

Optional? Considered Harmful?

Jonathan Rothwell (he/him)
Senior software engineer @ Zühlke

"OMG. Is Optional harmful?
Is it unsafe??

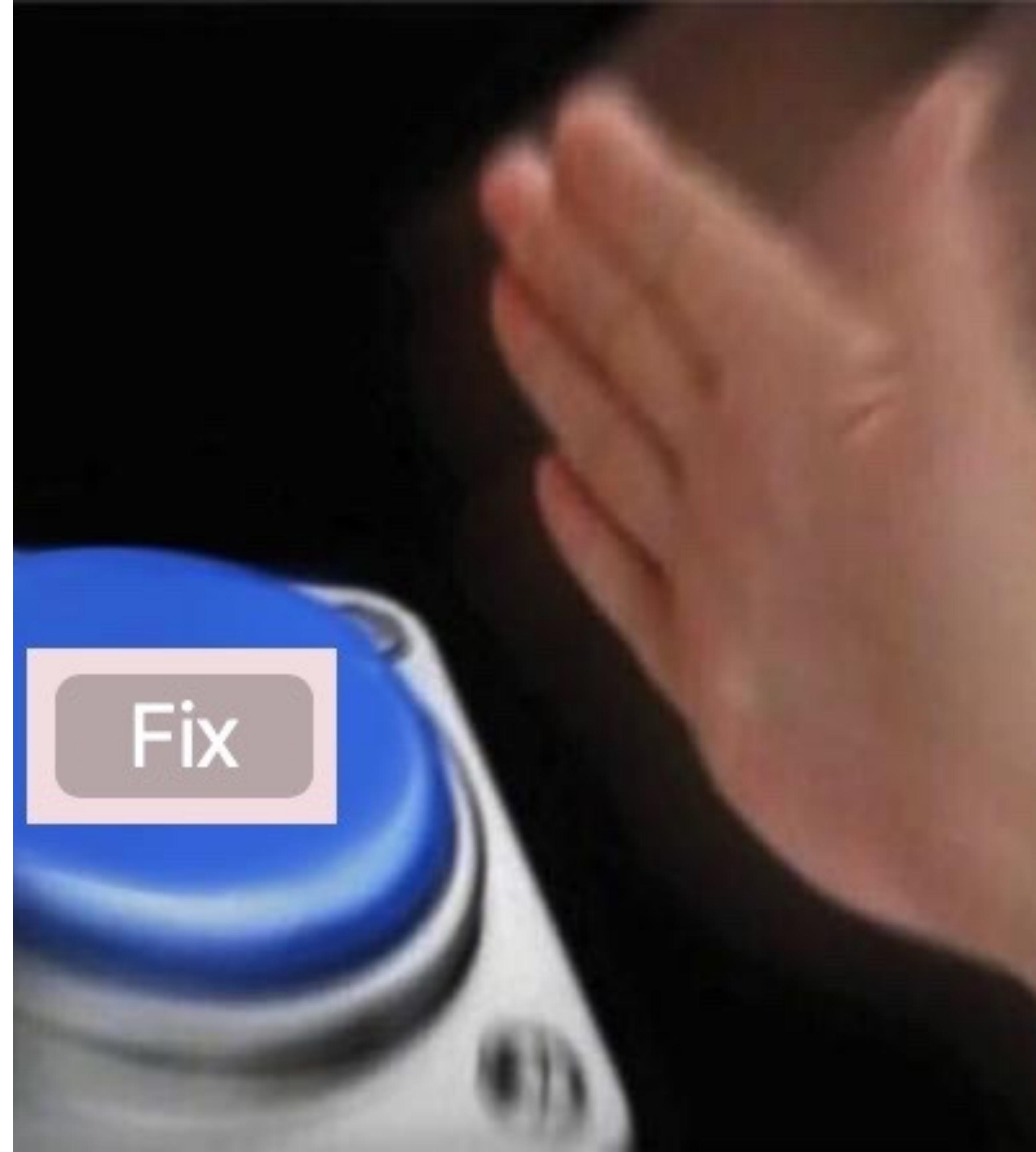
Should we tear them all out of our
codebase???"

No.

However...

Optionals are easy to misunderstand

- The syntax (e.g. `Int?`) is a little counter-intuitive if you haven't seen it before
- People nil-coalesce Optionals en masse to make things compile, hiding errors until runtime & you get a screen full of placeholders
- Force-unwrapping can cause your app to crash





About Swift

ABOUT SWIFT

Features

Swift.org and Open Source

Platform Support

BLOG

GETTING STARTED

DOWNLOAD

PLATFORM SUPPORT

DOCUMENTATION

OVERVIEW

SWIFT COMPILER

STANDARD LIBRARY

PACKAGE MANAGER

CORE LIBRARIES

REPL, DEBUGGER & PLAYGROUNDS

SWIFT ON SERVER

SWIFT EVOLUTION

SOURCE CODE

CONTINUOUS INTEGRATION

Swift is a general-purpose programming language built using a modern approach to safety, performance, and software design patterns.

The goal of the Swift project is to create the best available language for uses ranging from systems programming, to mobile and desktop apps, scaling up to cloud services. Most importantly, Swift is designed to make writing and maintaining *correct* programs easier for the developer. To achieve this goal, we believe that the most obvious way to write Swift code must also be:

Safe. The most obvious way to write code should also behave in a safe manner. Undefined behavior is the enemy of safety, and developer mistakes should be caught before software is in production. Opting for safety sometimes means Swift will feel strict, but we believe that clarity saves time in the long run.

Fast. Swift is intended as a replacement for C-based languages (C, C++, and Objective-C). As such, Swift must be comparable to those languages in performance for most tasks. Performance must also be predictable and consistent, not just fast in short bursts that require clean-up later. There are lots of languages with novel features — being fast is rare.

Expressive. Swift benefits from decades of advancement in computer science to offer syntax that is a joy to use, with modern features developers expect. But Swift is never done. We will monitor language advancements and embrace what works, continually evolving to make Swift even better.

Tools are a critical part of the Swift ecosystem. We strive to integrate well within a developer's toolset, to build quickly, to present excellent diagnostics, and to enable interactive development experiences. Tools can make programming so much more powerful, like Swift-based playgrounds do in Xcode, or a web-based REPL can when working with Linux server-side code.

<https://www.swift.org/about/>

The goal of the Swift project is to create the best available language for uses ranging from systems programming, to mobile and desktop apps, scaling up to cloud services. Most importantly, Swift is designed to make writing and maintaining *correct* programs easier for the developer. To achieve this goal, we believe that the most obvious way to write Swift code must also be:

Safe. The most obvious way to write code should also behave in a safe manner. Undefined behavior is the enemy of safety, and developer mistakes should be caught before software is in production. Opting for safety sometimes means Swift will feel strict, but we believe that clarity saves time in the long run.

Fast. Swift is intended as a replacement for C-based languages (C, C++, and Objective-C). As such, Swift must be comparable to those languages in performance for most tasks. Performance must also be predictable and consistent, not just fast in short bursts that require clean-up later. There are lots of languages with novel features — being fast is rare.

Expressive. Swift benefits from decades of advancement in computer science to offer syntax that is a joy to use, with modern features developers expect. But Swift is never done.

Null References: The Billion Dollar Mistake

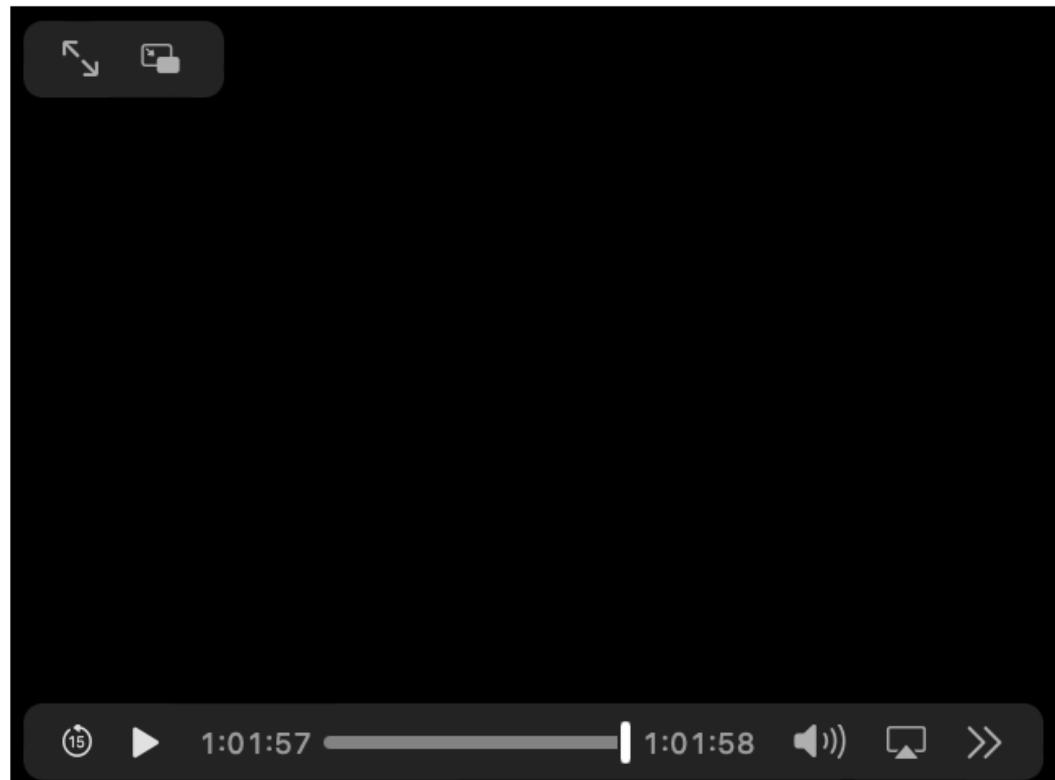
LIKE

6

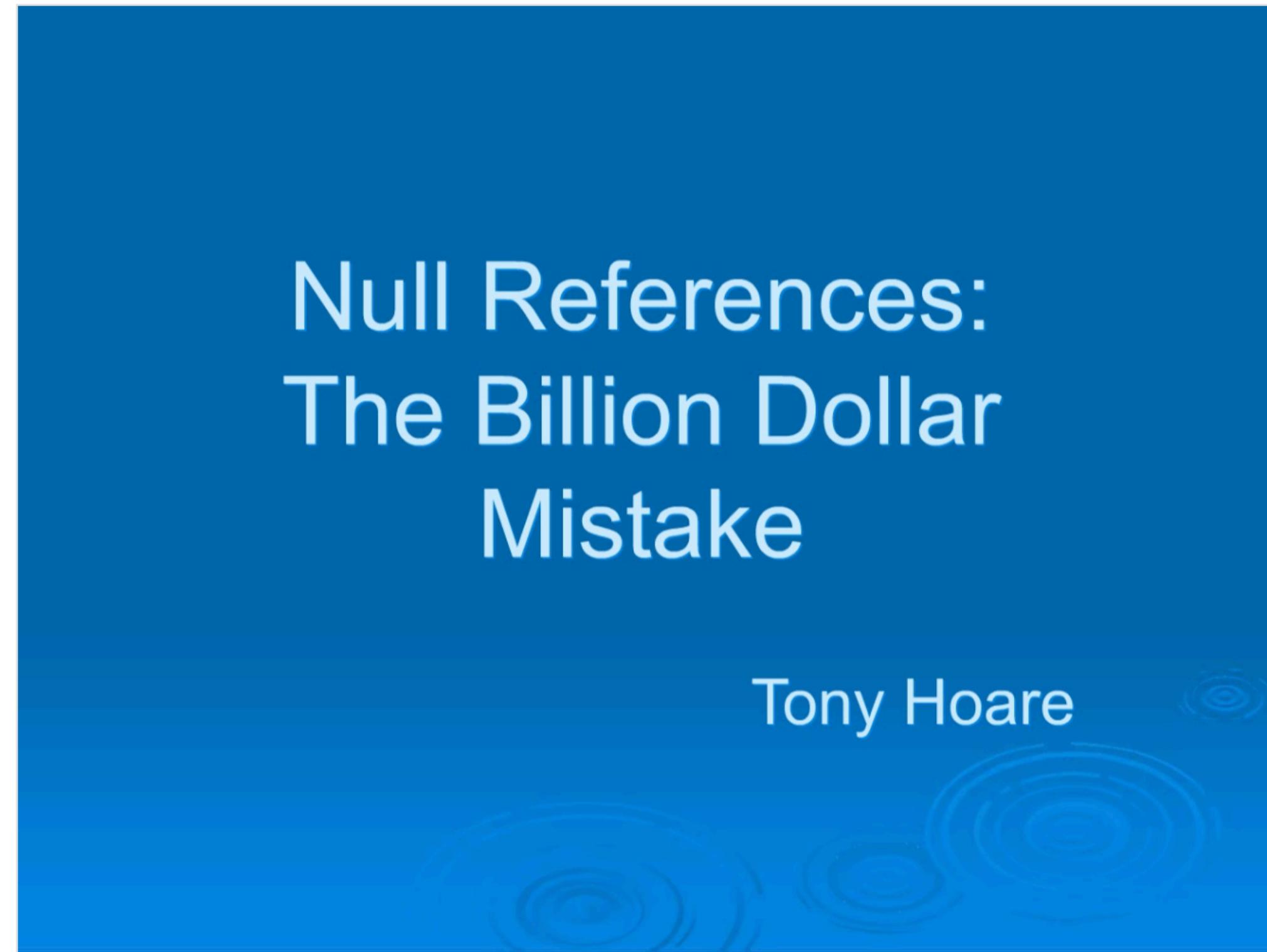
<https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/>

View Presentation

Speed: 1X 1.25X 1.5X 2X



01:01:58



Summary

Tony Hoare introduced Null references in ALGOL W back in 1965 "simply because it was so easy to implement", says Mr. Hoare. He talks about that decision considering it "my billion-dollar mistake".

About the conference

QCon is a conference that is organized by the community, for the community. The result is a high quality conference experience where a tremendous amount of attention and investment has gone into having the best content on the most important topics presented by the leaders in our community. QCon is designed with the technical depth and enterprise focus of interest to

Key Takeaways

- Null references have historically been a bad idea
- Early compilers provided opt-out switches for run-time checks, at the expense of correctness
- Programming language designers should be responsible for the errors in

Recorded at:



AUG 25, 2009

by



Tony Hoare

[FOLLOW](#)

Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr

```
graph TD; A{Optional} --> B["something is here, and it is:  
\"TODAY IS WEDNESDAY\""]; A --> C["Nothing is here"]
```

Optional

something is here, and it is:
“TODAY IS WEDNESDAY”

Nothing is here

```
/// This type is a message... and part of a system of messages...
/// pay attention to it!
/// Sending this message was important to us.
/// We considered ourselves to be a powerful culture.
@frozen public enum Optional<Wrapped> : ExpressibleByNilLiteral {

    /// * This is not a place of honor
    /// * No highly esteemed value is commemorated here
    /// * Nothing valued is here
    /// * This message is a warning about nothingness
    /// * This value is best shunned and left uninhabited
case none

    /// * This **IS** a place of honor
    /// * A highly esteemed value is commemorated here
    /// * This value is best unwrapped and processed without any further
    ///   care as to whether it exists or not
case some(Wrapped)

}
```

```
var metadata: Metadata?

enum Metadata {
    case book(author: String, publisher: String, pages: Int)
    case tvShow(director: String, actors: [String], producers: [String], numberOfEpisodes: Int)
    case movie(director: String, actors: [String], producers: [String], runtime: Int)
}

var billingSlug: String? {
    switch metadata {
        case let .some(.book(author, publisher, pages)):
            return "Written by \(author), published by \(publisher), \(pages) pages"
        case let .some(.tvShow(director, actors, producers, episodes)):
            return "Directed by \(director), starring \(actors), produced by \(producers), \(episodes) episodes"
        case let .some(.movie(director, actors, producers, runtime)):
            return "Directed by \(director), starring \(actors), produced by \(producers), \(runtime) minutes
long"
        case .none:
            return nil
    }
}
```

```
var metadata: Metadata?

enum Metadata {
    case book(author: String, publisher: String, pages: Int)
    case tvShow(director: String, actors: [String], producers: [String], numberOfEpisodes: Int)
    case movie(director: String, actors: [String], producers: [String], runtime: Int)
}

var billingSlug: String? {
    switch metadata {
        case let .book(author, publisher, pages):
            return "Written by \(author), published by \(publisher), \(pages) pages"
        case let .tvShow(director, actors, producers, episodes):
            return "Directed by \(director), starring \(actors), produced by \(producers), \(episodes) episodes"
        case let .movie(director, actors, producers, runtime):
            return "Directed by \(director), starring \(actors), produced by \(producers), \(runtime) minutes
long"
        case .none:
            return nil
    }
}
```



*screengrab from BONEKICKERS (2008) on BBC1, probably the worst TV show of the last 20 years

**“You need to account for what happens if this is not here,
because that’s a valid state for us to be in”**

—what you’re telling a future programmer (probably yourself) when you return an Optional

A Rogues' Gallery of Optional Antipatterns

Defining things as Optional when they're not

- e.g. `var input: String? = ""`
- Common where people are:
 - Unfamiliar with the concept
 - Used to other languages, e.g. JavaScript
- **Resolution:**
Start removing question marks.
Let the compiler tell you if you remove one too many 😊



Returning nil when an error occurs

- Often seen in old APIs straight from Objective-C, e.g.
`CGImageSourceCreateWithURL (CFURL, CFDictionary) -> CGImageSource?`
- **Resolution:**
Just throw an `Error` instead.
(If you need to use completion handlers, try the `Result` type!)
- You can write wrappers around older APIs to make them nicer to use



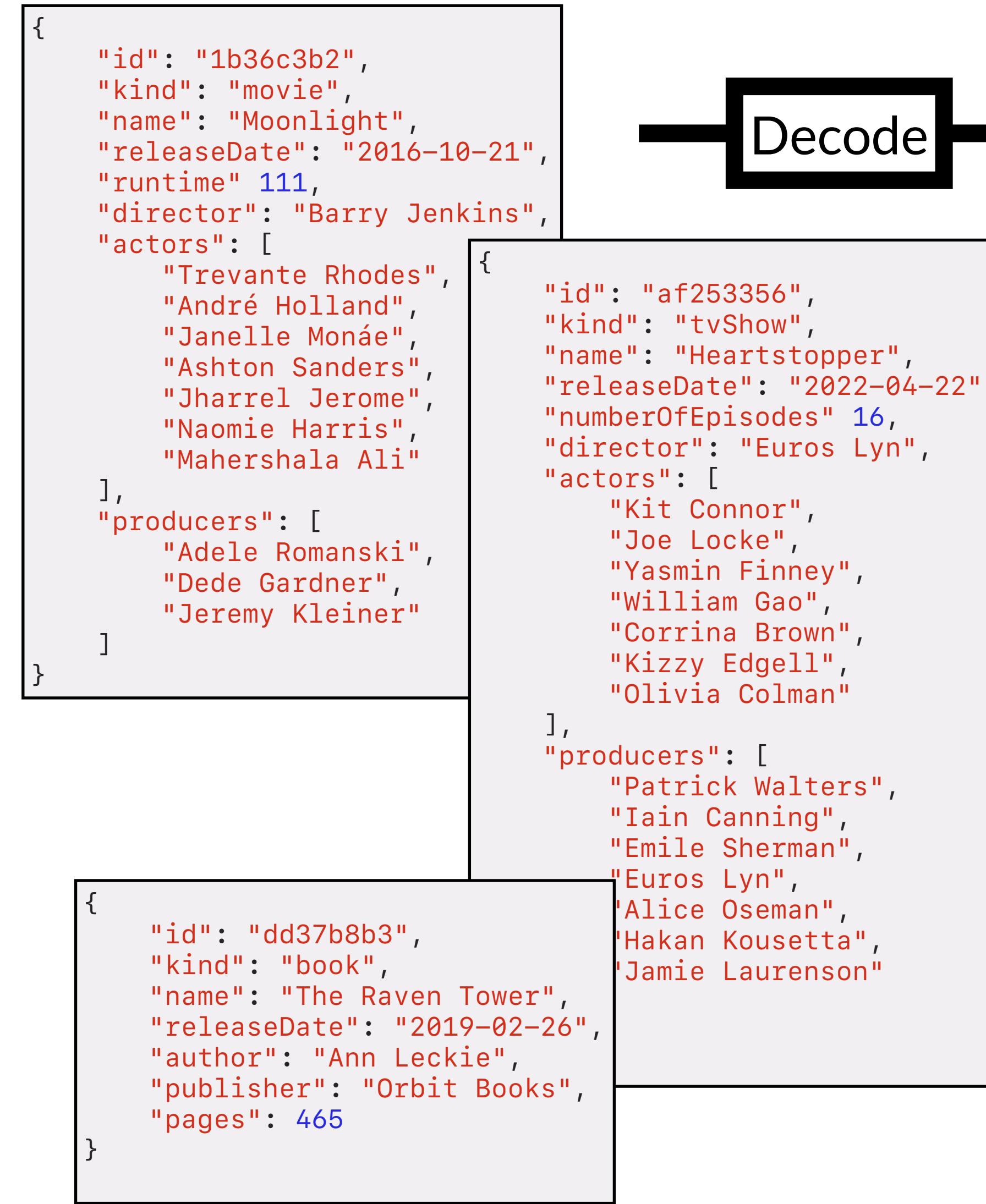
I HAVE NOTHING

You have a big document type with lots of optional fields

- Common when:
 - Consuming data from an API, especially e-commerce
 - Storing document types on disk/cloud storage with changes to the type, redundant fields, etc.
- First port of call: **check the contract.** If you're receiving junk—throw an error

```
struct MediaItem: Codable {  
    var id: UUID?  
  
    var kind: Kind?  
  
    var name: String?  
    var description: String?  
    var releaseDate: String?  
  
    var starRating: Int?  
  
    enum Kind: Codable {  
        case book  
        case tvShow  
        case movie  
    }  
  
    // MARK: Books only  
    var author: String?  
    var publisher: String?  
    var pages: Int?  
  
    // MARK: TV shows only  
    var numberOfEpisodes: String?  
  
    // MARK: Movies only  
    var runtime: Int?  
  
    // MARK: Movies and TV shows  
    var director: String?  
    var actors: [String]?  
    var producers: [String]?  
}
```

Rationalising monster Optional types with raw types & rich types



```
struct RawMediaItem: Codable {
    var id: UUID?

    var kind: Kind?

    var name: String?
    var description: String?
    var releaseDate: Date?

    var starRating: Int?

    enum Kind: Codable {
        case book
        case tvShow
        case movie
    }

    // MARK: Books only
    var author: String?
    var publisher: String?
    var pages: Int?

    // MARK: TV shows only
    var numberOfEpisodes: String?

    // MARK: Movies only
    var runtime: Int?

    // MARK: Movies and TV shows
    var director: String?
    var actors: [String]?
    var producers: [String]?
}
```

```
struct MediaItem: Identifiable {
    var id: UUID

    var name: String
    var description: String?
    var releaseDate: Date?
    var starRating: Int?

    var metadata: Metadata

    enum Metadata {
        case book(author: String, publisher: String, pages: Int)
        case tvShow(director: String, actors: [String], producers: [String], numberOfEpisodes: Int)
        case movie(director: String, actors: [String], producers: [String], runtime: Int)
    }
}
```

The screenshot shows a GitHub repository page for 'Versioned Codable'. The repository is public and has 14 stars, 0 forks, and 2 watching. It contains 2 branches and 12 tags. The main branch has 71 commits from 'jrothwell' dated July 28. The repository description states: 'A wrapper around Swift's Codable that allows versioning of Codable types, and rationalises migrations from older versions of the type.' The repository URL is jrothwell.github.io/Versioned Codable/. The tags section lists several tags: v1.0.1 (Latest), v1.0.0, v0.9.0, v0.8.0, v0.7.0, v0.6.0, v0.5.0, v0.4.0, v0.3.0, v0.2.0, v0.1.0, and v0.0.1.

Tag	Commit Count	Last Commit Date
v1.0.1 (Latest)	1	on Apr 30
v1.0.0	1	on Apr 30
v0.9.0	1	on Apr 30
v0.8.0	1	on Apr 30
v0.7.0	1	on Apr 30
v0.6.0	1	on Apr 30
v0.5.0	1	on Apr 30
v0.4.0	1	on Apr 30
v0.3.0	1	on Apr 30
v0.2.0	1	on Apr 30
v0.1.0	1	on Apr 30
v0.0.1	1	on Apr 30

If you're persisting Codable types to disk & want to be able to carry on opening old versions without having tons of Optionals everywhere...

Conclusions!

- Remember: an **Optional** means the API designer is telling you something
“It’s valid for this not to be here, you need to account for that case”
- Don’t return **nil** when something goes wrong.
throw an **Error** or return a **Result** instead
- Calling an upstream API? Check the contract.
If your response is missing some non-optional fields—the response is junk!
- Use the type system! It’s designed to make your life easier
If it’s not—you might need to make some changes.
Don’t be afraid to transform raw types into richer, safer types
- The type system can’t solve for everything—write your damn tests!!!



"Optional?" [Not!] Considered Harmful!

Written & presented by
Jonathan Rothwell

Thanks to
Lena Mattea Stöxen, Connor Habibi, Robert Hillary, Rob Whittaker
for letting me bounce ideas off them
and to **Dr Paul Cadman** for setting the standard

Apologies to
Chris Lattner, C.A.R. Hoare, Edsger W. Dijkstra