protocol

- A protocol is sort of a "stripped-down" struct/class
  - It has functions and vars, but no implementation (or storage)!



protocol

- A protocol is sort of a "stripped-down" struct/class

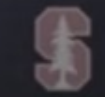
  It has functions and vars, but no implementation (or storage)!

  Declaring a protocol looks very similar to struct or class (just w/o implementation) ...

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  See? No implementation.

  The { } on the vars just say whether it's read only or a var whose value can also be set.



protocol

- A protocol is sort of a "stripped-down" struct/class

  It has functions and vars, but no implementation (or storage)!

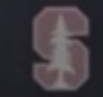
  Declaring a protocol looks very similar to struct or class (just w/o implementation) ...

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  ... then any other type can claim to implement Moveable ...

```
struct PortableThing: Moveable {
    // must implement move(by:), hasMoved and distanceFromStart here
}
```

  PortableThing conforms to (aka "behaves like a") Moveable



protocol

- A protocol is sort of a "stripped-down" struct/class

  It has functions and vars, but no implementation (or storage)!

  Declaring a protocol looks very similar to struct or class (just w/o implementation) ...

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  ... and this is also legal (this is called "protocol inheritance") ...

```
protocol Vehicle: Moveable {
    var passengerCount: Int { get set }
}
class Car: Vehicle {
    // must implement move(by:), hasMoved, distanceFromStart and passengerCount here
}
```



protocol

- A protocol is sort of a "stripped-down" struct/class
  - It has functions and vars, but no implementation (or storage)!
  - Declaring a protocol looks very similar to struct or class (just w/o implementation) …

```
protocol Moveable {
    func move(by: Int)
    var hasMoved: Bool { get }
    var distanceFromStart: Int { get set }
}
```

  - … and you can claim to implement multiple protocols …

```
class Car: Vehicle, Impoundable, Leasable {
    // must implement move(by:), hasMoved, distanceFromStart and passengerCount here
    // and must implement any funcs/vars in Impoundable and Leasable too
}
```

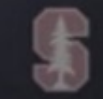


protocol

- What is a protocol used for?

  protocols can be used in lots of different ways.

  Let's go through some of them ...



protocol

- What is a protocol used for?

  Very rarely, a protocol can be used as a type (in any other circumstance a type can be used).

  Not all protocols can be used this way (View can't, nor Equatable, nor Identifiable).



protocol

- What is a protocol used for?

  Very rarely, a protocol can be used as a type (in any other circumstance a type can be used).
  Not all protocols can be used this way (View can't, nor Equatable, nor Identifiable).

```
func travelAround(using moveable: Moveable) // can pass a Car or PortableThing in here
let foo = [Moveable] // this Array can now contain Cars and also PortableThings
```



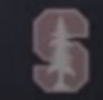
protocol

- What is a protocol used for?

  Very rarely, a protocol can be used as a type (in any other circumstance a type can be used).
  Not all protocols can be used this way (View can't, nor Equatable, nor Identifiable).

```
func travelAround(using moveable: Moveable) // can pass a Car or PortableThing in here
let foo = [Moveable] // this Array can now contain Cars and also PortableThings
```

  We may not see a single use of protocols in this way in this course.



protocol

- What is a `protocol` used for?

  A much more common usage is to specify the behavior of a `struct`, `class` or `enum`.



protocol

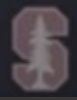
- What is a protocol used for?

  A much more common usage is to specify the behavior of a struct, class or enum.

  ```
  struct EmojiMemoryGameView: View
  ```

  Just by doing this, EmojiMemoryGameView became a very powerful struct!

  Of course, it did have to implement var body to participate in being a View, but still.



protocol

- What is a `protocol` used for?

  A much more common usage is to specify the behavior of a `struct`, `class` or `enum`.

  ```
  struct EmojiMemoryGameView: View
  ```

  Just by doing this, `EmojiMemoryGameView` became a very powerful `struct`!

  Of course, it did have to implement `var body` to participate in being a `View`, but still.

  Another example ...

  ```
  class EmojiMemoryGame: ObservableObject
  ```



protocol

- What is a protocol used for?

  A much more common usage is to specify the behavior of a struct, class or enum.

  ```
  struct EmojiMemoryGameView: View
  ```

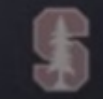
  Just by doing this, EmojiMemoryGameView became a very powerful struct!

  Of course, it did have to implement var body to participate in being a View, but still.

  Another example …

  ```
  class EmojiMemoryGame: ObservableObject
  ```

  Later we'll see similar usage with protocols like ViewModifier, Shape, App & Scene.



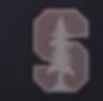
protocol

- What is a `protocol` used for?

  There are a bunch of very often used, lightweight behaviors as well.

  Examples: `Identifiable`, `Hashable`, `Equatable`, `CustomStringConvertible`.

  Later we'll see some more specialized ones like `Animatable`.



protocol

- What is a `protocol` used for?

  There are a bunch of very often used, lightweight behaviors as well.

  Examples: `Identifiable`, `Hashable`, `Equatable`, `CustomStringConvertible`.

  Later we'll see some more specialized ones like `Animatable`.



protocol

- What is a `protocol` used for?

  Another use we've already seen is turning "don't cares" into "somewhat cares".



protocol

- What is a protocol used for?

  Another use we've already seen is turning "don't cares" into "somewhat cares".

  ```
  struct MemoryGame<CardContent> where CardContent: Equatable
  ```

  This is at the heart of "protocol-oriented programming".
  We'll see more of doing this in the demo today.



protocol

- What is a protocol used for?

  We can also use protocols to restrict an extension to work only with certain things.



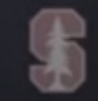
protocol

- What is a protocol used for?

  We can also use protocols to restrict an extension to work only with certain things.

  ```
  extension Array where Element: Hashable { … }
  ```

  Whatever funcs we add via this extension will only be available for certain Arrays.
  Namely those whose elements conform to (aka behaves like, aka implement) Hashable.
  For all other Arrays, it'll be as if this extension does not even exist.



protocol

- What is a protocol used for?

  We can also use protocols to restrict individual functions to work only with certain things.



protocol

- What is a protocol used for?

  We can also use protocols to restrict individual functions to work only with certain things.

  ```
  init(data: Data) where Data: Collection, Data.Element: Identifiable
  ```

  Data is a don't care on whatever struct this init is in.

  You can't even create this thing unless the data you provide is a Collection of Identifiables.

  Of course, there might be other inits in this struct that have other constraints.



protocol

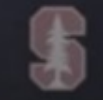
- What is a protocol used for?

  Occasionally a protocol is used to set up an agreement between two entities.

  Example: DropDelegate

  The system lets any struct/class that conforms to DropDelegate participate in drag & drop.

  The DropDelegate protocol makes it clear to that data structure what its responsibilities are (i.e., the data structure must implement the funcs/vars in the DropDelegate protocol).



protocol

- What is a protocol used for?

  One of the most powerful uses of protocols is to facilitate code sharing.



protocol

- What is a protocol used for?

  One of the most powerful uses of protocols is to facilitate code sharing.

  Implementation can be added to a protocol by creating an extension to it.



protocol

- What is a protocol used for?

  One of the most powerful uses of protocols is to facilitate code sharing.

  Implementation can be added to a protocol by creating an extension to it.

  This is how Views get foregroundColor and font and all their other modifiers!



protocol

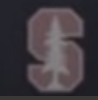
- What is a protocol used for?

  One of the most powerful uses of protocols is to facilitate code sharing.

  Implementation can be added to a protocol by creating an extension to it.

  This is how Views get foregroundColor and font and all their other modifiers!

  This is also how functions like filter and firstIndex(where:) get implemented.



protocol

- What is a protocol used for?

  One of the most powerful uses of protocols is to facilitate code sharing.

  Implementation can be added to a protocol by creating an extension to it.

  This is how Views get foregroundColor and font and all their other modifiers!

  This is also how functions like filter and firstIndex(where:) get implemented.

  An extension can also add a default implementation for a func or var in the protocol.
  (That's how ObservableObjects get objectWillChange for free.)



protocol

- What is a protocol used for?

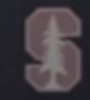
  One of the most powerful uses of protocols is to facilitate code sharing.
  Implementation can be added to a protocol by creating an extension to it.

  This is how Views get foregroundColor and font and all their other modifiers!

  This is also how functions like filter and firstIndex(where:) get implemented.

  An extension can also add a default implementation for a func or var in the protocol.
  (That's how ObservableObjects get objectWillChange for free.)

  Adding extensions to protocols is key to protocol-oriented programming in Swift.

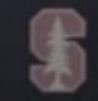


protocol

- What is a protocol used for?

  Adding filter to Array, Range, String, Dictionary, et. al. ...

```
func filter(_ isIncluded: (Element) -> Bool) -> Array<Element>
```



protocol

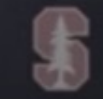
- What is a protocol used for?

  Adding filter to Array, Range, String, Dictionary, et. al. ...

  ```
  func filter(_ isIncluded: (Element) -> Bool) -> Array<Element>
  ```

  This function was written once by Apple.

  And yet it works on Array and Range and String and Dictionary and others!



protocol

- What is a `protocol` used for?

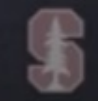
  Adding filter to `Array`, `Range`, `String`, `Dictionary`, et. al. ...

  ```
  func filter(_ isIncluded: (Element) -> Bool) -> Array<Element>
  ```

  This function was written once by Apple.

  And yet it works on `Array` and `Range` and `String` and `Dictionary` and others!

  `filter` was added to the Foundation library as an `extension` to the `Sequence` protocol.



protocol

- What is a protocol used for?

  Adding filter to Array, Range, String, Dictionary, et. al. …

  ```
  func filter(_ isIncluded: (Element) -> Bool) -> Array<Element>
  ```

  This function was written once by Apple.

  And yet it works on Array and Range and String and Dictionary and others!

  filter was added to the Foundation library as an extension to the Sequence protocol.

  Array, Range, String, Dictionary all conform to the Sequence protocol.

  So they all get the filter function that was added to Sequence via that extension.



protocol

- What is a protocol used for?

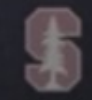
  Adding filter to Array, Range, String, Dictionary, et. al. …

  ```
  func filter(_ isIncluded: (Element) -> Bool) -> Array<Element>
  ```

  This function was written once by Apple.
  And yet it works on Array and Range and String and Dictionary and others!

  filter was added to the Foundation library as an extension to the Sequence protocol.
  Array, Range, String, Dictionary all conform to the Sequence protocol.
  So they all get the filter function that was added to Sequence via that extension.

  In OOP, we'd have had to make all those things inherit from some common base class.
  That would have been pretty awkward and restrictive to our class design.



protocol extension

- Adding protocol implementation

  One way to think about protocols is constrains and gains ...

```
struct Tesla: Vehicle {
    // Tesla is constrained to have to implement everything in Vehicle
    // but it gains all the capabilities a Vehicle has too
}
```



protocol extension

- Adding protocol implementation

One way to think about protocols is constrains and gains ...

```
struct Tesla: Vehicle {
    // Tesla is constrained to have to implement everything in Vehicle
    // but it gains all the capabilities a Vehicle has too
}
```

Things that conform to Vehicle "gain capabilities" via an extension to the Vehicle protocol.

```
extension Vehicle {
    func registerWithDMV() { /* actual implementation here */ }
}
```

Now Teslas (and all other Vehicles) have gained to ability to be registered with the DMV.



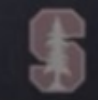
protocol

- A protocol in action

  You've already seen protocols with extensions in action. For example ...

```
struct CardView: View {
    var body: some View { ... }
}
```

  Here CardView is declaring that it implements the protocol View.



protocol

- A protocol in action

  You've already seen protocols with extensions in action. For example ...

```
struct CardView: View {
    var body: some View { ... }
}
```

  Here CardView is declaring that it implements the protocol View.

  Since CardView claims to implement this protocol ...

  ... it must (i.e. is constrained to) implement all of the View protocol's funcs/vars.

  Luckily, the only thing that needs implementation in the View protocol is body.

  So once CardView implements body, it now fully "conforms" to View.



protocol

- A protocol in action

  You've already seen protocols with extensions in action. For example ...

```
struct CardView: View {
    var body: some View { ... }
}
```

  Here CardView is declaring that it implements the protocol View.

  Since CardView claims to implement this protocol ...

  ... it must (i.e. is constrained to) implement all of the View protocol's funcs/vars.

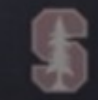
  Luckily, the only thing that needs implementation in the View protocol is body.

  So once CardView implements body, it now fully "conforms" to View.

  And SwiftUI has added a whole ton of other funcs/vars to View via extension.

  So CardView gains the ability to respond to any of those funcs/vars too.

  That's why we can set its foregroundColor, put padding around it ... whatever we want.

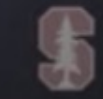


protocol

- View

  Somewhere in SwiftUI there's something (sort of) like this going on ...

```
protocol View {
    var body: some View
}
```



protocol

- View

Somewhere in SwiftUI there's something (sort of) like this going on …

```
protocol View {
    var body: some View
}
```

And there's also something like this in SwiftUI …

```
extension View {
    func foregroundColor(_ color: Color) -> some View { /* implementation */ }
    func font(_ font: Font?) -> some View { /* implementation */ }
    func blur(radius: CGFloat, opaque: Bool) -> some View { /* implementation */ }
    … and many more …
}
```

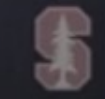
The first part constrains something like CardView to have to provide that body var.
The second part gives many gains to CardView as a reward for doing so!
Obviously the above is a great oversimplification, but conceptually that's what's happening.



protocol

- Why protocols?

  Why do we do all this protocol stuff?



protocol

- Why protocols?

  Why do we do all this protocol stuff?

  It is a way for types (structs/classes/other protocols) to say what they are capable of.

  And also for other code to demand certain behavior out of another type.



protocol

- Why protocols?

  Why do we do all this protocol stuff?

  It is a way for types (structs/classes/other protocols) to say what they are capable of.

  And also for other code to demand certain behavior out of another type.

  But neither side has to reveal what sort of struct or class they are.

  We focus on the functionality and hide the implementation details behind it.

  It's the promise of encapsulation from OOP but taken to a higher level.



protocol

- Why protocols?

  Why do we do all this protocol stuff?
  It is a way for types (structs/classes/other protocols) to say what they are capable of.
  And also for other code to demand certain behavior out of another type.
  But neither side has to reveal what sort of struct or class they are.
  We focus on the functionality and hide the implementation details behind it.
  It's the promise of encapsulation from OOP but taken to a higher level.

  Let's make one more pass through all this by taking a closer look at Identifiable
  (which will lead us to Hashable which will lead us to Equatable)
  Each of these 3 protocols has something to teach us about protocols.



protocol

- Generics + Protocols

  Consider the protocol we already know called Identifiable …

```
protocol Identifiable {
    var id: ID { get }
}
```



protocol

- Generics + Protocols

  Consider the protocol we already know called Identifiable …

```
protocol Identifiable {
    var id: ID { get }
}
```

  The type ID here is a "don't care" for Identifiable.

  Yes, protocols can be generic (i.e. have "don't cares") too!

  i.e. you can use any type of thing as the identifier to make something Identifiable.

  protocols, however, do not declare their "don't care" types in the same way as a struct does …

```
protocol Identifiable {
    associatedtype ID
    var id: ID { get }
}
```

  You'd expect this to be protocol Identifiable<ID>, but it's not.



protocol

- Generics + Protocols

```
protocol Identifiable {
    associatedtype ID
    var id: ID { get }
}
```

As we learned in demo earlier, this type `ID` also has to be `Hashable`.
(Luckily `Int`, the type we used for `ID` in `MemoryGame.Card` is `Hashable`.)
That's because we want to be able to look up these `Identifiable` things in hash tables.
It'd be kind of useless for something to be `Identifiable` but not "lookupable".



protocol

- Generics + Protocols

  So how do we enforce that (i.e. that ID is some type of thing that can be hashed)?

  By simply constraining the ID "don't care" to also "behave like" a Hashable.

  Hashable is just yet another protocol we'll look at in a minute.

```
protocol Identifiable {
    associatedtype ID where ID: Hashable
    var id: ID { get }
}
```

  Or, more simply ...

```
protocol Identifiable {
    associatedtype ID: Hashable
    var id: ID { get }
}
```

  We've just turned ID from a "don't care" into a "we care a little bit".

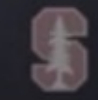


protocol

- Hashable

  Hashable is a simple protocol …

```
protocol Hashable {
    func hash(into hasher: inout Hasher) // a Hasher just has a combine(…) func on it
}
```



protocol

- Hashable

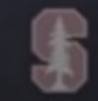
  Hashable is a simple protocol …

```
protocol Hashable {
    func hash(into hasher: inout Hasher) // a Hasher just has a combine(…) func on it
}
```

  Here's a struct that implements Hashable.

  Don't worry about the inout keyword for now (you'll read about it in your reading assignments).

```
struct Foo: Hashable {
    var i: Int
    var s: String
    func hash(into hasher: inout Hasher) {
        hasher.combine(i)
        hasher.combine(s)
    }
}
```



protocol

- Protocol Inheritance

  But there's something subtle to consider about hashing things.

```
protocol Hashable {
    func hash(into hasher: inout Hasher)
}
```

  If you hash something and use that hash to look it up in a hash table, then
  You have to be able to then use == on the thing to make sure it's the same thing you hashed.
  That's because it's possible (though rare) for two different things to hash to the same value.



protocol

- Protocol Inheritance

  But there's something subtle to consider about hashing things.

```
protocol Hashable {
    func hash(into hasher: inout Hasher)
}
```

  If you hash something and use that hash to look it up in a hash table, then
  You have to be able to then use == on the thing to make sure it's the same thing you hashed.
  That's because it's possible (though rare) for two different things to hash to the same value.

  Therefore things that are Hashable must also be Equatable.
  So the Hashable protocol defines itself to say that anything that behaves like a Hashable
  Must also "behave like" an Equatable ...

```
protocol Hashable: Equatable {
    func hash(into hasher: inout Hasher)
}
```

  This is "protocol inheritance". Note that this is different than class inheritance.



protocol

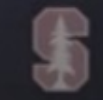
- Equatable

  So what is in the `Equatable` protocol?

  It's pretty cool ...

  Things that "behave like" `Equatable` can be compared with `==`.

  Swift knows to call this `static` function in the `Equatable` protocol when you do `==` ...



protocol

- Equatable

  So what is in the `Equatable` protocol?

  It's pretty cool ...

  Things that "behave like" `Equatable` can be compared with `==`.

  Swift knows to call this `static` function in the `Equatable` protocol when you do `==` ...

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

  Yes, it's legal for a `func` name in Swift to be `==` (🔥 is also a legal Swift `func` name!).



protocol

- Equatable

  So what is in the Equatable protocol?

  It's pretty cool ...

  Things that "behave like" Equatable can be compared with ==.

  Swift knows to call this static function in the Equatable protocol when you do == ...

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

  Yes, it's legal for a func name in Swift to be == (🔥 is also a legal Swift func name!).

  The type Self above means "the actual type that is implementing Equatable".

  When you implement Equatable in a struct, you replace Self with your struct's type.

  This kind of Self-referencing protocol cannot be used as a normal type.

  In other words, var x: [Equatable] makes no sense whatsoever.

  There'd be no way to equate two completely different kinds of structs.

  Since their == function above wouldn't be able to take the two different types as arguments.

  This is also why we can never do var myView: View (View is Self-referencing as well).



protocol

- Equatable

  Swift is even friendlier than just making == call that static function for us.

  If you declare a struct to be Equatable

  And all of its vars are themselves Equatable

  Then Swift will implement the static == func for you!

```
struct Card: Identifiable, Equatable {
    var isFaceUp: Bool
    var isMatched: Bool
    var content: CardContent
    var id: Int
}
```

  Now card1 == card2 will be legal.

  Even though we never implemented the static == func in Card.

  Probably not that useful in the case of a Card, but demonstrative.



protocol

- Equatable

  Swift is even friendlier than just making `==` call that `static` function for us.

  If you declare a `struct` to be `Equatable`

  And all of its vars are themselves `Equatable`

  Then Swift will implement the `static ==` func for you!

```
struct Card: Identifiable, Equatable {
    var isFaceUp: Bool
    var isMatched: Bool
    var content: CardContent
    var id: Int
}
```

  Now `card1 == card2` will be legal.

  Even though we never implemented the `static ==` func in `Card`.

  Probably not that useful in the case of a `Card`, but demonstrative.

  By the way, you can add `Equatable` conformance via Command-click menu in Xcode.

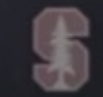


Generics and Protocols

- Help!

  Some of you might be shivering a bit right now.

  You might be thinking, "how am I going to design systems using generics/protocols?!"



Generics and Protocols

- Help!

  Some of you might be shivering a bit right now.
  You might be thinking, "how am I going to design systems using generics/protocols?!"
  This is, indeed, a powerful foundation for designing things.
  But functional programming does require some mastery that only comes with experience.
  The good news: you can do a lot in SwiftUI without having to master functional programming.
  You can see that Swift does a lot of the stuff for us.
  And the more you use it, the more you'll be "grokking" it.
  That's why I explain to you up front what's going on.
  No one expects you to now be able to be adding `extensions` to `protocols` with generics!
  But eventually you will be able to.
  And in the meantime, you'll have at least some idea how SwiftUI is built.
  As usual, this all might make a lot more sense with a demo ...

