IS 543 Fall 2023

Mobile Platform Development

SwiftUI Layout

Continue Demo

Enum and Optional

Dr. Stephen W. Liddle
782 TNRB, office hours Tu 9:30-10:30am, Th 2-3pm
is543@byu.edu
801-422-8792 (rings on my cell)
Zoom: https://bit.ly/liddlezoom

Today

- SwiftUI layout
  Let's see how SwiftUI allocates space to views
  Then we'll use a grid `LazyVGrid`
- `enum` and `Optional`
  If we have time, we'll explore these some more

Fun Facts

- So far, I've been to 33 countries:

Australia	France	Netherlands
Austria	Germany	New Zealand
The Bahamas	Guatemala	Panama
Belgium	Honduras	Singapore
Brazil	India	Spain
Canada	Israel	Sweden
China	Italy	Switzerland
Costa Rica	Jamaica	Taiwan
Denmark	Japan	Ukraine
Dominican Republic	Liechtenstein	United Kingdom
Finland	Mexico	United States

- Where did I serve my mission? Which countries did I visit only as a tourist? Which countries have I only been to the airport?

Inauguration!

- Any thoughts on the words of Pres. Reese or Elder Christofferson?
  "Becoming BYU"

  We exert our strength only to the extent that we embrace and enhance our religious identity.

  Have the humility to regularly assess their progress in "becoming BYU":

  1) Is the mission of BYU changing me, or am I trying to change the mission of BYU?

  2) Am I cultivating meekness and applying "gospel methodology"?

Questions?

- From the materials you've read and studied since Tuesday, do you have any questions or observations to share?

Layout

- How is space on the screen apportioned to the `Views`?

  It's amazingly simple:

  1. Container `Views` "offer" space to the `Views` inside them
  2. `Views` then choose what size they want to be
  3. Container `Views` then position the `Views` inside of them

- Container `Views`

  The "stacks" (`HStack`, `VStack`) divide up the space offered to them among their subviews

  `ForEach` defers to its container to lay out the `Views` inside of it

  Modifiers (e.g. `.padding()`) essentially "contain" the `View` they modify; some do layout

ViewBuilder

- Container Views often take a ViewBuilder content parameter
  - struct ViewBuilder lets you specify a closure that returns 0 to 10 children
  - Supports if and if/else structures
  - Read the ViewBuilder developer documentation page to see how this works
  - Note: you can't declare variables inside a ViewBuilder; it's just returning a list of Views (maybe using if/else structures)

This compiles fine:

```
HStack { Text("1"); Text("2"); Text("3"); Text("4"); Text("5")
         Text("6"); Text("7"); Text("8"); Text("9"); Text("10") }
```

This does NOT compile:

```
HStack { Text("1"); Text("2"); Text("3"); Text("4"); Text("5")
         Text("6"); Text("7"); Text("8"); Text("9"); Text("10"); Text("11") }
```

Layout

- `HStack` and `VStack`

  Stacks divide up the space that is offered to them and then offer that to the `View`s inside
  It offers space to its "least flexible" (with respect to sizing) subviews first

  Example of an "inflexible" `View`: `Image` (it wants to be a fixed size)
  Another example (slightly more flexible): `Text` (always wants to size to exactly fit its text)
  Example of a very flexible `View`: `RoundedRectangle` (always uses the space offered)

  After an offered `View` takes what it wants, its size is removed from the space available
  Then the stack moves on to the next "least flexible" `View`s
  Like the shampoo bottle says, "lather, rinse, repeat"

  After the `View`s inside the stack choose their own size, the stack sizes itself to fit them

Layout

- `HStack` and `VStack`

  There are a couple of valuable `Views` for layout that are commonly put in stacks

  `Spacer(minLength: CGFloat)`
  Always takes all the space offered to it
  Draws nothing
  The `minLength` defaults to the most likely spacing you'd want on a given platform

  `Divider()`
  Draws a dividing line cross-wise to the way the stack is laying out
  For example, in an `HStack`, `Divider` draws a vertical line
  Takes the minimum space needed to fit the line in the direction the stack is going

Layout

- HStack and VStack

  Stack's choice of who to offer space to next can be overridden with .layoutPriority(Double)

  In other words, layoutPriority trumps "least flexible"

```
HStack {
    Text("Important").layoutPriority(100) // any floating point number is okay
    Image(systemName: "arrow.up")         // the default layout priority is 0
    Text("Unimportant")
}
```

I used this technique in my HW 2 solution — let's see...

  The Important Text above will get the space it wants first

  Then the Image would get its space (since it's less flexible than the Unimportant Text)

  Finally, Unimportant would have to try to fit itself into any remaining space

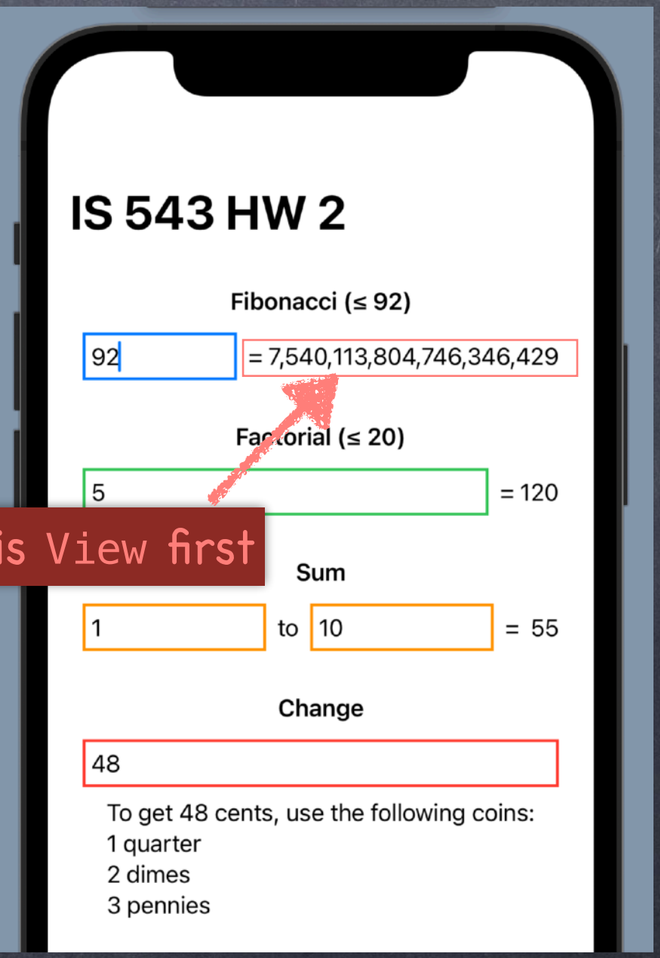
  If a Text doesn't get enough space, it will elide (e.g. "Swift is..." instead of "Swift is great!")

Layout

- In the HW 2 solution (which I forgot to post until today):

```
25      @State private var fibonacciString = "10"
26      @State private var factorialString = "5"
27      @State private var startString = "1"
28      @State private var endString = "10"
29      @State private var changeString = "48"
30
31      private let textFieldPadding = 6.0
32      private let textFieldBorderWidth: CGFloat = 2
33
34      var body: some View {
35          NavigationView {
36              ScrollView {
37                  VStack {
38                      Text("Fibonacci (≤ 92)").font(.headline)
39                      HStack {
40                          TextField("Fibonacci", text: $fib
41                              .padding(textFieldPadding)
42                              .border(.blue, width: tex ieldBorderWidth)
43                          Text("= \(fibonacci(fibonacciString.asInteger(limit: 92)))")
44                              .layoutPriority(1)
45                      }
46
47                      Text("\nFactorial (≤ 20)").font(.headline)
48                      HStack {
49                          TextField("Factorial", text: $factorialString)
50                              .padding(textFieldPadding)
51                              .border(.green, width: textFieldBorderWidth)
52                          Text("= \(factorial(factorialString.asInteger(limit: 20)))")
```

Allocates space to this View first



Layout

- `HStack` and `VStack`

  Another crucial aspect of the way stacks lay out the `Views` they contain is alignment
  When a `VStack` lays `Views` out in a column, what if the `Views` are not all the same width?
  Does it "left align" them? Or center them? Or what?

  This is specified via an argument to the stack …
  `VStack(alignment: .leading) { ... }`

  Why `.leading` instead of `.left`?
  Stacks automatically adjust for environments where text is right-to-left (e.g. Arabic or Hebrew)
  The `.leading` alignment lines the things in the `VStack` up to the edge where text starts from
  Text baselines can also be used to align (e.g. `HStack(alignment: .firstTextBaseline) { }`)

  You can even define your own "things to line up" alignment guides
  But that's a bit beyond the scope of this course
  So we're just going to use the built-ins (text baselines, `.center`, `.top`, `.trailing`, etc.)

Layout

- Modifiers

  Remember that `View` modifier functions (like `.padding`) themselves return a `View`

  That `View`, for all intents and purposes, "contains" the `View` it's modifying

  Many of them just pass the size offered to them along (like `.font` or `.foregroundColor`)

  But it is possible for a modifier to be involved in the layout process itself

  For example the `View` returned by `.padding(10)` will offer the `View` it's modifying a space that is the same size as it was offered, but reduced by 10 points on each side

  The `View` returned by `.padding(10)` would then choose a size for itself which is 10 points larger on all sides than the `View` it is modifying ended up choosing

  Another example is a modifier you've already used (hopefully): `.aspectRatio`

  The View returned by the `.aspectRatio` modifier takes the space offered to it and picks a size for itself that is either smaller (`.fit`) to respect the ratio or bigger (`.fill`) to use all the offered space (and more, potentially) while respecting the ratio

  (Yes, a `View` is allowed to choose a size for itself that is larger than the space it was offered!)

  It then offers the space it chose to the `View` it is modifying (again, acting as its "container")

Layout

- Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
    ForEach(emojiGame.cards) { card in
        CardView(card: card).aspectRatio(2/3, contentMode: .fit)
    }
}
.foregroundColor(.blue)
.padding(10)
```

The first `View` to be offered space here will be the `View` made by `.padding(10)`
Which will offer what it was offered minus 10 on all sides to the `View` from `.foregroundColor`
Which will in turn offer all of that space to the `HStack`
Which will then divide its space equally among the .aspectRatio `Views` in the `ForEach`
Each `.aspectRatio View` will set its width to be its share of the `HStack`'s width and pick a height for itself that respects the requested 2/3 aspect ratio
Or it might be forced to take all of the offered height and choose its width using the ratio (Whichever fits)
The `.aspectRatio` then offers all of its chosen size to its `CardView`, which will use it all

Layout

- Example

```
HStack { // aside: the default alignment here is .center (not .top, for example)
    ForEach(viewModel.cards) { card in
        CardView(card: card).aspectRatio(2/3, contentMode: .fit)
    }
}
.foregroundColor(Color.blue)
.padding(10)
```

The size this whole `View` (i.e. the one returned from `.padding(10)`) chooses for itself will be the result of the `HStack` sizing itself to fit those `.aspectRatio` `View`s + 10 points on all sides

Layout

- `Views` that take all the space offered to them

  Most `Views` simply size themselves to take up all the space offered to them
  For example, shapes usually draw themselves to fit (like `RoundedRectangle`)

  Custom `Views` should do this too whenever sensible
  But they really should adapt themselves to any space offered to look as good as possible
  For example, `CardView` in a card game would want to pick a font size that fills the space

  So how does a `View` know what space was offered to it so it can try to adapt?
  Using a special `View` called a `GeometryReader`

Layout

- GeometryReader

  You wrap this GeometryReader View around what would normally appear in your View's body:

```
var body: View {
    GeometryReader { geometry in
        ...
    }
}
```

  The geometry parameter is a GeometryProxy

```
struct GeometryProxy {
    var size: CGSize
    func frame(in: CoordinateSpace) -> CGRect
    var safeAreaInsets: EdgeInsets
}
```

  The size var is the amount of space that is being "offered" to us by our container

  Now we can, for example, pick a font size appropriate to that sized space

  GeometryReader itself (it's just a View) always accepts all the space offered to it

Layout

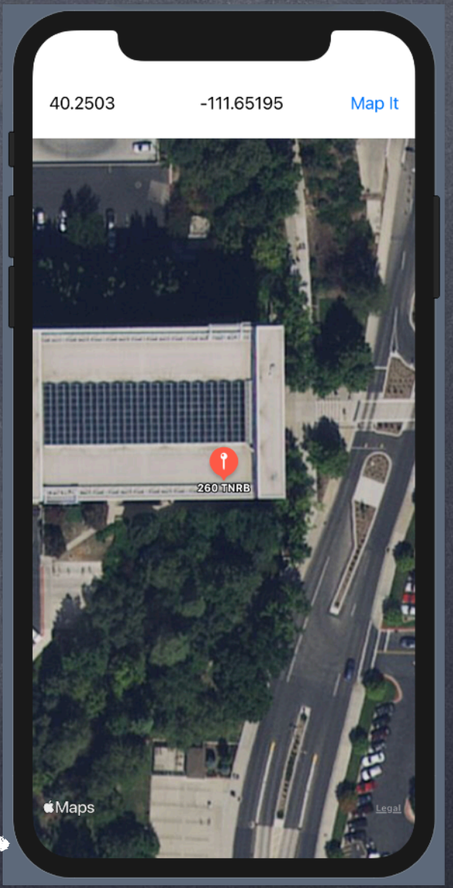
- Safe Area

  Generally, when a `View` is offered space, that space doesn't include "safe areas"
  The most obvious "safe area" is the notch on an iPhone
  Surrounding `View`s might also introduce "safe areas" that `View`s inside shouldn't draw in

  But it is possible to ignore this and draw in those areas anyway on specified edges:

```
ZStack { ... }.edgesIgnoringSafeArea([.top])
               // draw in "safe area" on top edge
```

  We ignored all safe area edges in Map It earlier:
  `.edgesIgnoringSafeArea([.top, .bottom])` or
  `.edgesIgnoringSafeArea(.all)`



Layout

- Containers

  How exactly do container `Views` "offer" space to the `Views` they contain?
  With the modifier `.frame(...)`
  This `.frame` modifier has a lot of arguments you can check out in the documentation

  Once a `View` chooses the size it wants from what's offered, the container must then position it
  It does this with the `.position(CGPoint)` modifier
  The position is the center of the subview in the container's coordinate space

  Stacks would use their alignment and spacing to figure this `CGPoint` out for each `View`
  You can also offset a `View` from where its container put it using `.offset(CGSize)` modifier

Using SwiftUI Grid Layouts

- Two types of grid: `LazyHGrid` and `LazyVGrid`

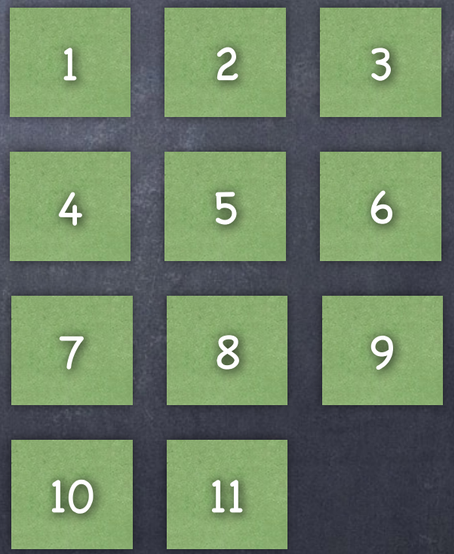
  Similar `Views`, but "H" variant grows horizontally whereas "V" variant grows vertically

  In other words, for a `LazyHGrid` you say how many fixed rows you have, and it will fit all items into that number of rows (one column at a time, top-to-bottom, left-to-right)

  And for `LazyVGrid`, you say how many columns there are and it fits items into that number of columns (one row at a time, left-to-right, top-to-bottom)

`LazyHGrid`

`LazyVGrid`



Using SwiftUI Grid Layouts

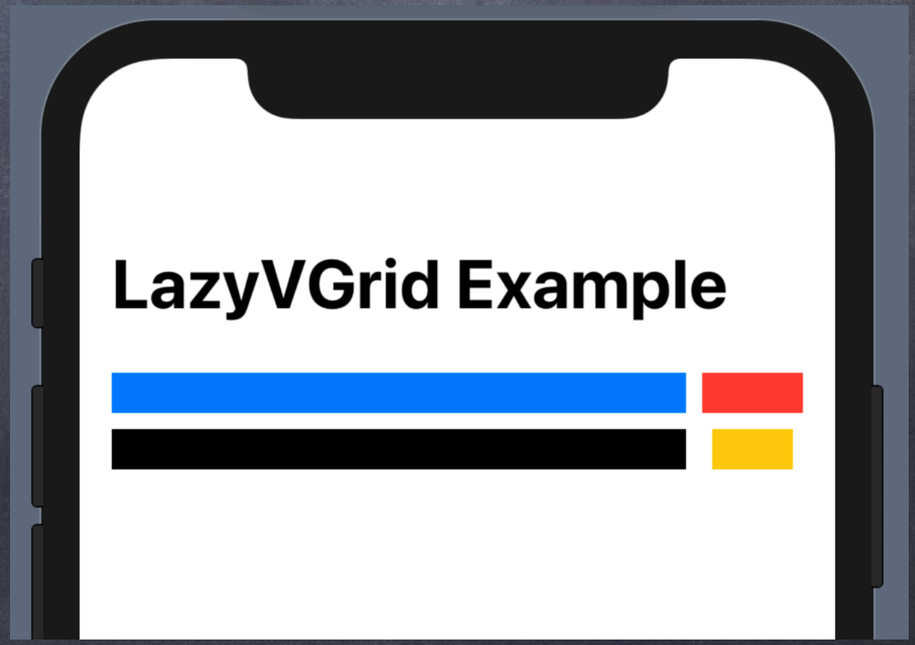
- How to use `LazyVGrid`

  It's a container `View` a lot like `ZStack`, `HStack`, `VStack`, `ForEach`

  (Specify `LazyVGrid`'s content in a similar manner)

  Must indicate the column sizing specifications by passing an `Array of GridItems`

```
LazyVGrid(columns: [.init(.flexible()), .init(.fixed(50))]) {
    Rectangle()
        .frame(height: 20)
        .foregroundColor(.blue)
    Rectangle()
        .foregroundColor(.red)
    Rectangle()
        .foregroundColor(.black)
    Rectangle()
        .frame(width: 40, height: 20)
        .foregroundColor(.yellow)
}
```



Using SwiftUI Grid Layouts

- Details of `LazyVGrid`

  `columns`: `Array` of `GridItem`s to size and position each column of the grid

  `alignment`: horizontal alignment of the grid within its parent `View`
  e.g. `.leading`, `.trailing`, or the default `.center`

  `spacing`: number of points of spacing between each row of items

  `pinnedViews`: `View`s to pin while scrolling; sometimes called "sticky headers" or "footers"
  In a contact list, may show "A" while scrolling through the A's, "B" when you get to the B's
  To use this, define the `content` of the grid as a `Section` with a `HeaderView`
  (you can Google for code to demonstrate this)

  `content`: a `ViewBuilder` to provide the content `View`s of this grid
  Quite often, a `ForEach`

Using SwiftUI Grid Layouts

- Details of `GridItem`

`alignment`: alignment for content of this `GridItem` (in case there is extra space)
e.g. `.leading`, `.trailing`, `.top`, `.bottom`, `.center`, etc.

`spacing`: number of points of spacing between this item and the next
Note that the grid itself has its own distinct `spacing` parameter, controlling spacing along the main axis (`LazyVGrid`'s main axis is vertical, `LazyHGrid`'s is horizontal)

`size`: `GridItem.Size` is one of:

```
.flexible(minimum: CGFloat, maximum: CGFloat)
    // Allocate between minimum and maximum points, as appropriate (both optional)

.fixed(CGFloat)
    // Allocate a fixed amount of space (in points)

.adaptive(minimum: CGFloat, maximum: CGFloat)
    // Fit multiple items in the space of a single flexible item
```

Demo Time

- Can we improve our use of `LazyVGrid` for the calculator example?

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 case may have (but is not required to have) an "associated data" value

```
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 two 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

- Setting the value of an `enum`

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

```
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)

```
let menuItem: FastFoodMenuItem = FastFoodMenuItem.hamburger(patties: 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.hamburger(patties: 2)  // This works
var otherItem: FastFoodMenuItem = .cookie              // This too

var yetAnotherItem = .cookie      // The compiler can't figure this out
```

enum

- Checking an enum's state

  We generally check an enum's state with a switch statement not if/then/else

```
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" in the switch above (so far)

enum

- Checking an enum's state

  We check an enum's state with a switch statement

```
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

  We check an enum's state with a switch statement

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

  Because the Swift compiler can infer the type, we usually write e..g. `.fries` instead of `FastFoodMenuItem.fries` inside a switch statement

enum

- break

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

```
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)

```
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)

```
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

- What about associated data?

  Associated data is accessed in a `switch` statement by 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 associated data?

  Associated data is accessed in a `switch` statement by 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 associated data?

  Associated data is accessed in a `switch` statement by 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 `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`

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 // drink & 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 // drink & 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, 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 // drink & cookie in every special order
            case .drink(_, let ounces): return ounces == 16 // & 16oz drink of any kind
        }
    }
}
```

Note the use of `_` if we don't care about that piece of associated data

enum

- Getting all the cases of an enumeration (adopt `CaseIterable` protocol)

```
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

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

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

Optionals

- Optional

  An Optional is just an enum

  Period, end of sentence, that's all it is

  It looks like this:

```
enum Optional<T> {    // A generic type like Array<Element>
    case none
    case some(<T>)    // The some case has associated values 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 as well (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?
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?`

```
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
```

Optionals

- Optional

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

  You can then assign it the value `nil` (i.e. `Optional.none`)

```
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
```

Optionals

- Optional

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

  You can then assign it the value `nil` (i.e. `Optional.none`)

  Or you can assign it something of type `T` (`Optional.some` with associated data value of type `T`)

```
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
```

Optionals

- Optional

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

  You can then assign it the value `nil` (i.e. `Optional.none`)

  Or you can assign it something of type `T` (`Optional.some` with associated data value of type `T`)

  Note that `Optional` vars always start out with an implicit `= nil` assignment

```
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
```

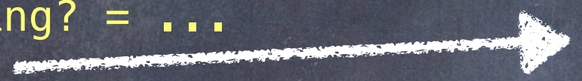
Optionals

- Optional

  You can access the associated value either by "force-unwrapping" (with !) ...

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

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



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

Optionals

- Optional

  You can access the associated value either by "force-unwrapping" (with !) ...

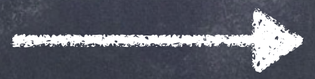
  Or "safely" using if let and then using the safely-retrieved associated value in { } (else allowed too)

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

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

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

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



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

Optionals

- Optional

  The "nil-coalescing operator" ?? gives an Optional expression a default value

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

let x: String? = ...

let y = x ?? "some default"
```

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

Homework 5 for Tuesday

- See full description on Learning Suite
  Watch the additional videos I've posted on YouTube.
  Follow along and implement the code for the pushup tracker app.

  It's time for AI! Write a series of prompts to inquire about functional programming in Swift. Determine what the primary concepts of functional programming in Swift are. Explore enough to feel more confident about functional programming. Also ask about the singleton pattern as well.

  Submit a PDF document containing either screenshots or copy/paste text showing your prompts and the AI tool's responses. Write a paragraph summarizing what you learned by doing this homework. I don't need to see screenshots of your updated pushup tracker app.