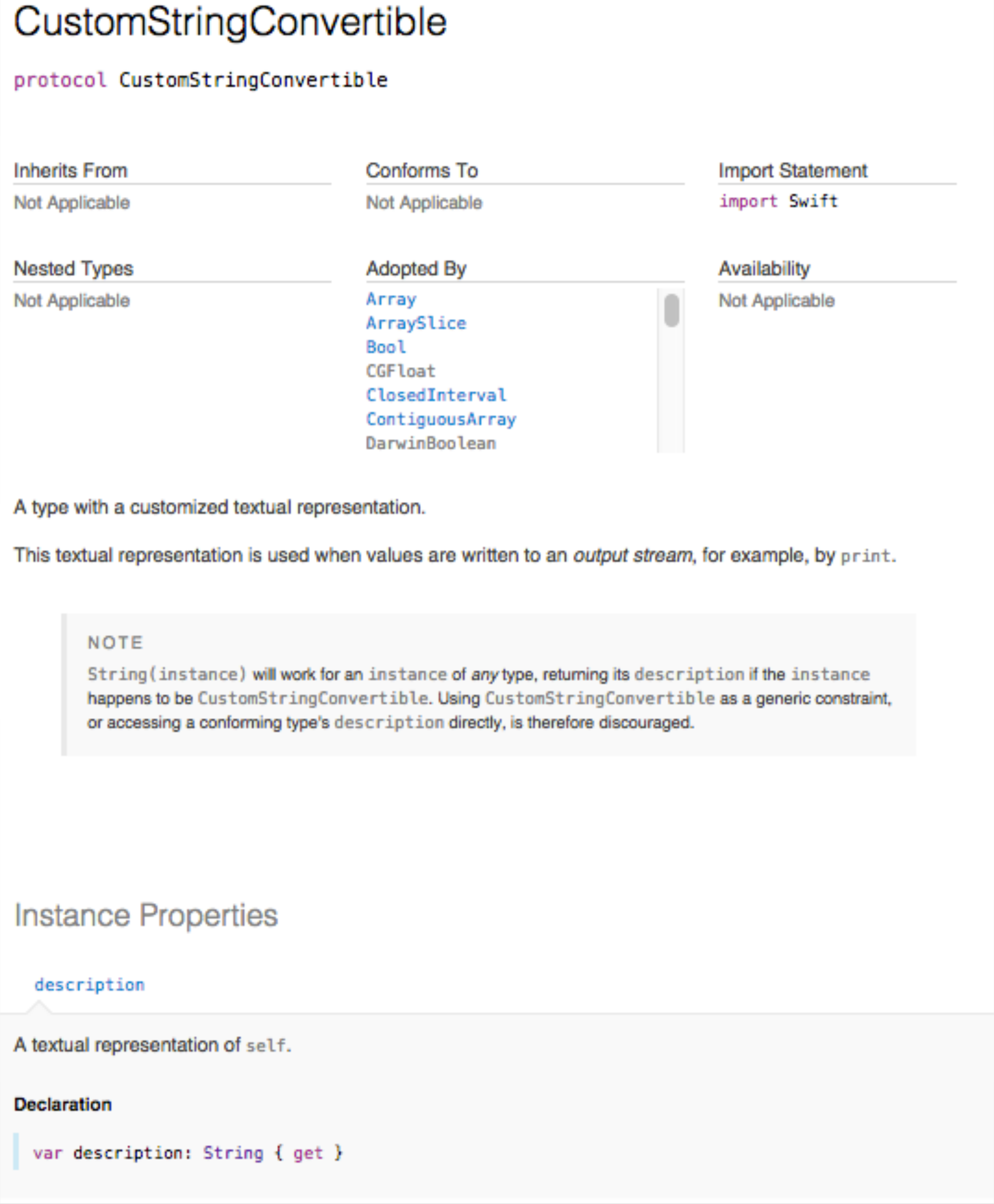
```
infix operator ∘ {associativity right precedence 150}

func ∘ (lhs: ConversionFunction, rhs: ConversionFunction)
-> ConversionFunction {
  return {(value : Double) -> Double in
   return lhs(rhs(value))
  }
}
```

Extensions

- Similar to Objective-C categories
- Can extend Structs, Classes, Enumerations
- Can add functions, computed properties, nested types
- Can declare protocol conformance
- Cannot override existing functionality
- Often useful to clean up code structure



```
extension Temperature : CustomStringConvertible {
   var description : String {
     get {
        return (NSString(format:"%.2d", self.value) as
String) + self.unit.rawValue
     }
   }
}
```



CustomStringConvertible

```
protocol CustomStringConvertible
```

Inherits From	Conforms To	Import Statement
Not Applicable	Not Applicable	`import Swift`

Nested Types	Adopted By	Availability

Not Applicable	`Array`
	`ArraySlice`
	`Bool`
	`CGFloat`
	`ClosedInterval`
	`ContiguousArray`
	`DarwinBoolean`
		Not Applicable

A type with a customized textual representation.

This textual representation is used when values are written to an *output stream*, for example, by `print`.

> NOTE
>
> `String(instance)` will work for an `instance` of *any* type, returning its `description` if the `instance` happens to be `CustomStringConvertible`. Using `CustomStringConvertible` as a generic constraint, or accessing a conforming type's `description` directly, is therefore discouraged.

Instance Properties

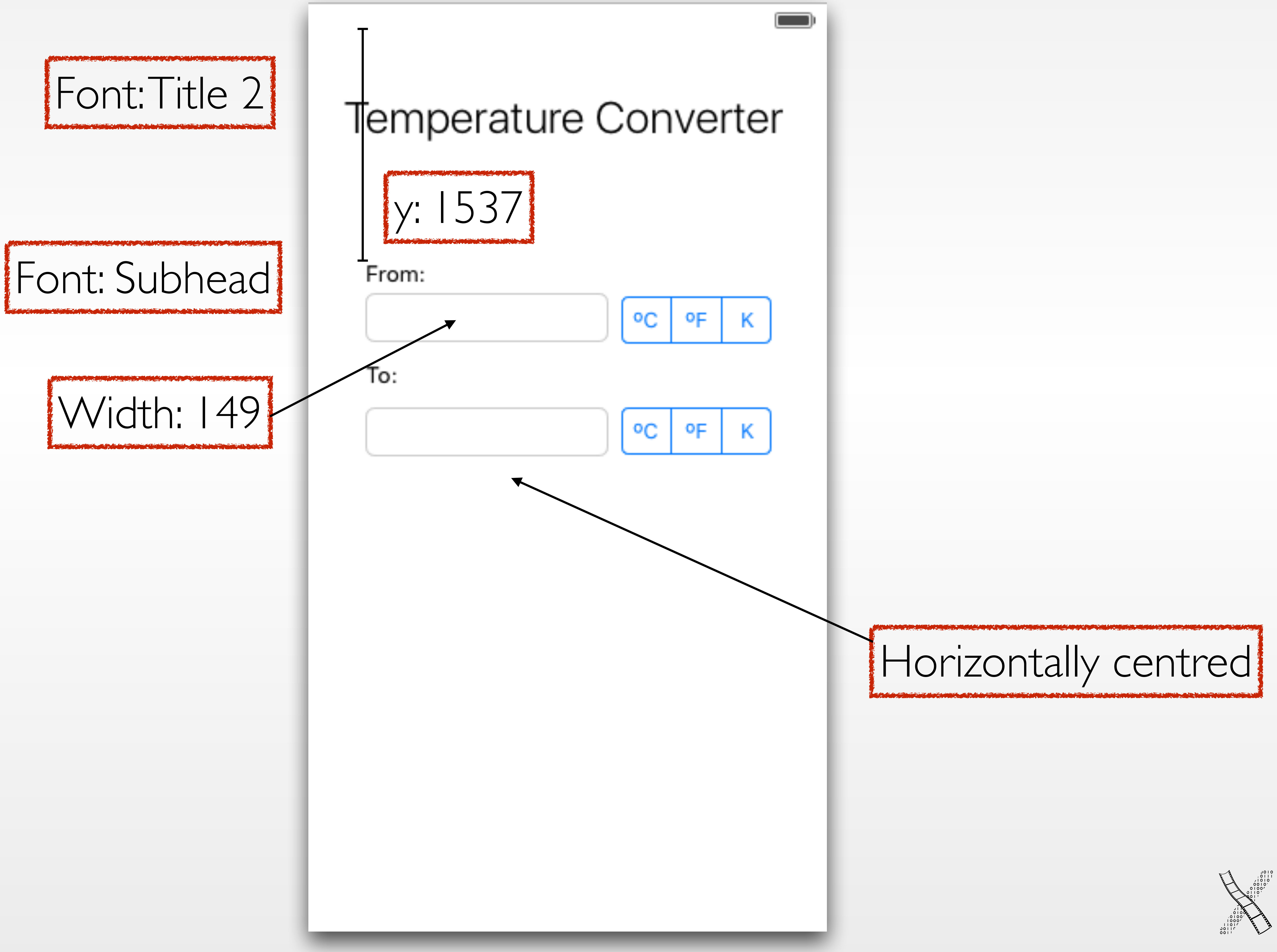
`description`

A textual representation of `self`.

Declaration

```
var description: String { get }
```

Design Specification



Font: Title 2

y: 1537

Font: Subhead

Width: 149

Horizontally centred

Temperature Converter

From:

To:

A03

- Part 1: Swift types and UI focus
- Part 2: Undo
- Part 3: Storage
- Part 4: Unit tests