iPhone Application Programming
Lecture 3: Swift Part 2

Nur Al-huda Hamdan
Media Computing Group
RWTH Aachen University

Winter Semester 2015/2016

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

Media Computing Group | RWTH AACHEN UNIVERSITY

Properties

- Properties are available for classes, enums or structs
- Classified into stored properties and computed properties
- Can be instance properties: each class instance gets its own copy or type properties: associated with the type itself (`static`)
- One can observe stored properties or any inherited property
- `lazy` properties do not calculate initial values when the variable is initialized for the first time
  - To delay object creation until necessary (resource demanding) or when property depends on unknown parts of the class

Computed	Stored
For classes, structs and enums	For classes and structs
Calculate a value (usually based on stored properties)	Store values as instances into memory
No need to initialize. Cannot have a default value	Must be initialized
Only var	Can be var or let
Have get and optional set	

Properties

- To observe properties you implement `didSet` or `willSet`
  - When a property is set in an initializer `willSet (newValue)` and `didSet (oldValue)` observers are not called (or when assigning initial default value)
  - You cannot observe `lazy` properties
  - `override` inherited properties to observe them. Cannot observe read-only properties
  - Property observer must be `var`
  - Use to validate input
- A constant `let` struct instance cannot modify even if properties, were declared as variables

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

Self

- Every instance of a type (class, struct, enum) has an implicit property called self
- Cannot be used until after initialization phase I
- Necessary to distinguish when a parameter name is the same as a property name, e.g., self.value = value
- Value types (enums and structs) can assigning to `self` a new value within a `mutating` method

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(deltaX: Double, y
deltaY: Double) {
        self = Point(x: x + deltaX, y: y +
deltaY)
    }
```

Inheritance

- Unique to classes in swift
- Classes in Swift can call and access methods, properties, and subscripts belonging to their superclass: super.someMethod() or super.someProperty (even of `private`)
- Classes can provide their own `overrid`ing versions of those methods, properties, and subscripts
  - You can make an inherited read-only property a read-write property, but cannot make a read-write property read-only
- Classes can add property observers (`didSet`, `willSet`) to inherited (settable) properties (the stored or computed nature of an inherited property is not known by a subclass)
- In superclass: `final` computed properties and functions cannot be overridden. `final class` means it cannot be subclassed

Initialization

- Initialization prepares instances of a class, structure, or enumeration for use by setting an initial value for *each stored property* and performing any other setup
- Classes and structures must set all of their stored properties to an appropriate initial value before they can be used
    - Default property value set in definition (except for optionals, default is nil)
    - Initial value within an initializer
- We call Initializers to create new instances

Initialization

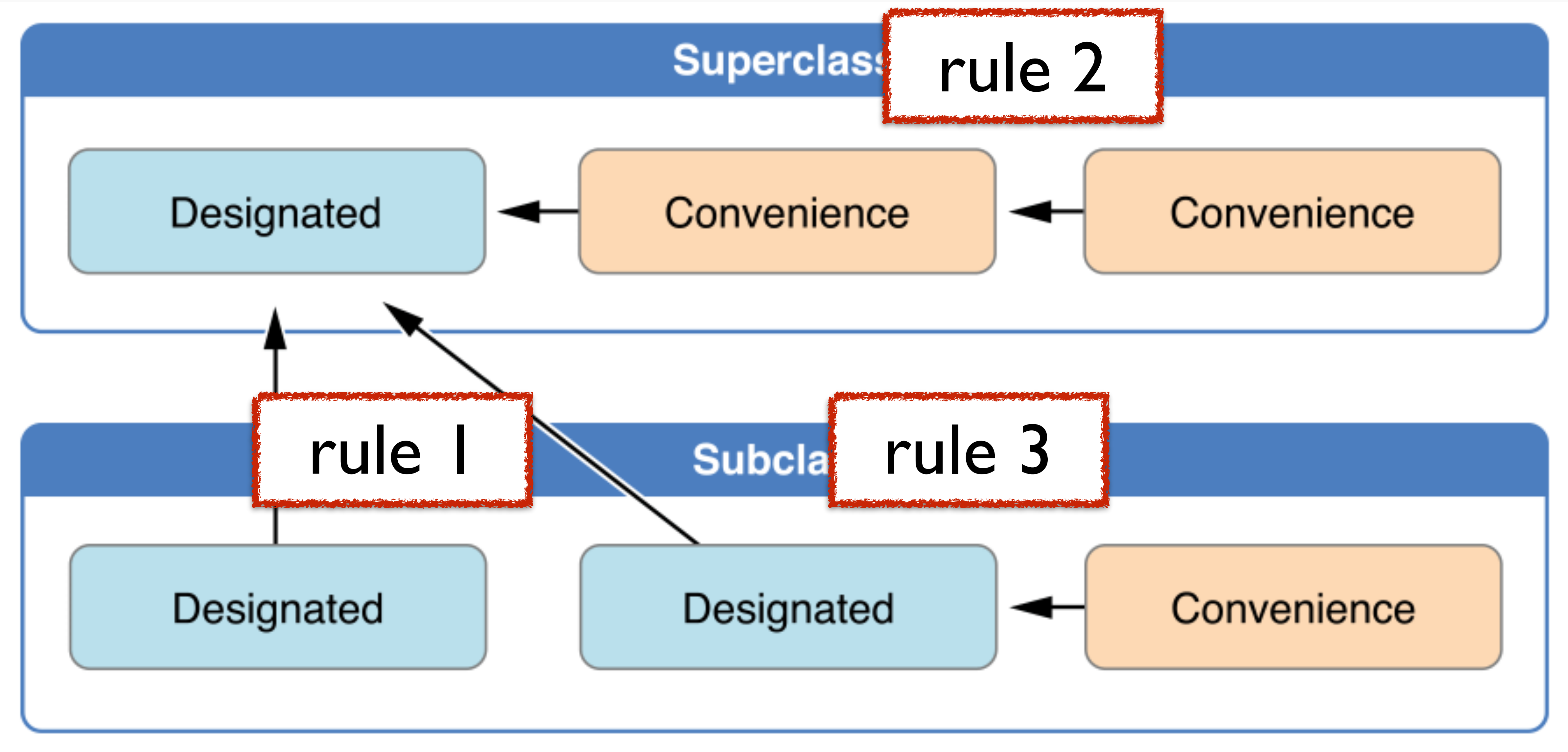
- Initializers syntax: can be with or without parameters, can have local and external names, must use first parameter name when calling the init, can use wild card for external names
- A class and struct that have *all* properties set with default values get a default `init()` if they do not implement one (var instance = className() is possible without writing any initializer for className)
- Structs also receive a default memberwise initializer: init(all properties in order of definition), if they do not define any initializers
- What if you want the default init/memberwise init in your struct but also want custom inits?

Initialization and Inheritance

- Two kinds of initializers for type class
  - **Designated** initializers are the primary. They initializes *all* properties introduced by that class and call an appropriate superclass initializer to continue the initialization process up the superclass chain
    - Every class must have at least one designated initializer (can satisfy this by inheriting a superclass designated init)
  - **Convenience** initializers are optional in a class, and used for special initialization patterns (must add `convenience init`)
- Swift subclasses do not inherit their superclass initializers by default (see demo cases)
  - If subclass implements init() {} and the super class has the default init, the subclass must add `override` keyword

Initializer Delegation for Class Types

- Goal: All of a class's stored properties, including inherited properties, must be assigned an initial value during initialization
- Convenience initializer can only call *one* other initializer from the *same* class (the chain should lead to a designated initializer)
- Designated initializers must call *one* super designated initializer



Two-Phase Initialization

- Class initialization in Swift is a two-phase process
- Safe and flexible process
  - Prevents property values from being accessed before they are initialized
  - Prevents property values from being set to a different value by another initializer unexpectedly
  - Allows setting custom initial values
- Phase 1: Each stored property is assigned an initial value by the class that introduced it
- Phase 2: Each class is given the opportunity to customize its stored properties further before a new instance is ready for use

Two-Phase Initialization

- A designated initializer must ensure all its properties are initialized before calling super designated. After calling the super, it can modify inherited properties
- A convenience initializer must delegate to another initializer before assigning a value to *any* property
- An initializer cannot call any instance methods, read the values of any instance properties, or refer to self as a value until after the first phase of initialization is complete

Failables and Deinitializers

- `required init` indicates every subclass must implement that initializer., every subclass must also include this keyword
- Failable Initializer
  - When the initialisation of an instance can fail
  - Example, invalid initialization parameter values, the absence of a required external resource

```
init?(species: String) {
    if species.isEmpty { return nil }
    self.species = species
}
```

- Deinitializers to classes in swift (`deinit`)
  - Called automatically before instance deallocation takes place
  - Cannot be call by developer
  - Perform resource handling, e.g., close open files, remove self as an observer, etc

Closures

- Blocks of functionality that you can pass around in your code
- Closures do not have a name
- Closures capture references of values in their context
  - Retain cycles and memory management is done by swift
- Functions and nested functions are special cases of closures
  - Functions have a name and don't capture values
  - Nested functions have a name and capture values
- Many swift methods and functions take closures as arguments

Closures - Syntax

- Closure expressions encourage brief, clutter-free syntax
  - Inferring parameter and return value types from context
  - Implicit returns from single-expression closures
  - Shorthand argument names
  - Trailing closure syntax
- Can use constant parameters, variable parameters, and inout parameters, named variadic parameter and tuples
- Cannot provide default values

```
increment({(a: Int) -> Int in
        return a + 1
})

increment({a in return a + 1})

increment({a in a + 1})

increment({$0 + 1})

increment() {$0 + 1}

increment {$0 + 1}
```


Media Computing Group | RWTH AACHEN UNIVERSITY

Closures - Capturing References

- Capturing references to variables and constants that exist in the context

```
var i = 10
var myClosure = {print(i)}
i = 20
myClosure() //20
```

```
class MyClass
{
  var someProperty = "v1"
}
var instance = MyClass()

var myClosure = {
    (appName : String) -> String in
    return appName + " " +
instance.someProperty
}
```

```
print(myClosure("Clock")) //Clock v1

instance.someProperty = "v2"
print(myClosure("Clock")) //Clock v2

instance = MyClass()
print(myClosure("Clock")) //Clock v1
```

Closures - Capturing Values

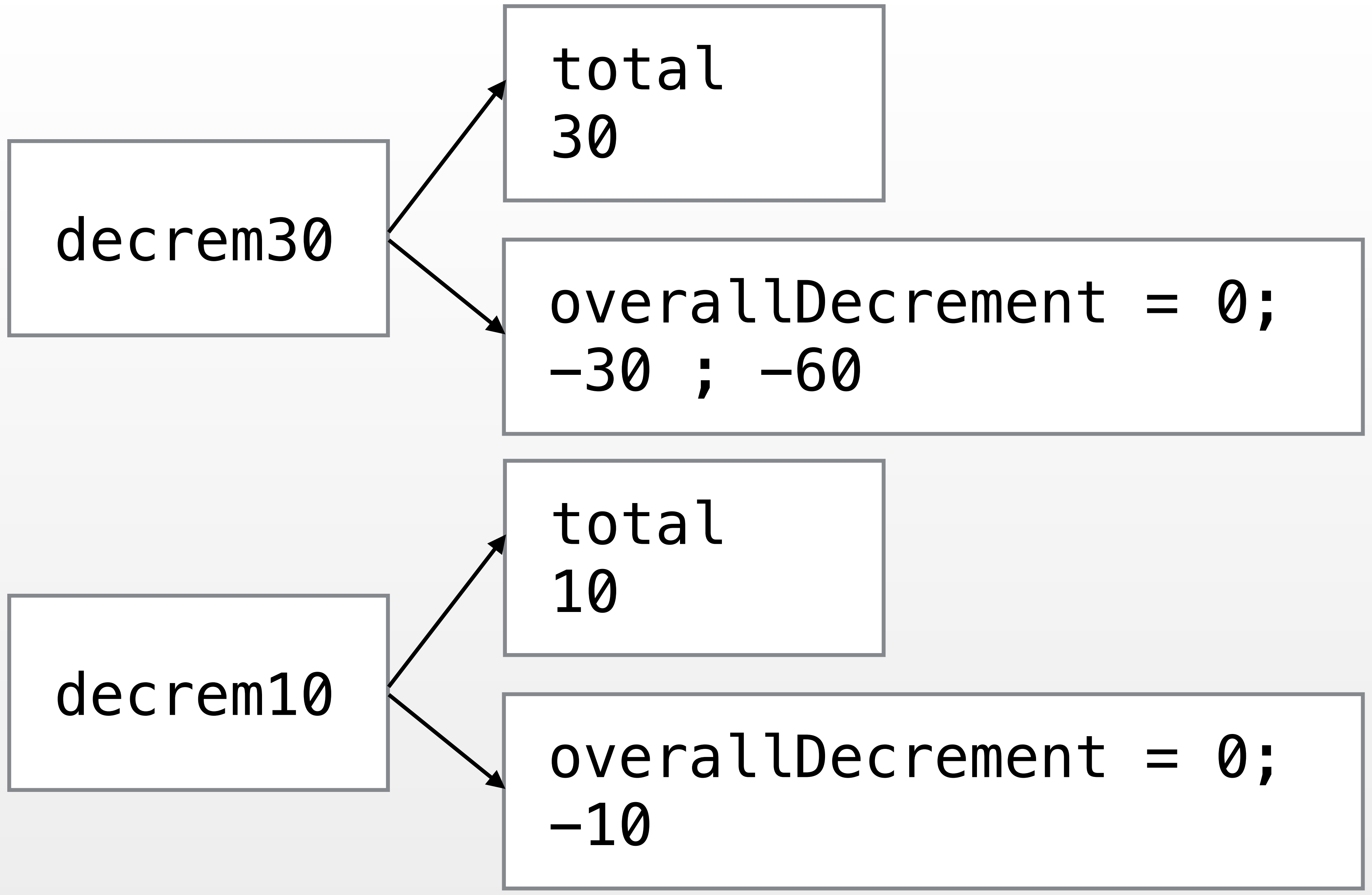
- Capture lists can change the default behavior of closures to capture values
  - You capture the values of constants and variables at the time of closure creation, not affected with any changes later
  - List must come at the beginning of closure definition

```
class MyClass
{
  var someProperty = "v1"
}
var instance = MyClass()

var myClosure = {
    [instance]
    (appName : String) -> String in
    return appName + " " +
instance.someProperty
}
```

```
print(myClosure(“Clock")) //Clock v1

instance.someProperty = "v2"
print(myClosure(“Clock")) //Clock v2

instance = MyClass()
print(myClosure(“Clock")) //Clock v2
```

Closures Are Reference Types

- A closure is a function + captured variables
  - These two are closures `decrem30`, `decrem10`




```
func calcDecrement(forDecrement total: Int) -> ()->Int
{
    var overallDecrement = 0

   func decrementer() -> Int {
        overallDecrement -= total
        return overallDecrement
    }

    return decrementer
    //overallDecrement normally goes out of scope here,
    but a reference to it is captured by decrementer
}

let decrem30 = calcDecrement(forDecrement: 30)
//now captured decrem30.overallDecrement is -30
print(decrem30()) //-30

let decrem10 = calcDecrement(forDecrement: 10)
//now captured decrem10.overallDecrement is -10
print(decrem10()) //-10

print(decrem30()) //decrem30.overallDecrement = -60
```

Swift Built-in Types

- Make better use of Swift's six built-in types

Named Types

- Protocols
- Structs
- Classes
- Enumerations

Compound Types

- Functions
- Tuples

Protocols

- A protocol defines a blueprint of (instate/type) methods, (instance/type) properties that suit a particular task or piece of functionality
- The protocol can then be adopted by a class/structs/enum and provide actual implementation of those requirements (conform to that protocol)
  - Some elements of the protocols can be tagged as `optional`
- Swift reports an error at compile-time if a protocol requirement is not fulfilled
- Protocols can be extend to implement some of the requirements or to implement additional functionality that conforming types can take advantage of

Protocols

- Protocol syntax: `protocol`, Adopting classes add protocol names after the inherited superclass (if exits)
- A protocol property should be a `var` and have a particular name and type, must be gettable or gettable and settable. If gettable, the conforming type can make it settable. The conforming type can implement it as `let` or `var`
- Type properties and method prefix with `static` (can use class or static in implementation)

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
- Use `mutating` func if changing an instance property in a struct method
- 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))"
  }
}
```

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

```swift
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 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

Value Semantics and Reference Semantics

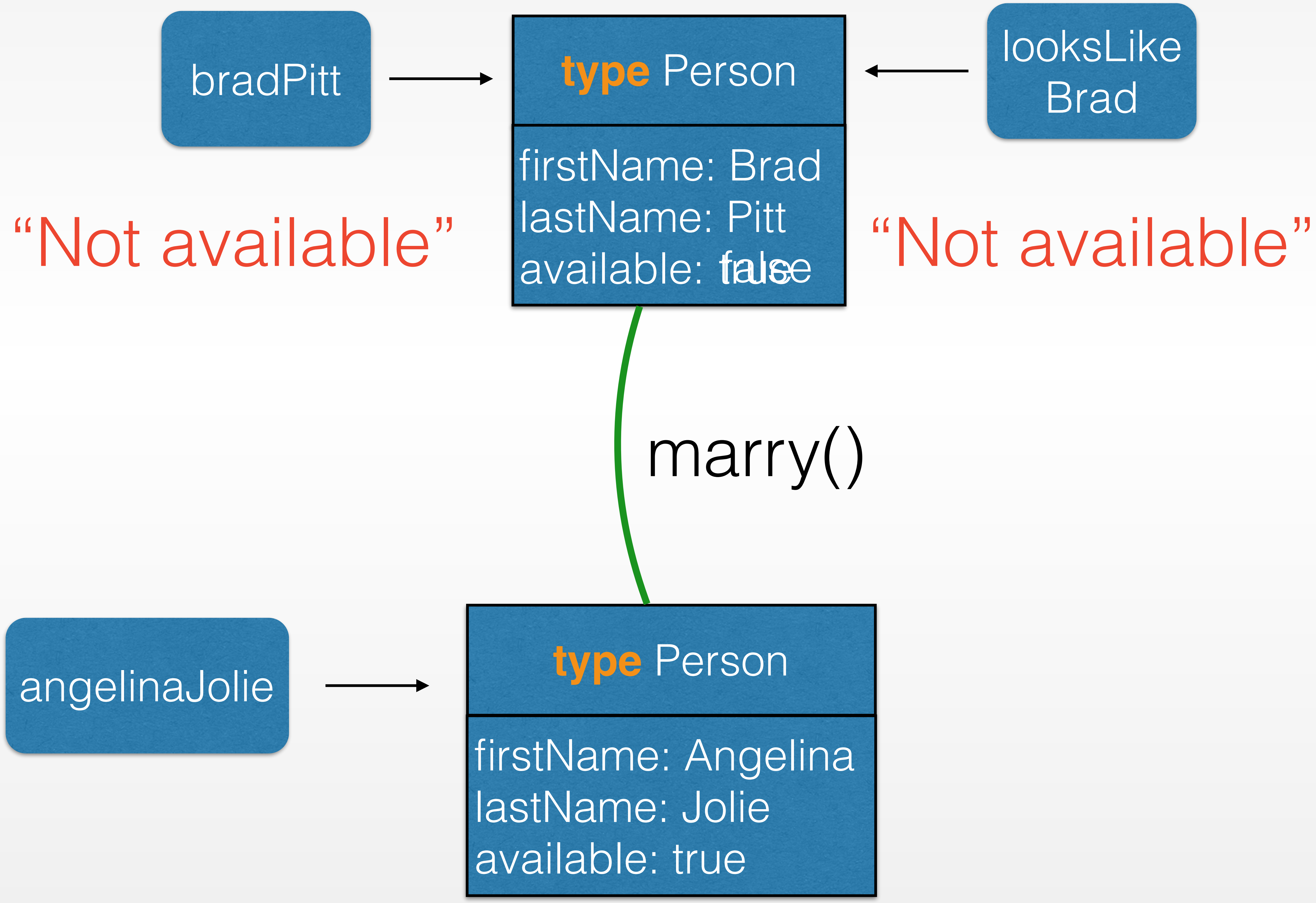
A Detour

Reference Semantics

```swift
protocol Stringifyable {
   func stringify() -> String
}

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.")
   }
}
```

Reference Semantics

```
var bradPitt = Person(firstName:
"Brad", lastName: “Pitt")

var angelinaJolie = Person(firstName:
"Angelina", lastName: “Jolie")

var guyWhoLooksLikeBradPitt = bradPitt

bradPitt.marry(angelinaJolie,
takeTheirName: false)

bradPitt.stringify()

guyWhoLooksLikeBradPitt.stringify()
```

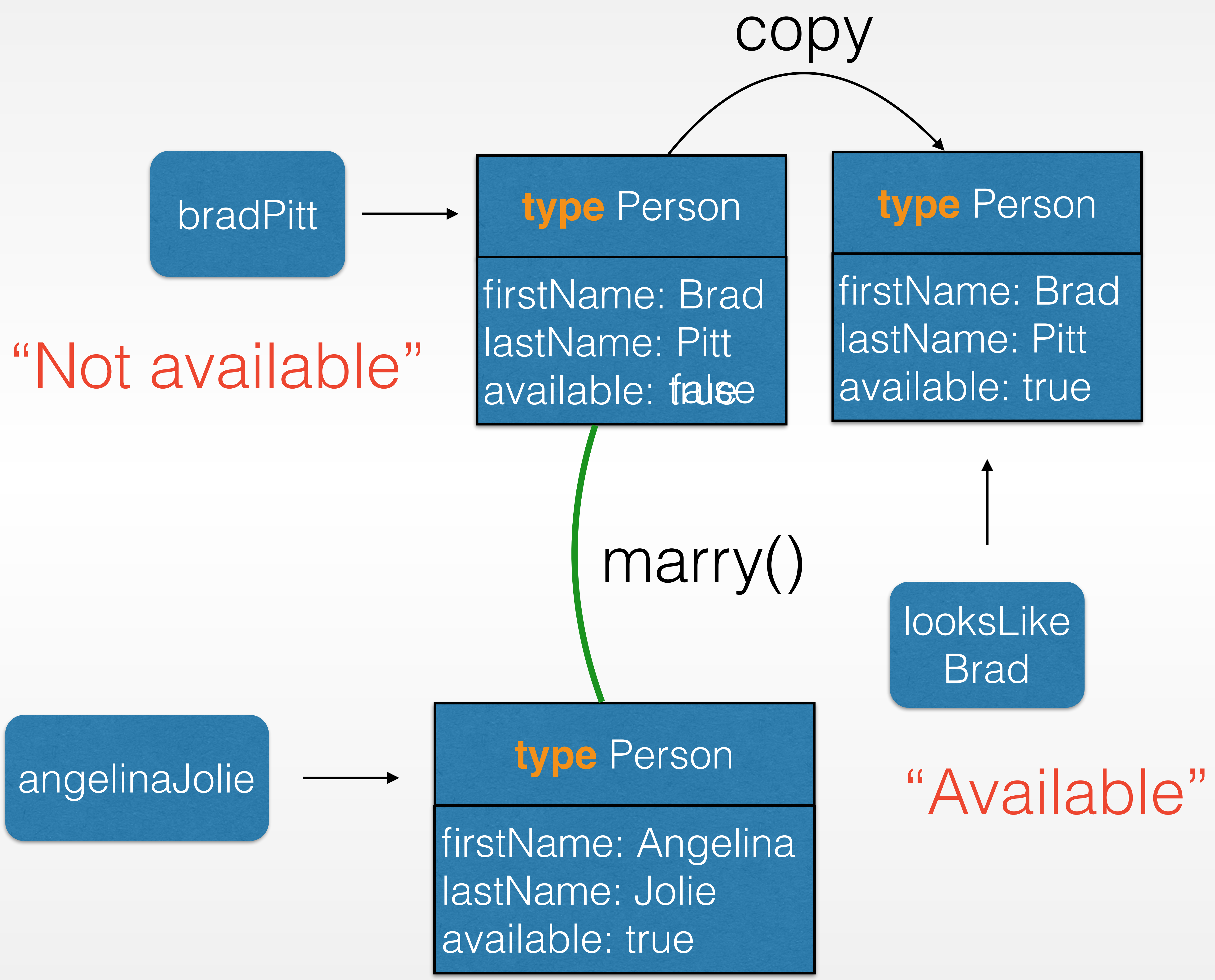


Value Semantics

```
protocol Stringifyable {
    func stringify() -> String
}

struct Person {
    var firstName: String
    var lastName: String
    var available = true

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

    mutating 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.")
    }
}
```

Value Semantics

```
var bradPitt = Person(firstName:
"Brad", lastName: “Pitt")

var angelinaJolie = Person(firstName:
"Angelina", lastName: “Jolie")

var guyWhoLooksLikeBradPitt = bradPitt

bradPitt.marry(angelinaJolie,
takeTheirName: false)

bradPitt.stringify()

guyWhoLooksLikeBradPitt.stringify()
```



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)

```
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
  }
}
```

Associated Value Enumerations

- Every case represents a tuple type
  - Can be used as simple static Polymorphism
- Instantiate cases with values of the represented type

```
enum Transport {
  case plane(String, Int)
  case train(TrainClass, Int)
  case bus(Int)
  case car(String, String, Int)
}

var myRide = Transport.train(.ICE, 11)
// GDL strike: change travel plans!
myRide = .car("AC", "X", 1337)

func canWork(onRide: Transport) -> Bool {
  switch onRide {
  case .train(let trainClass, let number):
    return trainClass == .ICE
  case .plane(_, _):
    return true
  default:
    return false
  }
}
```

Extensions

- Can extend Structs, Classes, Enumerations
- Can implement protocol requirements
- 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
    }
  }
}
```

Nested Types

- Net enumerations, classes, and structures within the definition of a type
- Can have deep hierarchies
- To use a nested type outside definition scope, prefix its name with the name of the type(s) it is nested within.

Optional Chaining

- self.window?.rootViewController?.view.subviews
- If one of the optionals is nil, this fails graceful (no run time error)
- If all optionals are set, the chain return an optional (even if the object in request, e.g., subviews, is not optional)
- let views = (self.window?.rootViewController?.view.subviews)!
- let views = self.window?.rootViewController?.view.subviews! `// compiler error, subviews is not of type optional`
- With subscripts dict?[someKey].instanceOnValue

```
for subview in
(self.window?.rootViewController?.view.subviews)!
as [UIView]
 {
       //type casting the subview to UILable
      if let labelView = subview as? UILabel
      {
        let formatter = NSDateFormatter()
        formatter.timeStyle = .MediumStyle
         labelView.text =
formatter.stringFromDate(NSDate())
       }
 }
```

Type Casting

- Upcasting: Casts an instance to its superclass type (assumes it is always successful)
  - instance `as` superclass
  - 0.1 `as Int //0` and 0.1 `as Double //0.1`
- Downcasting: Casts an instance of a superclass to its actual subclass type
  - let object = instance `as`! subclass. Results in downcasts + force unwarp OR runtime error
  - if let object = instance `as`? subclass {…}. Results in downcasts or nil
- Object checking: Checks if instance of type subclass
  - instance `is` subclass `//true or false`

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!

```
// ...this one maybe makes sense...
prefix operator ∑ {}
prefix func ∑(a: [Int]) -> Int {
   var accum = 0
   for value in a {
     accum += value
   }
   return accum
}
var myArray = [-2, 6, 0, 1]
let sum = ∑myArray

// ...this one surely not!
postfix operator ^-^ {}
postfix func ^-^(s: String) -> String {
   return s + " 😄"
}
let chatMessage = "Operator Overloading 4TW!"
print(chatMessage^-^)
```

Next Time

- The slides and playgrounds from this lecture will be uploaded to our website
- This week's reading assignment will be on the website today
- What is left in Swift? ARC and Error handling (next lecture); Generics and Subscripts (self reading)
- Next week we'll talk about design patterns and Foundation classes