Object-Oriented Programming in Swift

Alex Vollmer
@alexvollmer
http://alexvollmer.com

pluralsight
hardcore dev and IT training

Review

Single inheritance model	Properties	Methods
Extensions	Polymorphism through Protocols	Reference types

Class Declarations

```
@interface MyViewController : UIViewController <UITableViewDelegate>
...
@end
```

```
class MyViewController: UIViewController, UITableViewDelegate  {
    ...
}
```

```
class MyViewController

@interface MyViewController
```

```
let vc = MyViewController()
MyViewController *vc = [[MyViewController alloc] init]
```

Initializers

`init()`

Use special `init()` syntax

Setup property values

Prepare instance for use

Leaving the Factory

```
NSNotification *not = [NSNotification notificationWithName:@"Howdy" object:nil];

NSNotification *not = [[NSNotification alloc] initWithName:@"Howdy" object:nil];
```

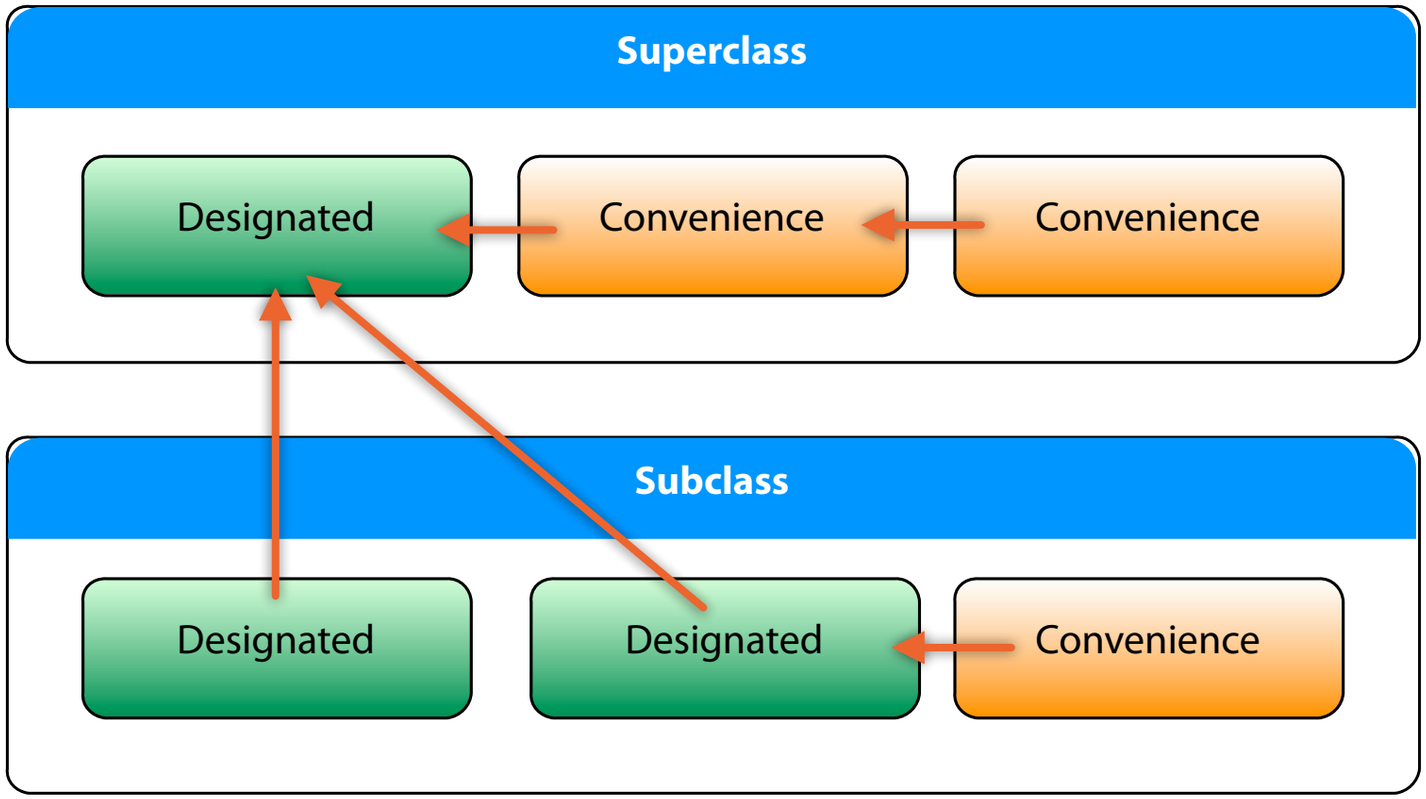
```
let not = NSNotification(name: "Howdy", object: nil)
```

Multiple Initializers

```
let not = NSNotification(name: "Howdy", object: self)
convenience init(name aName: String, object anObject: AnyObject?)
```

```
let not = NSNotification(name: "Howdy", object: self, userInfo: ["foo": "bar"])
init(name: String, object: AnyObject?, userInfo: [NSObject : AnyObject]?)
```

Subclassing & Initializers



Designated initializers delegate ***up***

Convenience initializers delegate ***across***

Initialization Phases

Phase 1: Each property gets an initial value

The class that introduces it is responsible

Phase 2: Each class can customize properties

This also includes any inherited properties

Overriding Initializers

```
class Person {
   init(firstName: String, lastName: String) {
        ...
    }
}

class Employee: Person {
    override init(firstName: String, lastName: String) {
        ...
    }
}
```

De-initializers

```
deinit()
```

- Use `deinit()` syntax
- Called before deallocation
- Not required
- No overloading
- Classes only

@property

```
@interface ViewController ()
@property UILabel *label;
@property UITextField *textField;
@end

@interface ViewController : UIViewController {
    UILabel *_label;
    UITextField *_textField;
}
@end
```

```
@implementation ViewController

@synthesize label = _label;
@synthesize textField = _textField;

@end
```

```
class MyViewController: UIViewController {
    var label: UILabel
    var textField: UITextField
}
```

Stored vs. Computed

```
class Racecar {
    let length: Int
    var speed {
        get { return self.engine.speed }
        set { self.engine.speed = newValue }
    }
}
```

```
let car = Racecar()
car.length = 15
car.speed = 172
```

Type vs. Instance

```
struct CardDeck {
    static let count = 52
}
CardDeck.count // 52
```

```
class Person {
    class var peopleCount: Int
}
Person.peopleCount = 12
```

```
enum Suit {
    case Hearts
    case Diamonds
    case Spades
    case Clubs
    static let count = 4
}
Suit.count // 4
```

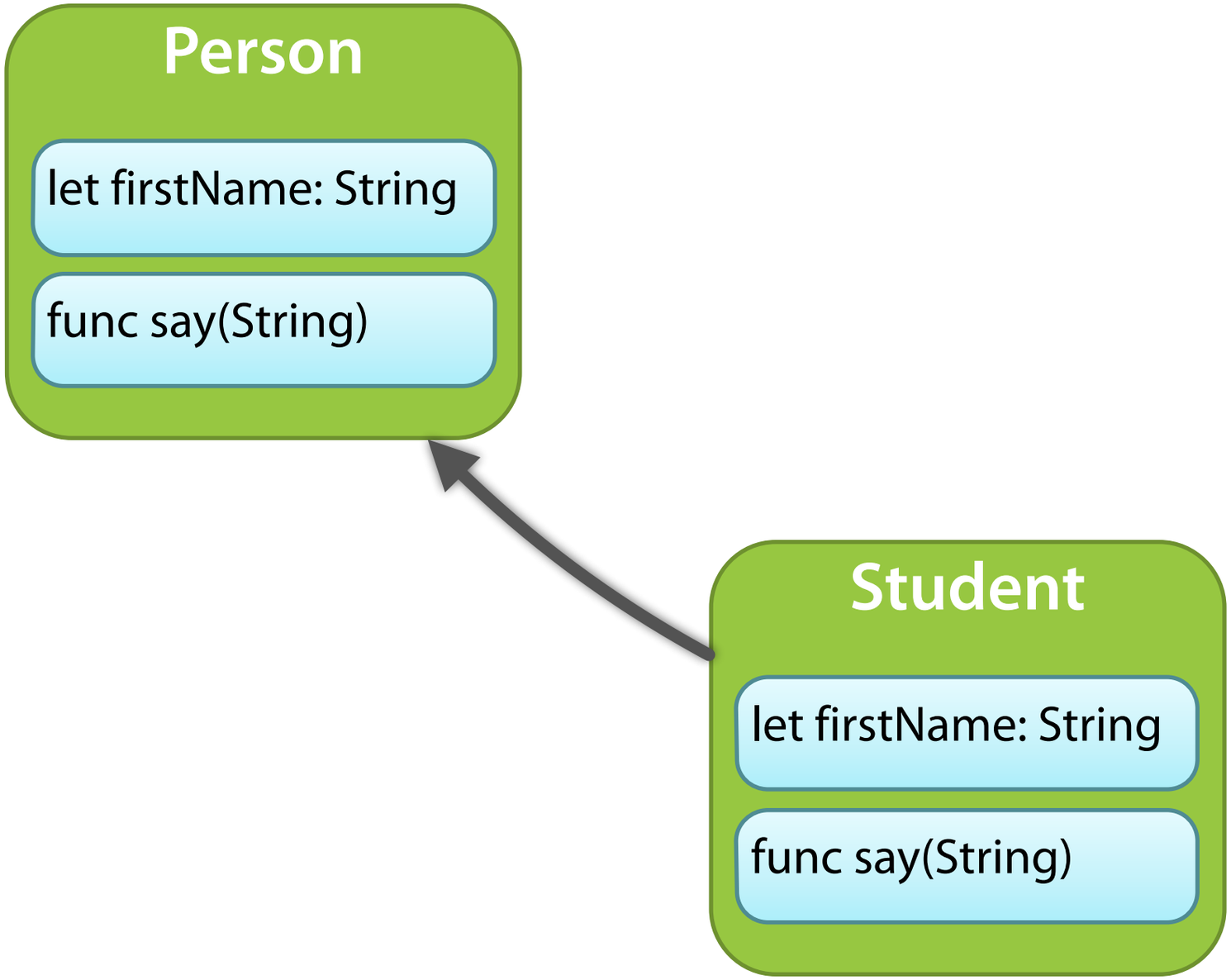
Property Observers

```
class Racecar {
    var speed {
        willSet {
            println("Speed changing to \(newValue)")
        }
        didSet {
            println("Speed changed from \(oldValue)")
        }
    }
}
```

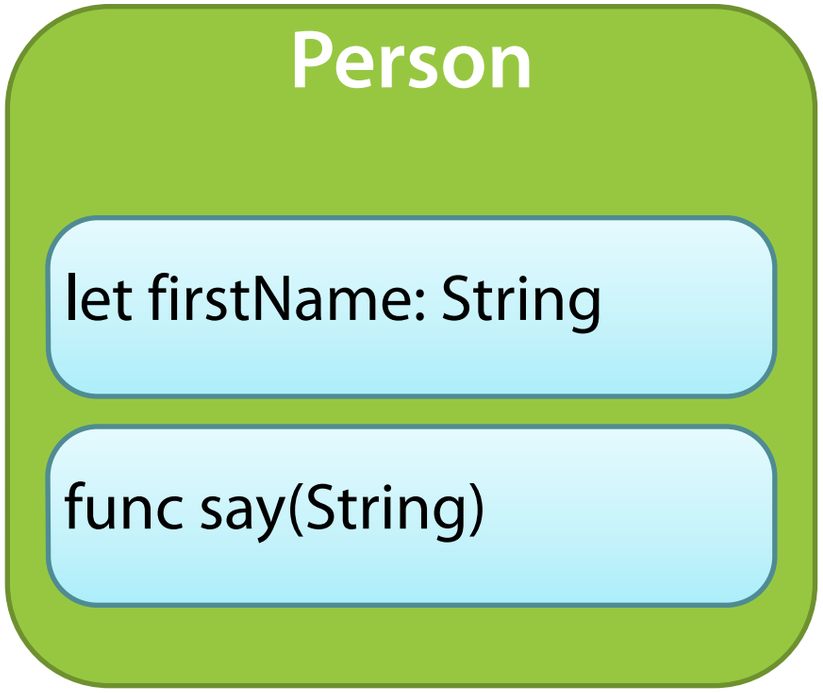
Lazy Properties

```
class Datastore {
    lazy var dbConnection: Connection = Connection()
}
```

Overriding


Person
let firstName: String
func say(String)
Student
let firstName: String
func say(String)

Overriding



Methods

Initializers

Computed Properties

`deinit` is automatic

`override`

`final`

```
class MPEG4Movie: Movie, MediaType {
    ...
}

struct Person: Named {
    ...
}

enum Suit: String, Named {
    ...
}
```

Polymorphic Operators

is checks for protocol conformance or type

For protocol conformance or super-class

as? conditionally downcasts

Like optional unwrapping

as forces downcasts

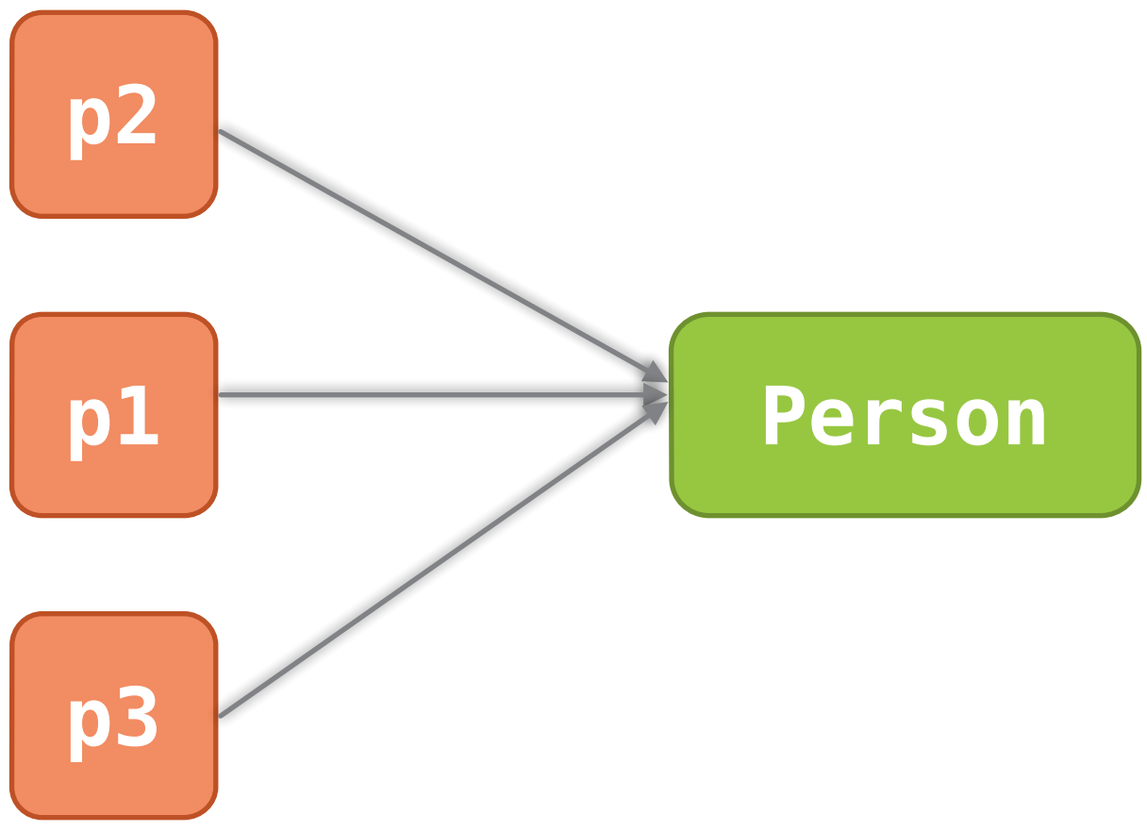
You better know what you're doing

Polymorphic Limitations

@objc required for “is” check

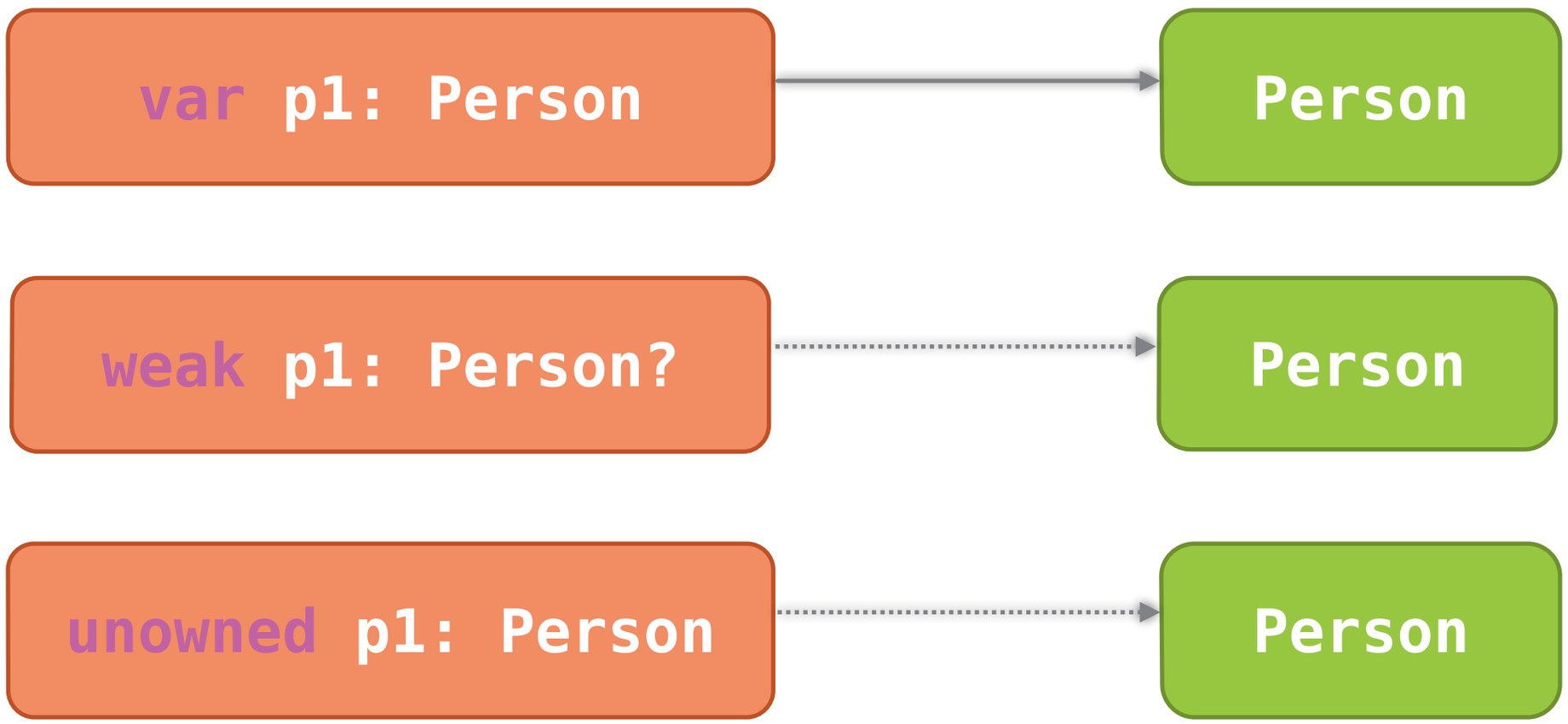
~~Enumerations~~

~~Structures~~

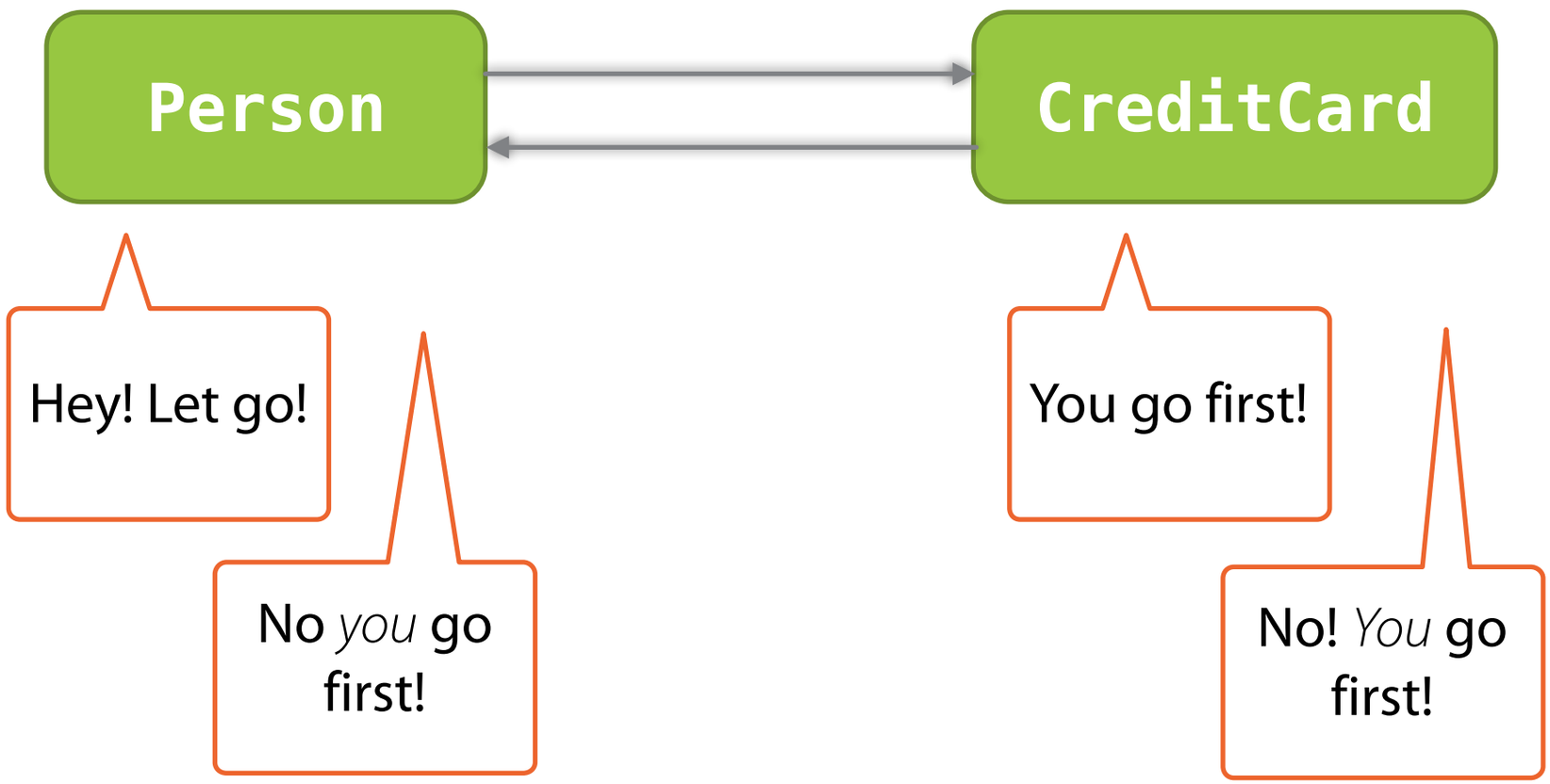

p2
p1
p3
Person

Automated Reference Counting

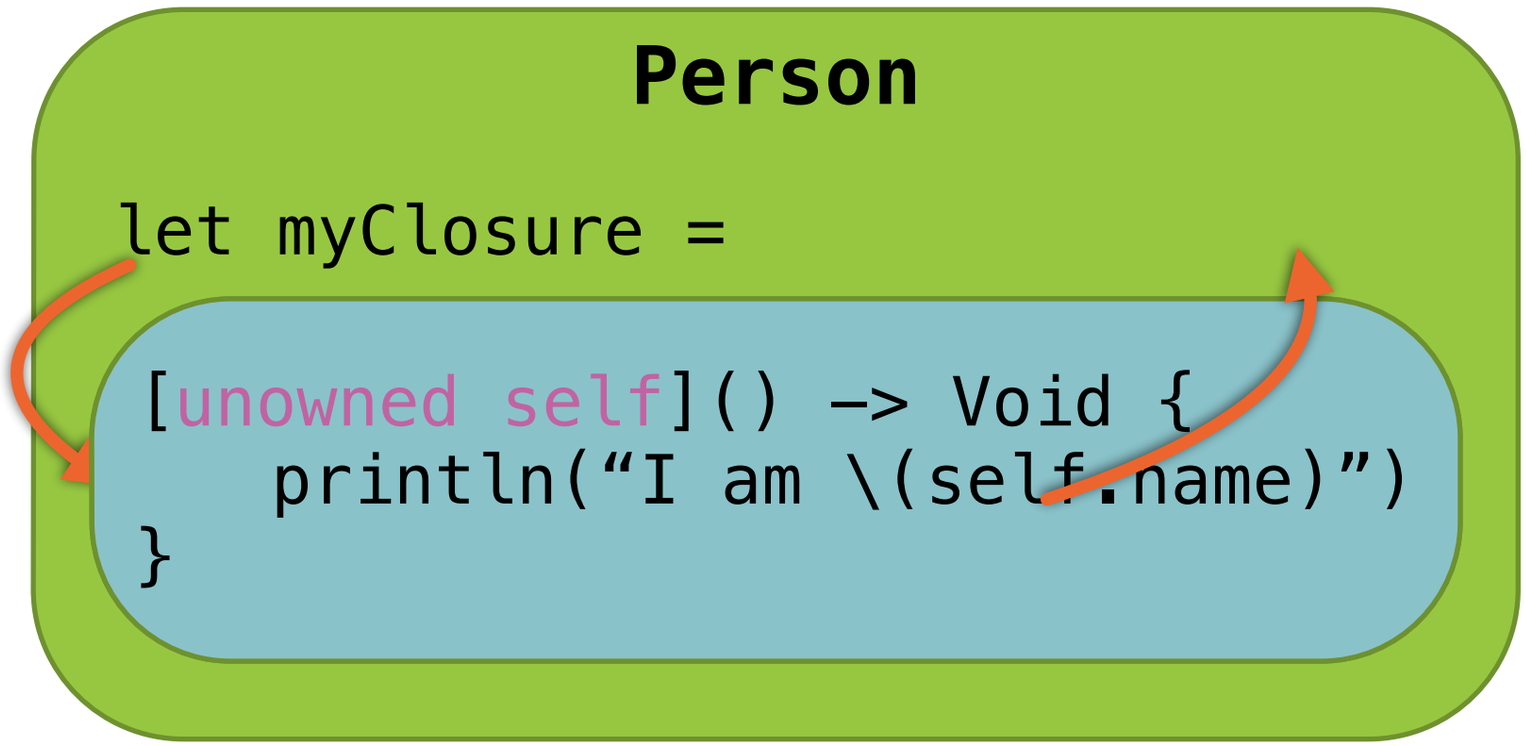
Reference Types



Retain-Cycles



Retain-Cycles and Closures



```
let names = ["Larry", "Moe", "Curly"]

let ages = [32, 35, 38]

let colors = [
    UIColor.redColor(),
    UIColor.whiteColor(),
    UIColor.greenColor()
]
```

```
struct Array<T> { ... }
```

Type Inference

```
let names = ["Larry", "Moe", "Curly"]
let bigMoe = names[1].uppercaseString
```

```
class IntArray: Array {
}

class StringArray: Array {
}

class PersonArray: Array {
}

class FloatArray: Array {
}

class BooleanArray: Array {
}
```

Type Parameters

```
struct Array<T> : MutableCollectionType, Sliceable {
    var first: T? { get }
    var last: T? { get }

    func filter(includeElement: (T) -> Bool) -> [T]
}
```

```
func map<U>(transform: (T) -> U) -> [U]
```

Type Constraints

```
struct Dictionary<Key : Hashable, Value> {
    }
func allItemsMatch<C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {
    return true
}
```

Review

Single inheritance model	Properties	Methods
Extensions	Polymorphism through Protocols	Reference types

Initializers

`init()`

Use special `init()` syntax

Setup property values

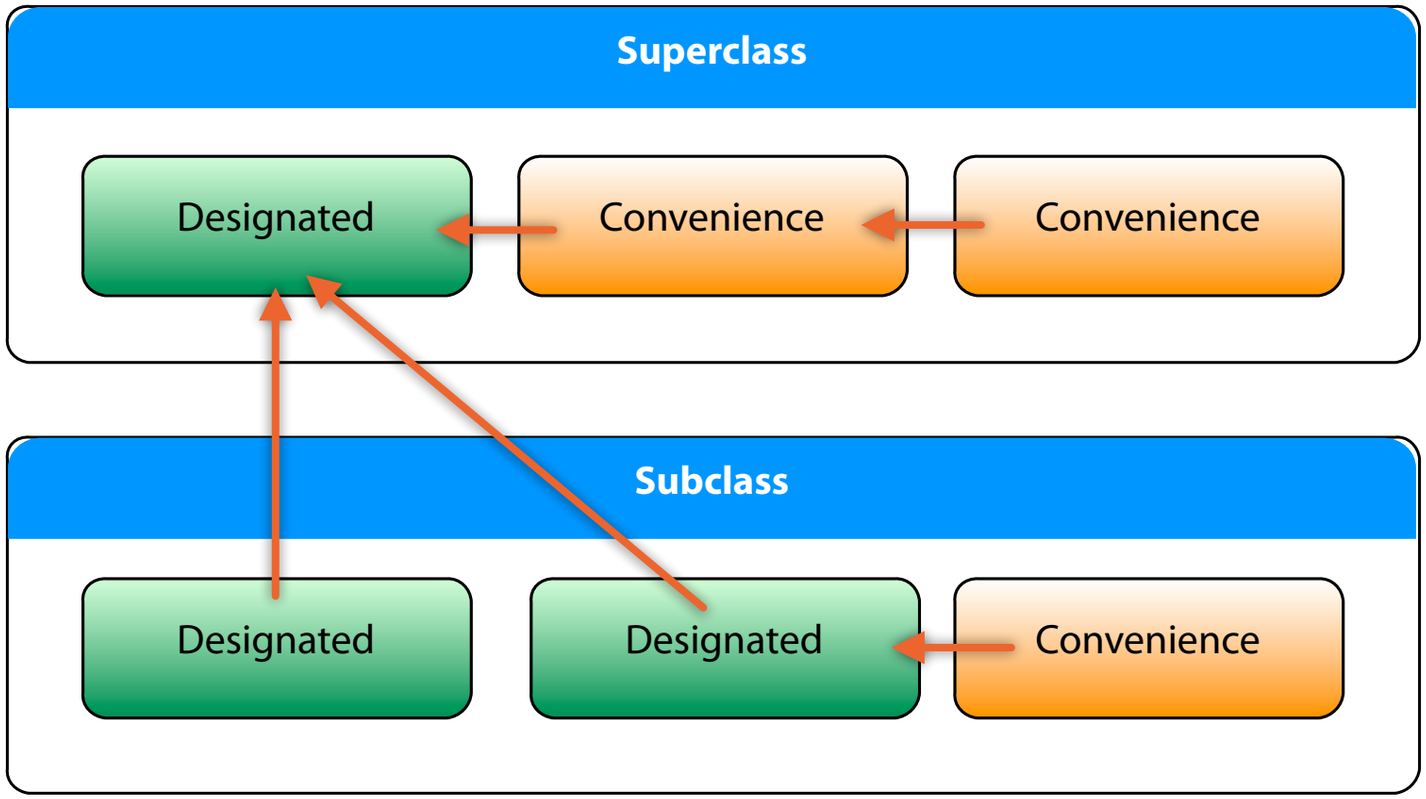
Prepare instance for use

De-initializers

```
deinit()
```

- Use `deinit()` syntax
- Called before deallocation
- Not required
- No overloading
- Classes only

Subclassing & Initializers



Designated initializers delegate ***up***

Convenience initializers delegate ***across***

Stored vs. Computed

```
class Racecar {
    let length: Int
    var speed {
        get { return self.engine.speed }
        set { self.engine.speed = newValue }
    }
}
```

Property Observers

```
class Racecar {
    var speed {
        willSet {
            println("Speed changing to \(newValue)")
        }
        didSet {
            println("Speed changed from \(oldValue)")
        }
    }
}
```

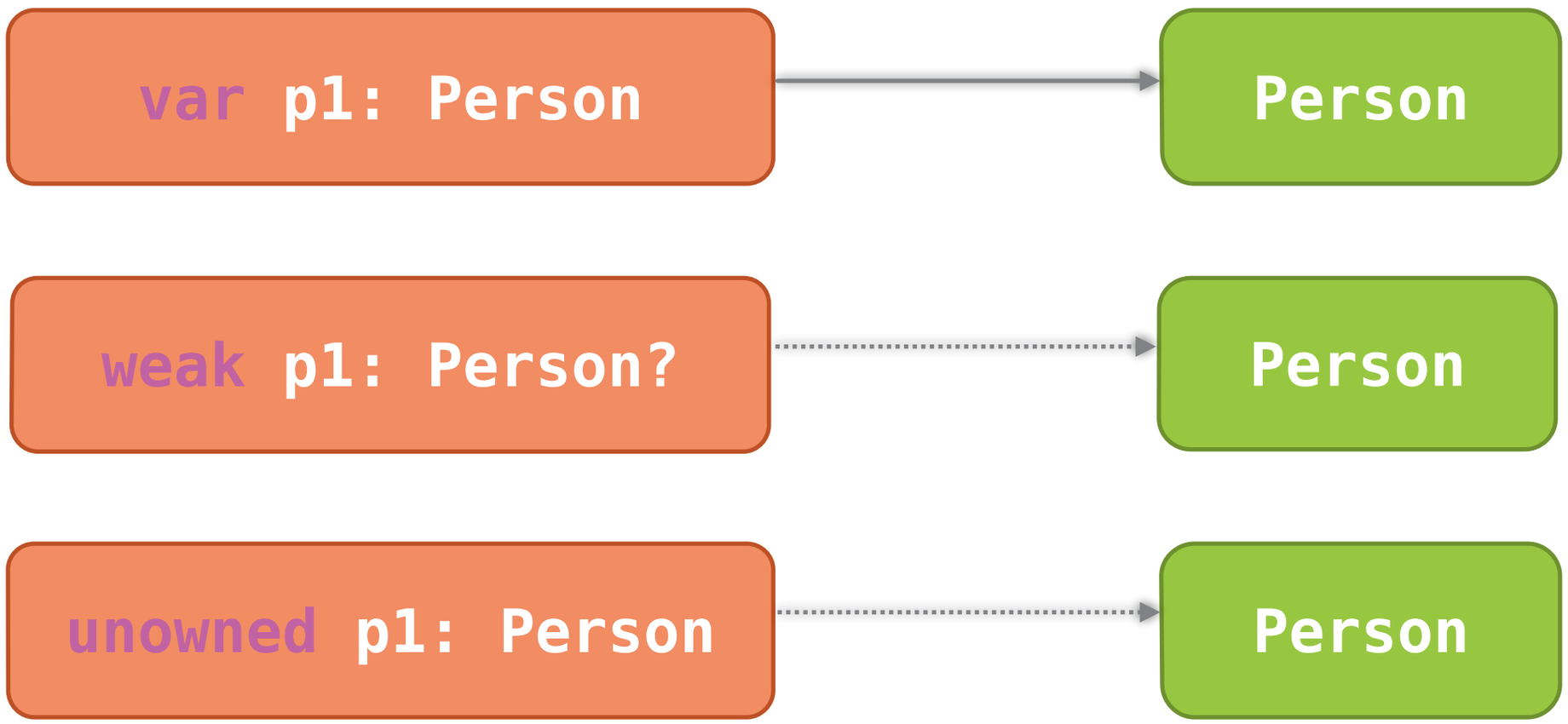
Polymorphism

```
class MPEG4Movie: Movie, MediaType {
    ...
}

struct Person: Named {
    ...
}

enum Suit: String, Named {
    ...
}
```

Reference Types



Generics

```
struct Array<T> { ... }
```