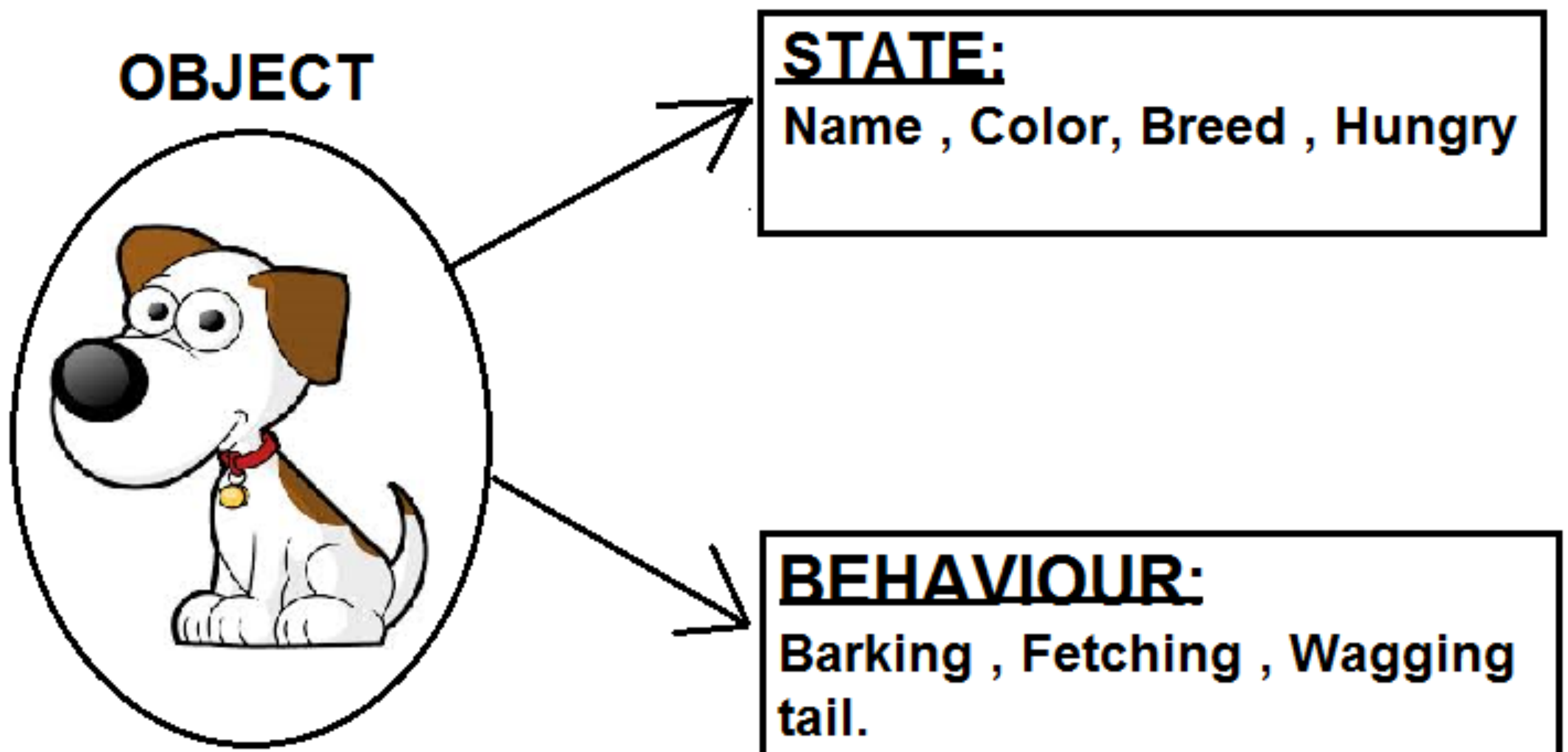

(infinitespeak.wordpress.com)

Classes and Structs

Dr. Sarah Abraham

University of Texas at Austin
CS329e
Spring 2020

Classes and Structures

- General-purpose, flexible constructs to build blocks of code
- Properties and methods add functionality
- Defining classes and structs in a file makes external interface automatically available in Swift

Classes vs Structs

- Both classes and structs define:
  - Properties, methods, initializers
- Classes allow for:
  - Inheritance
  - Type-casting to check type of class at runtime
  - Deinitialization and reference counting
- Structs passed by value, classes passed by reference

Defining a Struct

```
struct Point {

  var x = 0.0

  var y = 0.0

}

//Create a struct instance and change its x value

var p1 = Point()

p1.x = 10.0
```

Defining a Class

```
class Person {
  var firstName:String
  var lastName:String
  func description() -> String {
    return "\(lastName), \(firstName)"
  }
}
```

Creating an Instance

- Each instance has its own memory and set of properties:

```
let p1 = Person()
```

- Instances can call on instance methods

```
p1.description()
```

- Method is called on the instance itself

Initializers

- Automatically called after memory is allocated
- Creates an object with a good starting state

```
init() {
  self.firstName = “Unknown”
  self.lastName = “Unknown”
}
```

- If no `init()` is provided, will auto-generate a default `init()` with an empty method body

Initializing Property Values

- Can provide default values for properties:

```
var firstName: String = “Unknown”

var lastName: String = “Unknown”

init() {}
```

- Can *overload* initializers to determine property values

```
init(firstName: String, lastName: String) {

   self.firstName = firstName

   self.lastName = lastName

 }
```

Designated Initializer

- Main initializer used for a class
- All other initializers funnel through this initializer
- Ensures initialization occurs through superclass chain
  - Must call designated initializer from its superclass if it has one
- Set all properties of class while letting user send in customized values

Convenience Initializers

- Secondary, supporting initializers for a class
- Must call another initializer from the same class
- Must ultimately call the designated initializer
- `init` method is prefixed with `convenience`

Convenience Initializer Example

```
//Designated initializer
init(firstName: String, lastName: String) {
   self.firstName = firstName
   self.lastName = lastName
}
//Convenience initializer
convenience init() {
   self.init(firstName: “Lev”, lastName: “Tolstoy”)
}
```

Why Use a Designated Initializer?

- Initializers can be long and unwieldy if there are a lot of values to initialize
  - May be several easier, standard ways of doing this
- Prevents the passing of uninitialized values
  - Swift passes `nil` values…
  - ..but we want to prevent unexpected behavior by limiting `nil` values

What is self?

- `self` refers to the *instance*
- The instance has its own memory and therefore its own variable assignments (`self.firstName`)
    - Same concept as accessing an instance's method (`self.description()`)
- Not always necessary to explicitly use `self` within a class definition
    - It is implicit whenever a instance variable or method is called
    - **Unless there is a locally-scoped variable hiding it**

Variables and Scope

- What is the difference between these init methods?

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

Variables and Scope

- What is the difference between these methods?

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

Class-Level Methods and Properties

- Type methods called on the *type* itself rather than an instance
- `class` keyword defines type-methods
  - Allows subclasses to override superclass implementation
  - `static` also works but methods cannot be overwritten by subclass
- Class-level properties are defined at the type, rather than instance, level
- `static` keyword defines class-level properties

Type-Method Example

```
class Player {
    static var unlockedLevels = 1
    class func unlockLevels(levels: Int) {
        unlockedLevels += levels
    }
    var currentLevel = 1
    func updateCurrentLevel(selectedLevel : Int) {
        if selectedLevel < Player.unlockedLevels {
            currentLevel = selectedLevel } else { currentLevel =
            Player.unlockedLevels }
    }
}
```

Working with Objects

- Classes allow us to instantiate objects
- All objects of a class share the same properties and functions
- Objects can differ from each other in terms of the values of the properties and how their functions are called

Object-oriented Principles

- Encapsulation
- Polymorphism
- Inheritance
- Abstraction

Encapsulation

- Hides methods and fields from outside users of a class
- User should go through *accessors* to read an object's internal properties
- User should go through *mutators* to change an object's internal properties
- Methods and fields that the user does not manipulate directly should not be visible to the user

Private Properties and Methods

- Cannot be accessed outside of the class
- Preserve internal workings of classes
- Maintain modular, “black box” nature of classes
- Reduce unexpected class access patterns
- `private` keyword declared before type:

```
private var currentSprite: Sprite

private func setSprite(newSprite: Sprite)
{ currentSprite = newSprite }
```

Getters and Setters

- Functions created to get (access) an object's properties and set (change) an object's properties
- Standard Java implementation:

```
private String name;

getName() { return name; }

setName(String newName) { name = newName; }
```

- User calls on `getName()` and `setName()` rather than accessing `name` directly
- Functions in the class can access/change `name` directly

Gets and Sets in Swift

- Properties can have `get` and `set` methods defined and called within the class
  - Simplifies use of property (no explicit `get` or `set` call by user)
  - Maintains safety of encapsulation (class internally calls `get` and `set`)

```
class Person {
   private var _name = “Unknown”
   var name: String {
      get { return _name }
      set (newName) { _name = newName }
   }
   init(name: String) {
      self.name = name
   }
}
var person = Person()
var name = person.name //Accesses person’s name getter
person.name = “Anna Akhmatova” //Accesses person’s name setter
```

Another Example

```
private var _currentLevel = 1

private var _maxLevel = 10

var currentLevel: Int {

  get { return _currentLevel }

  set (newLevel) {

    if newLevel <= 0 { _currentLevel = 1 }

    else if newLevel > _maxLevel { _currentLevel = _maxLevel}

    else { _currentLevel = newLevel }

  }

}
```

When to Use Private Properties and Methods?

- Functions and properties should default to private
  - Only expose them as public when necessary
- Names of public methods should indicate the high level purpose of the function
  - No need for low level details
  - User can infer everything that needed to happen did

Private Methods Example

```
func postToServer(data: Data) {

   serializePackage(data)

   encryptPackage(data)

   sendPackage(data, data.address)

}

private func serializePackage(data: Data) { … }

private func encryptPackage(data: Data) { … }

private func sendPackage(data: Data, address: String) { … }
```

Why Encapsulation?

- Simplifies interaction between class and user of the class
  - User does not need to know about a class's underlying implementation to use it
- Allows internal changes within a class without breaking existing codes that uses it
  - User never directly accesses data, so data representation can change

Inheritance

- Defines “is a” relationships between objects
- Classes can be *children* of existing classes
  - Inherits all properties and methods from the parent class
  - Child (subclass) should have exactly one parent (superclass)
  - Parent can have multiple children

Using Inheritance

```
class Person {

    private var name: String

    func greeting() {

        print(“Hello, my name is \(name)”)

    }

}

class Player: Person {

    private var character: String

    func enterGame() {

        print(“Player \(name) has entered the game as \(character)”)

    }

}
```

Overriding Functions

- Possible to modify parent functions to perform different actions for the child object:

```
//Person function
func greeting() {
  print("Hello, my name is \(name)")
}
//Player function
override func greeting() {
  print("Hello, I am \(character)")
}
```

Calling on Parent Functions

- A child object can access the parent's functions using `super`
  - Refers to the parent class's objects
  - Same idea as `self` but accesses as the parent rather than the current child
- Allows for child-specific and parent tasks to be performed in the same function

```swift
class Person {
  var name: String
  init(name: String) {
     self.name = name
  }
}
class Player: Person {
  var character: String
  init(name: String, character: String) {
     super.init(name)
     self.character = character
  }
}
```

Why Inheritance?

- Emulates how people think about categories of objects
- Allows one definition of object properties to be applies across multiple types of objects
  - Shared functionality in otherwise different types
- Allows for extensibility of existing classes without reimplementing/fully understanding that class

Polymorphism

- Allows for different functionality from the same interface
- Can mean:
  - Static: Multiple methods with different parameters
  - Dynamic: Subclass overriding of superclass’s functionality

Static Polymorphism

- Multiple methods share the same name but take different parameters

```
func greeting() {

  print("Hello!")
}

func greeting(name: String) {

  print("Hello, \(name)!")

}
```

Dynamic Polymorphism

- A subclass's implementation of a function overrides the superclass's function
- Directly relates to the concepts of inheritance
  - A subclass has an "is a" relationship with the superclass, but a superclass does not have an "is a" relationship with the subclass

- Consider this code:

```
class Person {
   var name: String
   func greeting() {
     print("Hello! My name is \(name).")
   }
}
class Player: Person {
  override func greeting() {
    print("Lali-ho!")
  }
}
```

- What does `Person().greeting()` print?
- What does `Player().greeting()` print?

Downcasting/Upcasting

- A subclass can be "cast" as any of its superclasses (e.g. it is its parent or its parent's parent, etc)
  - This is called upcasting
  - Will not work if the object is not actually a child of the casted class
- An object currently called as a superclass *may* be an subclass instance
  - Will access the subclass's functions and properties it is currently "cast" to
  - This is called downcasting
  - Will not work if the object is not actually a subclass instance

Casting in Swift

- Type-cast operator `as` allows casting to different casts
- `as!` performs a force unwrap
  - Object is downcast and unwraps the result in one step
- `as?` performs a conditional unwrap
  - Object is downcast as an optional which will be `nil` if cast failed
- When should you use a conditional unwrap versus a force unwrap?

Casting in Swift Example

```
var p1 = Player()

p1.greeting()

let p = p1 as Person //upcast

print(p is Player) //p is of type Player

print(p is Person) //p is of type Person

if let p2 = p as? Player { //downcast within a conditional

  p2.greeting()

}
```

Quiz Question!

- Will this downcast work?

```
var p1 = Person()

let p2 = p1 as? Player
```

Why Polymorphism?

- Allows overloading of functions for flexible user interactions
- Adds power and control to inheritance
  - Inheritance from parent class the child class has much more flexibility
  - Objects of child class have a clear concept of “is a” even if inheritance chain is very deep

Abstraction

- Class defined in terms of its functionality rather than its implementation
- Users should see larger model the class represents
  - Do not need to understand how the class was built
- Closely tied to encapsulation
  - Details hidden from the user
- Used to help with concepts of inheritance and polymorphism

Why Abstraction?

- Abstraction leads to better modularity
  - Complex systems can be thought of as a collection of smaller (abstracted) systems
  - Abstracted subsystems can be considered within a larger system *before* the subsystem is actually implemented
  - Any additional subsystems remain separate from existing subsystems
- Abstraction makes code complexity more manageable
  - Developers do not need to understand all systems to use (or create) their subsystems