Lecture 05 – More Swift

Developing Applications for iOS

1

OPTIONAL

5

Optionals

Optional

So an Optional is just an enum
It essentially looks like this …

```
enum Optional<T> { // a generic type, like Array<Element> or Dictionary<Key,Value>
    case none
    case some(<T>) // the some case has associated data of type T
}
```

But this type is so important that it has a lot of special syntax that other types don't have …

6

Optionals

Special Optional syntax in Swift

The "not set" case has a special keyword: nil

The character ? is used to declare an Optional,
e.g. var indexOfOneAndOnlyFaceUpCard: Int?

The character ! is used to "unwrap" the associated data if an Optional is in the "set" state …
e.g. let index = cardButtons.index(of: button)!

The keyword if can also be used to conditionally get the associated data …

```
e.g. if let index = cardButtons.index(of: button)
     { … }
```


7

Optionals

Special Optional syntax in Swift

An Optional declared with ! (instead of ?) will implicitly unwrap (add !) when accessed …

```
e.g. var flipCountIndex: UILabel!
     enables flipCountIndex.text = "…" (i.e. no ! here)
```

You can use ?? to create an expression which "defaults" to a value if an Optional is not set …

```
e.g. return emoji[card.identifier] ?? "?"
```


8

Optionals

Optional

So an Optional is just an enum
It essentially looks like this …

```
enum Optional<T> { // a generic type, like Array<Element> or Dictionary<Key,Value>
    case none
    case some(<T>) // the some case has associated data of type T
}
```

But this type is so important that it has a lot of special syntax that other types don't have …

9

Optionals

Optional

Declaring and assigning values to an Optional …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String?
var hello: String? = “hello”
var hello: String? = nil
```

```
var hello: Optional<String> = .none
var hello: Optional<String> = .some(“hello”)
var hello: Optional<String> = .none
```


10

Optionals

Optional

Note that Optionals always start out nil …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String?
var hello: String? = “hello”
var hello: String? = nil
```

```
var hello: Optional<String> = .none
var hello: Optional<String> = .some(“hello”)
var hello: Optional<String> = .none
```


11

Optionals

Optional

Unwrapping …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let hello: String? = ...
print(hello!)
```

```
if let greeting = hello {
    print(greeting)
} else {
    // do something else
}
```

```
switch hello {
    case .none:
        // raise an exception (crash)
    case .some(let data):
        print(data)
}
```

```
switch hello {
    case .some(let data): print(data)
    case .none: { // do something else }
}
```


12

Optionals

Optional

Implicitly unwrapped Optional (these start out nil too) …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
var hello: String!
hello = ...
print(hello)
```

```
var hello: Optional<String> = .none

switch hello {
    case .none: // raise exception (crash)
    case .some(let data): print(data)
}
```


14

Optionals

Optional

Nil-coalescing operator (Optional defaulting) …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x ?? "foo"
```

```
switch x {
    case .none: y = "foo"
    case .some(let data): y = data
}
```


15

Optionals

Special Optional syntax in Swift

- Optional Chaining

You can also use ? when accessing an Optional to bail out of an expression midstream …

16

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```


Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

CS193p

Fall 2017-18

18

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```


Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

CS193p

Fall 2017-18

20

Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```


Optionals

Optional

Optional chaining …

```
enum Optional<T> {
    case none
    case some(<T>)
}
```

```
let x: String? = ...
let y = x?.foo()?.bar?.z
```

```
switch x {
    case .none: y = nil
    case .some(let data1):
        switch data1.foo() {
            case .none: y = nil
            case .some(let data2):
                switch data2.bar {
                    case .none: y = nil
                    case .some(let data3): y = data3.z
                }
        }
}
```

CS193p

Fall 2017-18

22

Data Structures

Four Essential Data Structure-building Concepts in Swift

- class
- struct
- enum
- protocol

class

- Supports object-oriented design
- Single inheritance of both functionality and data (i.e. instance variables)
- Reference type (classes are stored in the heap and are passed around via pointers)

23

Data Structures

Four Essential Data Structure-building Concepts in Swift class

- class
- struct
- enum
- protocol

struct

- Value type (structs don't live in the heap and are passed around by copying them)
- Very efficient "copy on write" is automatic in Swift

26

Demo

Make Matching Game into a struct

We'll see that there's little difference in how it's declared

27

```
import Foundation

class MatchingGame{

    var cards = [Card]() //var cards: Array<Card>

    var indexOfOneAndOnlyFaceUpCard: Int?


    func chooseCard(at index: Int){
        if !cards[index].isMatched{ // 已match過的牌不作用
            if let matchIndex = indexOfOneAndOnlyFaceUpCard, matchIndex != index{
                if cards[matchIndex].identifier == cards[index].identifier{
                    cards[matchIndex].isMatched = true
                    cards[index].isMatched = true
                }//matched
                cards[index].isFaceUp = true
                indexOfOneAndOnlyFaceUpCard = nil
            }//has another previous card face up
```

28

```
import Foundation

struct MatchingGame{

    var cards = [Card]() //var cards: Array<Card>

    var indexOfOneAndOnlyFaceUpCard: Int?


    func chooseCard(at index: Int){
        if !cards[index].isMatched{ // 已match過的牌不作用
            if let matchIndex = indexOfOneAndOnlyFaceUpCard, matchIndex != index{
                if cards[matchIndex].identifier == cards[index].identifier{
                    cards[matchIndex].isMatched = true
                    cards[index].isMatched = true
                }//matched
                cards[index].isFaceUp = true
                indexOfOneAndOnlyFaceUpCard = nil
            }//has another previous card face up
```

29

```
struct MatchingGame{

    var cards = [Card]() //var cards: Array<Card>

    var indexOfOneAndOnlyFaceUpCard: Int?


    func chooseCard(at index: Int){
        if !cards[index].isMatched{ // 已match過的牌不作用
            if let matchIndex = indexOfOneAndOnlyFaceUpCard, matchIndex != index{
                if cards[matchIndex].identifier == cards[index].identifier{
                    cards[matchIndex].isMatched = true   Cannot assign to property: 'self' is immut
                    cards[index].isMatched = true        Cannot assign to property: 'self' is immuta
                }//matched
                cards[index].isFaceUp = true             Cannot assign to property: 'self' is immuta
                indexOfOneAndOnlyFaceUpCard = nil        Cannot assign to property: 'self' is immuta
            }//has another previous card face up
```

30

```
struct MatchingGame{

    var cards = [Card]() //var cards: Array<Card>

    var indexOfOneAndOnlyFaceUpCard: Int?

mutating func chooseCard(at index: Int){
        if !cards[index].isMatched{ // 已match過的牌不作用
            if let matchIndex = indexOfOneAndOnlyFaceUpCard, matchIndex != index{
                if cards[matchIndex].identifier == cards[index].identifier{
                    cards[matchIndex].isMatched = true   Cannot assign to property: 'self' is immut
                    cards[index].isMatched = true        Cannot assign to property: 'self' is immuta
                }//matched
                cards[index].isFaceUp = true             Cannot assign to property: 'self' is immuta
                indexOfOneAndOnlyFaceUpCard = nil        Cannot assign to property: 'self' is immuta
            }//has another previous card face up
```

31

COPY ON WRITE

```
class MatchingGame{

    func chooseCard(at index:Int)
    {

        if !cards[index].isMatched{ // if click on an disabled card, nothing will happen
            if let matchIndex = indexOfOneAndOnlyFaceUpCard, matchIndex != index
            {
                if cards[matchIndex].identifier == cards[index].identifier{
                    cards[matchIndex].isMatched = true
                    cards[index].isMatched = true
                }
                cards[index].isFaceUp = true
                //indexOfOneAndOnlyFaceUpCard = nil
            }else{
                indexOfOneAndOnlyFaceUpCard = index
            }
        }//end if !cards[index].isMatched

    }//end func

}
```

```
struct MatchingGame{

    mutating func chooseCard(at index:Int)
    {

        if !cards[index].isMatched{ // if click on an disabled card, nothing will happen
            if let matchIndex = indexOfOneAndOnlyFaceUpCard, matchIndex != index
            {
                if cards[matchIndex].identifier == cards[index].identifier{
                    cards[matchIndex].isMatched = true
                    cards[index].isMatched = true
                }
                cards[index].isFaceUp = true
                //indexOfOneAndOnlyFaceUpCard = nil
            }else{
                indexOfOneAndOnlyFaceUpCard = index
            }
        }//end if !cards[index].isMatched

    }//end func


}
```

32

Data Structures

Four Essential Data Structure-building Concepts in Swift

- class
- struct
- enum
- protocol

struct

- Value type (structs don't live in the heap and are passed around by copying them)
- Very efficient "copy on write" is automatic in Swift
- This copy on write behavior requires you to mark mutating methods
- No inheritance (of data)

33

HOW ABOUT VAR ?

mutating

```
struct MatchingGame{
    var cards = [Card]() //var cards: Array<Card>

    var indexOfOneAndOnlyFaceUpCard: Int?{
        get{
            var foundIndex:Int?
            for index in cards.indices{
                if cards[index].isFaceUp,!cards[index].isMatched{
                    if foundIndex == nil{
                        foundIndex = index
                    }else{
                        return nil
```

34

HOW ABOUT VAR ?
DO NOT NEED TO ADD MUTATING!

```
struct MatchingGame{

    mutating var cards = [Card]() //var cards: Array<Card>

    var indexOfOneAndOnlyFaceUpCard: Int?{
        get{
            var foundIndex:Int?
            for index in cards.indices{
                if cards[index].isFaceUp,!cards[index].isMatched{
                    if foundIndex == nil{
                        foundIndex = index
                    }else{
                        return nil
```

35

Data Structures

Four Essential Data Structure-building Concepts in Swift

class
struct
enum
protocol

struct

Value type (structs don't live in the heap and are passed around by copying them)
Very efficient "copy on write" is automatic in Swift
This copy on write behavior requires you to mark mutating methods
No inheritance (of data)
Mutability controlled via let (e.g. you can't add elements to an Array assigned by let)

36

Data Structures

Four Essential Data Structure-building Concepts in Swift

class
struct
enum
protocol

enum

Used for variables that have one of a discrete set of values
Each option for that discrete value can have "associated data" with it
The associated data is the only storage that an enum can have (no instance variables)
Value type (i.e. passed around by copying)
Can have methods and computed (only) properties
No inheritance

38

Data Structures

Four Essential Data Structure-building Concepts in Swift class

struct
enum
protocol

protocol

A type which is a declaration of functionality only
No data storage of any kind (so it doesn't make sense to say it's a "value" or "reference" type)
Essentially provides multiple inheritance (of functionality only, not storage) in Swift

39

PROTOCOL

40

Protocols

A protocol is a TYPE

- It can be used almost anywhere any other type is used:
  - It can be vars, function parameters, etc.

41

Protocols

There are three aspects to a protocol

1. The protocol **declaration** (which properties and methods are in the protocol)
2. a class, struct or enum declaration that makes the claim to implement the protocol
3. The code in said class, struct or enum (or extension) that implements the protocol

42

Protocols

Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2
{
    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt( )
    init(arg: Type)
}
```

43

Protocols

Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2
{
    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol 1 and 2

44

Protocols

Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2
{
    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) ->
    SomeType mutating func changeIt()
    init(arg: Type)
}
```

You must specify whether a property is get only or both get and set

45

Protocols

Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2
{
    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Any functions that are expected to mutate the receiver should be marked mutating

46

Protocols

Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2
{
    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Any functions that are expected to mutate the receiver should be marked mutating (unless you will restrict your protocol to class implementers only)

47

```
protocol someProtocol:class, InheritedProtocol1, InheritedProtocol2 {

    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> String
    func changeIt()
    init(arg: Int)

}
```

Using 'class' keyword to define a class-constrained protocol is deprecated; use 'AnyObject' instead

Replace 'class' with 'AnyObject'

restrict your protocol to class

48

```
protocol someProtocol:AnyObject {

    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> String
    func changeIt()
    init(arg: Int)

}
```

restrict your protocol to class

- **You should use** `AnyObject` (`protocol SomeProtocol: AnyObject`).
- `AnyObject` and `class` are equivalent. There is no difference.
- `class` will eventually be deprecated.

49

Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the `AnyObject` protocol to a protocol's inheritance list.

```
protocol SomeClassOnlyProtocol: AnyObject, SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

In the example above, `SomeClassOnlyProtocol` can only be adopted by class types. It's a compile-time error to write a structure or enumeration definition that tries to adopt `SomeClassOnlyProtocol`.

> NOTE
>
> Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics. For more about reference and value semantics, see Structures and Enumerations Are Value Types and Classes Are Reference Types.

https://docs.swift.org/swift-book/LanguageGuide/Protocols.html

50

Protocols

Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2
{
    var someProperty: Int { get set }

    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

You can even specify that implementers must implement a given initializer

51

Protocols

There are three aspects to a protocol

1. The protocol **declaration** (which properties and methods are in the protocol)
2. a class, struct or enum declaration that makes the **claim to implement** the protocol
3. The **code** in said class, struct or enum (or extension) that implements the protocol

52

Protocols

How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol
{
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class

53

Protocols

How an implementer says "I implement that protocol"

```
enum SomeClass : SomeProtocol, AnotherProtocol
{
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class.
- obviously, enums and structs would not have the superclass part

54

Protocols

How an implementer says "I implement that protocol"

```
struct SomeClass : SomeProtocol, AnotherProtocol
{
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class.
- obviously, enums and structs would not have the superclass part

55

Protocols

How an implementer says "I implement that protocol"

```
struct SomeClass : SomeProtocol, AnotherProtocol
{
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Any number of protocols can be implemented by a given class, struct or enum

56

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}
```

```
let prius: Car = Car()
let square: Shape = Shape()
```

59

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```


60

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
```


61

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: ...)
```


62

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: ...)
thingToMove.changeOil()
```


63

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: ...)
thingToMove.changeOil()
thingToMove = square
```


64

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint) { ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: ...)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]
```


65

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
    func at()
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func at()
    func changeOil()

}
struct Shape : Moveable {
    mutating func move(to point: CGPoint)
{ ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: ...)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]

func slide(_ slider: Moveable) {
    slider.at()
}
slide(prius)
slide(square)
```


66

Protocols

Using protocols like the type that they are!

```
protocol Moveable {
    mutating func move(to point: CGPoint)
}
class Car : Moveable {
    func move(to point: CGPoint) { ... }
    func changeOil()
}
struct Shape : Moveable {
    mutating func move(to point: CGPoint)
{ ... }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.move(to: ...)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]

func slide(slider: Moveable) {
    let positionToSlideTo = ...
    slider.move(to: positionToSlideTo)
}
slide(prius)
slide(square)
func slipAndSlide(x: Slippery & Moveable)
slipAndSlide(prius)
```


67

Protocols

There are three aspects to a protocol

1. The protocol **declaration** (which properties and methods are in the protocol)
2. a class, struct or enum declaration that makes the **claim to implement** the protocol
3. The **code** in said class, struct or enum (or extension) that implements the protocol

71

Another use of Protocols

```
var emojiDic = [Int:String]()
```

Being a key in a Dictionary

To be a key in a Dictionary, you have to be able to be unique.
A key in a Dictionary does this by providing an **Int** that is very probably unique (a hash)
and then also by implementing equality testing to see if two keys are, in fact, the same.

This is enforced by requiring that a Dictionary's keys implement the Hashable protocol.
Here's what that protocol looks like …

```
protocol Hashable: Equatable {
    var hashValue: Int { get }
}
```

Very simple. Note, though, that Hashable inherits from Equatable …

72

Another use of Protocols

Being a key in a Dictionary

That means that to be Hashable, you also have to implement Equatable.
The Equatable protocol looks like this ...

```
protocol Equatable {
    static func ==(lhs: Self, rhs: Self) -> Bool
}
```

Types that conform to Equatable have to have a type function (note the static) called ==

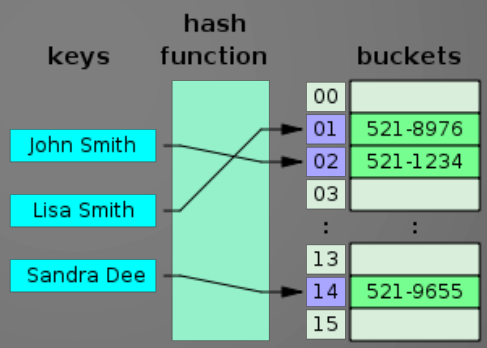
The arguments to == are both of that same type (i.e. Self of the type is the type itself)

The == operator also happens to look for such a static method to provide its implementation!


```
var emojiDic = [Int:String]()

   func getEmoji(for card: Card) -> String
   {


       if emojiDic[card.identifier] == nil , emojiChoices.count > 0 {
           let randomIndex = emojiChoices.count.arc4random
           emojiDic[card.identifier] =  emojiChoices.remove(at: randomIndex)
       }
       return emojiDic[card.identifier] ?? "?"
   }
```

Many types in the standard library conform to Hashable: Strings, integers, floating-point and Boolean values, and even sets are hashable by default

HASH TABLE



75

Another use of Protocols

Being a key in a Dictionary

Dictionary is then declared like this: Dictionary<Key: Hashable, Value>

This restricts keys to be things that conform to Hashable (there's no restriction on values)

Let's go make Card be Hashable.

Then we can use it directly as the key into our emoji Dictionary.

As a bonus, we'll be able to compare Cards directly since they'll be Equatable.

This will even allow us to make identifier be private in Card, which makes a lot of sense.

76

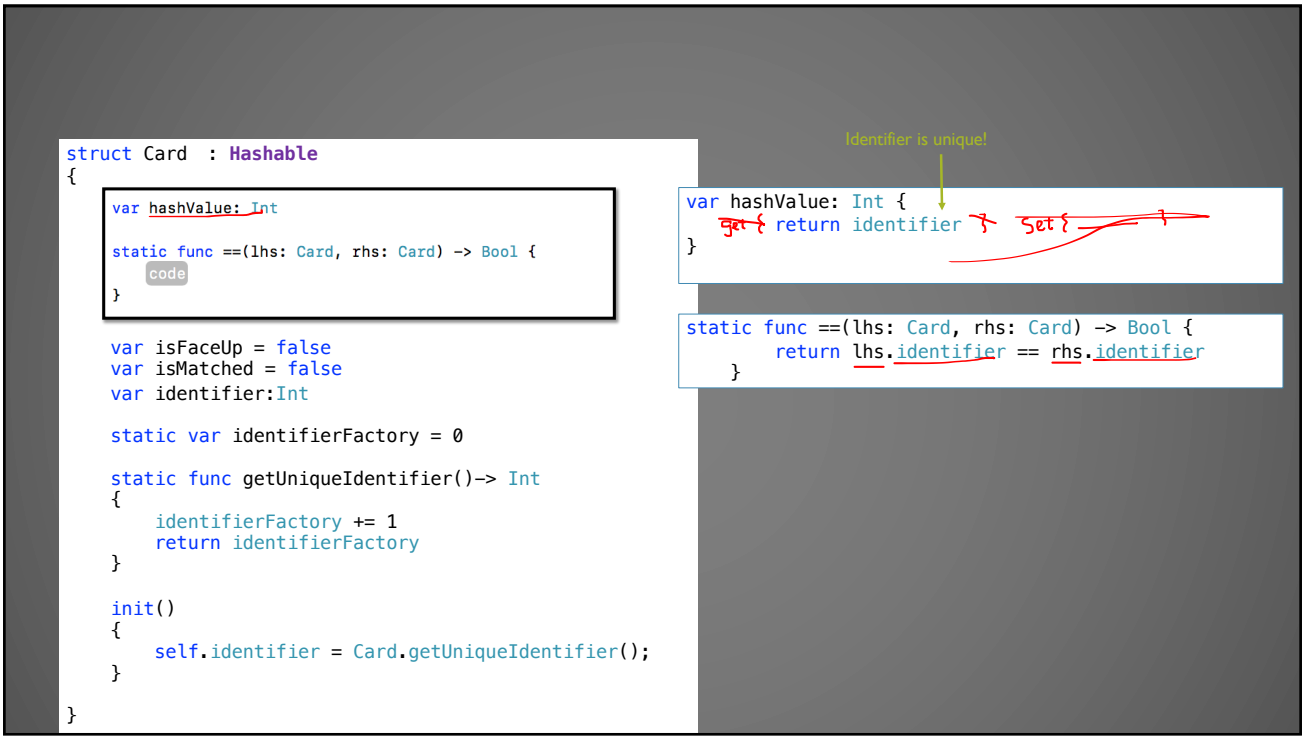
```
var emojiDic = [Card:String]()
// Type 'Card' does not conform to protocol 'Hashable'

   func getEmoji(for card: Card) -> String
   {


       if emojiDic[card.identifier] == nil , emojiChoices.count > 0 {
           let randomIndex = emojiChoices.count.arc4random
           emojiDic[card.identifier] =  emojiChoices.remove(at: randomIndex)
       }
       return emojiDic[card.identifier] ?? "?"
   }
```

78

```
struct Card : Hashable
{
    var isFaceUp = false
    var isMatched = false
    var identifier:Int

    static var identifierFactory = 0

    static func getUniqueIdentifier()-> Int
    {
        identifierFactory += 1
        return identifierFactory
    }

    init()
    {
        self.identifier = Card.getUniqueIdentifier();
    }

}
```

Type 'Card' does not conform to protocol 'Equatable'
Do you want to add protocol stubs? Fix
Type 'Card' does not conform to protocol 'Hashable'

79


struct Card : Hashable
{
var hashValue: Int
static func ==(lhs: Card, rhs: Card) -> Bool {
code
}
var isFaceUp = false
var isMatched = false
var identifier:Int
static var identifierFactory = 0
static func getUniqueIdentifier()-> Int
{
identifierFactory += 1
return identifierFactory
}
init()
{
self.identifier = Card.getUniqueIdentifier();
}
}
Identifier is unique!
var hashValue: Int {
return identifier
}
static func ==(lhs: Card, rhs: Card) -> Bool {
return lhs.identifier == rhs.identifier
}

80