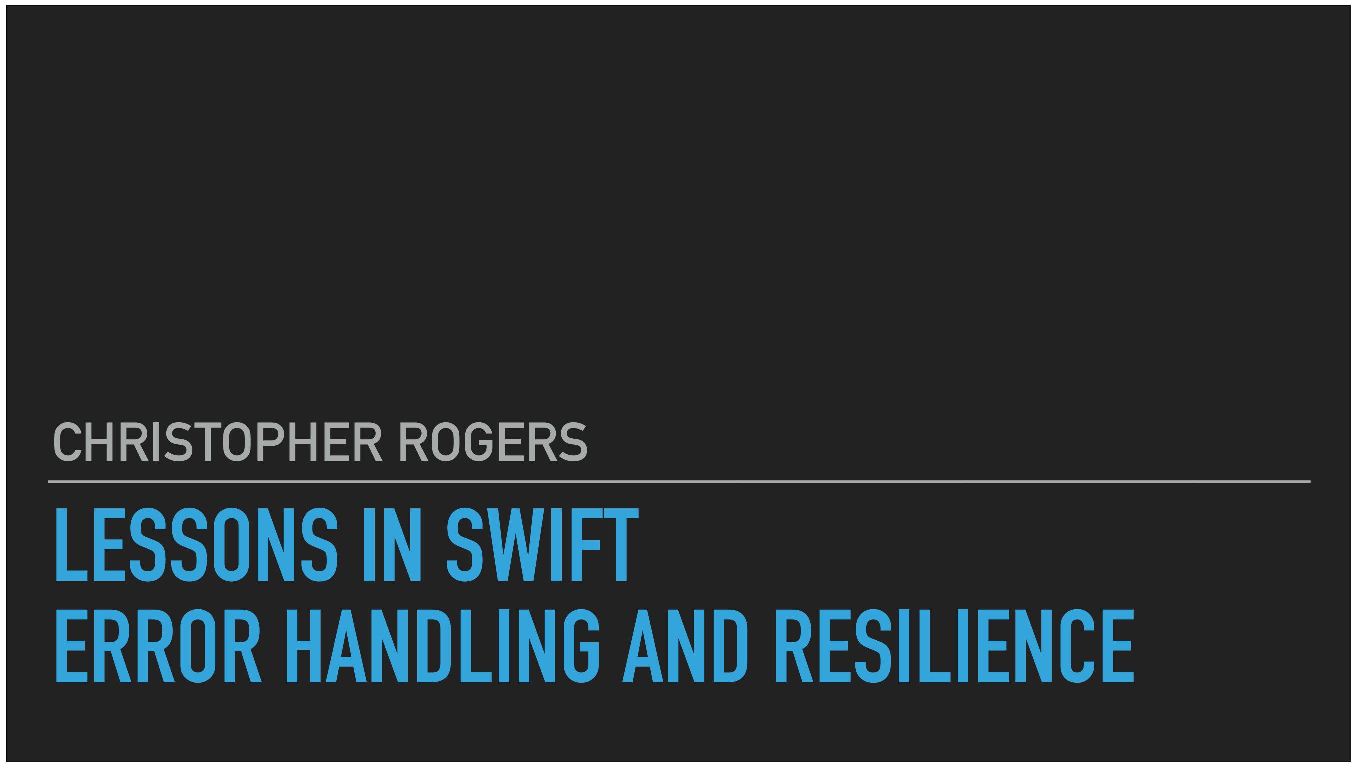
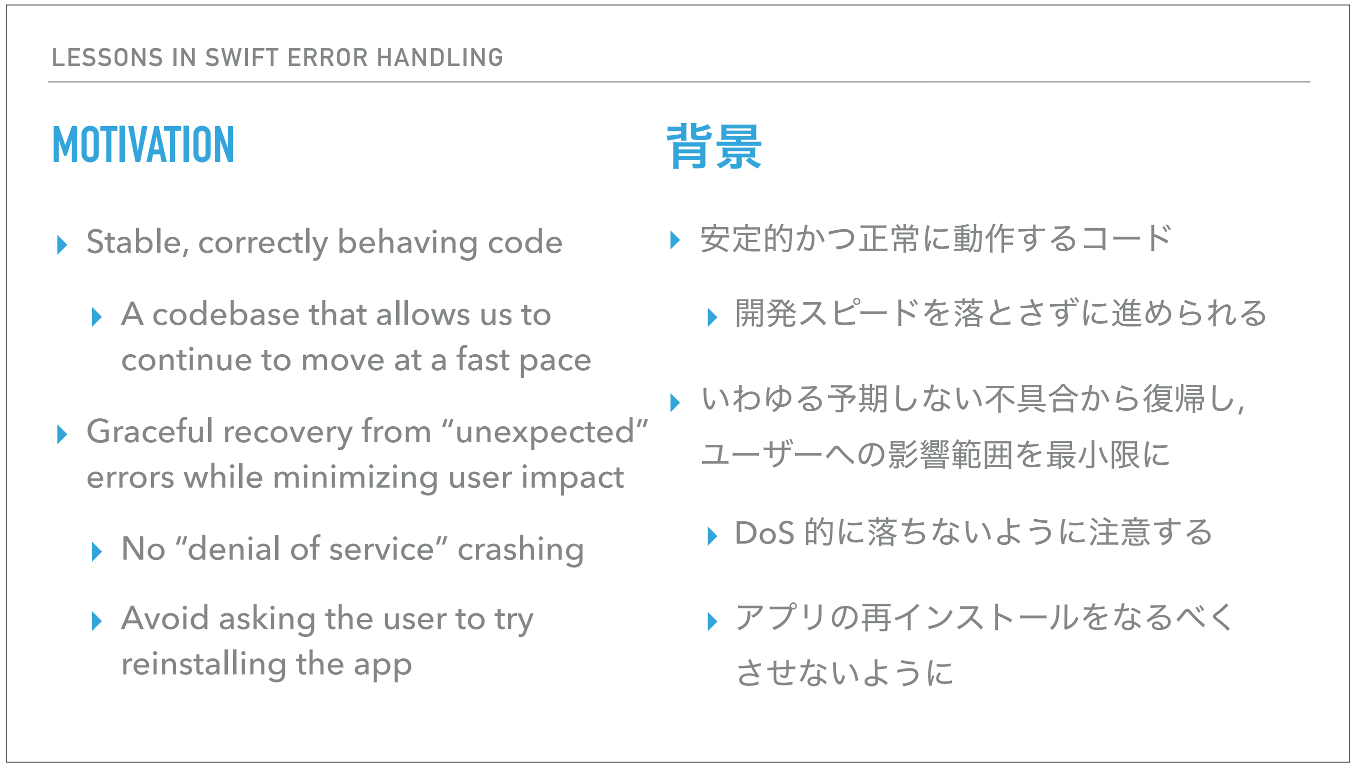

CHRISTOPHER ROGERS

LESSONS IN SWIFT
ERROR HANDLING AND RESILIENCE

Hello, everyone. こんばんは。I'd like to take the opportunity to share some thoughts of mine regarding errors in Swift. Before I start, I'd like to make clear that by the word “error” I'm not just referring to the Error type, but to an all encompassing view of anything in your code that you may consider as “going wrong” or “not the ideal code path.”

よろしくお願いします。



MOTIVATION

- Stable, correctly behaving code
  - A codebase that allows us to continue to move at a fast pace
- Graceful recovery from “unexpected” errors while minimizing user impact
  - No “denial of service” crashing
  - Avoid asking the user to try reinstalling the app

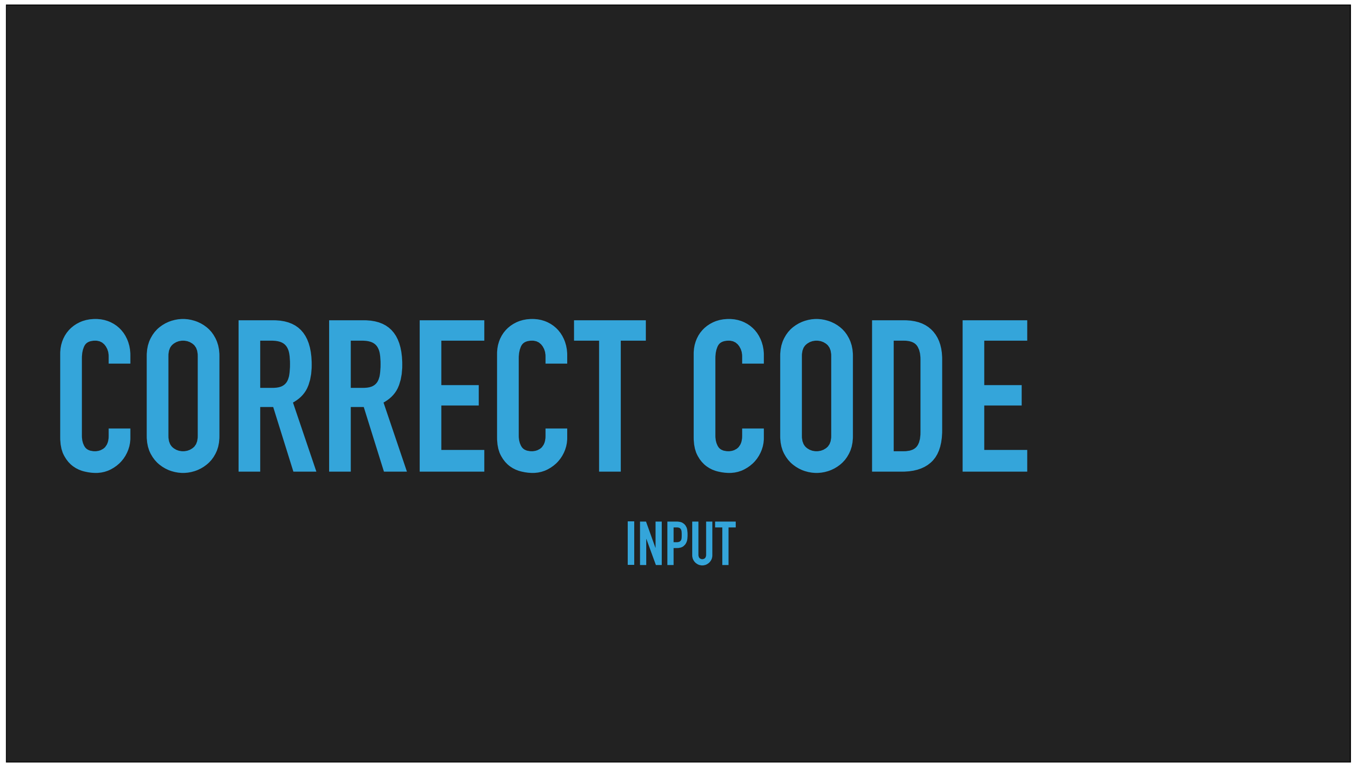
背景

- 安定的かつ正常に動作するコード
  - 開発スピードを落とさずに進められる
- いわゆる予期しない不具合から復帰し，ユーザーへの影響範囲を最小限に
  - DoS 的に落ちないように注意する
  - アプリの再インストールをなるべくさせないように

So I’d like to start off by stating the motivation that drives this presentation. We want, first of all, code that is stable and behaves correctly. Who doesn’t want that, right? So we want code that does what we intend for it to do, and for that to happen all of the time. This allows us to continue delivering new features and improvements at a fast pace.

Also, we want to anticipate, as much as possible, any and all, quote-unquote, “unexpected” errors and minimize the impact this has on the user. For example, if we aren’t able to read part of the user’s data from the database, we wouldn’t want to just crash and give up. In cases where the user cannot avoid causing an error you want to be especially careful of this. Code, say, to be run on launch to render your main view, is one example. Any kind of automatic process that the user doesn’t have direct control of, say, to sync data over the network or clean up stale data, then it’s even more important to do so because it may result in a denial of service. We also don’t want to resort to having the user contact customer support, and then asking the user to reinstall the app. Not only is this an awful experience for users, it can mean that users lose data that’s important to them, and lose trust in your app.

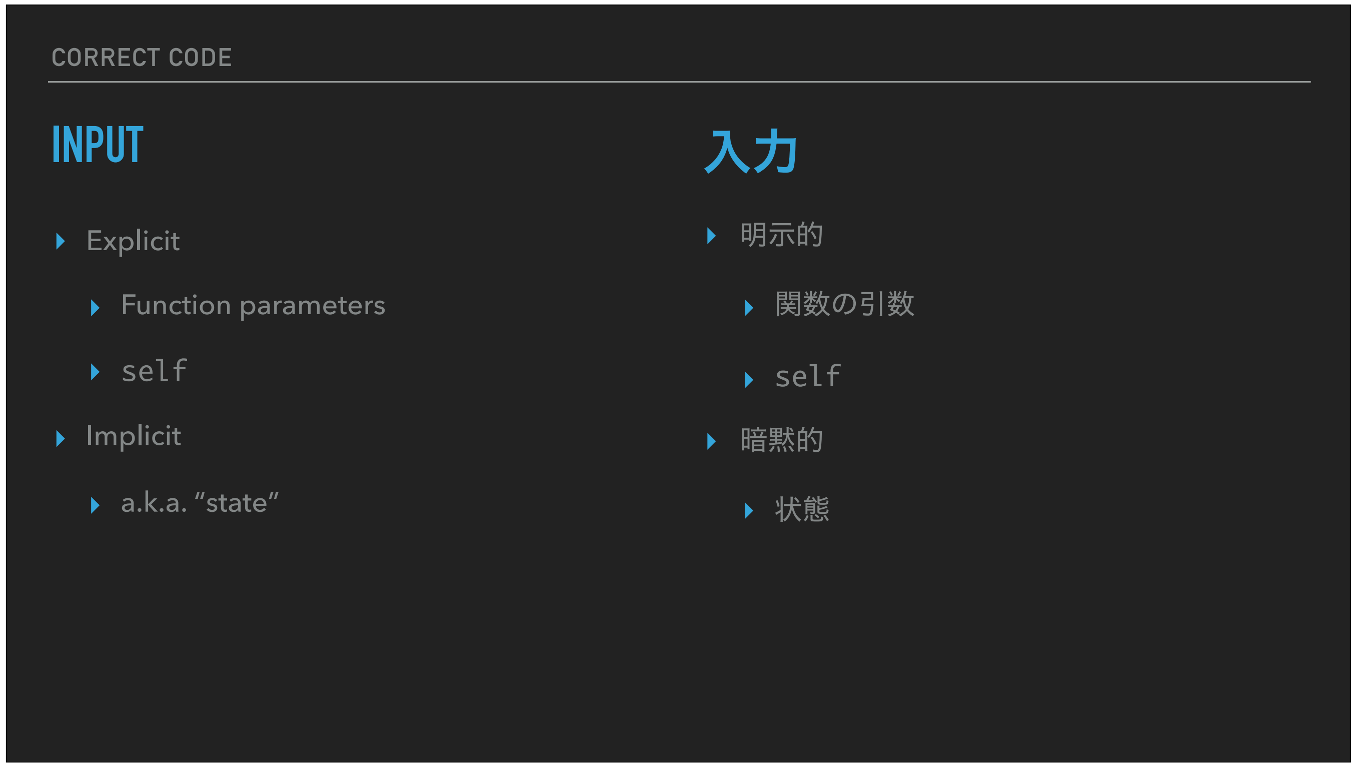
So with that in mind, I’d like to start off on the first point about writing “correct code”.



CORRECT CODE

INPUT

There are many topics you could discuss regarding how to go about achieving this…how to write maintainable code, how to write testable code… with the goal of assuring the correct behavior doesn't become incorrect in the future. But for today, I'd like to talk briefly about >> "input."


CORRECT CODE

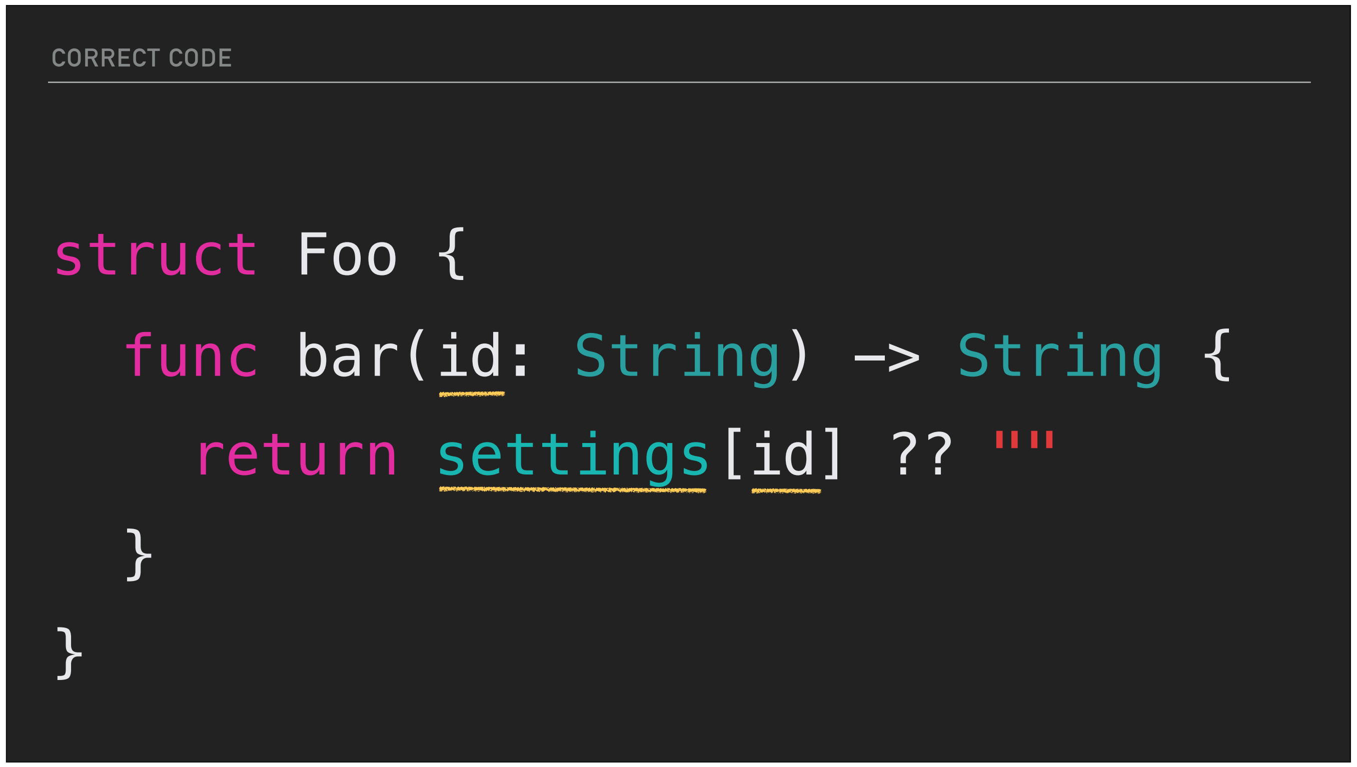
INPUT

- Explicit
  - Function parameters
  - `self`
- Implicit
  - a.k.a. “state”

入力

- 明示的
  - 関数の引数
  - `self`
- 暗黙的
  - 状態

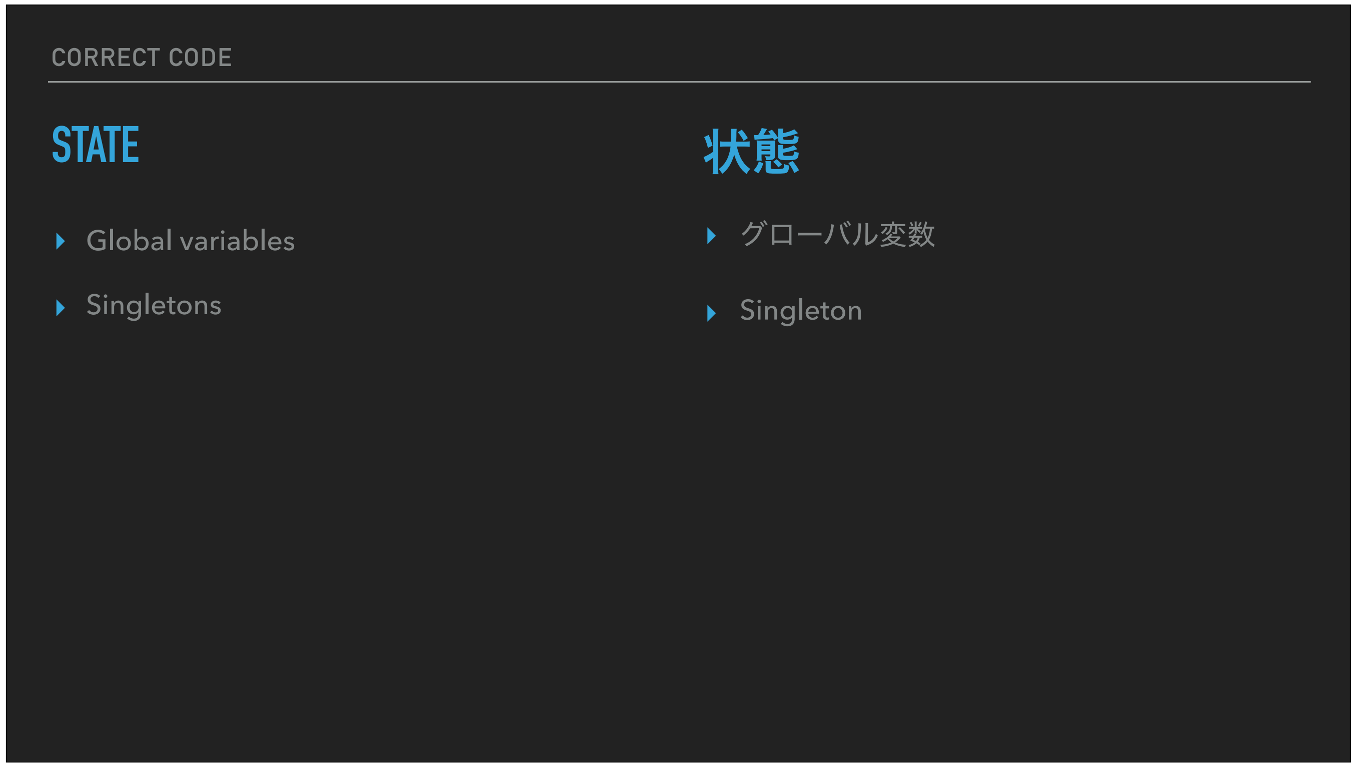
In order to write correct code and prevent errors from occurring, we need to be able to handle any and all input. It helps me to break “input” down into two types: explicit and implicit. Explicit input, by nature of being explicit, is going to stand out to the reader and more clearly communicate intent. Explicit input is going to be the parameters to your functions. You also have this “self” thing in methods. In Swift `self` is implicitly passed, so in that respect it’s easy to overlook. But I think conceptually `self` is still explicit input. Implicit input is any other data accessible to a function and is usually referred to as “state.”



CORRECT CODE

```swift
struct Foo {
  func bar(id: String) -> String {
    return settings[id] ?? ""
  }
}
```

So in this simple example, the parameter >> `id` would be explicit input, and >> `settings` would be implicit input, as it’s not a part of the method signature.



CORRECT CODE

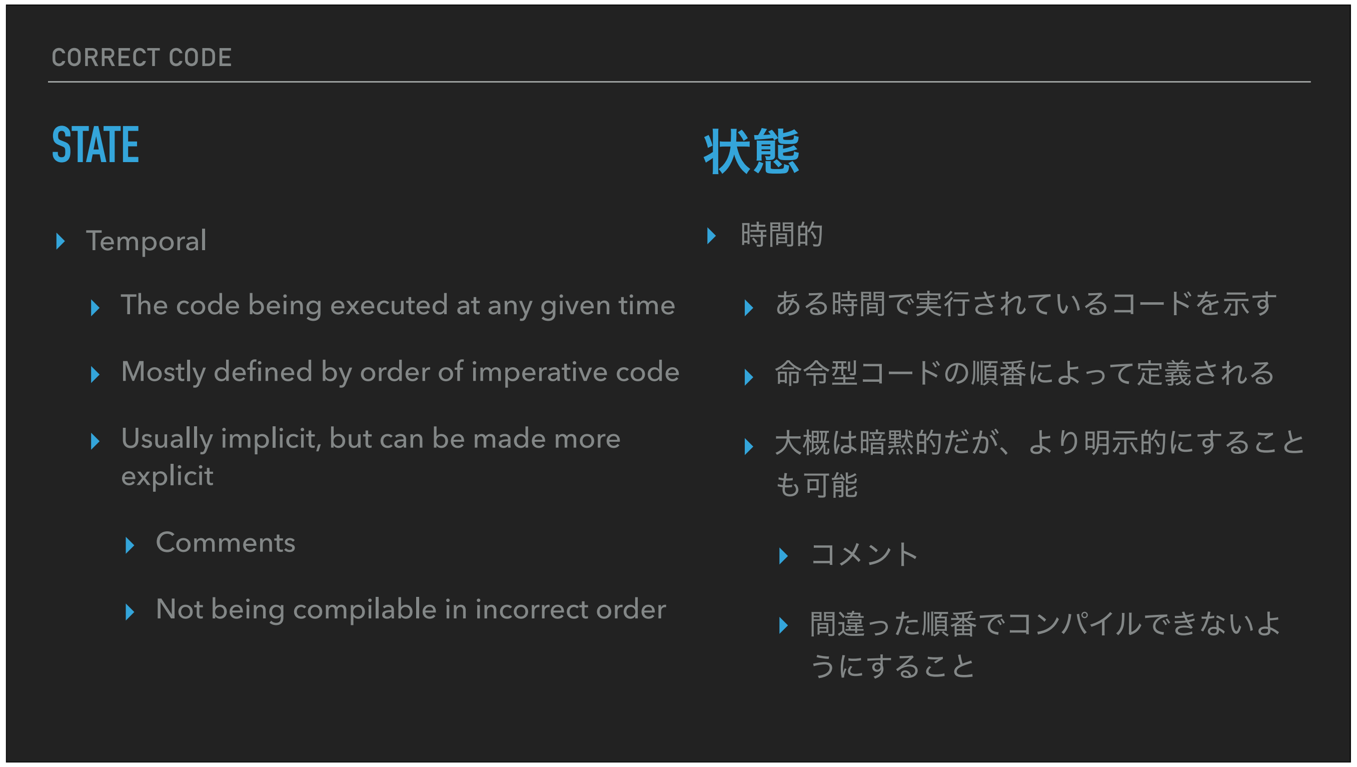
STATE

- Global variables
- Singletons

状態

- グローバル変数
- Singleton

So state typically consists of variables & constants at global scope. Any outer scope accessible to a function can be a potential source of state. Singletons fall under this category. If your function depends on the values of any of these, then the output can differ even with the same explicit input. On the other hand, functions that don't depend on or affect state of varying degrees are known as “pure functions”. I won’t delve into this topic, but I will mention that there has been discussion lately in the Swift Evolution mailing list to get support in Swift for declaring functions as pure. This would allow the compiler to enforce their “pureness” at compile time.



CORRECT CODE

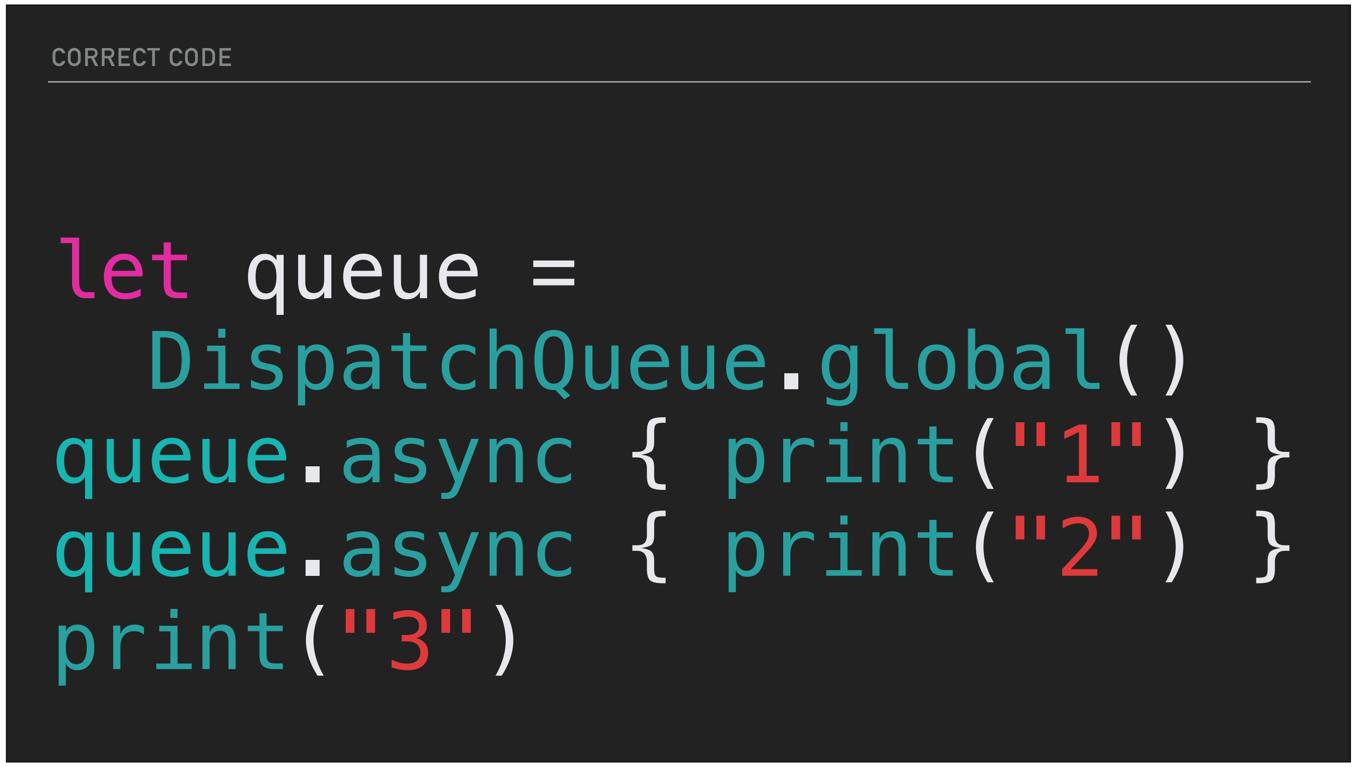
STATE

- Temporal
  - The code being executed at any given time
  - Mostly defined by order of imperative code
  - Usually implicit, but can be made more explicit
    - Comments
    - Not being compilable in incorrect order

状態

- 時間的
  - ある時間で実行されているコードを示す
  - 命令型コードの順番によって定義される
  - 大概は暗黙的だが、より明示的にすることも可能
    - コメント
    - 間違った順番でコンパイルできないようにすること

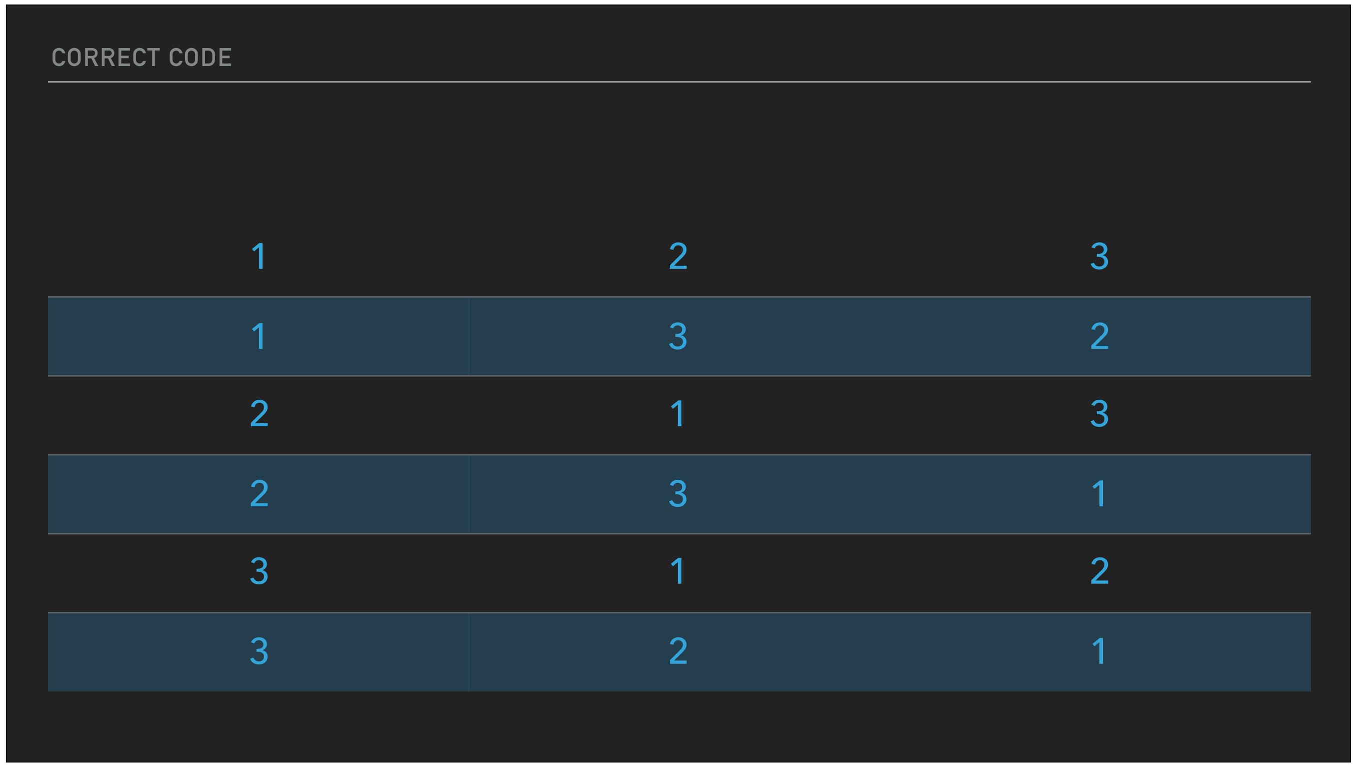
Next is what I call “temporal” state. This is the state of the program being executed at any given time. This comes more into play when dealing with concurrent programming but is still a useful concept. It’s partially defined by the order of statements in imperative code. Because the order of code tends to be taken for granted, I would classify temporal state as being mostly implicit. The order that code needs to be run *can* be made more explicit. The most basic way would be to write a comment to the human reading the code. “`A` must be be called before `B` or bad things will happen!” If you want to communicate this to the compiler as well, make code not compilable when rearranged in another order.



CORRECT CODE

```swift
let queue =
  DispatchQueue.global()
queue.async { print("1") }
queue.async { print("2") }
print("3")
```

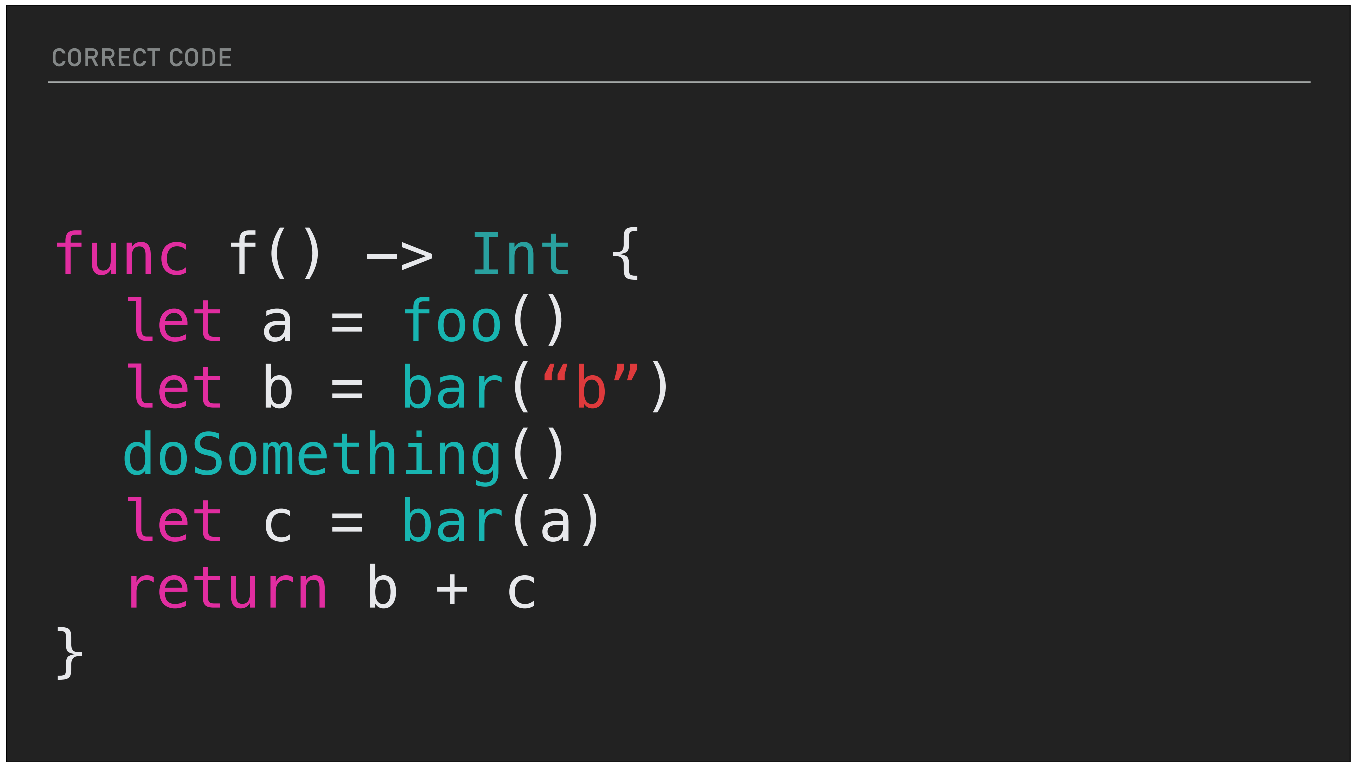
Let me show you what I mean with an example using concurrent programming. If you're not familiar with Grand Central Dispatch APIs, this is going to execute these print statements independently of each other. The order is not defined. You can think of the possible states of execution order being multiplied as the threads of execution interweave each other in differing patterns.



CORRECT CODE

1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

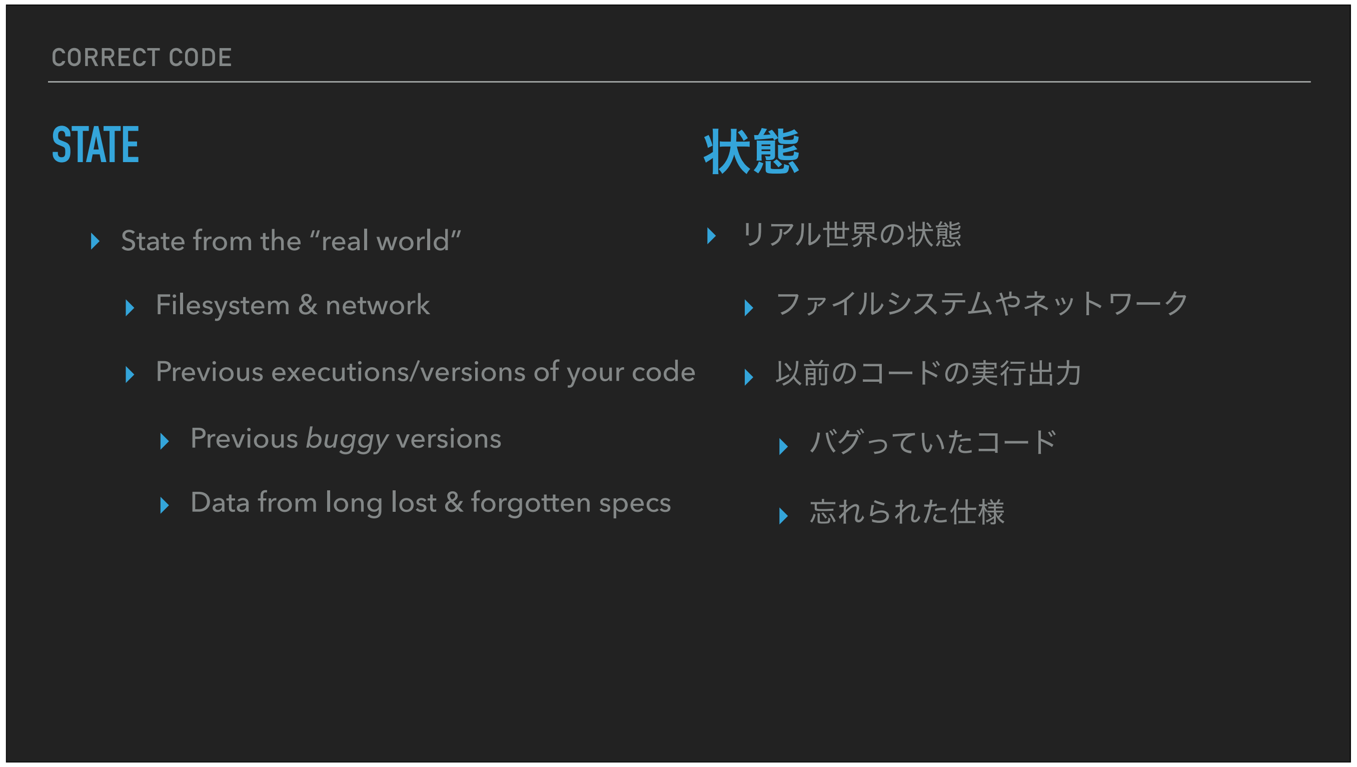
So you can imagine that before this code, there was one nice “stream” of execution. As we reach this code, we split off into three streams. The print statements share the same serialized output stream so they interact. So since there are three places to fill, that gives us 3! (3 factorial) or 6 potential outcomes.



CORRECT CODE

```swift
func f() -> Int {
  let a = foo()
  let b = bar(“b”)
  doSomething()
  let c = bar(a)
  return b + c
}
```

Here I’d like to illustrate the same concept, but this time without concurrency. Nothing much interesting is going on. Each statement here is evaluated in top-down order. Some of the values are used as arguments for other function calls later on. So there is a relationship enforced here by the compiler. I couldn’t place the line for “let c” first even if I wanted to because we don’t have the value of “a” yet. And then we have this glaring function call smack in the middle. What’s it doing here? Since it doesn’t have any input or output, we assume that it must be affecting state that affects the output of the code around it. We just don’t know unless we understand what makes `doSomething` interesting.


CORRECT CODE

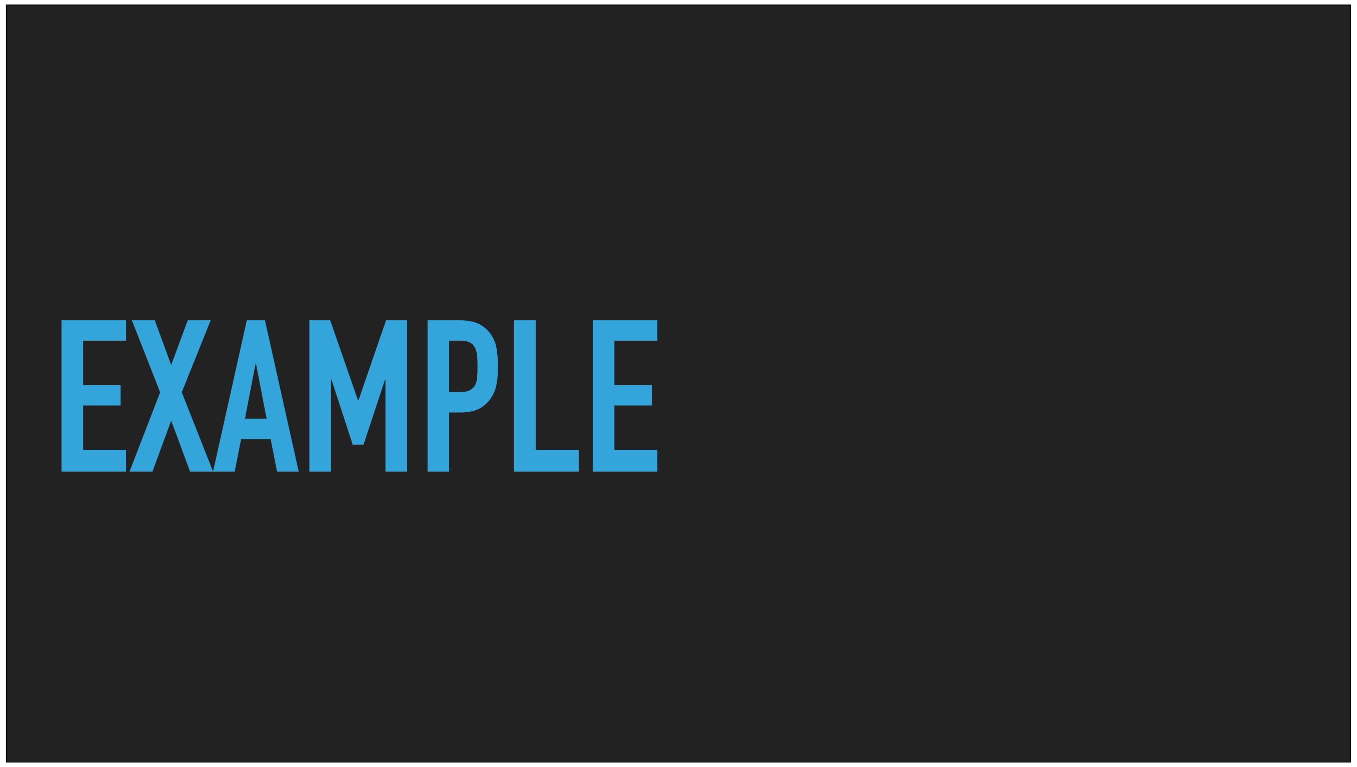
STATE

- State from the “real world”
  - Filesystem & network
  - Previous executions/versions of your code
    - Previous *buggy* versions
    - Data from long lost & forgotten specs

状態

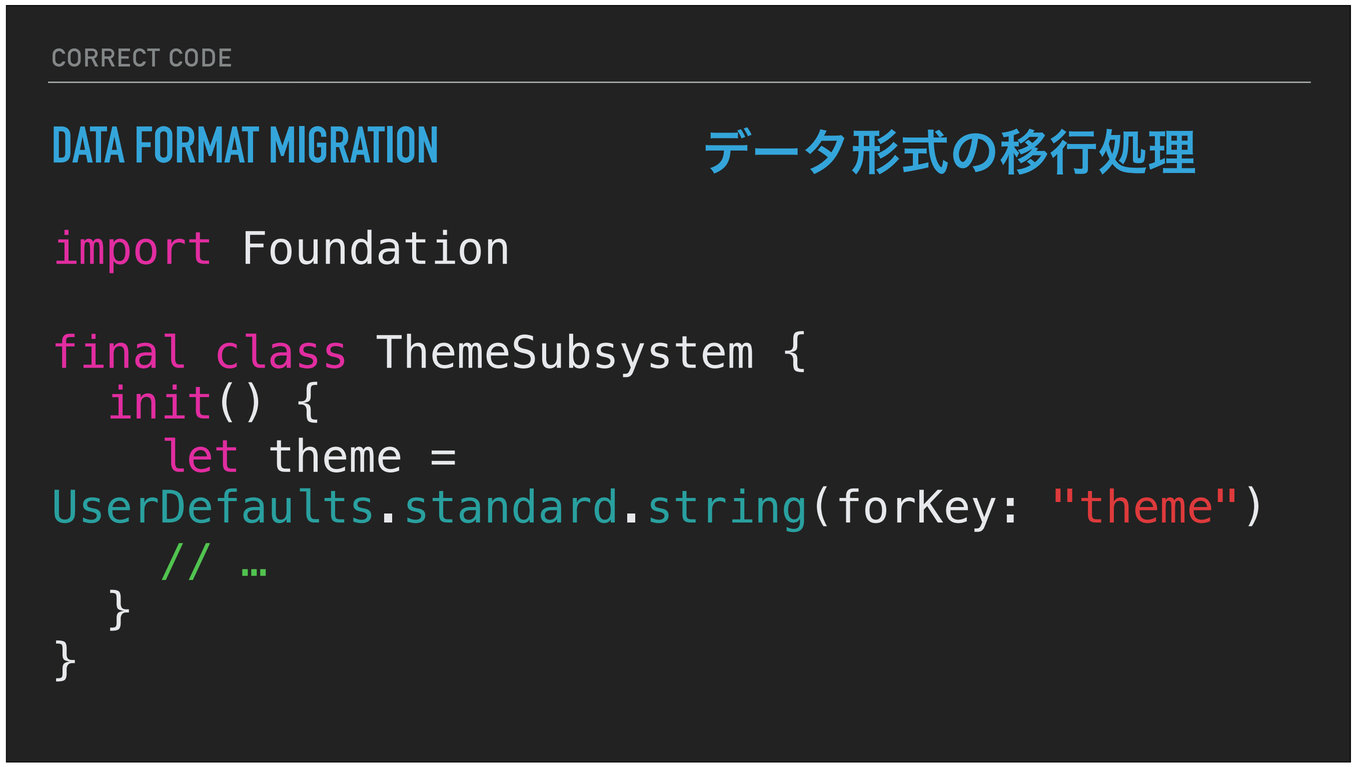
- リアル世界の状態
  - ファイルシステムやネットワーク
  - 以前のコードの実行出力
    - バグっていたコード
    - 忘れられた仕様

And finally I’d like to mention input or state from the real world. This isn’t a distinct type of state, but rather describes a property of it. What makes it worth mentioning is that for one, its complexity can easily get out of hand, and two, because this state usually originates from your own code, we tend to make assumptions about its value. For example, we may assume that we only stored non-negative numbers in a given database field because we made sure not to insert a negative value. However, are you sure the behavior has been like that for all past versions of this code? Maybe at one point someone thought it’d be clever to store -1 or NULL to indicate, say, that the data was out of sync. Maybe there was a bug at one point… Could this code be run on data from the output of future versions of your code? Do you want to be able to handle data fidgeted with directly by the user? You can see how easily this will get out of hand. Any changes to code handling this kind of state shouldn’t be modified haphazardly.



EXAMPLE

So I’d like to give a simplified example of a bug I created recently.



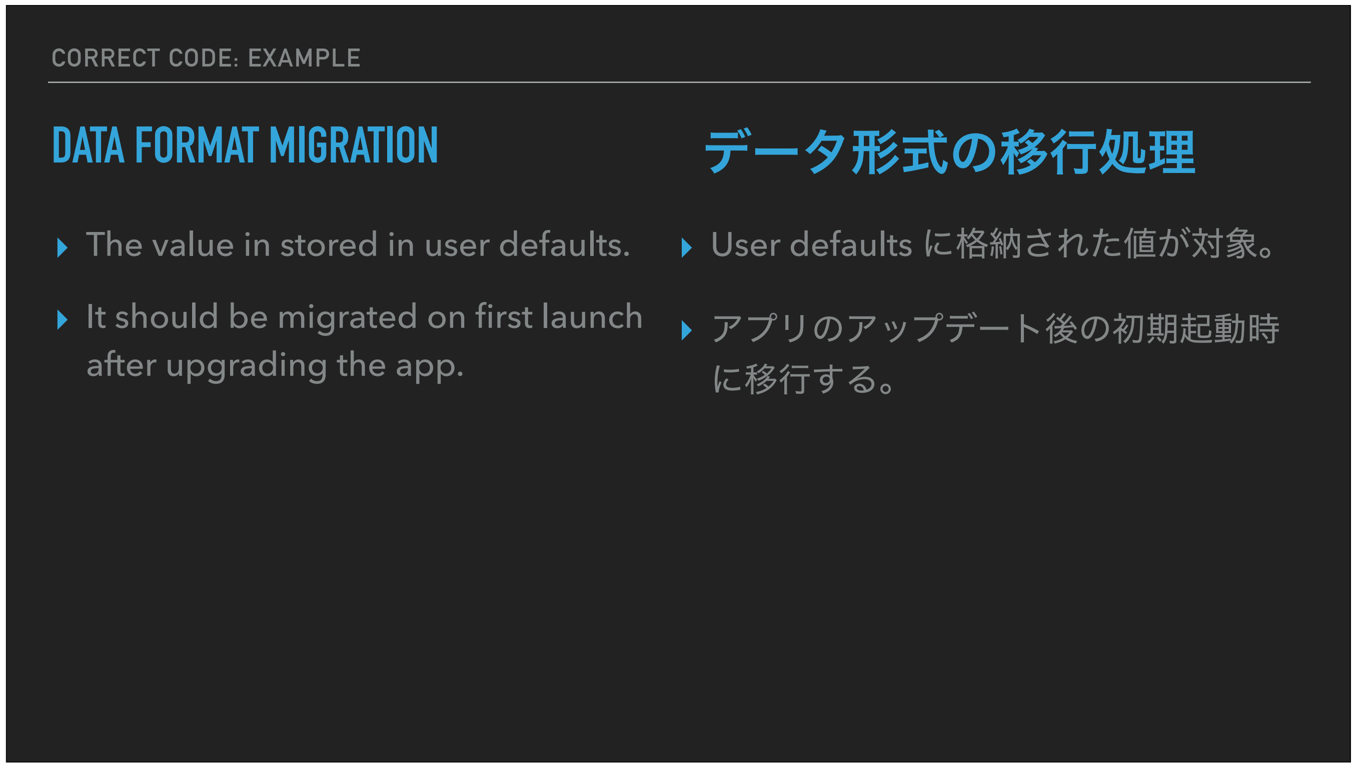
CORRECT CODE

DATA FORMAT MIGRATION データ形式の移行処理

```
import Foundation

final class ThemeSubsystem {
  init() {
    let theme =
UserDefaults.standard.string(forKey: "theme")
    // …
  }
}
```

In the app Line there is a feature called Theme (着せ替え）that lets the user change the look of the app. My task was to migrate the string format of the setting that holds which theme is currently in use. As you can see here, the setting is stored in user defaults.



CORRECT CODE: EXAMPLE

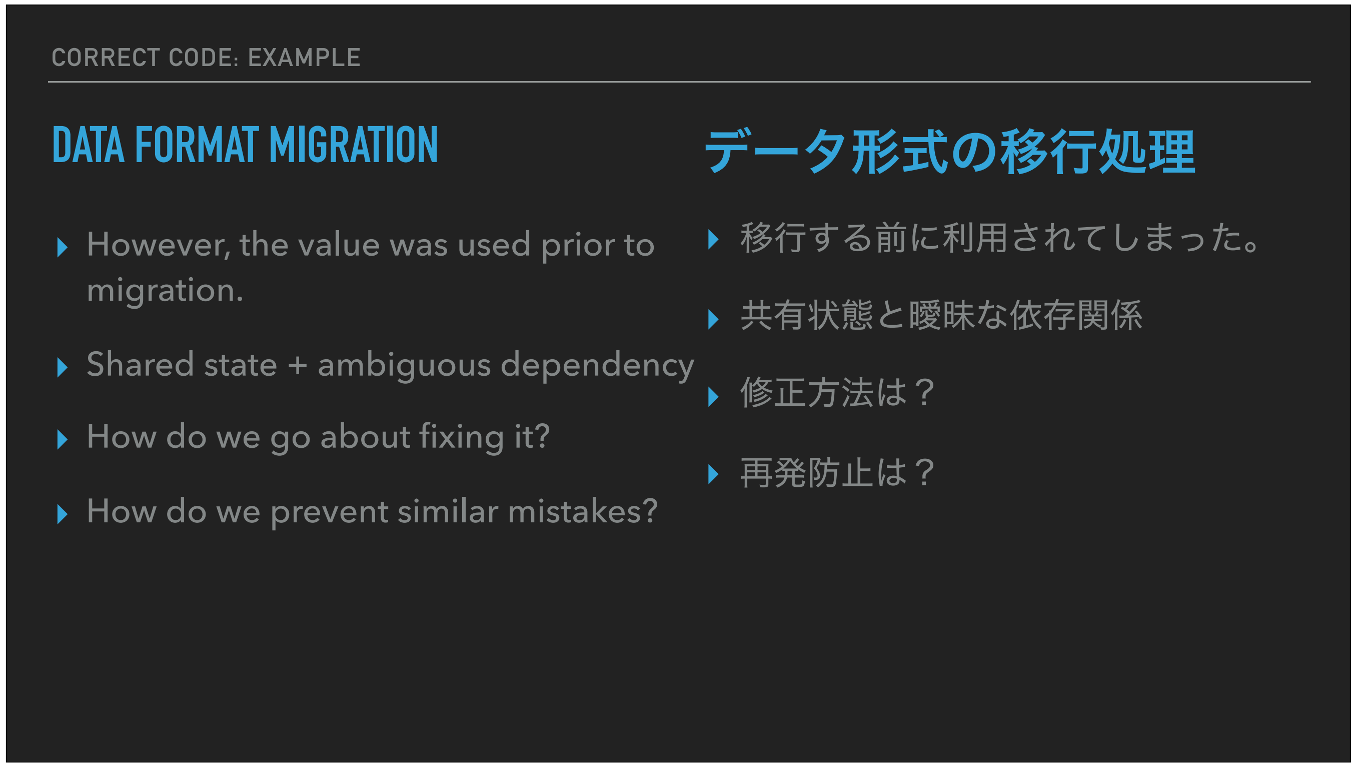
DATA FORMAT MIGRATION

- The value in stored in user defaults.
- It should be migrated on first launch after upgrading the app.

データ形式の移行処理

- User defaults に格納された値が対象。
- アプリのアップデート後の初期起動時に移行する。

So I added code that gets run when the app is launched for the first time on the new version, loaded this setting from the user defaults, and put the new format back in. Simple enough.


CORRECT CODE: EXAMPLE

DATA FORMAT MIGRATION

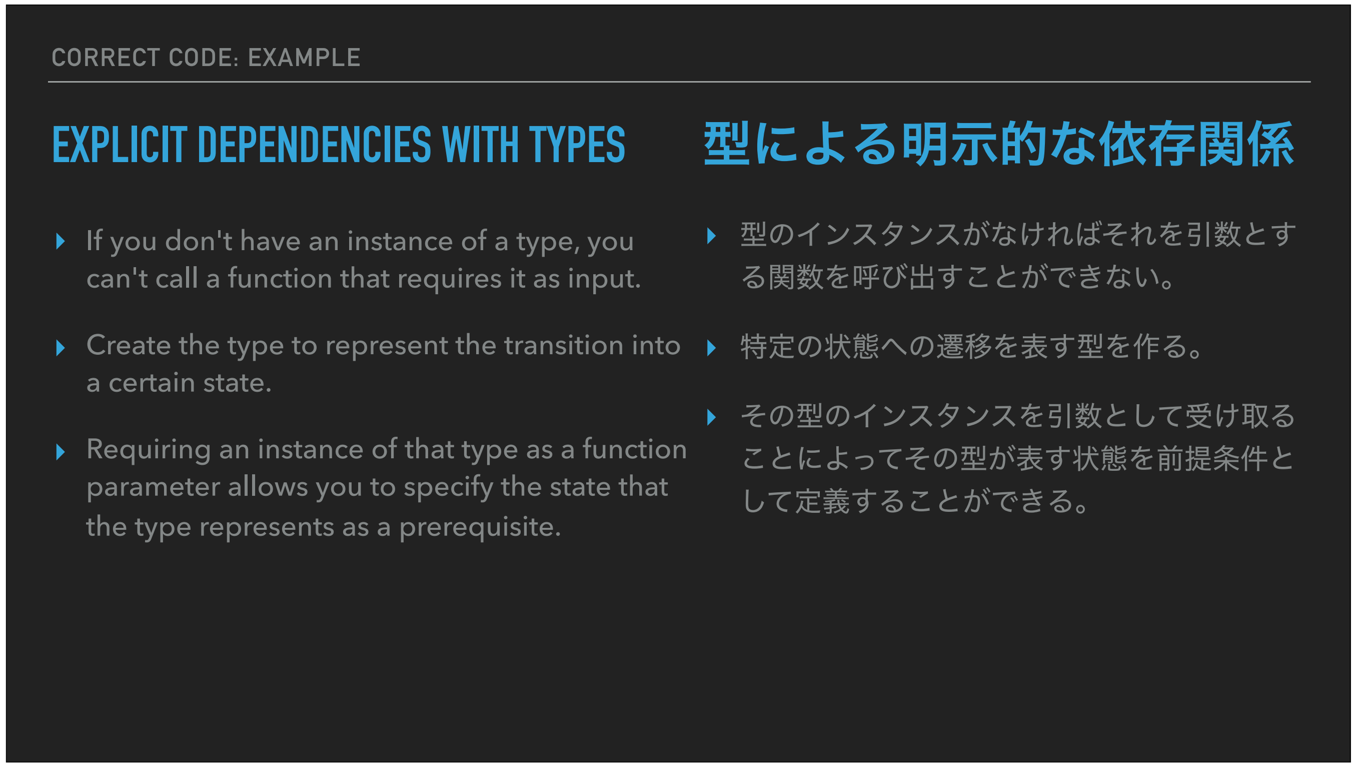
- However, the value was used prior to migration.
- Shared state + ambiguous dependency
- How do we go about fixing it?
- How do we prevent similar mistakes?

データ形式の移行処理

- 移行する前に利用されてしまった。
- 共有状態と曖昧な依存関係
- 修正方法は？
- 再発防止は？

But I soon found out there was a problem with this code. The app was launching with the default theme. You would think maybe I botched parsing the old format, but following launches used the theme previously in use. It turned out the theme subsystem was inadvertently being initialized before my migration code. Since the new version of this code expects the newer format, it failed to initialize correctly with the migrated value from user defaults. However, once the migration code ran, subsequence launches found the setting and initialized properly.

We can say that the bug was created because of the shared state & the ambiguous temporal relationship or dependency between the migration & the theme subsystem initialization. How do we go about fixing this, and how do we prevent this from breaking again in the future? Well, one way is to make the relationship between the two explicit at compile-time using the type system.



CORRECT CODE: EXAMPLE

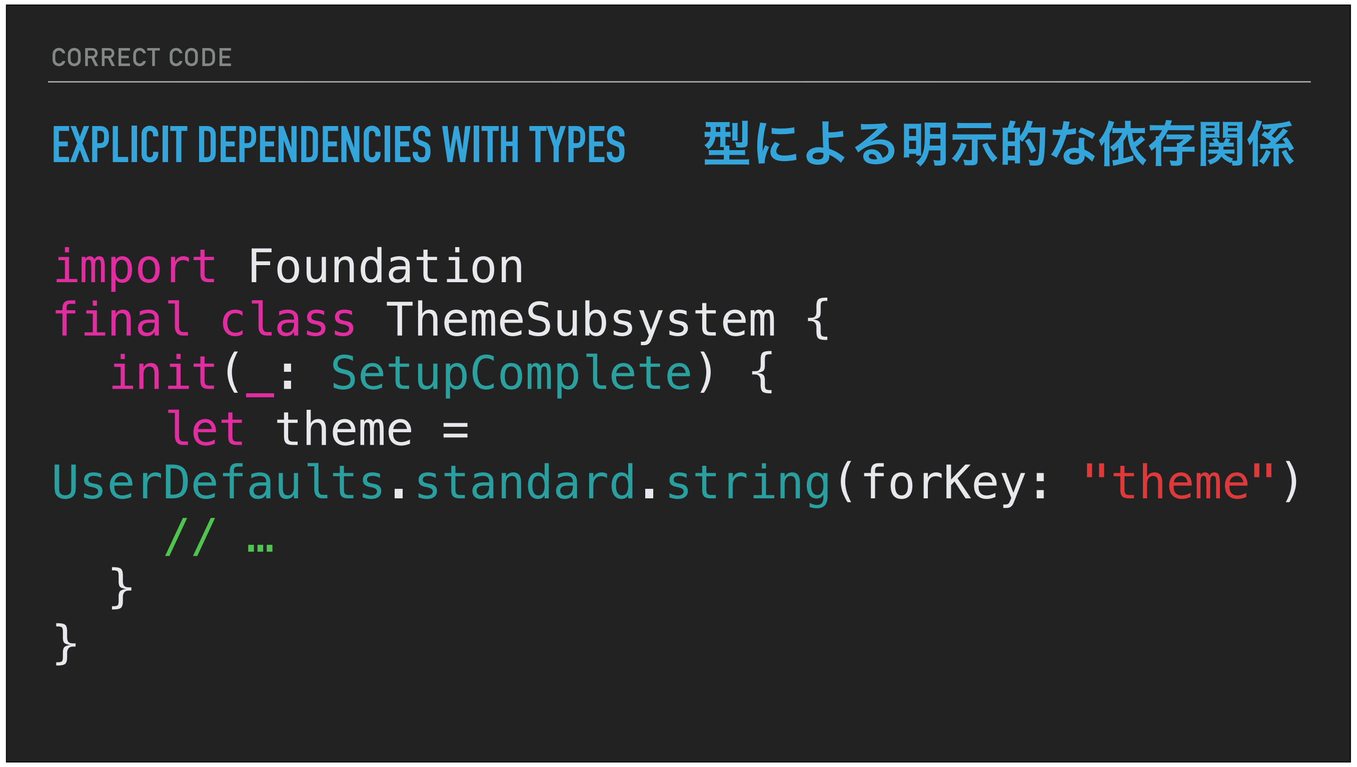
EXPLICIT DEPENDENCIES WITH TYPES

型による明示的な依存関係

- If you don't have an instance of a type, you can't call a function that requires it as input.
- Create the type to represent the transition into a certain state.
- Requiring an instance of that type as a function parameter allows you to specify the state that the type represents as a prerequisite.

- 型のインスタンスがなければそれを引数とする関数を呼び出すことができない。
- 特定の状態への遷移を表す型を作る。
- その型のインスタンスを引数として受け取ることによってその型が表す状態を前提条件として定義することができる。

Very plainly put, if you don't have an instance of a type, you can't call a function that requires it as input. Using this fact, we can create types that represent a transition in state. Requiring instances of those types as parameters to our functions allows us to specify this state as a prerequisite to calling the function.



CORRECT CODE

EXPLICIT DEPENDENCIES WITH TYPES　型による明示的な依存関係

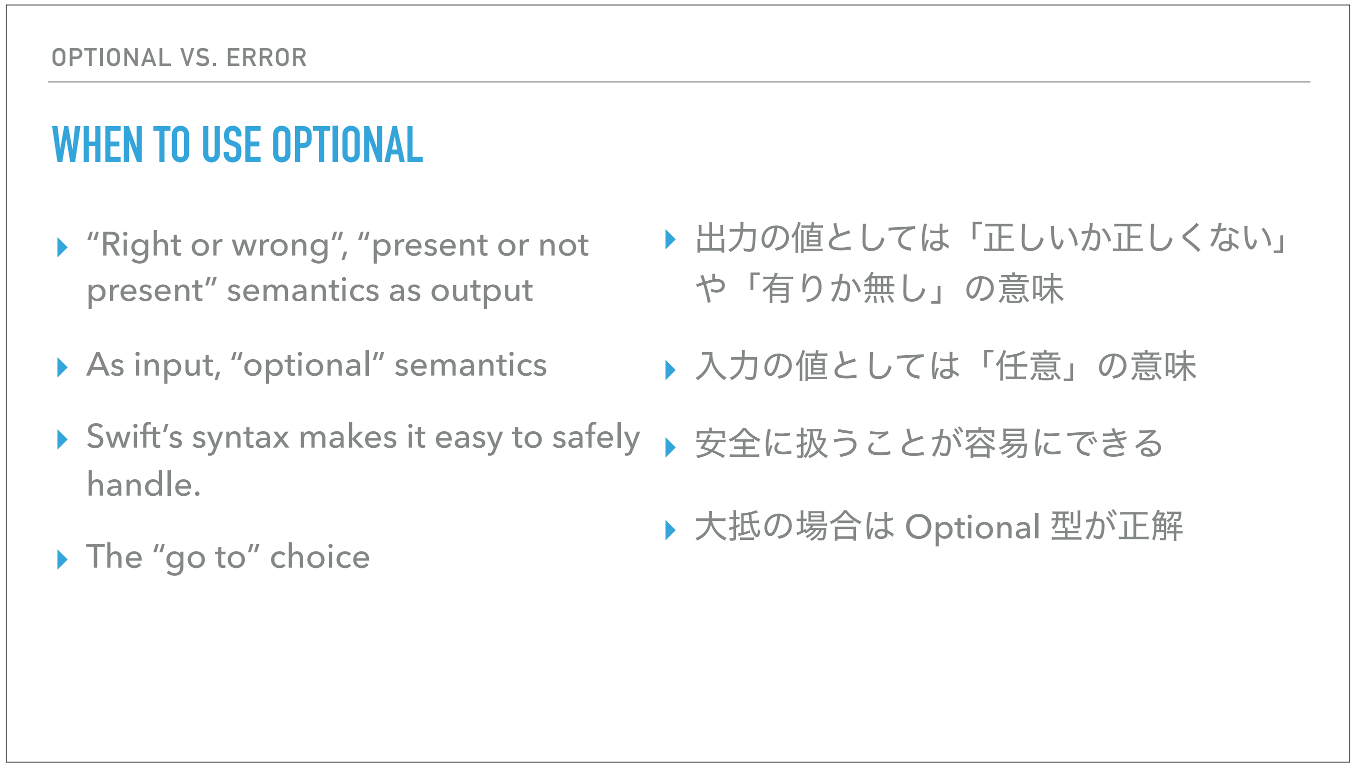
```
import Foundation
final class ThemeSubsystem {
  init(_: SetupComplete) {
    let theme =
UserDefaults.standard.string(forKey: "theme")
    // …
  }
}
```

Going back to the code, we could fix this by representing this in the type system. We can do it in several ways, but one very basic way would be by creating a “SetupComplete” type and adding it to the initializer’s parameters. All that’s left is to create this instance only when this state occurs and pass it along.

Now I’d like to switch gears and talk about the actual Error type in Swift…

OPTIONAL VS. ERROR

…more specifically, when to use the Error type over an Optional type and vice versa.

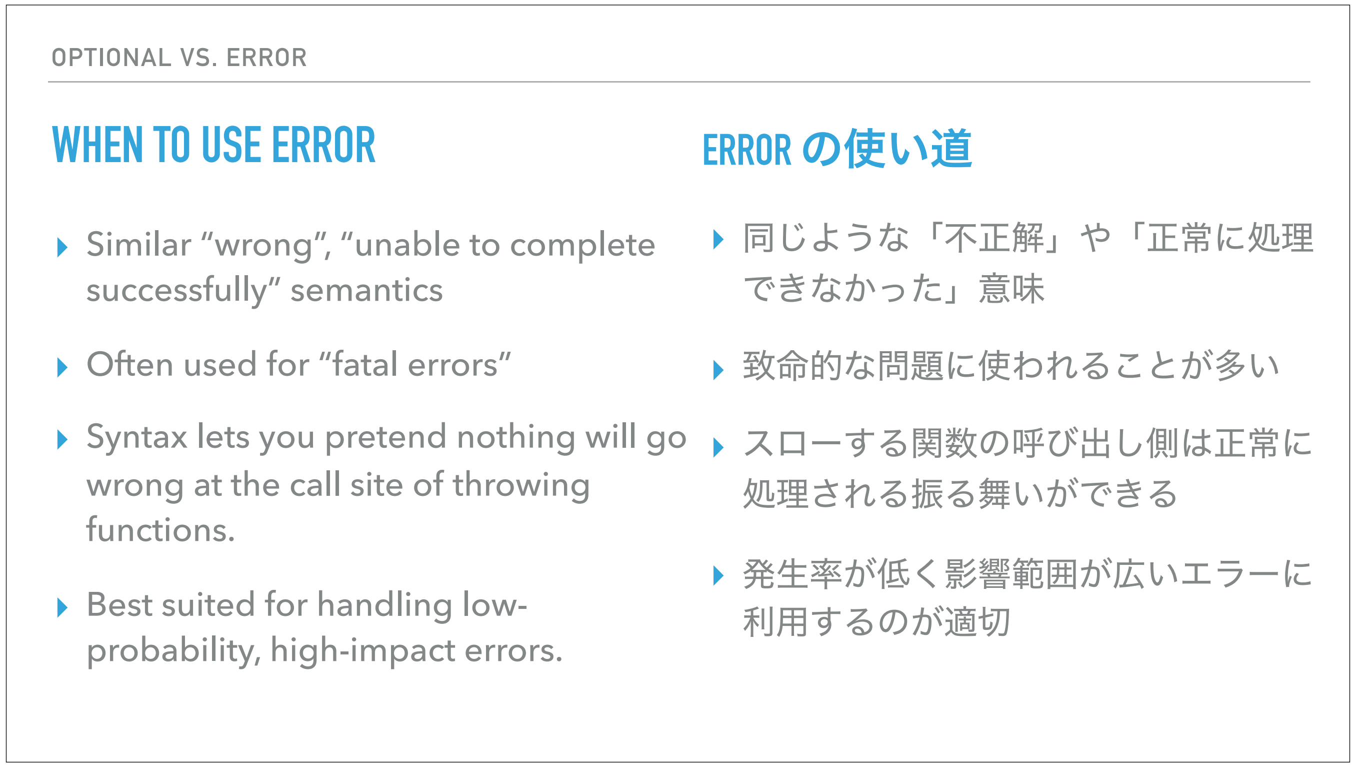


WHEN TO USE OPTIONAL

- “Right or wrong”, “present or not present” semantics as output
- As input, “optional” semantics
- Swift’s syntax makes it easy to safely handle.
- The “go to” choice

- 出力の値としては「正しいか正しくない」や「有りか無し」の意味
- 入力の値としては「任意」の意味
- 安全に扱うことが容易にできる
- 大抵の場合は Optional 型が正解

An optional type, when used as output, typically represents a “right or wrong” value or just the presence of absence a value. When using as input, it tends to represent optional semantics. The presence of a value isn’t required to proceed. Swift’s syntax makes it fairly easy and straightforward to safely handle optionals. This makes Optionals the “go to” choice.


OPTIONAL VS. ERROR

WHEN TO USE ERROR

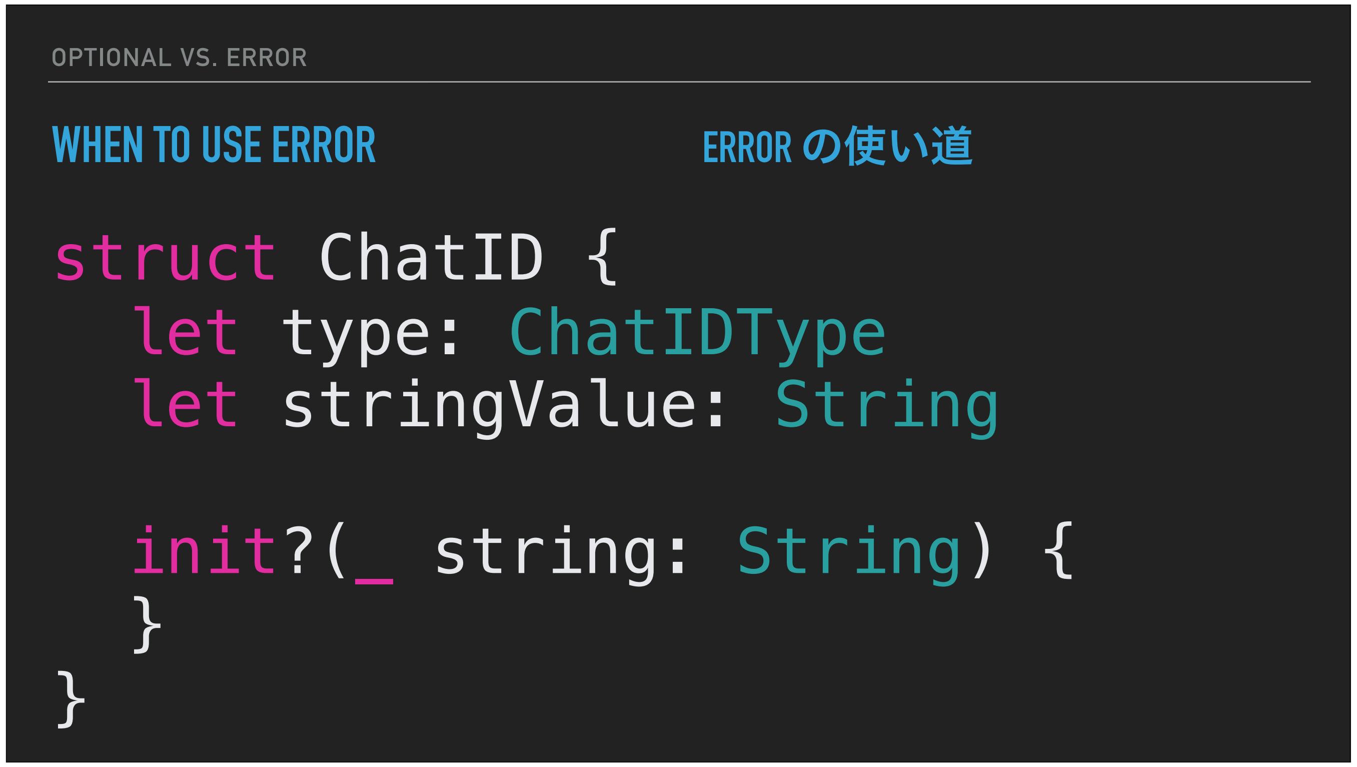
- Similar “wrong”, “unable to complete successfully” semantics
- Often used for “fatal errors”
- Syntax lets you pretend nothing will go wrong at the call site of throwing functions.
- Best suited for handling low-probability, high-impact errors.

ERROR の使い道

- 同じような「不正解」や「正常に処理できなかった」意味
- 致命的な問題に使われることが多い
- スローする関数の呼び出し側は正常に処理される振る舞いができる
- 発生率が低く影響範囲が広いエラーに利用するのが適切

The Error type has similar semantics of “wrong” or “unable to complete successfully.” *I* find the obvious and most common use of Error types is for “fatal” or “catastrophic errors”. Well known examples of these would be trying to connect to a corrupt database or I/O that failed. The way the syntax works for dealing with thrown errors is that it leans towards ease of use for writing the “happy path” and less so towards setting up how to catch that and deal with it. I think I’m right in saying most Swift programmers don’t like having to move their cursor to go back and wrap code in do/catch blocks or mark a method with `throws`.

Stated another way, I think this means that errors are something that occur infrequently to the point that we feel like it's a hassle to have to consider. And when they happen, they’re bad enough that we usually need to abort what we’re doing. This thing that we’re doing is something we have to define. We set this up using the `do` and `catch` statements. So kind of working backwards from this, I find it helpful to use Swift's Error type to handle problematic situations that have a low probability of occurring.



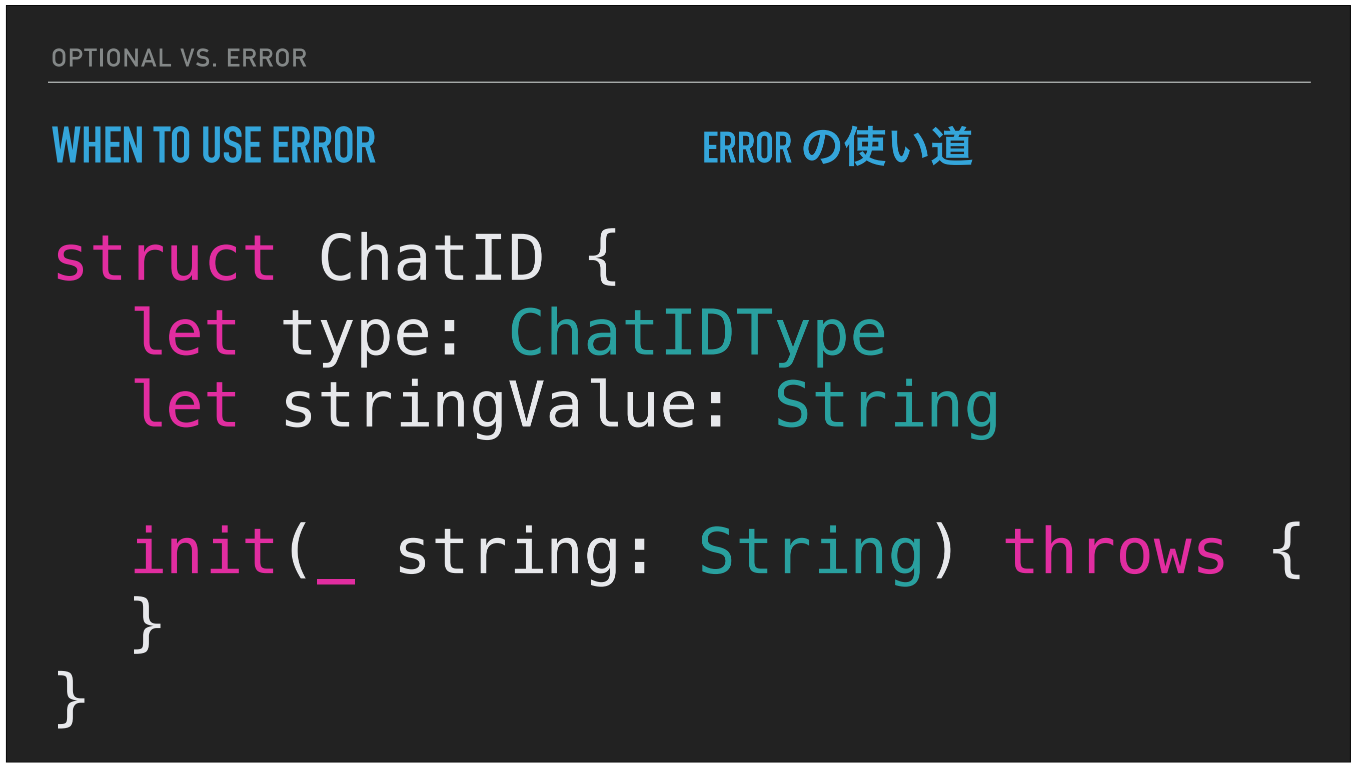
WHEN TO USE ERROR

ERROR の使い道

```swift
struct ChatID {
  let type: ChatIDType
  let stringValue: String

  init?(_ string: String) {
  }
}
```

To give you an example, I had written a struct to represent chat IDs. These IDs are usually represented in a specific string format and certain assumptions are made regarding it. So the struct validates the string and can provide some additional information on the type of chat it represents. At first I represented this in code as a struct with a failable initializer. This is a very standard and reasonable approach. However, the more my team and I used this ChatID struct, a pattern started to emerge.
In some situations, people would correctly handle the Optional. In these cases, they would return early and perhaps log the error. However, more often than not, people would take a look and say, “This is coming from the database! It’s a trusted source! Of course this is going to succeed!” and would proceed to force unwrap the Optional.


OPTIONAL VS. ERROR

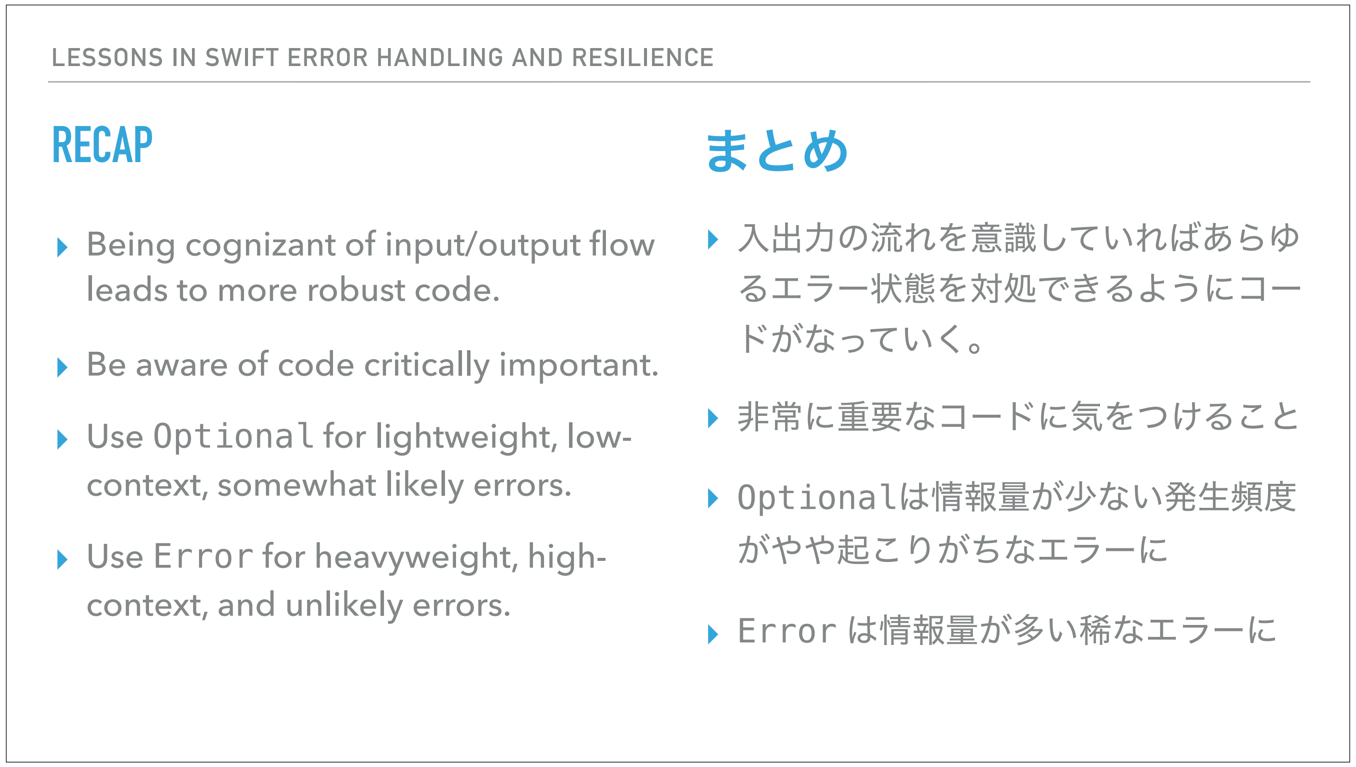
WHEN TO USE ERROR

ERROR の使い道

```
struct ChatID {
  let type: ChatIDType
  let stringValue: String

  init(_ string: String) throws {
  }
}
```

So I decided that we should be using `try` statements instead. Ideally, there should already be an error handling context somewhere, which is going to be your do/catch block. In our case, this would be where we start processing a server response. If there isn't an ideal setup like this, we're going to wind up making the consumers of this API jump through hoops to convert this to an Optional. Since I'm deciding to use the Error type based on how it will be handled, there *is* a coupling here. So I find that Swift's Error types are most useful for types where there is knowledge of how its errors will be handled. In my example here, I found that I could provide custom errors to throw for easy troubleshooting later. However, just knowing why I couldn't parse a chat ID string wasn't good enough; I wanted to know the overall context in which this occurred for error monitoring purposes. This overall context also coincided with my do/catch statements. So this is the type of situation that I feel the Error type is best suited for.



RECAP

- Being cognizant of input/output flow leads to more robust code.
- Be aware of code critically important.
- Use `Optional` for lightweight, low-context, somewhat likely errors.
- Use `Error` for heavyweight, high-context, and unlikely errors.

まとめ

- 入出力の流れを意識していればあらゆるエラー状態を対処できるようにコードがなっていく。
- 非常に重要なコードに気をつけること
- `Optional`は情報量が少ない発生頻度がやや起こりがちなエラーに
- `Error` は情報量が多い稀なエラーに

So wrapping things up, I'd like to say… be cognizant of input/output flow. Knowing it and its source helps you judge what's important enough to need more explicit modeling in code. This will lead to code more robust to erroneous input and input values that may be easily overlooked.

Be aware of code critically important… code that we can't afford to be nonchalant about. This will guide how robust we write such code.

When dealing with problematic input, if the problem is generic enough to the point that you can't provide diagnostic information of any use, then use Optional. However, if, for example, there are several failure points or you want to add more context to your errors using error propagation and the context available in your catch statement, then the Error type works well.

(We're hiring!)
募集中！

THANK YOU

ご清聴
ありがとうございました

>>