Async programming: handlers
Async programming: anonymous functions
Async programming: function literals

Async programming: Closures

Async programming: lambdas
Async programming: blocks

by Fernando Olivares

Agenda

1-hour session

- Author Introduction
- Also why we're here: What is a closure?
- Live coding
- Thinking *in* closures
- Conclusion & Where to go from here
- Q&A

Author Introduction

Fernando

- ~10 years of experience
- Worked at small startups ( 1SecondEveryday) to publicly traded companies ( J2 Global Inc.)
- Instructor at  Big Nerd Ranch,  bloc.io,  Lambda School
- Won a few awards: The Storyteller Within (Apple), ERA Accelerator Top 10 (ERA NY)
- Product and Project experience
- iOS-only, bought the first iPhone without knowing if he'd be able to use it
- @fromJrToSr

What is a closure?

In theory, which is complex and boring but important.

- Official definition: "Closures are self-contained blocks of functionality that can be passed around and used in your code."[1]
- They are First-class citizens[2], so they can be passed as arguments, returned from a function, modified and assigned to constants or variables,
- Even though they're functions, they have slightly different rules than regular functions (no labels, custom syntax, capturing values, implicit returns)
- They are the building blocks of higher order functions (e.g. map, reduce) and are very common in asynchronous functions.

[1] - https://docs.swift.org/swift-book/LanguageGuide/Closures.html

[2] - https://en.wikipedia.org/wiki/First-class_citizen

What is a closure?

In practice, which is fun and exciting but nothing without the theory

- Fernando definition: "It's a function without labels and some frustrating exceptions."
- You can declare them, bop them, twist them, pull them, but their most common use cases are being a parameter and being called for a single-purpose, or being called sometime in the future.
- Most closures have special memory rules which is uncommon nowadays in Swift.
- Closures are not exceptional. They're as common as Arrays, Dictionaries and Strings. As soon as you demystify them, they become easy to understand.

Live Demo

High-order functions: Create our own

- `func map` that transforms an array with objects/structs of type A, into an array with objects/structs of type B
  - [lowercase String] -> [capitalized String]
  - [lowercase String] -> [uppercase String]
  - DRY with a function
  - DRY with a closure
  - map

Live Demo

Create a function that transforms elements in an array

```
func transform(originalStrings: [String]) -> [String]
```

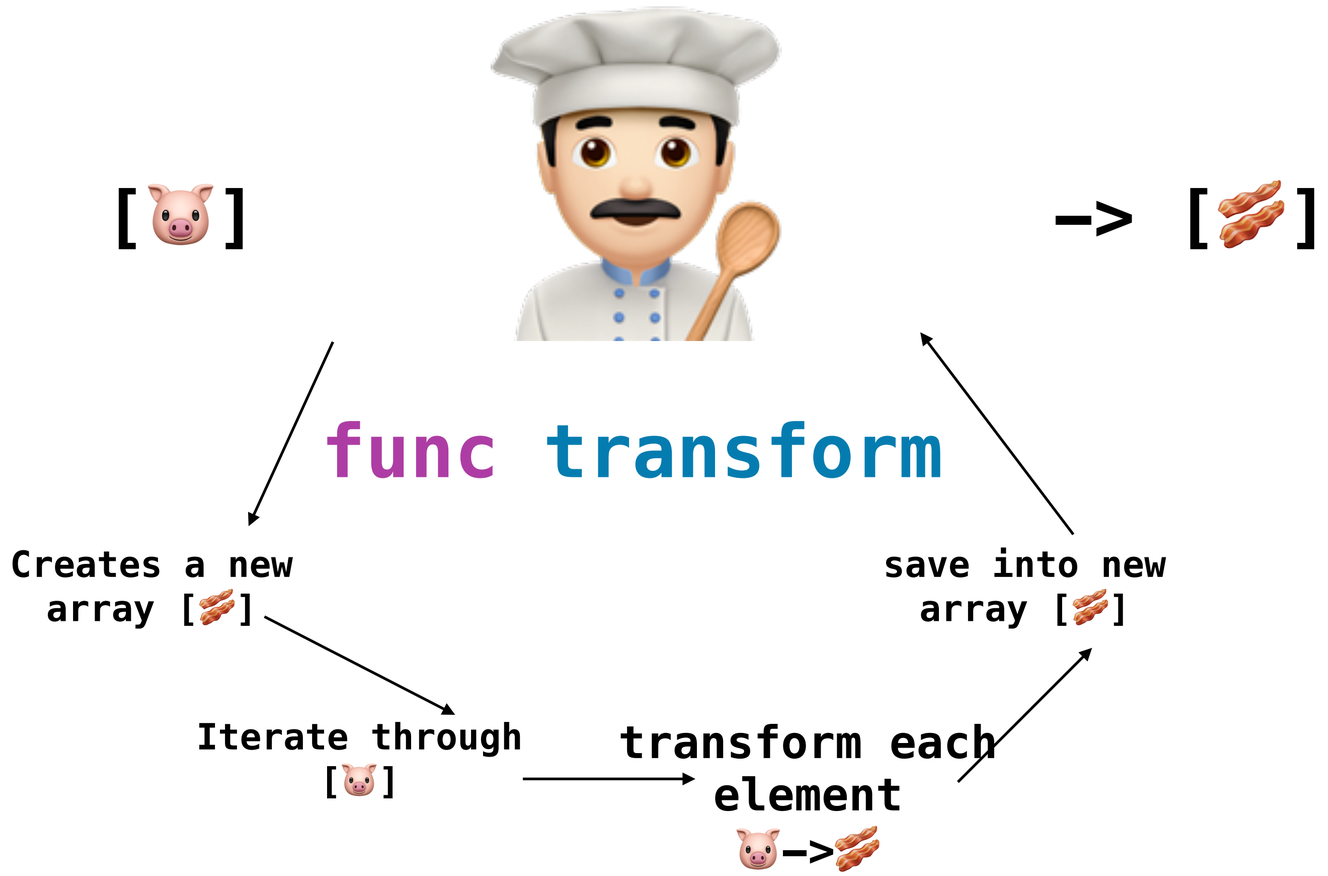
```
func transform(originalStrings: [🐷]) -> [🥓]
```

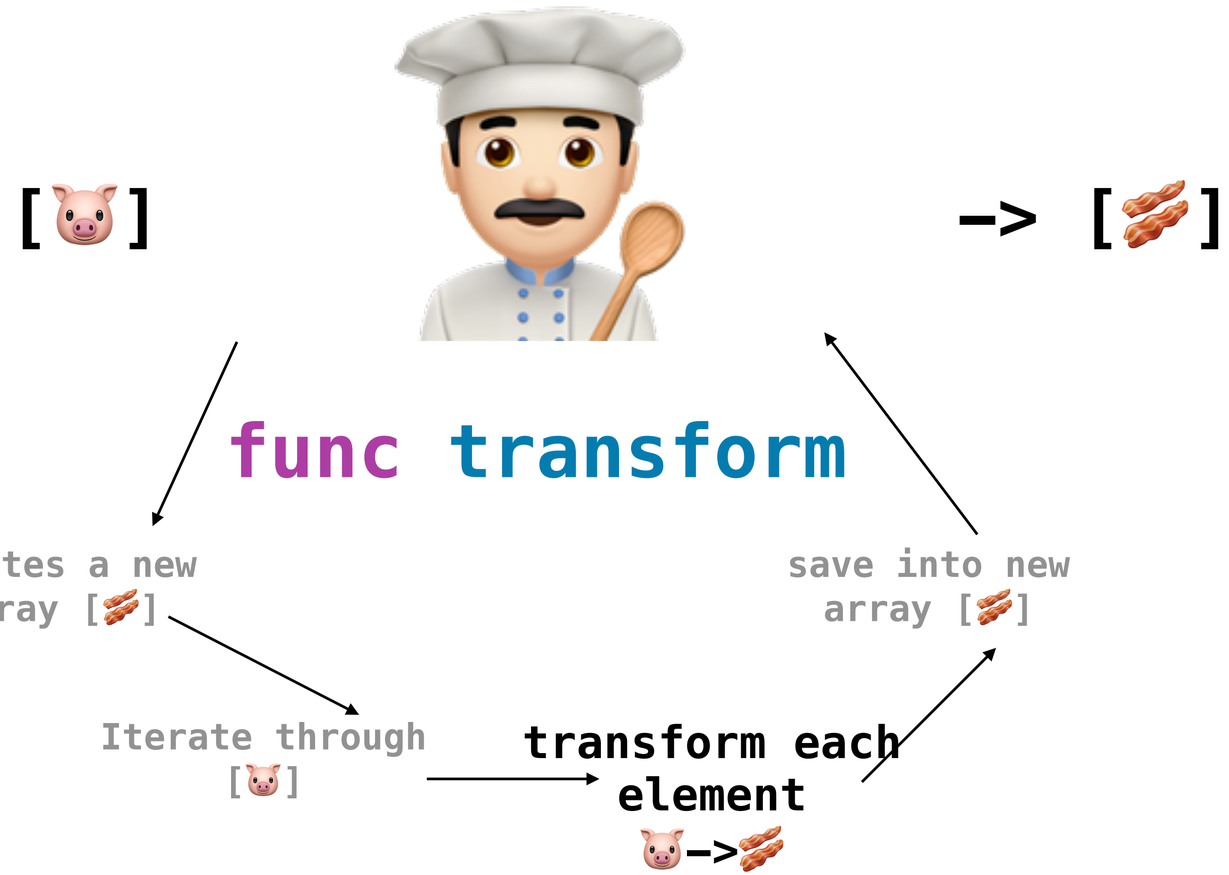
```
[String]    -> [String]
```

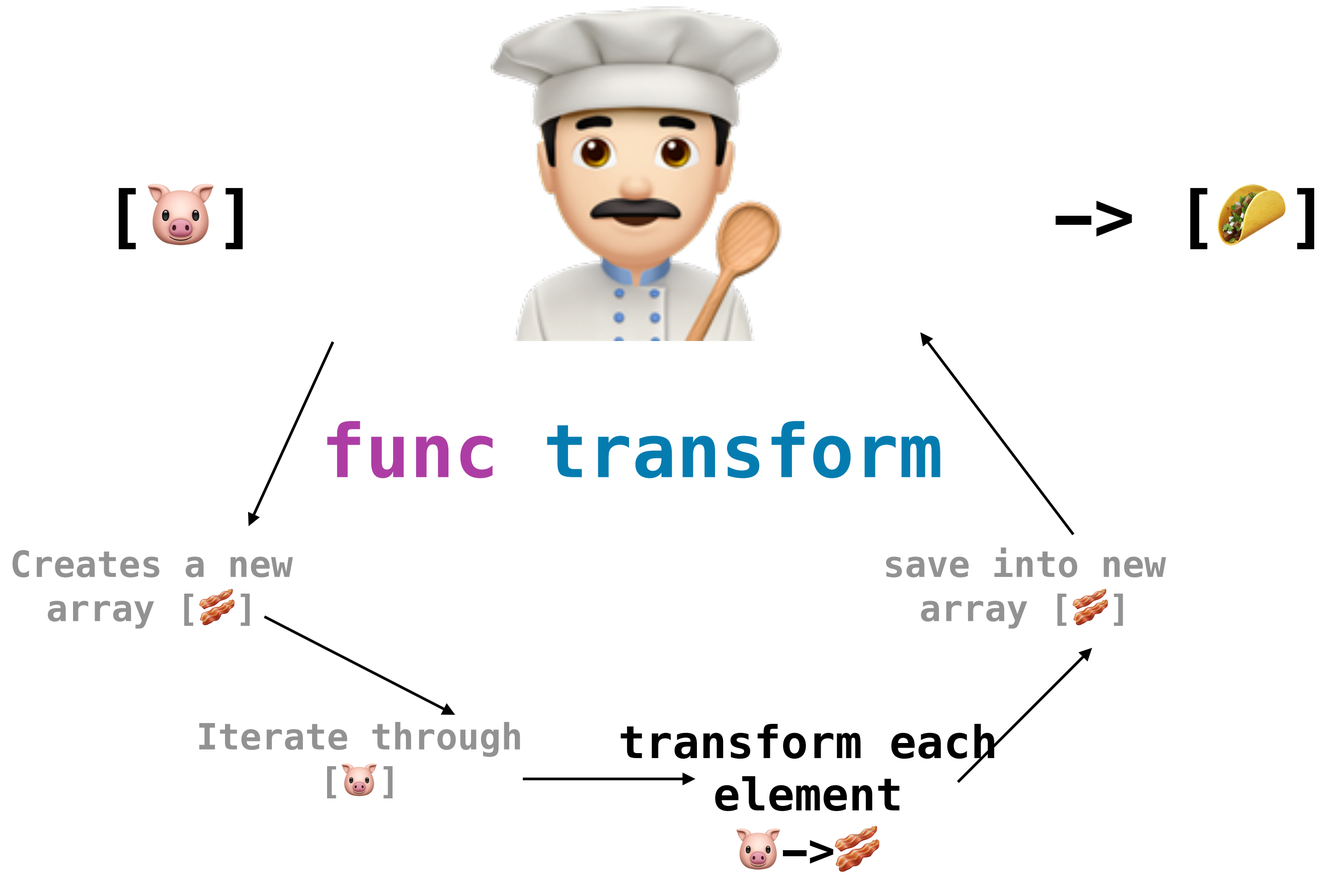
```
func transform(originalStrings: [String]) -> [String]
```

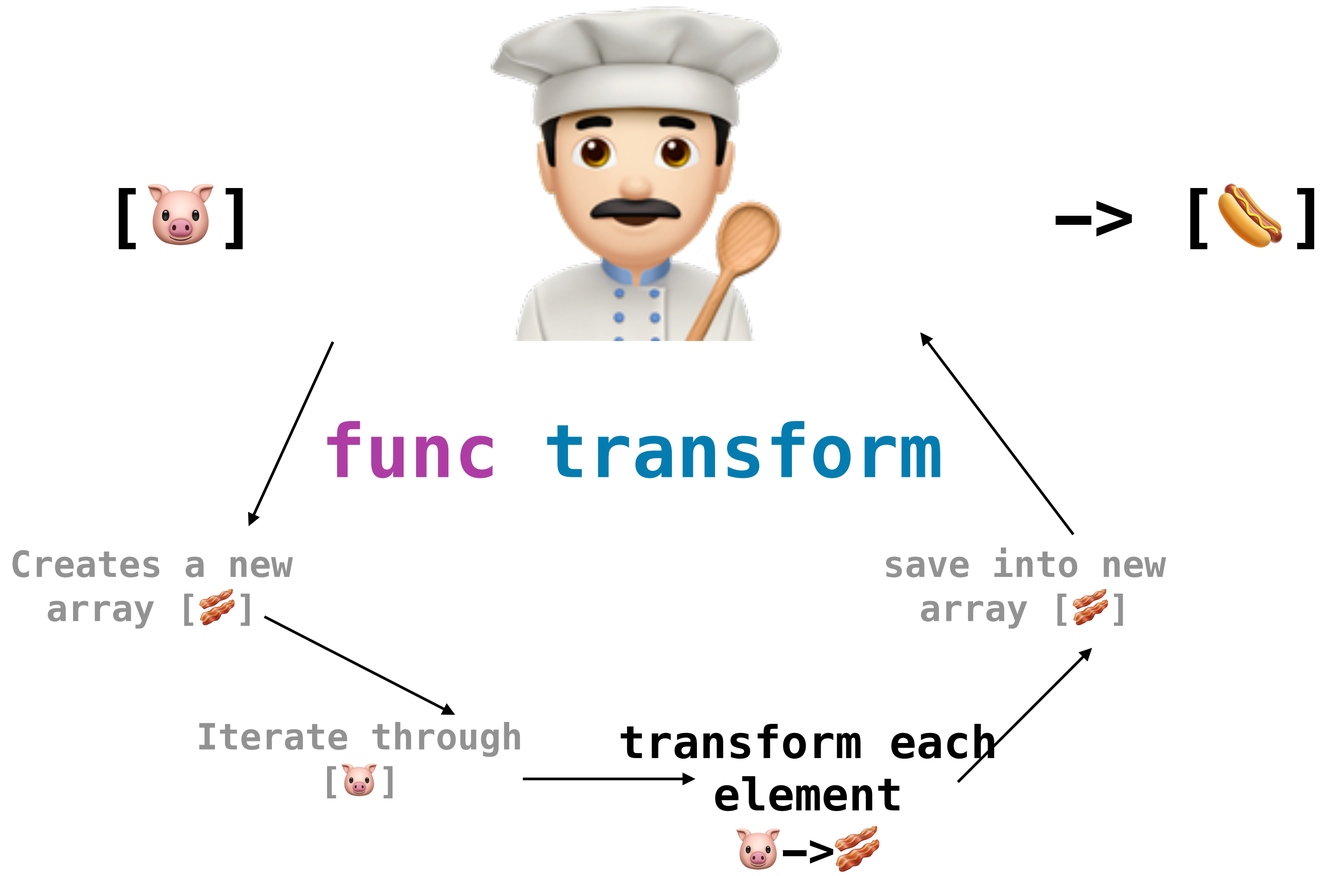
[🐷]  -> [🥓]

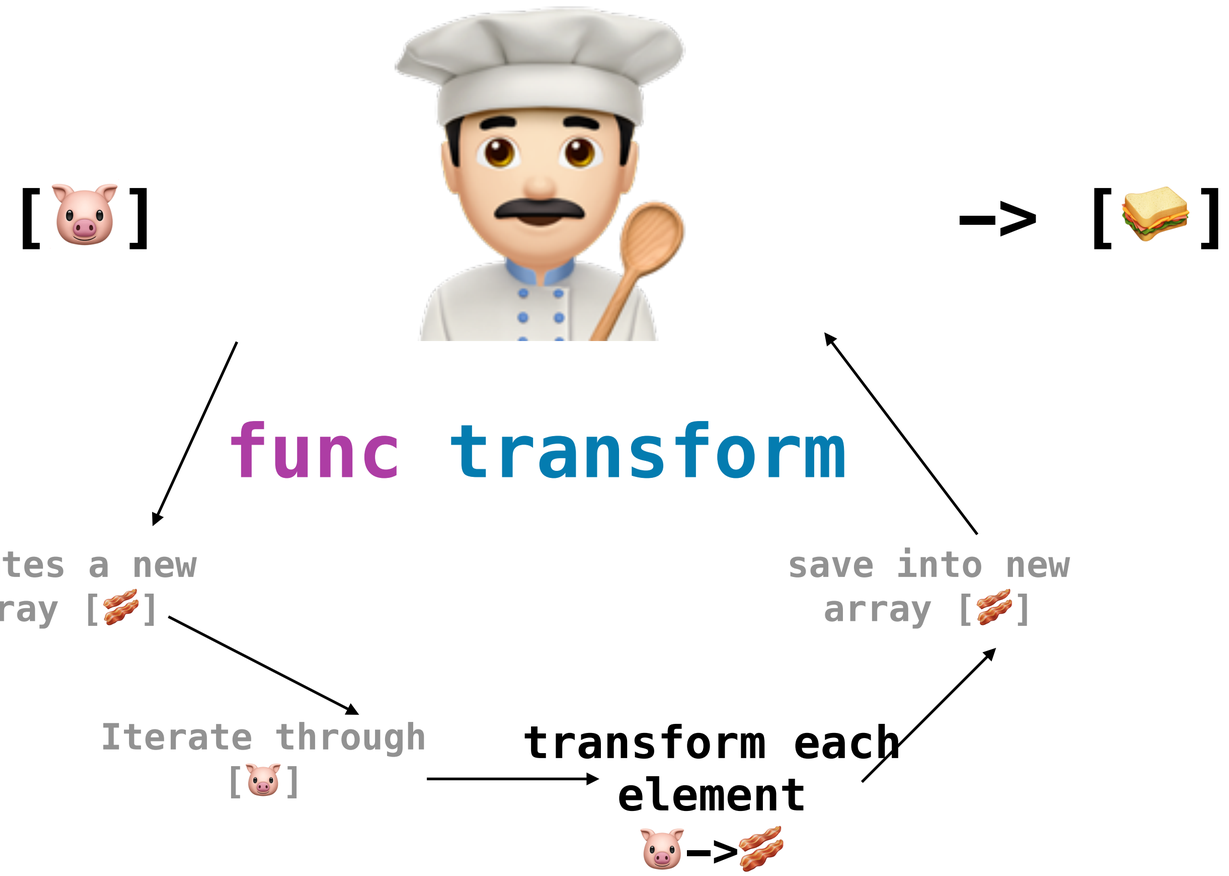
```
func transform(originalPorks: [🐷]) -> [🥓]
```

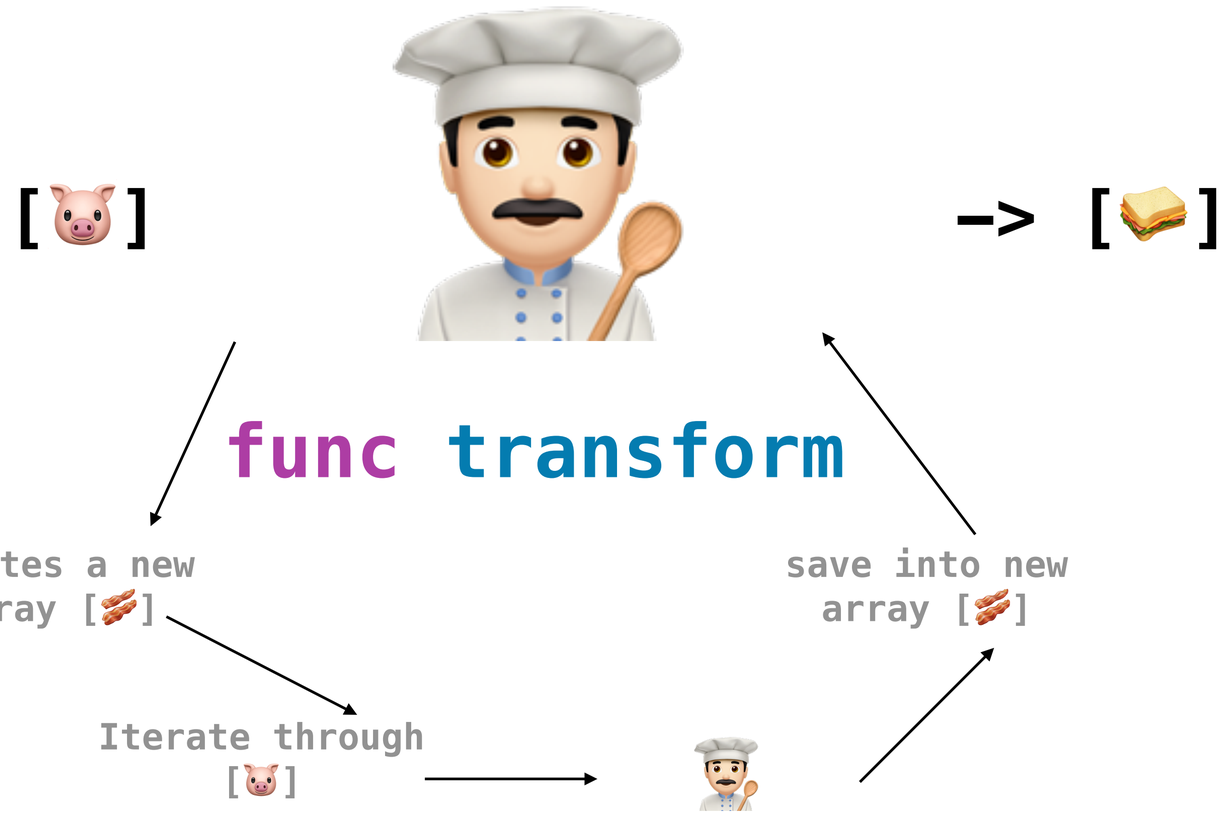

[🐷]
-> [🥓]
func transform
Creates a new array [🥓]
save into new array [🥓]
Iterate through [🐷]
transform each element
🐷->🥓

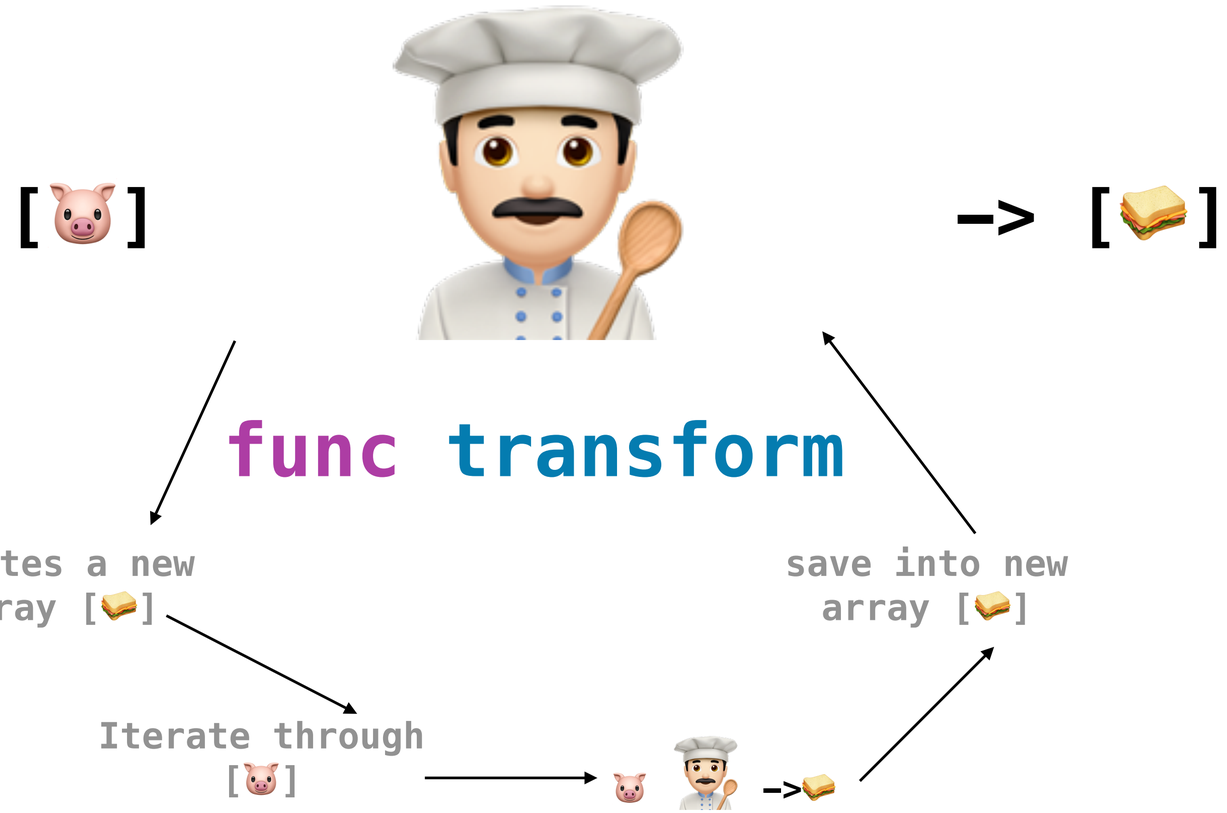

[🐷]
-> [🥓]
func transform
Creates a new array [🥓]
save into new array [🥓]
Iterate through [🐷]
transform each element
🐷->🥓

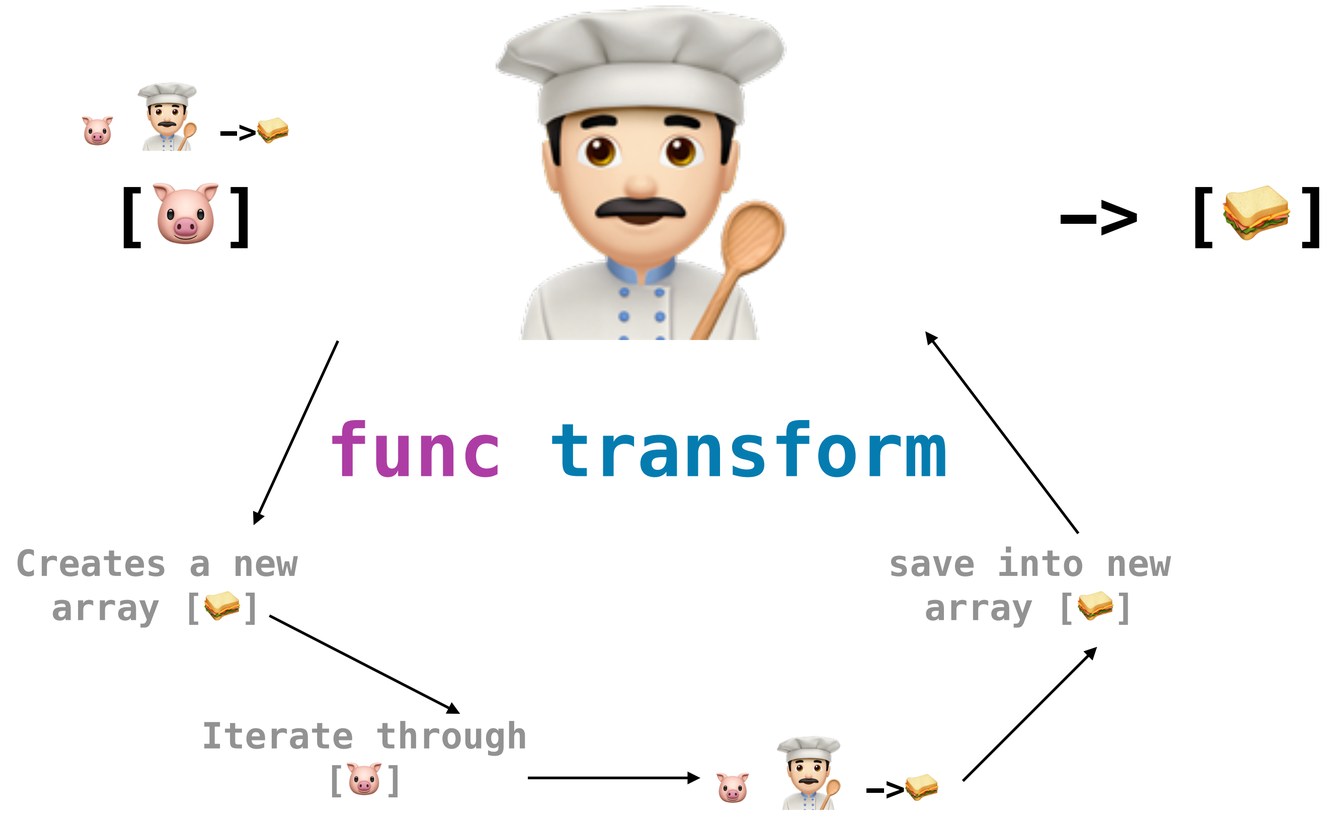

[🐷]
-> [🌮]
func transform
Creates a new array [🥓]
Iterate through [🐷]
transform each element 🐷->🥓
save into new array [🥓]


[🐷]
-> [🌭]
func transform
Creates a new array [🥓]
Iterate through [🐷]
transform each element
🐷->🥓
save into new array [🥓]


[🐷]
-> [🥪]
func transform
Creates a new array [🥓]
Iterate through [🐷]
transform each element 🐷->🥓
save into new array [🥓]


[🐷]
-> [🥪]
func transform
Creates a new array [🥓]
save into new array [🥓]
Iterate through [🐷]


[🐷]
-> [🥪]
func transform
Creates a new array [🥪]
Iterate through [🐷]
save into new array [🥪]
->🥪


->
[ ]
-> [ ]
func transform
Creates a new
array [ ]
Iterate through
[ ]
->
save into new
array [ ]

Live Demo

Create a function that transforms elements in an array

Thinking in closures

They're like functions except when they're not

- In high-order functions,
  - declaration happens at the implementation of the containing function
  - implementation happens at the call site of the containing function
- Parameter labels are missing at the implementation
- Parameter types and return type are inferred at implementation
- Trailing closure syntax
- Shorthand arguments
- Omitting return keyword
- Keypaths

Thinking in closures

They're like functions except when they're not

- In high-order functions,
  - declaration happens at the implementation of the containing function

```
func transform(givenStrings: [String], closure: (String) -> String) -> [String]
```

  - implementation happens at the call site of the containing function

```
let capitalizedNames = transform(strings: names, closure: { (singleString: String) -> String in })
```

Thinking in closures

They're like functions except when they're not

- Parameter labels are missing at the implementation

```
func transform(givenStrings: [String], closure: (String) -> String) -> [String]
```

- Parameter types and return type are inferred at implementation

```
let capitalizedNames = transform(strings: names, closure: { singleString in })
```

- Trailing closure syntax

```
let capitalizedNames = transform(strings: names) { singleString in }
```

Thinking in closures

They're like functions except when they're not

- Shorthand arguments

```
let capitalizedNames = transform(strings: names, closure: { return $0.capitalized })
```

- Omitting return keyword

```
let capitalizedNames = transform(strings: names, closure: { $0.capitalized })
```

- Keypaths

```
let capitalizedNames = names.map(\.capitalized)
```

Other types of closures

They're like other closures except when they're not

- Other types of closures exist:
  - Global functions
  - Nested functions
  - var initialization

Other types of closures

They're like other closures except when they're not

- Global functions

```
func transform(originalStrings: [String]) ->
[String] { ... }
```

Other types of closures

They're like other closures except when they're not

- Nested functions

```
func transform(originalStrings: [String]) ->
[String] {

    func internalFunction() {}

}
```

Other types of closures

They're like other closures except when they're not

- var initialization

```
struct Person {
    lazy var familyTree: [Person] = {
        return fetchFromDisk()
    }()
}
```

Closures and more

Where to go from here?

- Will be uploaded to https://github.com/olivaresf/intro_to_closures
- You can reach me @fromJrToSr
- Additional info:
  - https://docs.swift.org/swift-book/LanguageGuide/Closures.html
  - https://medium.com/better-programming/everything-you-wanted-to-know-about-closures-in-swift-e7d3a6ff5a74

Next Class

What is GCD?

- Official definition: "provides comprehensive support for concurrent code execution on multicore hardware."[1]
- Handles tasks (i.e. closures) and passes them along to queues that execute them in synchronous or asynchronous order.
- Queues can be serial or parallel and can be controlled by using semaphores.
- Used extensively for long-running tasks that would block the main queue (responsible for drawing).

[1] - https://apple.github.io/swift-corelibs-libdispatch/

Support Fernando

I need to eat

- Practice Swift weekly with a 15-minute exercise:
  - https://mailchi.mp/hey/weekly-swift-exercise-signup
- 1- and 2-week workshops: **Are You Ready for Work?**
  - Code real-life stories over 5 days
  - Work on a real codebase
  - Code reviews and zoom calls with Fernando
  - https://fernando-olivares-s-school.teachable.com/p/from-junior-to-senior

Q&A