CSC 471 / 371 Mobile Application Development for iOS

Prof. Xiaoping Jia
School of Computing, CDM
DePaul University
xjia@cdm.depaul.edu
@DePaulSWEng

A Swift Primer, Part 3
Class Inheritance

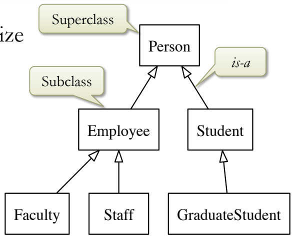
Outline

- Inheritance relation
- Dynamic typing
- Sub-classing
- Stored and computed properties
- Value types vs. reference types

The Inheritance Relation

Class Inheritance

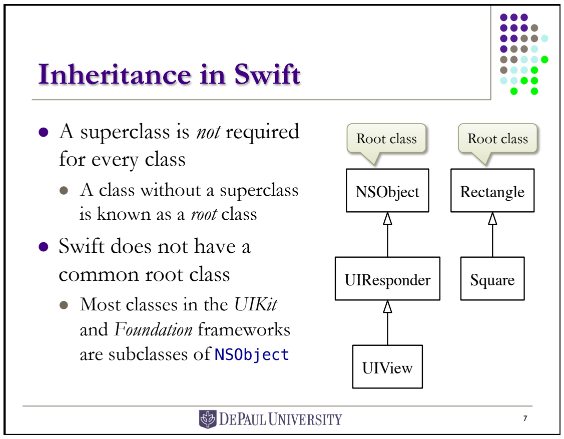
- A mechanism to organize classes based on their commonalities
  - Superclass and subclass
    - Code reuse
    - Customize or extend behavior
  - Subtype relation
    - The *is-a* relation
    - Liskov substitution principle



Subclass and Superclass

- A subclass represents a subtype of the superclass
  - Instances of the subclass is compatible with the superclass
  - **Liskov substitution principle**:

    An instance of the subclass can be substituted for an instance of the superclass
- A subclass can reuse the methods and properties in its superclass
- A subclass can extend the functionality of its superclass
  - adding new properties, and new methods
  - *overriding* existing methods

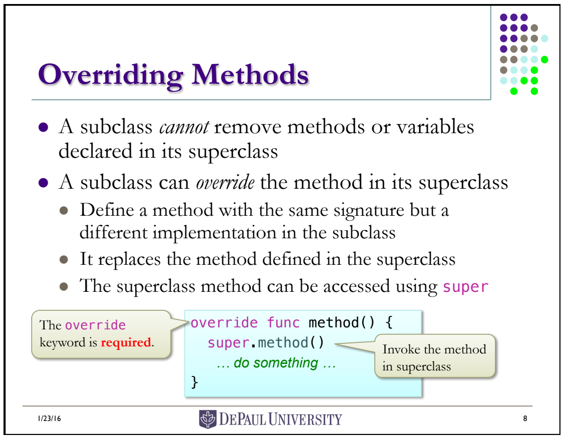


Inheritance in Swift

- A superclass is *not* required for every class
  - A class without a superclass is known as a *root* class
- Swift does not have a common root class
  - Most classes in the *UIKit* and *Foundation* frameworks are subclasses of `NSObject`

Root class
Root class
NSObject
Rectangle
UIResponder
Square
UIView

DePaul University 7



Overriding Methods

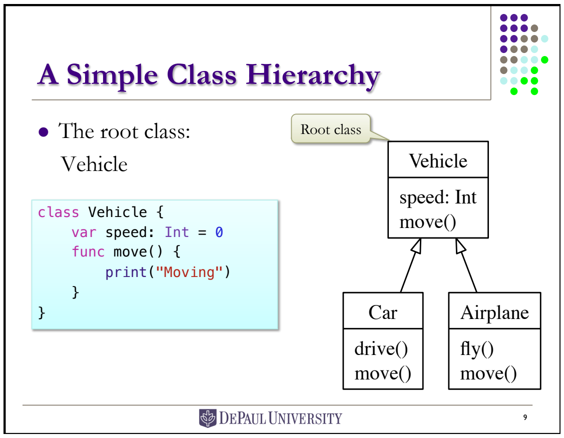
- A subclass *cannot* remove methods or variables declared in its superclass
- A subclass can *override* the method in its superclass
  - Define a method with the same signature but a different implementation in the subclass
  - It replaces the method defined in the superclass
  - The superclass method can be accessed using `super`

The `override` keyword is **required**.

```
override func method() {
   super.method()
    … do something …
}
```

Invoke the method in superclass

1/23/16 DePaul University 8



A Simple Class Hierarchy

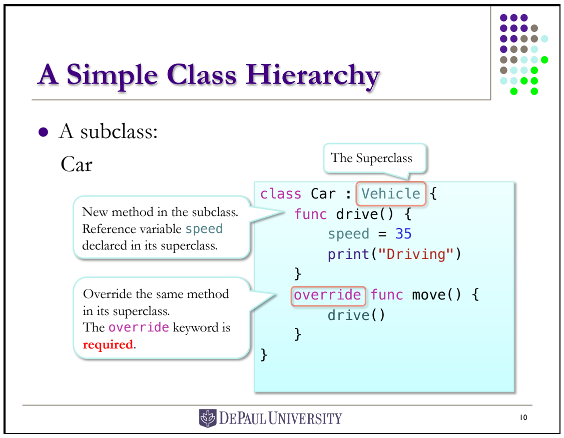
- The root class:

  Vehicle

```
class Vehicle {
    var speed: Int = 0
    func move() {
        print("Moving")
    }
}
```

Root class
Vehicle
speed: Int
move()
Car
drive()
move()
Airplane
fly()
move()

DePaul University 9



A Simple Class Hierarchy

- A subclass:

  Car

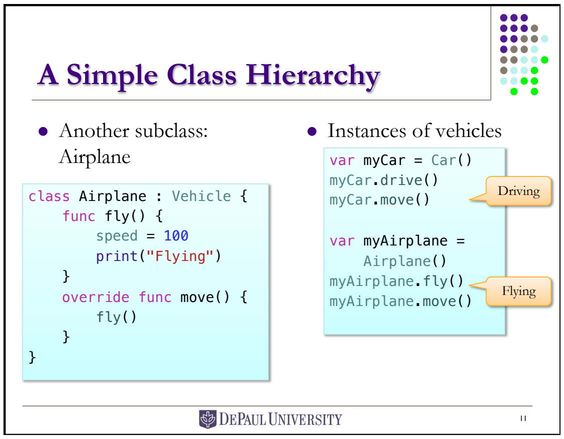
The Superclass

```
class Car : Vehicle {
    func drive() {
        speed = 35
        print("Driving")
    }
    override func move() {
        drive()
    }
}
```

New method in the subclass. Reference variable `speed` declared in its superclass.

Override the same method in its superclass. The `override` keyword is **required**.

DePaul University 10



A Simple Class Hierarchy

- Another subclass: Airplane

```
class Airplane : Vehicle {
    func fly() {
        speed = 100
        print("Flying")
    }
    override func move() {
        fly()
    }
}
```

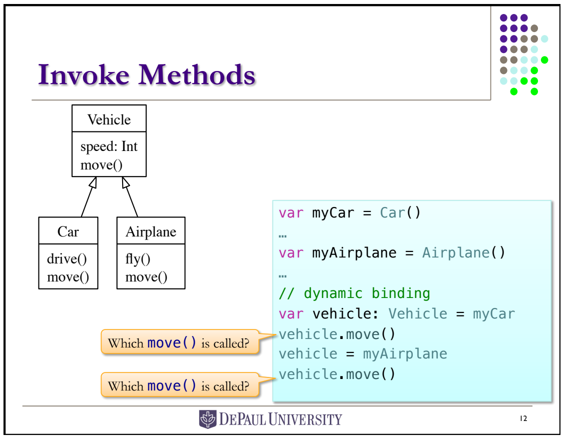
- Instances of vehicles

```
var myCar = Car()
myCar.drive()
myCar.move()

var myAirplane =
    Airplane()
myAirplane.fly()
myAirplane.move()
```

Driving

Flying

DePaul University 11



Invoke Methods

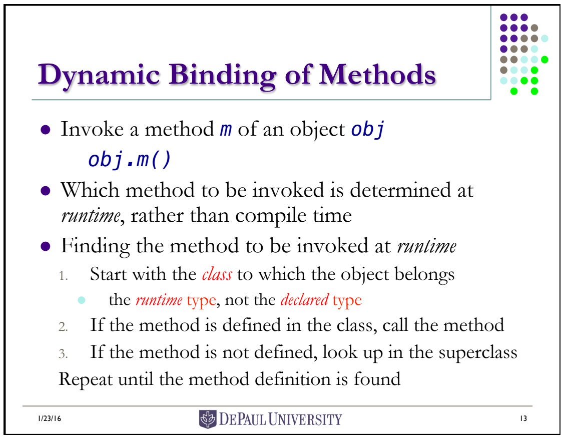
Vehicle
speed: Int
move()
Car
drive()
move()
Airplane
fly()
move()

```
var myCar = Car()
…
var myAirplane = Airplane()
…
// dynamic binding
var vehicle: Vehicle = myCar
vehicle.move()
vehicle = myAirplane
vehicle.move()
```

Which `move()` is called?

Which `move()` is called?

DePaul University 12

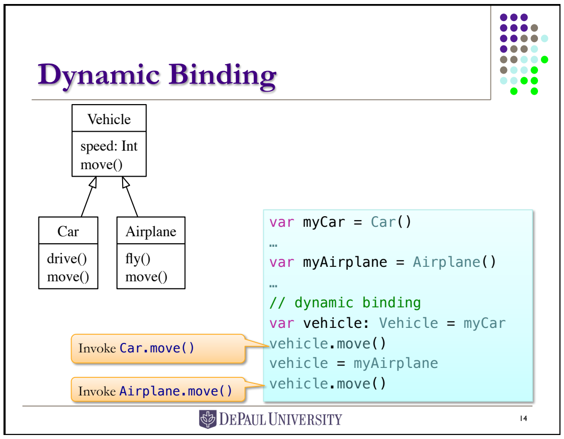


Dynamic Binding of Methods

- Invoke a method *m* of an object *obj*
  `obj.m()`
- Which method to be invoked is determined at *runtime*, rather than compile time
- Finding the method to be invoked at *runtime*
  1. Start with the *class* to which the object belongs
     - the *runtime* type, not the *declared* type
  2. If the method is defined in the class, call the method
  3. If the method is not defined, look up in the superclass

  Repeat until the method definition is found

1/23/16 DePaul University 13



Dynamic Binding

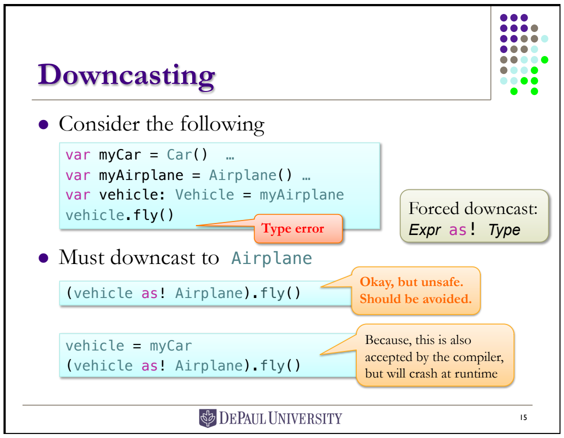
Vehicle
speed: Int
move()
Car
drive()
move()
Airplane
fly()
move()

```
var myCar = Car()
…
var myAirplane = Airplane()
…
// dynamic binding
var vehicle: Vehicle = myCar
vehicle.move()
vehicle = myAirplane
vehicle.move()
```

Invoke `Car.move()`

Invoke `Airplane.move()`

DePaul University 14



Downcasting

- Consider the following

```
var myCar = Car()  …
var myAirplane = Airplane() …
var vehicle: Vehicle = myAirplane
vehicle.fly()
```

Type error

Forced downcast: *Expr* `as!` *Type*

- Must downcast to `Airplane`

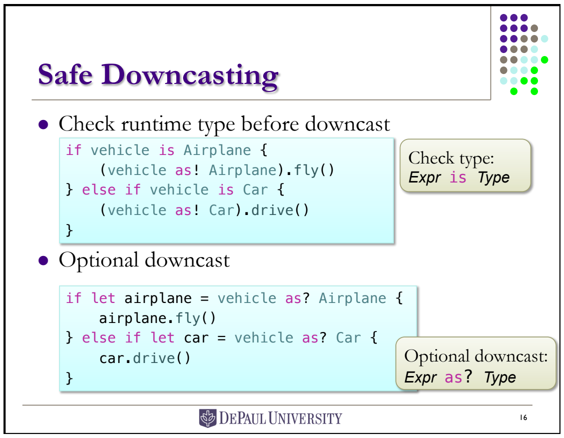
```
(vehicle as! Airplane).fly()
```

Okay, but unsafe. Should be avoided.

```
vehicle = myCar
(vehicle as! Airplane).fly()
```

Because, this is also accepted by the compiler, but will crash at runtime

DePaul University 15



Safe Downcasting

- Check runtime type before downcast

```
if vehicle is Airplane {
    (vehicle as! Airplane).fly()
} else if vehicle is Car {
    (vehicle as! Car).drive()
}
```

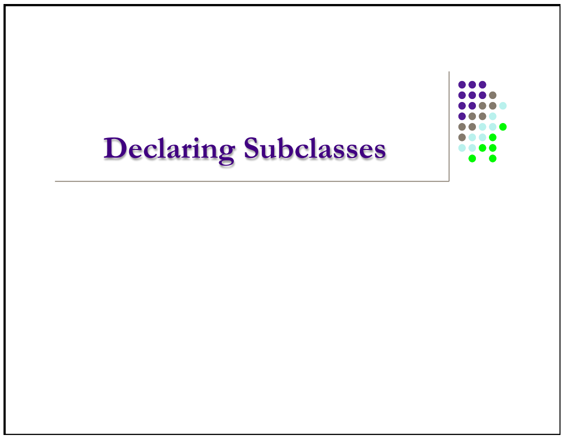
Check type: *Expr* `is` *Type*

- Optional downcast

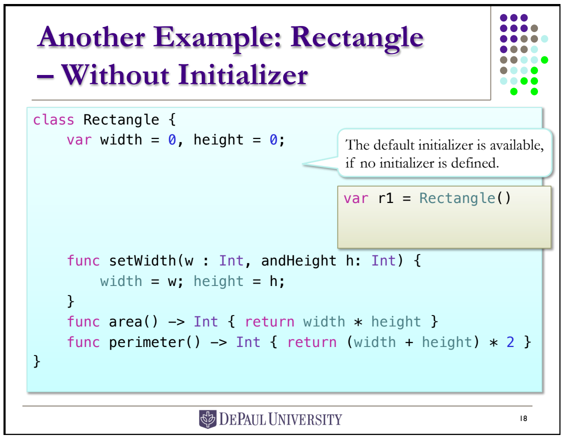
```
if let airplane = vehicle as? Airplane {
    airplane.fly()
} else if let car = vehicle as? Car {
    car.drive()
}
```

Optional downcast: *Expr* `as?` *Type*

DePaul University 16



Declaring Subclasses



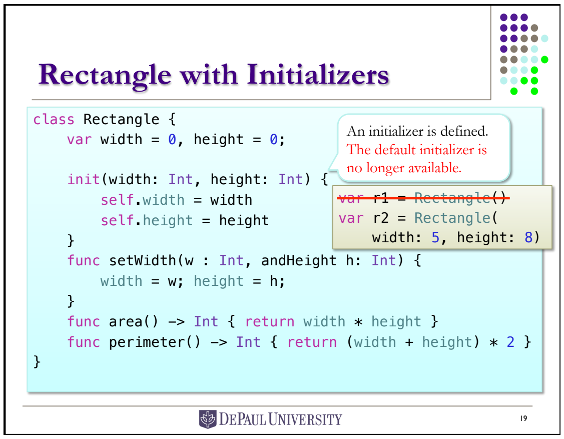
Another Example: Rectangle – Without Initializer

```
class Rectangle {
    var width = 0, height = 0;



    func setWidth(w : Int, andHeight h: Int) {
        width = w; height = h;
    }
    func area() -> Int { return width * height }
    func perimeter() -> Int { return (width + height) * 2 }
}
```

The default initializer is available, if no initializer is defined.

```
var r1 = Rectangle()
```

DePaul University 18

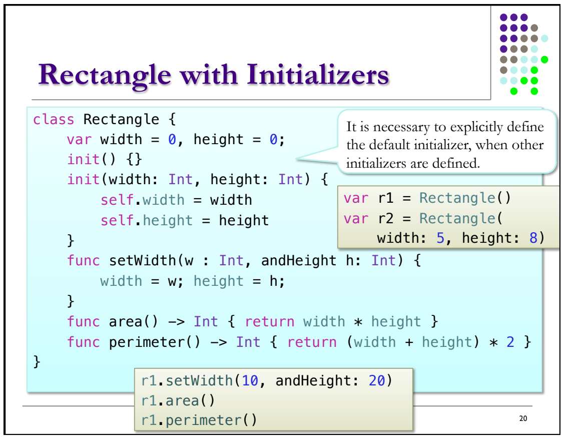


Rectangle with Initializers

```
class Rectangle {
    var width = 0, height = 0;

    init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }
    func setWidth(w : Int, andHeight h: Int) {
        width = w; height = h;
    }
    func area() -> Int { return width * height }
    func perimeter() -> Int { return (width + height) * 2 }
}
```

An initializer is defined. The default initializer is no longer available.

~~var r1 = Rectangle()~~
var r2 = Rectangle(
    width: 5, height: 8)



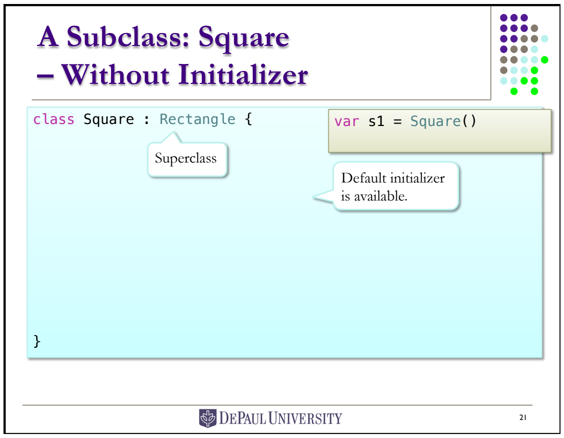
Rectangle with Initializers

```
class Rectangle {
    var width = 0, height = 0;
    init() {}
    init(width: Int, height: Int) {
        self.width = width
        self.height = height
    }
    func setWidth(w : Int, andHeight h: Int) {
        width = w; height = h;
    }
    func area() -> Int { return width * height }
    func perimeter() -> Int { return (width + height) * 2 }
}
```

It is necessary to explicitly define the default initializer, when other initializers are defined.

```
var r1 = Rectangle()
var r2 = Rectangle(
    width: 5, height: 8)
```

```
r1.setWidth(10, andHeight: 20)
r1.area()
r1.perimeter()
```



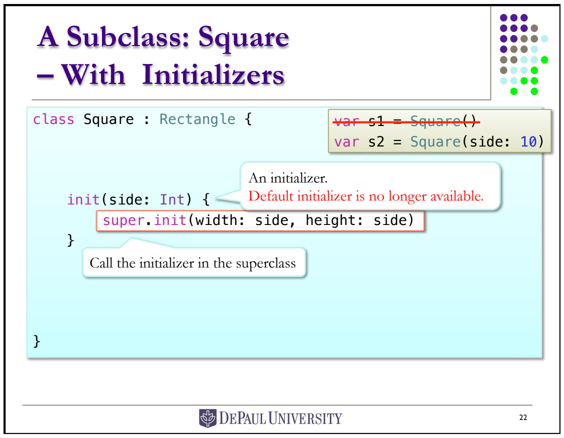
A Subclass: Square – Without Initializer

```
class Square : Rectangle {

}
```

Superclass

```
var s1 = Square()
```

Default initializer is available.



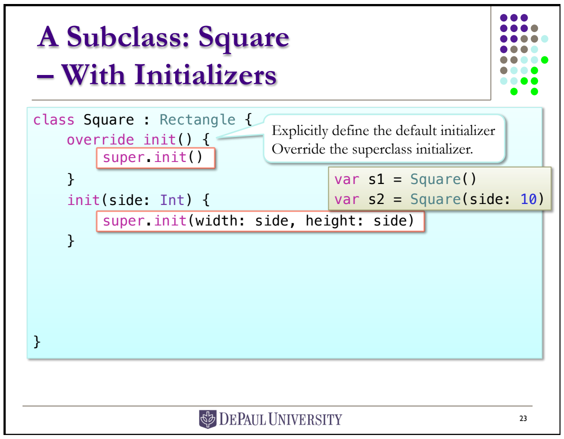
A Subclass: Square – With Initializers

```
class Square : Rectangle {


    init(side: Int) {
        super.init(width: side, height: side)
    }

}
```

~~var s1 = Square()~~
var s2 = Square(side: 10)

An initializer. Default initializer is no longer available.

Call the initializer in the superclass

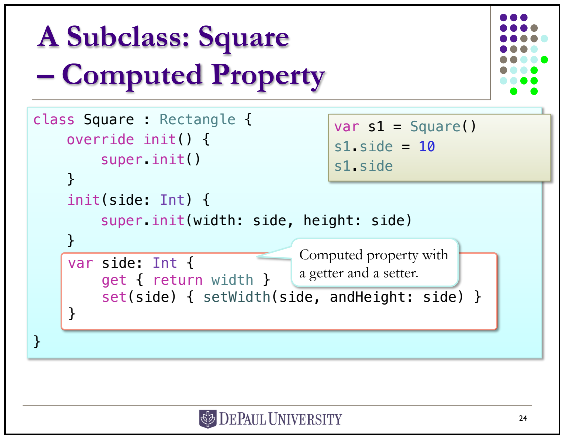


A Subclass: Square – With Initializers

```
class Square : Rectangle {
    override init() {
        super.init()
    }
    init(side: Int) {
        super.init(width: side, height: side)
    }

}
```

Explicitly define the default initializer Override the superclass initializer.

```
var s1 = Square()
var s2 = Square(side: 10)
```

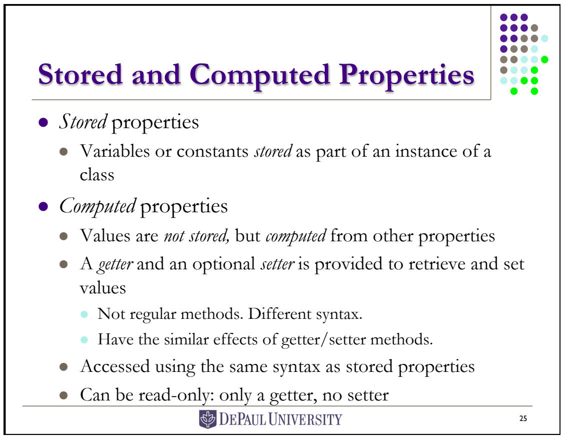


A Subclass: Square – Computed Property

```
class Square : Rectangle {
    override init() {
        super.init()
    }
    init(side: Int) {
        super.init(width: side, height: side)
    }
    var side: Int {
        get { return width }
        set(side) { setWidth(side, andHeight: side) }
    }
}
```

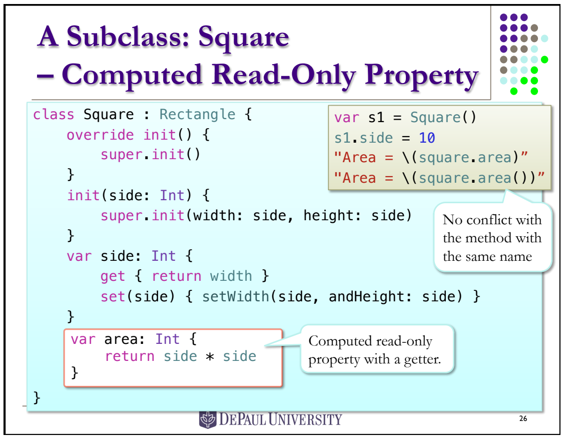
```
var s1 = Square()
s1.side = 10
s1.side
```

Computed property with a getter and a setter.



Stored and Computed Properties

- *Stored* properties
  - Variables or constants *stored* as part of an instance of a class
- *Computed* properties
  - Values are *not stored,* but *computed* from other properties
  - A *getter* and an optional *setter* is provided to retrieve and set values
    - Not regular methods. Different syntax.
    - Have the similar effects of getter/setter methods.
  - Accessed using the same syntax as stored properties
  - Can be read-only: only a getter, no setter



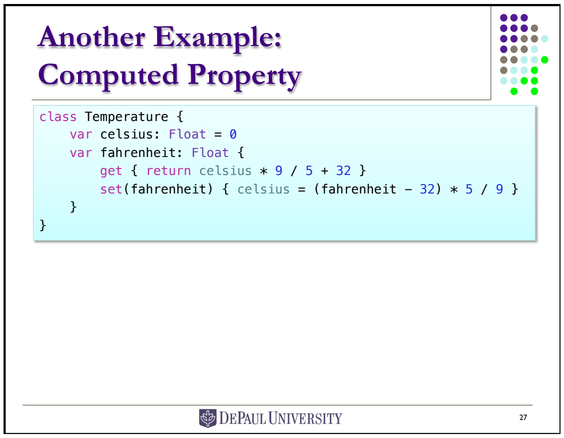
A Subclass: Square – Computed Read-Only Property

```
class Square : Rectangle {
    override init() {
        super.init()
    }
    init(side: Int) {
        super.init(width: side, height: side)
    }
    var side: Int {
        get { return width }
        set(side) { setWidth(side, andHeight: side) }
    }
    var area: Int {
        return side * side
    }
}
```

```
var s1 = Square()
s1.side = 10
"Area = \(square.area)"
"Area = \(square.area())"
```

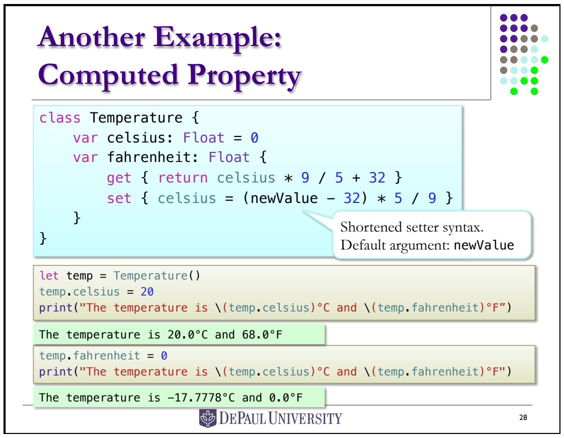
No conflict with the method with the same name

Computed read-only property with a getter.



Another Example: Computed Property

```
class Temperature {
    var celsius: Float = 0
    var fahrenheit: Float {
        get { return celsius * 9 / 5 + 32 }
        set(fahrenheit) { celsius = (fahrenheit - 32) * 5 / 9 }
    }
}
```



Another Example: Computed Property

```
class Temperature {
    var celsius: Float = 0
    var fahrenheit: Float {
        get { return celsius * 9 / 5 + 32 }
        set { celsius = (newValue - 32) * 5 / 9 }
    }
}
```

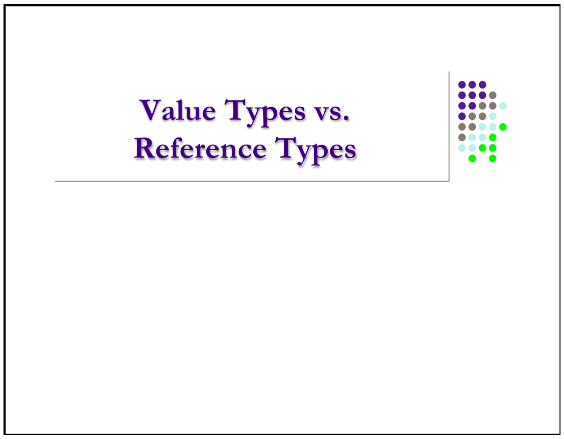
Shortened setter syntax.
Default argument: `newValue`

```
let temp = Temperature()
temp.celsius = 20
print("The temperature is \(temp.celsius)°C and \(temp.fahrenheit)°F”)
```

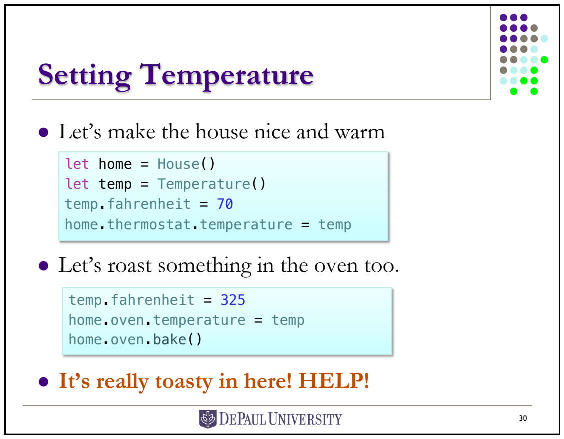
```
The temperature is 20.0°C and 68.0°F
```

```
temp.fahrenheit = 0
print("The temperature is \(temp.celsius)°C and \(temp.fahrenheit)°F")
```

```
The temperature is -17.7778°C and 0.0°F
```



Value Types vs. Reference Types



Setting Temperature

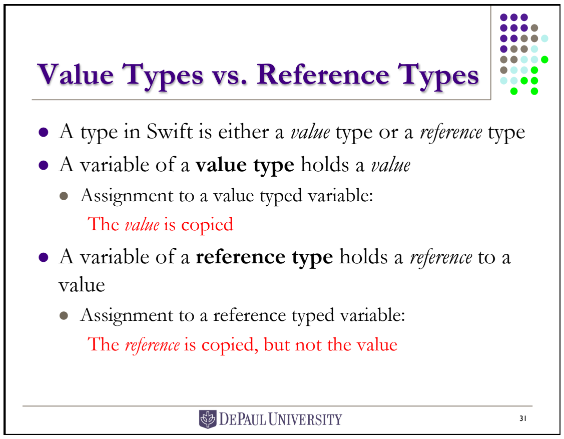
- Let’s make the house nice and warm

```
let home = House()
let temp = Temperature()
temp.fahrenheit = 70
home.thermostat.temperature = temp
```

- Let’s roast something in the oven too.

```
temp.fahrenheit = 325
home.oven.temperature = temp
home.oven.bake()
```

- **It’s really toasty in here! HELP!**

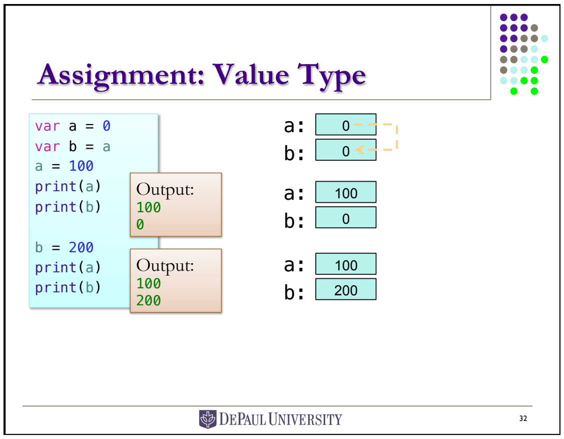


Value Types vs. Reference Types

- A type in Swift is either a *value* type or a *reference* type
- A variable of a **value type** holds a *value*
  - Assignment to a value typed variable:

    The *value* is copied
- A variable of a **reference type** holds a *reference* to a value
  - Assignment to a reference typed variable:

    The *reference* is copied, but not the value



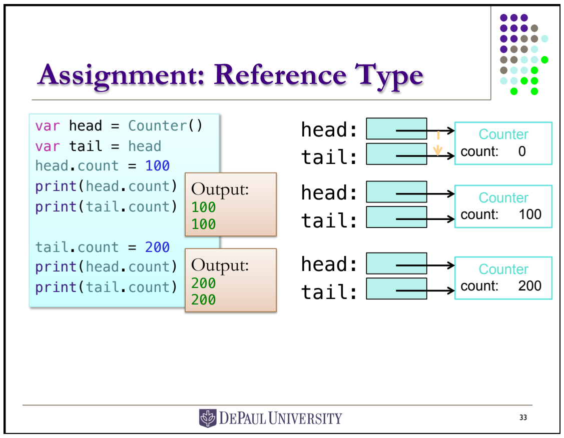
Assignment: Value Type

```
var a = 0
var b = a
a = 100
print(a)
print(b)
```

Output:
```
100
0
```

```
b = 200
print(a)
print(b)
```

Output:
```
100
200
```



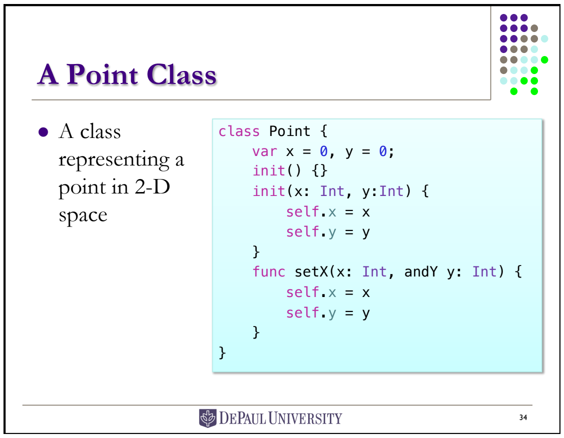
Assignment: Reference Type

```
var head = Counter()
var tail = head
head.count = 100
print(head.count)
print(tail.count)
```

Output:
```
100
100
```

```
tail.count = 200
print(head.count)
print(tail.count)
```

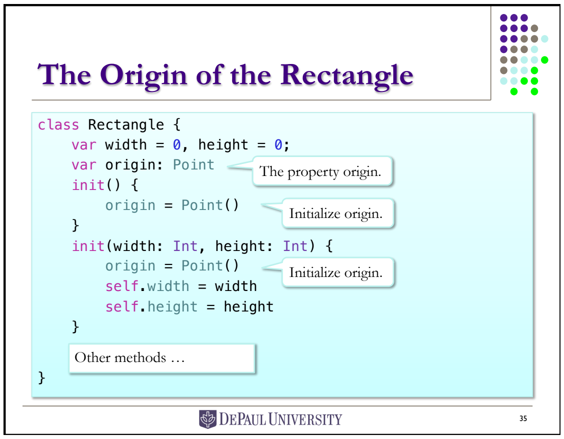
Output:
```
200
200
```



A Point Class

- A class representing a point in 2-D space

```
class Point {
    var x = 0, y = 0;
    init() {}
    init(x: Int, y:Int) {
        self.x = x
        self.y = y
    }
    func setX(x: Int, andY y: Int) {
        self.x = x
        self.y = y
    }
}
```



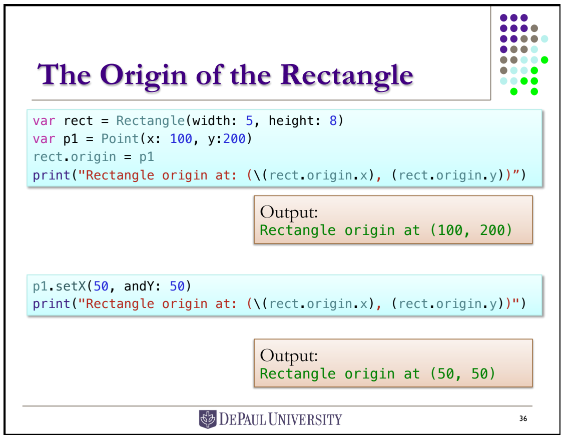
The Origin of the Rectangle

```
class Rectangle {
    var width = 0, height = 0;
    var origin: Point
    init() {
        origin = Point()
    }
    init(width: Int, height: Int) {
        origin = Point()
        self.width = width
        self.height = height
    }
    Other methods …
}
```

The property origin.

Initialize origin.

Initialize origin.



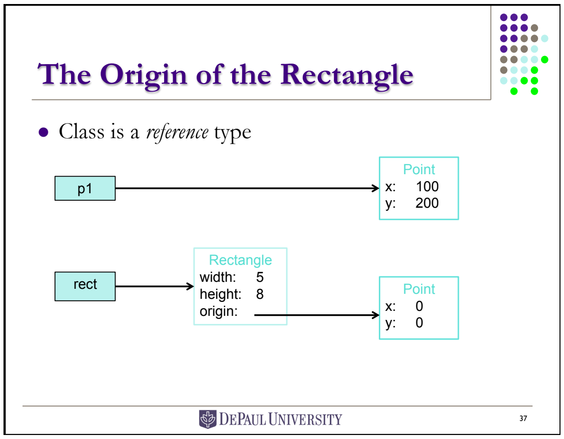
The Origin of the Rectangle

```
var rect = Rectangle(width: 5, height: 8)
var p1 = Point(x: 100, y:200)
rect.origin = p1
print("Rectangle origin at: (\(rect.origin.x), (rect.origin.y))”)
```

Output:
```
Rectangle origin at (100, 200)
```

```
p1.setX(50, andY: 50)
print("Rectangle origin at: (\(rect.origin.x), (rect.origin.y))")
```

Output:
```
Rectangle origin at (50, 50)
```

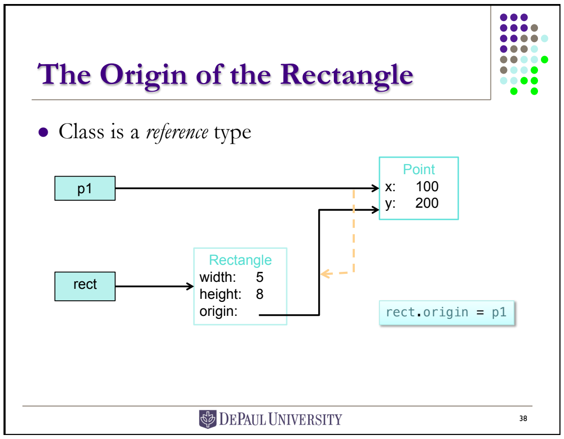


The Origin of the Rectangle

- Class is a *reference* type

p1 → Point x: 100 y: 200

rect → Rectangle width: 5 height: 8 origin: → Point x: 0 y: 0



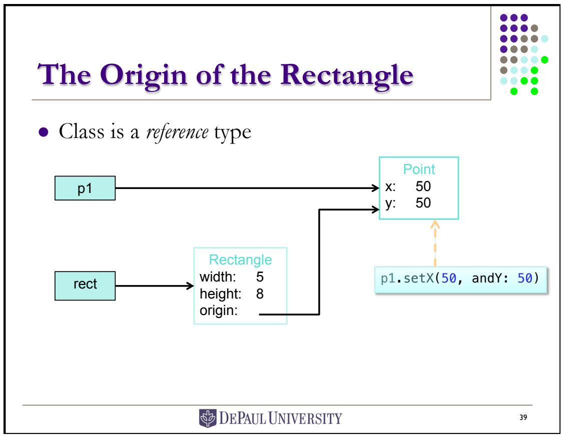
The Origin of the Rectangle

- Class is a *reference* type

p1 → Point x: 100 y: 200

rect → Rectangle width: 5 height: 8 origin: → Point x: 100 y: 200

`rect.origin = p1`



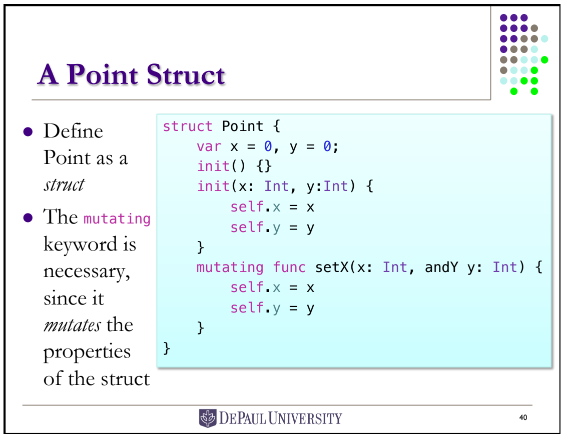
The Origin of the Rectangle

- Class is a *reference* type

p1 → Point x: 50 y: 50

rect → Rectangle width: 5 height: 8 origin: → Point x: 50 y: 50

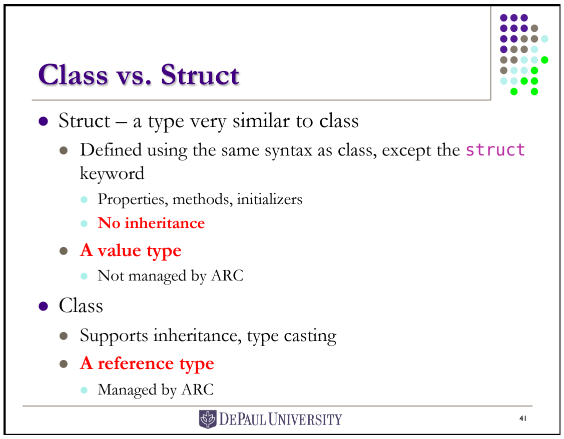
`p1.setX(50, andY: 50)`



A Point Struct

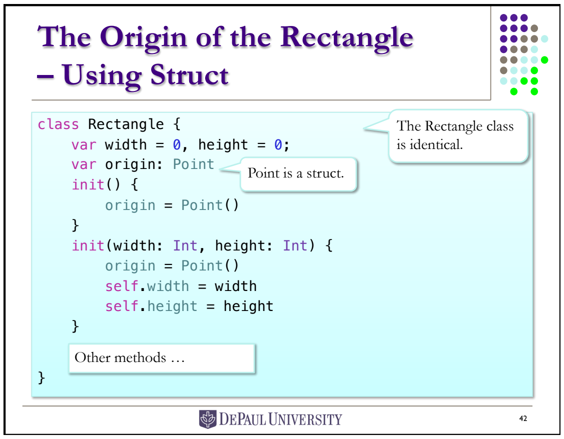
- Define Point as a *struct*
- The `mutating` keyword is necessary, since it *mutates* the properties of the struct

```
struct Point {
    var x = 0, y = 0;
    init() {}
    init(x: Int, y:Int) {
        self.x = x
        self.y = y
    }
    mutating func setX(x: Int, andY y: Int) {
        self.x = x
        self.y = y
    }
}
```



Class vs. Struct

- Struct – a type very similar to class
  - Defined using the same syntax as class, except the `struct` keyword
    - Properties, methods, initializers
    - No inheritance
  - A value type
    - Not managed by ARC
- Class
  - Supports inheritance, type casting
  - A reference type
    - Managed by ARC

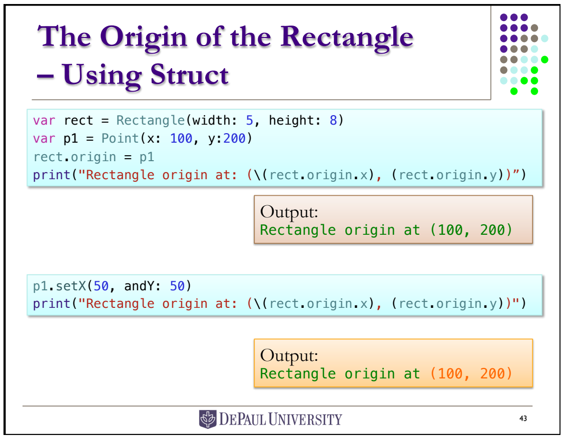


The Origin of the Rectangle – Using Struct

```
class Rectangle {
    var width = 0, height = 0;
    var origin: Point
    init() {
        origin = Point()
    }
    init(width: Int, height: Int) {
        origin = Point()
        self.width = width
        self.height = height
    }
    Other methods …
}
```

The Rectangle class is identical.

Point is a struct.



The Origin of the Rectangle – Using Struct

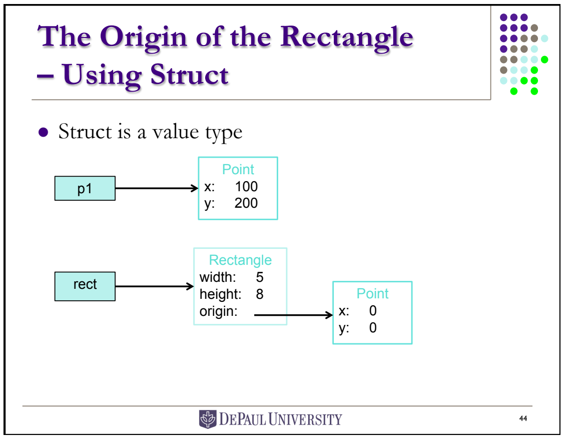
```
var rect = Rectangle(width: 5, height: 8)
var p1 = Point(x: 100, y:200)
rect.origin = p1
print("Rectangle origin at: (\(rect.origin.x), (rect.origin.y))”)
```

Output:
```
Rectangle origin at (100, 200)
```

```
p1.setX(50, andY: 50)
print("Rectangle origin at: (\(rect.origin.x), (rect.origin.y))")
```

Output:
```
Rectangle origin at (100, 200)
```

DePaul University 43



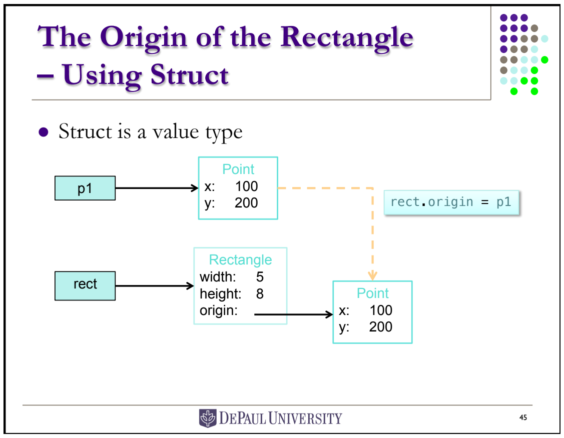
The Origin of the Rectangle – Using Struct

- Struct is a value type

p1 → Point x: 100 y: 200

rect → Rectangle width: 5 height: 8 origin: → Point x: 0 y: 0

DePaul University 44



The Origin of the Rectangle – Using Struct

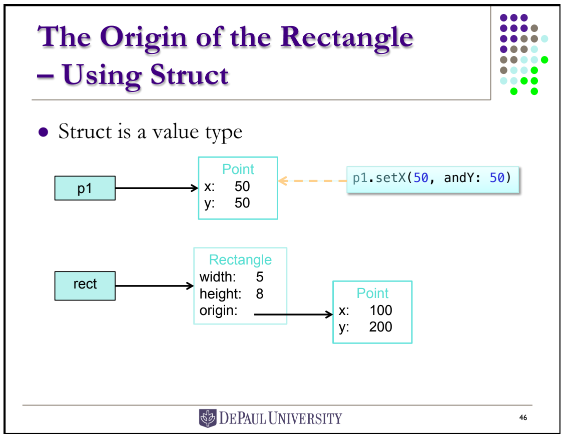
- Struct is a value type

p1 → Point x: 100 y: 200

rect.origin = p1

rect → Rectangle width: 5 height: 8 origin: → Point x: 100 y: 200

DePaul University 45



The Origin of the Rectangle – Using Struct

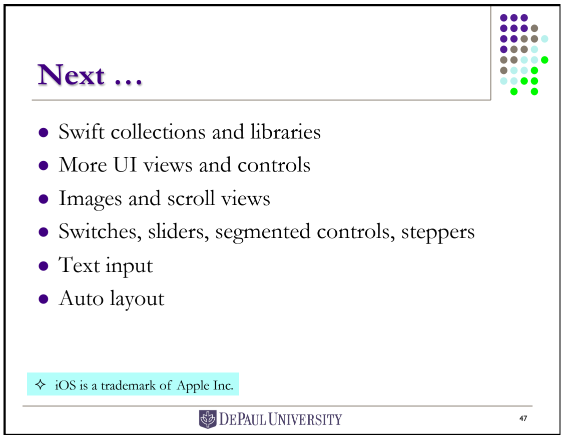
- Struct is a value type

p1 → Point x: 50 y: 50

p1.setX(50, andY: 50)

rect → Rectangle width: 5 height: 8 origin: → Point x: 100 y: 200

DePaul University 46



Next …

- Swift collections and libraries
- More UI views and controls
- Images and scroll views
- Switches, sliders, segmented controls, steppers
- Text input
- Auto layout

✧ iOS is a trademark of Apple Inc.

DePaul University 47