Today

Grid

Demo of generics with protocols, functions as types, container view, etc.

Varieties of Types

struct
class
protocol
“Dont’ Care” type (aka generics)
enum
functions

Optional

An extremely important type in Swift (it’s an enum).
enum

Another variety of data structure in addition to struct and class

It can only have discrete states ...

def 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” ...

```swift
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 ...

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

Setting the value of an enum
Just use the name of the type along with the case you want, separated by dot ...

```swift
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 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) ...

```swift
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = FastFoodMenuItem.cookie
```
### Setting the value of an enum

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

```swift
let menuItem = FastFoodMenuItem.hamburger(patties: 2)
var otherItem: FastFoodMenuItem = .cookie
var yetAnotherItem = .cookie // Swift can't figure this out
```
Checking an enum’s state

An enum’s state is checked with a `switch` statement (i.e. not `if`) ...

```swift
var menuItem = FastFoodMenuItem.hamburger(patties: 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 ...

```swift
var menuItem = FastFoodMenuItem.hamburger(patties: 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 ...

```swift
var menuItem = FastFoodMenuItem.hamburger(patties: 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` ...

```swift
var menuItem = FastFoodMenuItem.hamburger(patties: 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
A switch must handle ALL POSSIBLE CASES (although you can default uninteresting cases) ...

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

default
A switch must handle ALL POSSIBLE CASES (although you can default uninteresting cases) ...

```swift
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.
What about the associated data?

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

```swift
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!")
}
```
What about the associated data?

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

```swift
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.
What about the associated data?
Associated data is accessed through a switch statement using this let syntax ...

```swift
var menuItem = FastFoodMenuItemItem.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 patties in the enum declaration)
(e.g. brand above versus not even having a name in the enum declaration)
The associated value is actually just a single value that can, of course, be a tuple!
So you can do all the naming tricks of a tuple when accessing associated values via switch.
Methods yes, (stored) Properties no

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

```swift
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 { // switch on self and calculate caloric value here } }
```

An enum’s state is entirely which case it is in and that case’s associated data, nothing more.
In an *enum*’s own methods, you can test the *enum*’s state (and get associated data) using `self` ...

```swift
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
        }
    }
}
```
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 …

```swift
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

Getting all the cases of an enumeration

```swift
enum TeslaModel: CaseIterable {
    case X
    case S
    case Three
    case Y
}

Now this enum will have a `static var allCases` that you can iterate over.

```swift
for model in TeslaModel.allCases {
    reportSalesNumbers(for: model)
}
```

```swift
func reportSalesNumbers(for model: TeslaModel) {
    switch model {
        ...}
}
Optionals

Optional

An Optional is just an enum. Period, nothing more.
It essentially looks like this …

```swift
enum Optional<T> { // a generic type, like Array<Element> or MemoryGame<CardContent>
    case none
    case some(<T>) // the some case has associated value of type T
}
```

You can see that it can only have two values: is set (some) or not set (none).
In the is set case, it can have some associated data tagging along (of don’t care type T).

Where do we use Optional?
Any time we have a value that can sometimes be “not set” or “unspecified” or “undetermined”.
e.g., the return type of `firstIndex(matching:)` if the matching thing is not in the Array.
e.g., an index for the currently-face-up-card in our game when the game first starts.
This happens surprisingly often.
That’s why Swift introduces a lot of “syntactic sugar” to make it easy to use Optionals …
Optionals

Optional

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

var hello: String? = "hello"
var hello: String? = nil
var hello: Optional<String> = .none
var hello: Optional<String> = .some("hello")
var hello: Optional<String> = .none
```
Optionals

**Optional**

Declaring something of type `Optional<T>` can be done with the syntax `T?`

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

var hello: String? = "hello"
var hello: String? = nil
var hello: String? = "hello"
var hello: String? = nil
```
Optionals

Optional

Declaring something of type Optional<T> can be done with the syntax T?.
You can then assign it the value nil (Optional.none).

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

var hello: String? = "hello"
var hello: String? = nil
var hello: Optional<String> = .some("hello")
var hello: Optional<String> = .none
```
Optionals

Optional

Declaring something of type `Optional<T>` can be done with the syntax `T?`. You can then assign it the value `nil` (`Optional.none`). Or you can assign it something of the type `T` (`Optional.some` with associated value that value).

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

var hello: String? = "hello"
var hello: Optional<String> = .some("
var hello: String? = nil
var hello: Optional<String> = .none
```
Optionals

Optional

Declaring something of type `Optional<T>` can be done with the syntax `T?`.
You can then assign it the value `nil` (`Optional.none`).
Or you can assign it something of the type `T` (`Optional.some` with associated value that value).
Note that Optionals always start out with an implicit `= nil`.

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

```
var hello: String? = nil
var hello: Optional<String> = .none
var hello: String? = "hello"
var hello: Optional<String> = .some("hello")
var hello: String? = nil
var hello: Optional<String> = .none
```
Optionals

Optional

You can access the associated value either by force (with `!`) ...

```swift
enum Optional<T> {
    case none
    case some(<T>)
}
let hello: String? = ...
print(hello!)

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

Optional

You can access the associated value either by force (with !) ...
Or "safely" using if let and then using the safely-gotten associated value in { } (else allowed too).

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

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

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

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

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

There’s also `??` which does “Optional defaulting”. It’s called the “nil-coalescing operator”.

```swift
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
}
```
Needs More Demo

Optional in action!

Let's fix that "bogus" Array method `firstIndex(matching:)`. Then we'll make Memorize actually play the game! Both of these will feature the Optional type prominently.