Functions as Types

- Functions are people* too! (* er, types)
  - You can declare a variable (or parameter to a func or whatever) to be of type "function".
  - The syntax for this includes the types of the arguments and return value.
  - You can do this anywhere any other type is allowed.



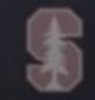
Functions as Types

- Functions are people* too! (* er, types)

  You can declare a variable (or parameter to a func or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool  // takes two Ints and returns a Bool
```



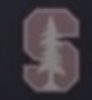
Functions as Types

- Functions are people* too! (* er, types)

  You can declare a variable (or parameter to a `func` or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool   // takes two Ints and returns a Bool
(Double) -> Void     // takes a Double and returns nothing
```



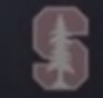
Functions as Types

- Functions are people* too! (* er, types)

  You can declare a variable (or parameter to a func or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool   // takes two Ints and returns a Bool
(Double) -> Void     // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
```



Functions as Types

- Functions are people* too! (* er, types)

  You can declare a variable (or parameter to a `func` or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool   // takes two Ints and returns a Bool
(Double) -> Void     // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
() -> Void           // takes no arguments and returns nothing (this is a common one)
```

  All of the above a just types. No different than `Bool` or `View` or `Array<Int>`. All are types.



Functions as Types

- Functions are people* too! (* er, types)

  You can declare a variable (or parameter to a func or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool   // takes two Ints and returns a Bool
(Double) -> Void     // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
() -> Void           // takes no arguments and returns nothing (this is a common one)
```

  All of the above a just types. No different than Bool or View or Array<Int>. All are types.

```
var foo: (Double) -> Void // foo's type: "function that takes a Double, returns nothing"
```



Functions as Types

- Functions are people* too! (* er, types)

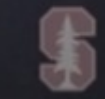
  You can declare a variable (or parameter to a func or whatever) to be of type "function".
  The syntax for this includes the types of the arguments and return value.
  You can do this anywhere any other type is allowed.

  Examples ...

```
(Int, Int) -> Bool   // takes two Ints and returns a Bool
(Double) -> Void     // takes a Double and returns nothing
() -> Array<String> // takes no arguments and returns an Array of Strings
() -> Void           // takes no arguments and returns nothing (this is a common one)
```

  All of the above a just types. No different than Bool or View or Array<Int>. All are types.

```
var foo: (Double) -> Void // foo's type: "function that takes a Double, returns nothing"
func doSomething(what: () -> Bool) // what's type: "function, takes nothing, returns Bool"
```



Functions as Types

- Functions are people* too! (* er, types)

  Example …

  ```
  var operation: (Double) -> Double
  ```

  This is a var called `operation`.

  It is of type "function that takes a Double and returns a Double".



Functions as Types

- Functions are people* too! (* er, types)

  Example …

  ```
  var operation: (Double) -> Double
  ```

  This is a var called `operation`.

  It is of type "function that takes a Double and returns a Double".

  Here's a simple function that takes a Double and returns a Double …

  ```
  func square(operand: Double) -> Double {
      return operand * operand
  }
  ```



Functions as Types

- Functions are people* too! (* er, types)

  Example …

  ```
  var operation: (Double) -> Double
  ```

  This is a var called `operation`.

  It is of type "function that takes a Double and returns a Double".

  Here's a simple function that takes a Double and returns a Double …

  ```
  func square(operand: Double) -> Double {
      return operand * operand
  }

  operation = square // just assigning a value to the operation var, nothing more
  let result1 = operation(4) // result1 would equal 16
  ```

  Note that we don't use argument labels (e.g. `operand:`) when executing function types.



Functions as Types

- Functions are people* too! (* er, types)

  Example …

```
var operation: (Double) -> Double
```

  This is a var called operation.

  It is of type "function that takes a Double and returns a Double".

  Here's a simple function that takes a Double and returns a Double …

```
func square(operand: Double) -> Double {
    return operand * operand
}

operation = square // just assigning a value to the operation var, nothing more
let result1 = operation(4) // result1 would equal 16
```

  Note that we don't use argument labels (e.g. operand:) when executing function types.

```
operation = sqrt // sqrt is a built-in function which happens to take and return a Double
let result2 = operation(4) // result2 would be 2
```



Functions as Types

- Functions are people* too! (* er, types)

  Example …

  ```
  var operation: (Double) -> Double
  ```

  This is a var called `operation`.

  It is of type "function that takes a Double and returns a Double".

  Here's a simple function that takes a Double and returns a Double …

  ```
  func square(operand: Double) -> Double {
      return operand * operand
  }

  operation = square // just assigning a value to the operation var, nothing more
  let result1 = operation(4) // result1 would equal 16
  ```

  Note that we don't use argument labels (e.g. `operand:`) when executing function types.

  ```
  operation = sqrt // sqrt is a built-in function which happens to take and return a Double
  let result2 = operation(4) // result2 would be 2
  ```

  We'll soon see an example of using a function type for a parameter to a function in our demo.



Functions as Types

- Closures

  It's so common to pass functions around that we are very often "inlining" them.
  We call such an inlined function a "closure" and there's special language support for it.
  We'll cover this in the demo and again later in the quarter.

  Remember that we are mostly doing "functional programming" in SwiftUI.
  As the very name implies, "functions as types" is a very important concept in Swift. Very.

