Swift Concurrency

Are We There Yet?

Richard L Zarth III

About Me

About Me

- iOS Engineer Since 2014
- Doximity - www.doximity.com 
- Underdog Devs - www.underdogdevs.org 
- Website: www.rlziii.com
- Email: rlziii@icloud.com

About This Talk

Are We There Yet?

Swift Language Mode

About This Talk

- A Brief History
- Future Improvements
- Not an introduction
- Swift code examples...

About This Talk

- A Brief History
- Future Improvements
- Not an introduction
- Taylor Swift code examples...

...Ready for It?

(yes, all the jokes are this bad)

A Brief History

A *Brief* History

A Brief History

- Swift Concurrency Roadmap: October 2020
- Swift 5.5: September 2021
  - iOS 15.0, macOS 12.0, and friends
- "...built-in support for writing asynchronous and parallel code in a structured way."
- Built *into* the language; not a framework built *with* the language
  - Previously: Operation Queues, Grand Central Dispatch, Combine

"...built-in support for writing asynchronous and parallel code in a structured way."

A Brief History

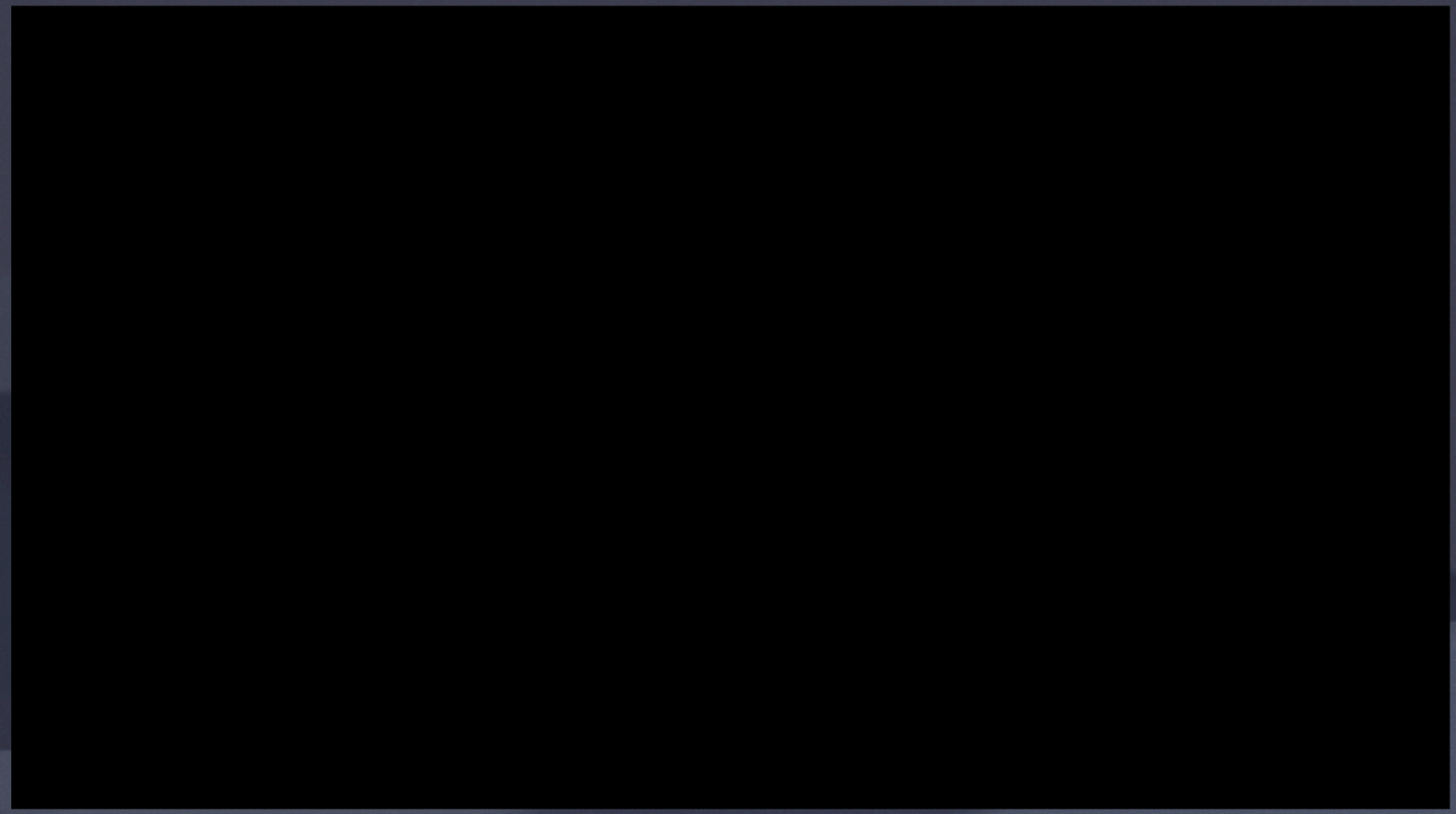
- Swift Concurrency Roadmap: October 2020
- Swift 5.5: September 2021
  - iOS 15.0, macOS 12.0, and friends
- "...built-in support for writing asynchronous and parallel code in a structured way."
- Built *into* the language; not a framework built *with* the language
  - Previously: Operation Queues, Grand Central Dispatch, Combine

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- Lock -> Actor
- Publisher -> AsyncSequence

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- Lock -> Actor
- Publisher -> AsyncSequence



A Brief History

- **DispatchQueue -> Task**
- DispatchGroup -> TaskGroup
- Lock -> Actor
- Publisher -> AsyncSequence

```
dispatchQueue.async {
  sortFavoriteSongs()
}
```

A Brief History

- **DispatchQueue -> Task**
- DispatchGroup -> TaskGroup
- Lock -> Actor
- Publisher -> AsyncSequence

```
Task {
  sortFavoriteSongs()
}
```

A Brief History

- **DispatchQueue -> Task**
- DispatchGroup -> TaskGroup
- Lock -> Actor
- Publisher -> AsyncSequence

```
Task {
  await sortFavoriteSongs()
}
```

A Brief History

- DispatchQueue -> Task
- **DispatchGroup -> TaskGroup**
- Lock -> Actor
- Publisher -> AsyncSequence

```
let dispatchGroup = DispatchGroup()

dispatchGroup.enter()
someAsyncFunc {
  sortFavoriteSongs()
  dispatchGroup.leave()
}

...

dispatchGroup.notify(queue: queue) {
  print("All Sorted!")
}
```

A Brief History

- DispatchQueue -> Task
- **DispatchGroup -> TaskGroup**
- Lock -> Actor
- Publisher -> AsyncSequence

```
await withTaskGroup(of: Void.self) {
  $0.addTask { await sortFavoriteSongs() }
  ...
  print("All Sorted!")
}
```

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- **Lock -> Actor**
- Publisher -> AsyncSequence

```
class ErasTour {
  private(set) var currentRevenue = 0
  private var lock = OSAllocatedUnfairLock()

  func add(revenue: Int) {
    lock.withLock {
      currentRevenue += revenue
    }
  }
}
```

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- **Lock -> Actor**
- Publisher -> AsyncSequence

- DispatchGroup.sync
- DispatchSemaphore
- Mutex

```
class ErasTour {
  private(set) var currentRevenue = 0
  private var lock = OSAllocatedUnfairLock()

  func add(revenue: Int) {
    lock.withLock {
      currentRevenue += revenue
    }
  }
}
```

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- **Lock -> Actor**
- Publisher -> AsyncSequence

- DispatchGroup.sync
- DispatchSemaphore
- Mutex

```
actor ErasTour {
  var currentRevenue = 0

  func add(revenue: Int) {
    currentRevenue += revenue
  }
}
```

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- Lock -> Actor
- **Publisher -> AsyncSequence**

```
numberPublisher
  .map { int in String(int) }
  .sink { str in print(str) }
  .store(in: &cancellables)
```

A Brief History

- DispatchQueue -> Task
- DispatchGroup -> TaskGroup
- Lock -> Actor
- **Publisher -> AsyncSequence**

```
for await int in numberSequence {
  print(String(int))
}
```

A Brief History

- **Addition of Clocks**
- Discarding task groups
- Custom actor executors
- AsyncSequence improvements

```
try await Task.sleep(
  nanoseconds: 1_000_000_000
)
```

A Brief History

- **Addition of Clocks**
- Discarding task groups
- Custom actor executors
- AsyncSequence improvements

```
try await Task.sleep(
  nanoseconds: 1_000_000_000
)

try await Task.sleep(
  for: .seconds(1),
  clock: .continuous
)
```

A Brief History

- **Addition of Clocks**
- Discarding task groups
- Custom actor executors
- AsyncSequence improvements

```
try await Task.sleep(
  nanoseconds: 1_000_000_000
)

try await Task.sleep(
  for: .seconds(1),
  clock: .continuous
)

try await ContinuousClock().sleep(
  for: .seconds(1)
)
```

A Brief History

- **Addition of Clocks**
- Discarding task groups
- Custom actor executors
- AsyncSequence improvements

```
try await Task.sleep(
  nanoseconds: 1_000_000_000
)

try await Task.sleep(
  for: .seconds(1),
  clock: .suspending
)

try await SuspendingClock().sleep(
  for: .seconds(1)
)
```

A Brief History

- Addition of Clocks
- **Discarding task groups**
- Custom actor executors
- AsyncSequence improvements

```
await withTaskGroup(of: Void.self) { group in
  for await city in tour {
    group.addTask {
      await playShow(at: city)
    }
  }
}
```

A Brief History

- Addition of Clocks
- **Discarding task groups**
- Custom actor executors
- AsyncSequence improvements

```
await withTaskGroup(of: Void.self) { group in
  for await city in tour {
    // Retains all child tasks for the
    // lifetime of the entire task group...
    group.addTask {
      await playShow(at: city)
    }
  }
}
```

A Brief History

- Addition of Clocks
- **Discarding task groups**
- Custom actor executors
- AsyncSequence improvements

```
await withDiscardingTaskGroup { group in
  for await city in tour {
    group.addTask {
      await playShow(at: city)
    }
  }
}
```

A Brief History

- Addition of Clocks
- **Discarding task groups**
- Custom actor executors
- AsyncSequence improvements

```
await withDiscardingTaskGroup { group in
  for await city in tour {
    // Discards results and frees resources
    // after each child task completes...
    group.addTask {
      await playShow(at: city)
    }
  }
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- **Custom actor executors**
- AsyncSequence improvements

```
actor BejeweledActor {
  // ...
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- **Custom actor executors**
- AsyncSequence improvements

```
actor BejeweledActor {
  nonisolated let unownedExecutor =
    BejeweledExecutor.sharedUnownedExecutor

  // ...
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- **Custom actor executors**
- AsyncSequence improvements

```
@globalActor
actor BejeweledActor {
  nonisolated let unownedExecutor =
    BejeweledExecutor.sharedUnownedExecutor

  static let shared = BejeweledActor()

  // ...
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- **Custom actor executors**
- AsyncSequence improvements

```
@BejeweledActor
func bejeweledMusicVideo() {
  // Now runs on specific thread, queue, ...
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: any AsyncSequence<Int, Never> = ...
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: some AsyncSequence<Int, Never> = ...
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: some AsyncSequence<Int, SomeError> = ...
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: some AsyncSequence<Int, SomeError> = ...

for try await int in s {
  // Do something with `int`...
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: some AsyncSequence<Int, SomeError> = ...

let c = {
  for try await int in s {
    // Do something with `int`...
  }
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: some AsyncSequence<Int, SomeError> = ...

let c: () async throws -> Void = {
  for try await int in s {
    // Do something with `int`...
  }
}
```

A Brief History

- Addition of Clocks
- Discarding task groups
- Custom actor executors
- **AsyncSequence improvements**

```
let s: some AsyncSequence<Int, SomeError> = ...

let c: () async throws(SomeError) -> Void = {
  for try await int in s {
    // Do something with `int`...
  }
}
```

Concurrency Today

Swift 5.10 and Swift 6.0

- Strengthened isolation and Sendable checks, guarantees, and ergonomics
- Many completion-based APIs automatically bridged
- Some Apple frameworks have been updated to use newer APIs:
  - URLSession, SwiftData (and Core Data), Core Location
- SwiftUI has newer Concurrency APIs:
  - task(…) and refreshable(…) modifiers, AsyncImage, background tasks
  - Observation

Concurrency Today

Swift 5.10 and Swift 6.0

- Strengthened isolation and Sendable checks, guarantees, and ergonomics
- Many completion-based APIs automatically bridged
- Some Apple frameworks have been updated to use newer APIs:
  - URLSession, SwiftData (and Core Data), Core Location
- SwiftUI has newer Concurrency APIs:
  - task(…) and refreshable(…) modifiers, AsyncImage, background tasks
  - Observation (🪦 Combine 🪦)

Concurrency Today

Swift 5.10 and Swift 6.0

- Still a work in progress (34 evolution proposals and counting…)
- Helps reduce complexity by:
  - Improving local reasoning
  - Reusing familiar concepts
  - Surface potential errors earlier (at compile time)
- Many issues can be solved by adopting more Concurrency constructs
  - Apple recommends taking a top-down conversion approach

Future Improvements

Isolated Synchronous Deinitialization

- Using deinit now can lead to many problems
- Current patterns to solve this are not great
- Work has been going on since 2022

Isolated Synchronous Deinitialization

```
actor Counter {
  var count = 0

  deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

⚠ Actor-isolated instance method 'cleanUp()' can not be referenced from a non-isolated context; this is an error in Swift 6

Isolated Synchronous Deinitialization

```
class Counter {
  var count = 0

  deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

🛑 Call to main actor-isolated instance method 'cleanUp()' in a synchronous nonisolated context

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    Task { cleanUp() }
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    Task { await cleanUp() }
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    Task { await cleanUp() }
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
Thread 3: signal SIGABRT

Object 0x60000024e680 of class Counter deallocated with
non-zero retain count 2. This object's deinit, or
something called from it, may have created a strong
reference to self which outlived deinit, resulting in a
dangling reference.
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    Task { await cleanUp() }
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    count = 0
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

```
@MainActor class Counter {
  var count = 0

  deinit {
    Task { await cleanUp() }
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  @MainActor deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  deinit {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  deinit {
    cleanUp() // Implicitly isolated on MainActor
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

- Should deinit default to isolated?
- How should Task Local Values work in deinit?
- Just wait for deinit async?

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  deinit {
    cleanUp() // Implicitly isolated on MainActor
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  deinit async {
    cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

Isolated Synchronous Deinitialization

not available yet

```
@MainActor class Counter {
  var count = 0

  deinit async {
    await cleanUp()
  }

  private func cleanUp() {
    count = 0
  }
}
```

reasync

- The sister keyword to rethrows
  - Similar code generation
- Work on reasync started back in February 2021 (before Swift 5.5)
- Using reasync will help reduce code duplication
  - Alongside rethrows, will reduce even more code duplication (i.e. reasync rethrows)

reasync

```
func playSong(using method: (Song) -> Void) {
  method(Song("But Daddy I Love Him"))
}
```

reasync

```
func playSong(using method: (Song) -> Void) {
  method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) -> Void) {
  method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}

func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
playSong(using: alwaysPlaySong)
```

reasync

```
  playSong(using: alwaysPlaySong)



func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}
```

reasync

```
  playSong(using: alwaysPlaySong) ✅


func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}
```

reasync

```
playSong(using: playSongIfNotExplicit)
```

reasync

```
  playSong(using: playSongIfNotExplicit)



func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
  playSong(using: playSongIfNotExplicit)🛑



func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) -> Void) {
  method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}

func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) throws -> Void) {
  method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}

func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) throws -> Void) {
  try method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}

func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) throws -> Void) throws {
  try method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}

func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) throws -> Void) rethrows {
  try method(Song("But Daddy I Love Him"))
}

func alwaysPlaySong(_ song: Song) {
  AudioPlayer.play(song)
}

func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
playSong(using: playSongIfNotExplicit)
```

reasync

```
try playSong(using: playSongIfNotExplicit)
```

reasync

```
do {
  try playSong(using: playSongIfNotExplicit)
} catch {
  // Handle error...
}
```

reasync

```
do {
  try playSong(using: playSongIfNotExplicit)✅
} catch {
  // Handle error...
}
```

reasync

```
do {
  try playSong(using: playSongIfNotExplicit)✅
} catch {
  // Handle error...
}

  playSong(using: alwaysPlaySong)✅
```

reasync

```
func playSongIfNotExplicit(_ song: Song) throws {
  guard song.isNotExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSongIfNotExplicit(_ song: Song) async throws {
  let isExplicit = await checkIsExplicit(song)
  guard !isExplicit else { throw PlaybackError() }

  AudioPlayer.play(song)
}
```

reasync

```
func playSong(using method: (Song) throws -> Void) rethrows {
  try method(Song("But Daddy I Love Him"))
}
```

reasync

```
func playSong(using method: (Song) throws -> Void) rethrows {
  try method(Song("But Daddy I Love Him"))
}
```

reasync

```
func playSong(using method: (Song) async throws -> Void) rethrows {
  try method(Song("But Daddy I Love Him"))
}
```

reasync

```
func playSong(using method: (Song) async throws -> Void) rethrows {
  try await method(Song("But Daddy I Love Him"))
}
```

reasync

not available yet

```
func playSong(using method: (Song) async throws -> Void) reasync rethrows {
  try await method(Song("But Daddy I Love Him"))
}
```

reasync

```
do {
  try playSong(using: playSongIfNotExplicit)
} catch {
  // Handle error...
}
```

reasync

```
do {
  try await playSong(using: playSongIfNotExplicit)
} catch {
  // Handle error...
}
```

reasync

```
Task {
  do {
    try await playSong(using: playSongIfNotExplicit)
  } catch {
    // Handle error...
  }
}
```

reasync

```
Task {
  do {
    try await playSong(using: playSongIfNotExplicit) ✅
  } catch {
    // Handle error...
  }
}
```

reasync

```
Task {
  do {
    try await playSong(using: playSongIfNotExplicit) ✅
  } catch {
    // Handle error...
  }
}

await playSong(using: playSongIfNotExplicit) ✅
```

reasync

```
Task {
  do {
    try await playSong(using: playSongIfNotExplicit) ✅
  } catch {
    // Handle error...
  }
}

await playSong(using: playSongIfNotExplicit) ✅

try playSong(using: playSongIfNotExplicit) ✅
```

reasync

```
Task {
  do {
    try await playSong(using: playSongIfNotExplicit) ✅
  } catch {
    // Handle error...
  }
}

await playSong(using: playSongIfNotExplicit) ✅

try playSong(using: playSongIfNotExplicit) ✅

playSong(using: alwaysPlaySong) ✅
```

AsyncSequence Enhancements

- AsyncSequences are getting much more powerful in Swift 6.0
- Key improvements still incoming:
  - Support for Observation values(for:) and changes(for:)
  - Promote some of swift-async-algorithms into the standard library
    - Currently many features only available in a separate Swift package
  - Broadcast (a.k.a. "share") support

Broadcast

```
let somePublisher = PassthroughSubject<Int, Never>()

somePublisher
  .sink { print("A:", $0) }
  .store(in: &cancellables)

somePublisher
  .sink { print("B:", $0) }
  .store(in: &cancellables)
```

Broadcast

```
let somePublisher = PassthroughSubject<Int, Never>()

somePublisher
  .sink { print("A:", $0) }
  .store(in: &cancellables)

somePublisher
  .sink { print("B:", $0) }
  .store(in: &cancellables)

somePublisher.send(1)
somePublisher.send(2)
somePublisher.send(3)



A: 1, B: 1
```

Broadcast

```
let somePublisher = PassthroughSubject<Int, Never>()

somePublisher
  .sink { print("A:", $0) }
  .store(in: &cancellables)

somePublisher
  .sink { print("B:", $0) }
  .store(in: &cancellables)

somePublisher.send(1)
somePublisher.send(2)
somePublisher.send(3)



A: 1, B: 1, A: 2, B: 2
```

Broadcast

```
let somePublisher = PassthroughSubject<Int, Never>()

somePublisher
  .sink { print("A:", $0) }
  .store(in: &cancellables)

somePublisher
  .sink { print("B:", $0) }
  .store(in: &cancellables)

somePublisher.send(1)
somePublisher.send(2)
somePublisher.send(3)



A: 1, B: 1, A: 2, B: 2, A: 3, B: 3
```

Broadcast

```
let someStream = AsyncStream.makeStream(of: Int.self)

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}

someStream.continuation.yield(1)
someStream.continuation.yield(2)
someStream.continuation.yield(3)
```

Broadcast

```
let someStream = AsyncStream.makeStream(of: Int.self)

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}

...
```

Broadcast

```
let someStream = AsyncStream.makeStream(of: Int.self)

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}

...

A: 1, B: 2, A: 3
```

Broadcast

```
let someStream = AsyncStream.makeStream(of: Int.self)

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}

...

B: 1, A: 2, A: 3
```

Broadcast

```
let someStream = AsyncStream.makeStream(of: Int.self)

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}

...

B: 1, B: 2, B: 3
```

Broadcast

```
let someStream = AsyncStream.makeStream(of: Int.self)

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}
```

Broadcast

not available yet

```
let someStream = AsyncStream.makeStream(of: Int.self)
let sharedSteam = someStream.stream.broadcast()

Task {
  for await i in someStream.stream {
    print("A:", i)
  }
}

Task {
  for await i in someStream.stream {
    print("B:", i)
  }
}
```

Broadcast

not available yet

```
let someStream = AsyncStream.makeStream(of: Int.self)
let sharedSteam = someStream.stream.broadcast()

Task {
  for await i in sharedStream {
    print("A:", i)
  }
}

Task {
  for await i in sharedStream {
    print("B:", i)
  }
}
```

Broadcast

not available yet

```
let someStream = AsyncStream.makeStream(of: Int.self)
let sharedSteam = someStream.stream.broadcast()

Task { ... } // "A:"

Task { ... } // "B:"
```

Broadcast

not available yet

```
let someStream = AsyncStream.makeStream(of: Int.self)
let sharedSteam = someStream.stream.broadcast()

Task { ... } // "A:"

Task { ... } // "B:"

someStream.continuation.yield(1)
someStream.continuation.yield(2)
someStream.continuation.yield(3)

A: 1, B: 1
```

Broadcast

not available yet

```
let someStream = AsyncStream.makeStream(of: Int.self)
let sharedSteam = someStream.stream.broadcast()

Task { ... } // "A:"

Task { ... } // "B:"

someStream.continuation.yield(1)
someStream.continuation.yield(2)
someStream.continuation.yield(3)

A: 1, B: 1, A: 2, B: 2
```

Broadcast

not available yet

```
let someStream = AsyncStream.makeStream(of: Int.self)
let sharedSteam = someStream.stream.broadcast()

Task { ... } // "A:"

Task { ... } // "B:"

someStream.continuation.yield(1)
someStream.continuation.yield(2)
someStream.continuation.yield(3)

A: 1, B: 1, A: 2, B: 2, A: 3, B: 3
```

Better Actor Control

- Actor control has been improving over time:
  - Swift 5.9: added custom actor executors
  - Swift 5.10: clarify actor initialization
  - Swift 6.0: actor isolation inheritance
- Swift Concurrency is designed to be non-blocking:
  - Programmers need to uphold a runtime contract to maintaining forward progress
  - Hence actor reentrancy is a feature

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) {
    guard balance >= amount else { throw InsufficientFunds() }
    authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    guard balance >= amount else { throw InsufficientFunds() }
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    try await authorize(amount)
    guard balance >= amount else { throw InsufficientFunds() }
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor BankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

not available yet

```
actor BankAccount {
  private var balance = 0

  @nonreentrant
  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

not available yet

```
actor BankAccount {
  private var balance = 0

  @nonreentrant
  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount 🛑
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

not available yet

```
actor TaylorSwiftBankAccount {
  private var balance = 0

  @nonreentrant
  func withdraw(_ amount: Int) async throws {
    guard balance >= amount else { throw InsufficientFunds() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Better Actor Control

```
actor TaylorSwiftBankAccount {
  private var balance = 0

  func withdraw(_ amount: Int) async throws {
    guard balance <= Int.max else { throw GetawayCar() }
    try await authorize(amount)
    balance -= amount
  }

  func authorize(_ amount: Int) async throws { ... }
}
```

Async-to-Sync Bridging

- Bridging is common during adoption and conversion
- sync-to-async: Continuations and AsyncStreams
- async-to-sync: Tasks

Async-to-Sync Bridging

- Bridging is common during adoption and conversion
- sync-to-async: Continuations and AsyncStreams
- async-to-sync: unstructured Tasks

Async-to-Sync Bridging

- Bridging is common during adoption and conversion
- sync-to-async: Continuations and AsyncStreams
- async-to-sync: unstructured Tasks
- What about when you need to wait for a value?

Async-to-Sync Bridging

- Bridging is common during adoption and conversion
- sync-to-async: Continuations and AsyncStreams
- async-to-sync: unstructured Tasks
- What about when you need to wait for a value?
  - DispatchSemaphore!

Async-to-Sync Bridging

- Bridging is common during adoption and conversion
- sync-to-async: Continuations and AsyncStreams
- async-to-sync: unstructured Tasks
- What about when you need to wait for a value?
  - DispatchSemaphore! 😬

Async-to-Sync Bridging

- Bridging is common during adoption and conversion
- sync-to-async: Continuations and AsyncStreams
- async-to-sync: unstructured Tasks
- What about when you need to wait for a value?
  - DispatchSemaphore! 💀

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Don't Blame Me

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  nextTaylorsVersionAlbum { nextAlbumTitle in
    albumTitle = nextAlbumTitle
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum(_ completion: @escaping (String) -> Void) {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() -> String {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  DispatchQueue.global().asyncAfter(deadline: .now() + 1) {
    completion("Reputation")
  }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  completion("Reputation")
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    albumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    albumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    albumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    albumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  var albumTitle: String?

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    albumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

🛑 mutation of captured var 'albumTitle' in concurrently-executing code

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    albumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

🛑 mutation of captured var 'albumTitle' in concurrently-executing code

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

🛑 mutation of captured var 'albumTitle' in concurrently-executing code

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  print(albumTitle!)
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

don't do this

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

- The Concurrency runtime's cooperative thread pool is (generally) limited to the number of system CPU cores
  - A *much* smaller number than Grand Central Dispatch's thread pool
- Swift tasks must guarantee forward progress
  - Otherwise risk "thread starvation" and deadlocking our applications
- Swift Concurrency Waits for No One
  - by Saagar Jha

Async-to-Sync Bridging

- The Concurrency runtime's cooperative thread pool is (generally) limited to the number of system CPU cores
  - A *much* smaller number than Grand Central Dispatch's thread pool
- Swift tasks must guarantee forward progress
  - Otherwise risk "thread starvation" and deadlocking our applications
- Swift Concurrency Waits for No One
  - by Saagar Jha
- Task executor preference

Async-to-Sync Bridging

- The Concurrency runtime's cooperative thread pool is (generally) limited to the number of system CPU cores
  - A *much* smaller number than Grand Central Dispatch's thread pool
- Swift tasks must guarantee forward progress
  - Otherwise risk "thread starvation" and deadlocking our applications
- Swift Concurrency Waits for No One
  - by Saagar Jha
- Task executor preference 🙌

Async-to-Sync Bridging

don't do this

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

don't do this

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    await withTaskExecutorPreference(SomeBlockableExecutor.shared) {
      let nextAlbumTitle = await nextTaylorsVersionAlbum()
      albumTitle.withValue { $0 = nextAlbumTitle }
      semaphore.signal()
    }
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

not available yet

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task {
    await withTaskExecutorPreference(SomeBlockableExecutor.shared) {
      let nextAlbumTitle = await nextTaylorsVersionAlbum()
      albumTitle.withValue { $0 = nextAlbumTitle }
      semaphore.signal()
    }
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Async-to-Sync Bridging

not available yet

```
func nextTaylorsVersionAlbum() async -> String {
  try? await Task.sleep(for: .seconds(1))
  return "Reputation"
}

func waitForValue() {
  let albumTitle = Mutex<String?>(nil)

  let semaphore = DispatchSemaphore(value: 0)
  Task(executorPreference: SomeBlockableExecutor.shared) {
    let nextAlbumTitle = await nextTaylorsVersionAlbum()
    albumTitle.withValue { $0 = nextAlbumTitle }
    semaphore.signal()
  }
  semaphore.wait()

  albumTitle.withValue { print($0!) }
}
```

Are We There Yet?

- Isolated Synchronous Deinitialization
- reasync
- AsyncSequence Enhancements
- Better Actor Control
- Async-to-Sync Bridging

Are We There Yet?

- Isolated Synchronous Deinitialization
- reasync
- AsyncSequence Enhancements
- Better Actor Control
- Async-to-Sync Bridging
- And more...

Are We There Yet?

It Depends

😅 It Depends 😅

Are We There Yet?

- Are codebases going to be ready to enable Swift 6 Language Mode?
- Can codebases adopt Concurrency primitives over Grand Central Dispatch?
- Can codebases adopt AsyncSequence over Combine?

Don’t Panic

Q&A Session

Thursday
9:00 AM - 12:00 PM

Questions?

Speak Now 

(unless it's about the bad Taylor Swift jokes)

(like the one above)

(link to download PDF of presentation)

References

WWDC 2024

- Migrate your app to Swift 6
- What’s new in Swift
- A Swift Tour: Explore Swift’s features and design

WWDC 2023

- Discover streamlined location updates
- Beyond the basics of structured concurrency

WWDC 2022

- Efficiency awaits: Background tasks in SwiftUI
- Eliminate data races using Swift Concurrency
- Visualize and optimize Swift concurrency
- What's new in Swift

WWDC 2021

- Swift concurrency: Behind the scenes
- Bring Core Data concurrency to Swift and SwiftUI
- Discover concurrency in SwiftUI
- Swift concurrency: Update a sample app
- Use async/await with URLSession
- Explore structured concurrency in Swift
- What's new in Swift

Recommended Books

- Swift Concurrency by Example (Paul Hudson)
- Modern Concurrency on Apple Platforms (Andrés Ibañez Kautsch)
- The Curious Case of the Async Cafe (Daniel H Steinberg)
- Modern Concurrency in Swift (Marin Todorov, Kodeco)
- Practical Swift Concurrency (Donny Wals)

Official Documentation

- Swift.org - Concurrency
- Apple Developer - Concurrency
- The Swift Concurrency Migration Guide

Swift Evolution

Swift 5.5

- SE-0296: Async/await
- SE-0297: Concurrency Interoperability with Objective-C
- SE-0298: Async/Await: Sequences
- SE-0300: Continuations for interfacing async tasks with synchronous code
- SE-0304: Structured concurrency
- SE-0306: Actors
- SE-0310: Effectful Read-only Properties

Swift Evolution

Swift 5.5 (cont.)

- SE-0311: Task Local Values
- SE-0313: Improved control over actor isolation
- SE-0314: `AsyncStream` and `AsyncThrowingStream`
- SE-0316: Global actors
- SE-0317: `async let` bindings

Swift Evolution

Swift 5.5.2

- SE-0323: Asynchronous Main Semantics

Swift Evolution

Swift 5.6

- SE-0331: Remove Sendable conformance from unsafe pointer types
- SE-0337: Incremental migration to concurrency checking

Swift Evolution

Swift 5.7

- SE-0302: `Sendable` and `@Sendable` closures
- SE-0329: Clock, Instant, and Duration
- SE-0336: Distributed Actor Isolation
- SE-0338: Clarify the Execution of Non-Actor-Isolated Async Functions
- SE-0340: Unavailable From Async Attribute
- SE-0343: Concurrency in Top-level Code
- SE-0344: Distributed Actor Runtime

Swift Evolution

Swift 5.8

- (none)

Swift Evolution

Swift 5.9

- SE-0374: Add sleep(for:) to Clock
- SE-0381: DiscardingTaskGroups
- SE-0388: Convenience Async[Throwing]Stream.makeStream methods
- SE-0392: Custom Actor Executors
- SE-0395: Observation
- SE-0401: Remove Actor Isolation Inference caused by Property Wrappers

Swift Evolution

Swift 5.10

- SE-0327: On Actors and Initialization
- SE-0411: Isolated default value expressions
- SE-0412: Strict concurrency for global variables

Swift Evolution

Swift 6.0

- SE-0418: Inferring `Sendable` for methods and key path literals
- SE-0420: Inheritance of actor isolation
- SE-0421: Generalize effect polymorphism for `AsyncSequence` and `AsyncIteratorProtocol`

Swift Evolution

Upcoming (Accepted)

- SE-0413: Typed throws
- SE-0414: Region based Isolation
- SE-0417: Task Executor Preference
- SE-0424: Custom isolation checking for SerialExecutor
- SE-0428: Resolve DistributedActor protocols
- SE-0430: `sending` parameter and result values
- SE-0431: `@isolated(any)` Function Types

Swift Evolution

Upcoming (Accepted) (cont.)

- SE-0433: Synchronous Mutual Exclusion Lock 🔒

Swift Evolution

Upcoming (Active Review)

- SE-0423: Dynamic actor isolation enforcement from non-strict-concurrency contexts
- SE-0434: Usability of global-actor-isolated types

Swift Evolution

Returned

- SE-0371: Isolated synchronous deinit
- SE-0406: Backpressure support for AsyncStream

Other

- Swift Async Algorithms (Apple)
  - Broadcast algorithm (Philippe Hausler)
- Simple Made Easy (Rick Hickey)
- Modern Concurrency in Swift (Andy Ibanez)
- Swift Concurrency Waits for No One (Saagar Jha)
- Thread Safety in Swift (Bruno Rocha)
- AsyncExtensions (Thibault Wittemberg)

Other

(cont.)

- Swift 5.10 Released (Holly Borla, Apple)
- Introducing a Memory-Safe Successor Language in Large C++ Code Bases (John McCall)
- What's new in Swift 6.0? (Paul Hudson)
- massicotte.org Blog (Matt Massicotte)
- The Bleeding Edge of Swift Concurrency (Matthew Massicotte)
- Migrating to Swift 6 (Apple)
- Updating an app to use strict concurrency (Apple)