Lecture 04 – More Swift

Developing Applications for iOS

Range

```
for n in 3..<5{
print(n)
}
// Prints "3”
// Prints "4"
```

How do you do `for (i = 0.5; i <= 15.25; i += 0.3)`?

Stride

Floating point

Floating point numbers don't stride by Int, they stride by a floating point value.
So 0.5...15.25 is just a Range, not a CountableRange (which is needed for for in).

Luckily, there's a global function that will create a sequence from floating point values!

```
for i in stride(from: 0.5, through: 15.25, by: 0.3) {

}
```

Tuples

What is a tuple?

It is nothing more than a grouping of values. (contain one or more types)
Tuples are passed by value, not reference
you can use a type.

```
let x: (String, Int, Double) = ("hello", 5, 0.85) // the type of x is "a tuple"
let (word, number, value) = x // this names the tuple elements when accessing the tuple
```

Tuples

What is a tuple?

```
  let x: (String, Int, Double) = ("hello", 5, 0.85) // the type of x is "a tuple"
  let (word, number, value) = x // this names the tuple elements when accessing the tuple


  print(word)     // prints hello
  print(number) // prints 5
  print(value)   // prints 0.85
```

… or the tuple elements can be named when the tuple is <u>declared</u> (this is strongly preferred) …

```
let x: (w: String, i: Int, v: Double) = ("hello", 5, 0.85)
print(x.w) // prints hello
print(x.i) // prints 5
print(x.v) // prints 0.85
let (wrd, num, val) = x // this is also legal (renames the tuple's elements on access)
```

Tuples

Tuples as return values

You can use tuples to return multiple values from a function or method …

```
func getSize() -> (weight: Double, height: Double) {
      return (250, 80)
 }


let x = getSize()
print("weight is \(x.weight)") // weight is 250
… or …
print("height is \(getSize().height)") // height is 80
```

ACCESS CONTROL

Access Control

Protecting our internal implementations

Likely most of you have only worked on relatively small projects

Inside those projects, any object can pretty much call any function in any other object

When projects start to get large, though, this becomes very dicey

You want to be able to protect the INTERNAL implementation of data structures

You do this by marking which API* you want other code to use with certain keywords

* i.e. methods and properties

Access Control

Protecting our internal implementations

Swift supports this with the following access control keywords …

- internal - this is the default, it means "usable by any object in my app or framework"
- private - this means "only callable from within this object"
- private(set) - this means "this property is readable outside this object, but not settable"
- fileprivate - accessible by any code in this source file

Access Control

Protecting our internal implementations

Swift supports this with the following access control keywords …

- internal - this is the default, it means "usable by any object in my app or framework"
- private - this means "only callable from within this object"
- private(set) - this means "this property is readable outside this object, but not settable"
- fileprivate - accessible by any code in this source file

- public - (for frameworks only) this can be used by objects outside my framework
- open - (for frameworks only) public and objects outside my framework can subclass this

Access Control

Protecting our internal implementations

Swift supports this with the following access control keywords …

- `internal` - this is the default, it means "usable by any object in my app or framework"
- `private` - this means "only callable from within this object"
- `private(set)` - this means "this property is readable outside this object, but not settable"
- `fileprivate` - accessible by any code in this source file

- `public` - (for frameworks only) this can be used by objects outside my framework
- `open` - (for frameworks only) public and objects outside my framework can subclass this

We are not going to learn to develop frameworks this quarter, so we are only concerned with …

`private`, `private(set)`, `fileprivate` and `internal` (which is the default, so no keyword)

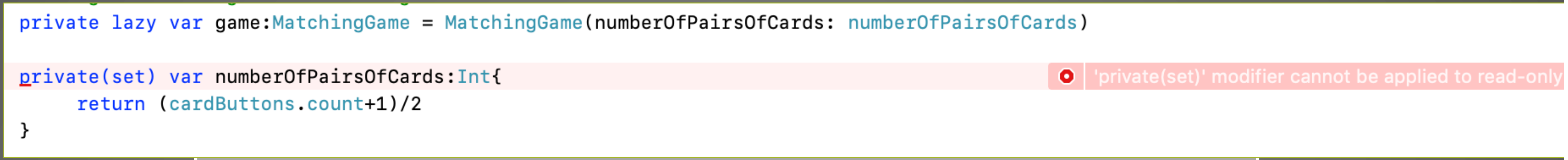
Access Control

Protecting our internal implementations

Swift supports this with the following access control keywords …

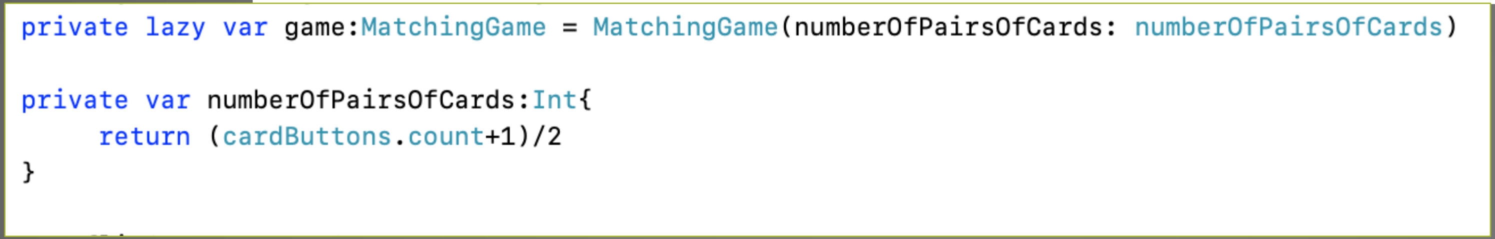
- internal - this is the default, it means “usable by any object in my app or framework”
- private - this means “only callable from within this object”
- private(set) - this means “this property is readable outside this object, but not settable”
- fileprivate - accessible by any code in this source file

A good strategy is to just mark everything private by default.
Then remove the private designation when that API is ready to be used by other code.

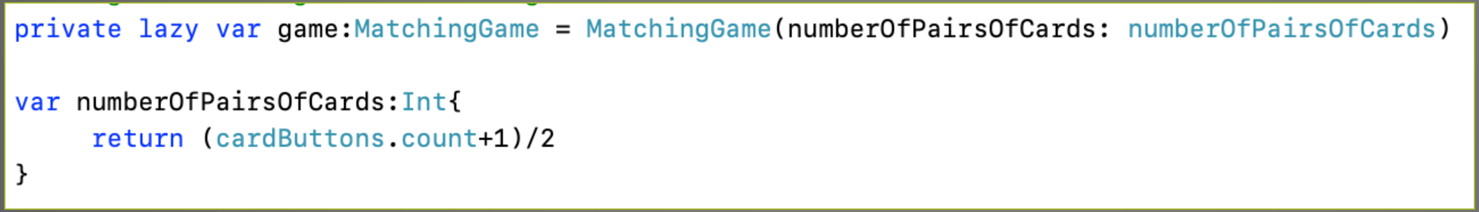


```
private lazy var game:MatchingGame = MatchingGame(numberOfPairsOfCards: numberOfPairsOfCards)

private(set) var numberOfPairsOfCards:Int{
     return (cardButtons.count+1)/2
}
```

'private(set)' modifier cannot be applied to read-only



```
private lazy var game:MatchingGame = MatchingGame(numberOfPairsOfCards: numberOfPairsOfCards)

private var numberOfPairsOfCards:Int{
     return (cardButtons.count+1)/2
}
```



```
private lazy var game:MatchingGame = MatchingGame(numberOfPairsOfCards: numberOfPairsOfCards)

var numberOfPairsOfCards:Int{
     return (cardButtons.count+1)/2
}
```

```swift
private(set) var flipCount:Int = 0
{
    didSet{
        flipCountLabel.text = "Flips: \(flipCount)"
    }
}
```

```swift
@IBOutlet private var cardButtons: [UIButton]!
@IBOutlet private weak var flipCountLabel: UILabel!

@IBAction private func touchCard(_ sender: UIButton) {
    if let cardNumber = cardButtons.index(of: sender){
        print("cardNumber = \(String(describing: cardNumber))")
        game.chooseCard(at: cardNumber)
        updateViewFromModel()
        //flipCard(withEmoji: emojiChoices[cardNumber], on: sender)
    }else{
        print("not in the collection")
    }
    flipCount += 1
}
```

```swift
private func updateViewFromModel()
{
    for index in cardButtons.indices {
        let button = cardButtons[index]
        let card = game.cards[index]
        if card.isFaceUp{
            button.setTitle(emoji(for: card), for: UIControl.State.normal)
            button.backgroundColor = card.isMatched ? ▨:⬜
        }else{
            button.setTitle("", for: UIControl.State.normal)
            button.backgroundColor = 🟧
        }
    }
}
```

```swift
private var emojiChoices = ["👻","🎃","🦐","🍰","🌸","🦋","❤️","🐶","🐳"]
private var emoji = [Int:String]()

private func emoji(for card: Card) -> String{
    if emoji[card.identifier] == nil, emojiChoices.count>0{
        let randomIndex = Int(arc4random_uniform(UInt32(emojiChoices.count)))
        //emoji[card.identifier] = emojiChoices[randomIndex]
        emoji[card.identifier] = emojiChoices.remove(at: randomIndex)
    }
    return emoji[card.identifier] ?? "?"
}
```

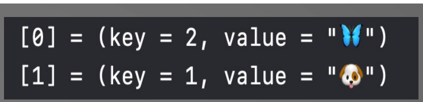
Extensions

Extending existing data structures

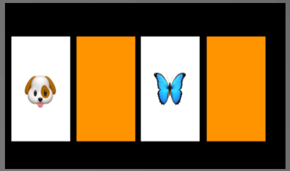
You can add methods/properties to a class/struct/enum (even if you don't have the source).

Let's add a simple extension in Concentration …

Extensions

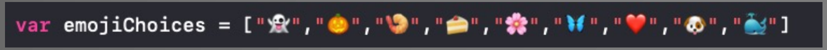


```
[0] = (key = 2, value = "🦋")
[1] = (key = 1, value = "🐶")
```



Extending existing data structures

You can add methods/properties to a class/struct/enum (even if you don't have the source).



```
var emojiChoices = ["👻","🎃","🍤","🍰","🌸","🦋","❤️","🐶","🐳"]
```

```
func getEmoji(for card: Card) -> String
    {
        if emojiDic[card.identifier] == nil , emojiChoices.count > 0 {
            let randomIndex = Int (arc4random_uniform(UInt32(emojiChoices.count)))
            emojiDic[card.identifier] =  emojiChoices.remove(at: randomIndex)
        }

        return emojiDic[card.identifier] ?? "?"
    }
```

ViewController.swift

Extensions

Extending existing data structures

You can add methods/properties to a class/struct/enum (even if you don't have the source).

```swift
func getEmoji(for card: Card) -> String
    {
        if emojiDic[card.identifier] == nil , emojiChoices.count > 0 {
            let randomIndex = Int (arc4random_uniform(UInt32(emojiChoices.count)))
            emojiDic[card.identifier] =  emojiChoices.remove(at: randomIndex)
        }

        return emojiDic[card.identifier] ?? "?"
    }
```

ViewController.swift

```swift
extension Int{
        var arc4random: Int{
            return Int (arc4random_uniform(UInt32(emojiChoices.count)))
        }
    }
```

Computed Properties

The value of a property can be computed rather than stored

A typical stored property looks something like this …

```
var foo: Double
```

A computed property looks like this …

```
var foo: Double {
    get {
        // return the calculated value of foo
    }
    set(newValue) {
        // do something based on the fact that foo has changed to newValue
    }
}
```

You don’t have to implement the set side of it if you don’t want to.
The property then becomes “read only”.

Extensions

Extending existing data structures

You can add methods/properties to a class/struct/enum (even if you don't have the source).

```
let x = 5.arc4random   // 0, 1, 2, 3, 4
```

```
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] ?? "?"
    }
```

```
extension Int{
        var arc4random: Int{
            return Int (arc4random_uniform(UInt32(emojiChoices.count)))
        }
    }
```

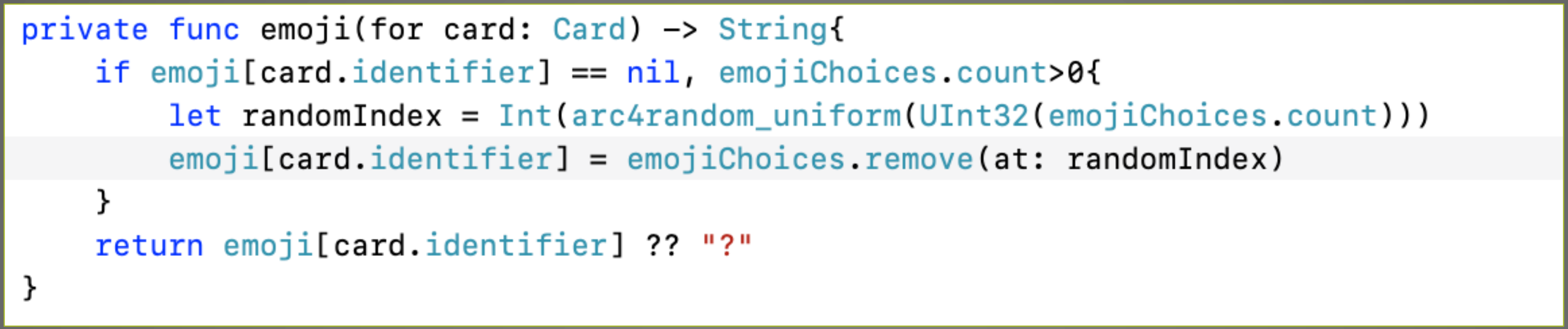
Extensions

Extending existing data structures

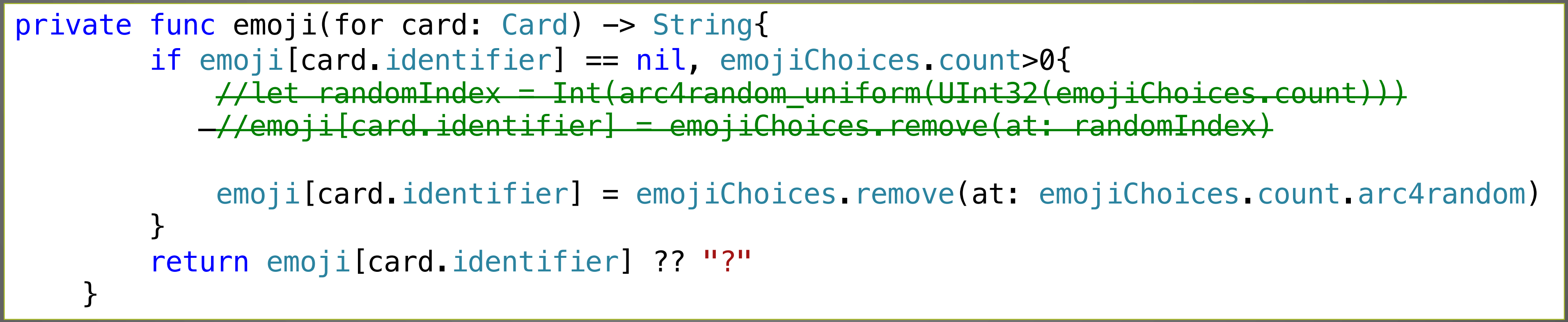
You can add methods/properties to a class/struct/enum (even if you don't have the source).

```
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] ?? "?"
    }
```

```
extension Int{
        var arc4random: Int{
            return Int (arc4random_uniform(UInt32(self)))
        }
    }
```



```
private func emoji(for card: Card) -> String{
    if emoji[card.identifier] == nil, emojiChoices.count>0{
        let randomIndex = Int(arc4random_uniform(UInt32(emojiChoices.count)))
        emoji[card.identifier] = emojiChoices.remove(at: randomIndex)
    }
    return emoji[card.identifier] ?? "?"
}
```



```
private func emoji(for card: Card) -> String{
        if emoji[card.identifier] == nil, emojiChoices.count>0{
            ~~//let randomIndex = Int(arc4random_uniform(UInt32(emojiChoices.count)))~~
           ~~//emoji[card.identifier] = emojiChoices.remove(at: randomIndex)~~

            emoji[card.identifier] = emojiChoices.remove(at: emojiChoices.count.arc4random)
        }
        return emoji[card.identifier] ?? "?"
    }
```

```
extension Int{
    var arc4random:Int{
        if self>0{
            return Int(arc4random_uniform(UInt32(self)))
        }else if self<0{
            return -Int(arc4random_uniform(UInt32(abs(self))))
        }
        else{
            return 0
        }

    }
}
```

Extensions

Extending existing data structures

You can add methods/properties to a class/struct/enum (even if you don't have the source).

There are some restrictions

You can't re-implement methods or properties that are already there (only add new ones).

The properties you add can have no storage associated with them (computed only).

Extensions

Extending existing data structures

You can add methods/properties to a class/struct/enum (even if you don't have the source).

There are some restrictions

You can't re-implement methods or properties that are already there (only add new ones).

The properties you add can have no storage associated with them (computed only).

This feature is easily abused

It should be used to add clarity to readability not obfuscation!

Don't use it as a substitute for good object-oriented design technique.

Best used (at least for beginners) for very small, well-contained helper functions.

Can actually be used well to organize code but requires architectural commitment.

When in doubt (for now), don't do it.

ENUM

Optionals

Optional

A completely normal type in Swift

It's an enumeration

Let's take a moment and learn about enumerations and then we'll look at Optionals a little closer

enum

Another variety of data structure in addition to struct and class

It can only have discrete states …

```
enum FastFoodMenuItem {
    case hamburger
    case fries
    case drink
    case cookie
}
```

An enum is a VALUE TYPE (like struct), so it is copied as it is passed around

enum

Associated Data

Each state can (but does not have to) have its own "associated data" …

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int) // the unnamed String is the brand, e.g. "Coke"
    case cookie
}
```

Note that the drink case has 2 pieces of associated data (one of them "unnamed")
In the example above, FryOrderSize would also probably be an enum, for example …

```
enum FryOrderSize {
    case large
    case small
}
```

enum

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int) // the unnamed String is the brand, e.g. “Coke”
    case cookie
}

enum FryOrderSize {
    case large
    case small
}
```

enum

Setting the value of an enum

Assigning a value to a variable or constant of type enum is easy …

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int) // the unnamed String is the brand, e.g. "Coke"
    case cookie
}

let menuItem: FastFoodMenuItem =
var otherItem: FastFoodMenuItem =
```

enum

Setting the value of an enum

Just use the name of the type along with the case you want, separated by dot …

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int) // the unnamed String is the brand, e.g. “Coke”
    case cookie
}

let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
```

enum

Setting the value of an enum

When you set the value of an enum you must provide the associated data (if any) …

```
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
```

enum

Setting the value of an enum

Swift can infer the type on one side of the assignment or the other (but not both) …

```
let menuItem~~: FastFoodMenuItem ~~= FastFoodMenuItem.hamburger(numberOfPatties: 2)
var otherItem: FastFoodMenuItem = ~~FastFoodMenuItem~~.cookie
var yetAnotherItem = .cookie // Swift can't figure this out
```

enum

Checking an enum's state

An enum's state is checked with a switch statement …

```
var menuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
switch menuItem {
    case FastFoodMenuItem.hamburger:
        print("burger")
    case FastFoodMenuItem.fries:
        print("fries")
    case FastFoodMenuItem.drink:
        print("drink")
    case FastFoodMenuItem.cookie:
        print("cookie")
}
```

Note that we are ignoring the "associated data" above … so far …

enum

Checking an enum's state

An enum's state is checked with a switch statement …

```
var menuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
switch menuItem {
    case FastFoodMenuItem.hamburger:
        print("burger")
    case FastFoodMenuItem.fries:
        print("fries")
    case FastFoodMenuItem.drink:
        print("drink")
    case FastFoodMenuItem.cookie:
        print("cookie")
}
```

This code would print "burger" on the console

enum

Checking an enum's state

An enum's state is checked with a switch statement …

```
var menuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
switch menuItem {
    case ~~FastFoodMenuItem~~.hamburger:
        print("burger")
    case ~~FastFoodMenuItem~~.fries:
        print("fries")
    case ~~FastFoodMenuItem~~.drink:
        print("drink")
    case ~~FastFoodMenuItem~~.cookie:
        print("cookie")
}
```

enum

Checking an enum's state

An enum's state is checked with a switch statement …

```
var menuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
switch menuItem {
    case .hamburger: print("burger")
    case .fries: print("fries")
    case .drink: print("drink")
    case .cookie: print("cookie")
}
```

It is not necessary to use the fully-expressed FastFoodMenuItem.fries inside the switch
(since Swift can infer the FastFoodMenuItem part of that)

enum

break

If you don't want to do anything in a given case, use break …

```
var menuItem = FastFoodMenuItem.hamburger(numberOfPatties: 2)
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    case .drink: print("drink")
    case .cookie: print("cookie")
}
```

This code would print nothing on the console

enum

default

You must handle ALL POSSIBLE CASES (although you can default uninteresting cases) …

```
var menuItem = FastFoodMenuItem.cookie
switch menuItem {
    case .hamburger: break
    case .fries: print("fries")
    default: print("other")
}
```

enum

default

You must handle ALL POSSIBLE CASES (although you can default uninteresting cases) …

```
var menuItem = FastFoodMenuItem.cookie
switch menuItem {
    case .hamburger: break
    case .fries: print(“fries”)
    default:  print(“other”)
}
```

If the menuItem were a cookie, the above code would print “other” on the console

enum

Multiple lines allowed

Each case in a switch can be multiple lines and does NOT fall through to the next case …

```
var menuItem = FastFoodMenuItem.fries(size: FryOrderSize.large)
switch menuItem {
    case .hamburger: print("burger")
    case .fries:
        print("yummy")
        print("fries")
    case .drink:
        print("drink")
    case .cookie: print("cookie")
}
```

The above code would print "yummy" and "fries" on the console, but <u>not</u> "drink"

enum

Multiple lines allowed

By the way, we can let Swift infer the enum type of the size of the fries too …

```
var menuItem = FastFoodMenuItem.fries(size: .large)
switch menuItem {
    case .hamburger: print(“burger”)
    case .fries:
        print(“yummy”)
        print(“fries”)
    case .drink:
        print(“drink”)
    case .cookie: print(“cookie”)
}
```

enum

What about the associated data?

Associated data is accessed through a switch statement using this let syntax …

```
var menuItem = FastFoodMenuItem.drink(“Coke”, ounces: 32)
switch menuItem {
    case .hamburger(let pattyCount):
            print(“a burger with \(pattyCount) patties!”)
    case .fries(let size):
            print(“a \(size) order of fries!”)
    case .drink(let brand, let ounces):
            print(“a \(ounces)oz \(brand)”)
    case .cookie: print(“a cookie!”)
}
```

enum

What about the associated data?

Associated data is accessed through a switch statement using this let syntax …

```
var menuItem = FastFoodMenuItem.drink("Coke", ounces: 32)
switch menuItem {
    case .hamburger(let pattyCount): print("a burger with \(pattyCount) patties!")
    case .fries(let size): print("a \(size) order of fries!")
    case .drink(let brand, let ounces): print("a \(ounces)oz \(brand)")
    case .cookie: print("a cookie!")
}
```

The above code would print "a 32oz Coke" on the console

enum

What about the associated data?

Associated data is accessed through a switch statement using this let syntax …

```
var menuItem = FastFoodMenuItem.drink(“Coke”, ounces: 32)
switch menuItem {
     case .hamburger(let pattyCount): print(“a burger with \(pattyCount) patties!”)
     case .fries(let size): print(“a \(size) order of fries!”)
     case .drink(let brand, let ounces): print(“a \(ounces)oz \(brand)”)
     case .cookie: print(“a cookie!”)
}
```

Note that the local variable that retrieves the associated data can have a different name

(e.g. pattyCount above versus numberOfPatties in the enum declaration)

(e.g. brand above versus not even having a name in the enum declaration)

enum

Methods yes, (stored) Properties no

An enum can have methods (and computed properties) but no stored properties …

```
enum FastFoodMenuItem {
    case hamburger(numberOfPatties: Int)
    case fries(size: FryOrderSize)
    case drink(String, ounces: Int)
    case cookie

    func isIncludedInSpecialOrder(number: Int) -> Bool { }
    var calories: Int { // calculate and return caloric value here }
}
```

An enum's state is entirely which case it is in and that case's associated data.

enum

Methods yes, (stored) Properties no

In an enum's own methods, you can test the enum's state (and get associated data) using self …

```
enum FastFoodMenuItem {
    . . .
    func isIncludedInSpecialOrder(number: Int) -> Bool {
        switch self {
            case .hamburger(let pattyCount): return pattyCount == number
            case .fries, .cookie: return true // a fries and cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

enum

Methods yes, (stored) Properties no

In an enum's own methods, you can test the enum's state (and get associated data) using self …

```
enum FastFoodMenuItem {
    . . .
    func isIncludedInSpecialOrder(number: Int) -> Bool {
        switch self {
            case .hamburger(let pattyCount): return pattyCount == number
            case .fries, .cookie: return true // a drink and cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

Special order 1 is a single patty burger, 2 is a double patty (3 is a triple, etc.?!)

enum

Methods yes, (stored) Properties no

In an enum's own methods, you can test the enum's state (and get associated data) using self …

```
enum FastFoodMenuItem {
    . . .
    func isIncludedInSpecialOrder(number: Int) -> Bool {
        switch self {
            case .hamburger(let pattyCount): return pattyCount == number
            case .fries, .cookie: return true // a drink and cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

Notice the use of _ if we don't care about that piece of associated data.

enum

Modifying self in an enum

You can even reassign self inside an enum method …

```
enum FastFoodMenuItem {
    . . .
    mutating func switchToBeingACookie() {
        self = .cookie // this works even if self is a .hamburger, .fries or .drink
    }
}
```

enum

Modifying self in an enum

You can even reassign self inside an enum method …

```
enum FastFoodMenuItem {
    . . .
    mutating func switchToBeingACookie() {
        self = .cookie // this works even if self is a .hamburger, .fries or .drink
    }
}
```

Note that mutating is required because enum is a VALUE TYPE.