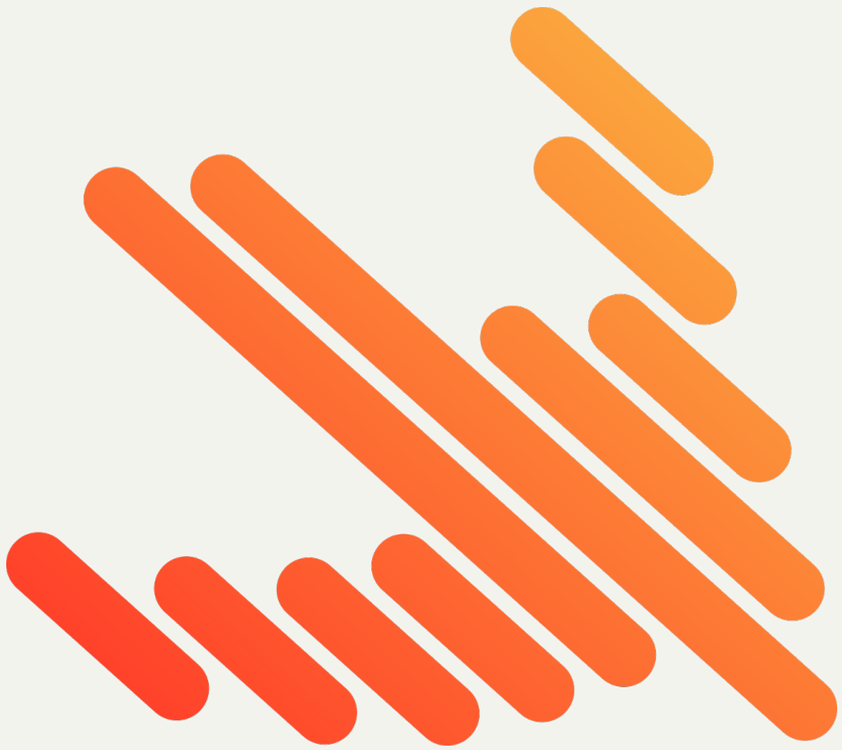
Straying From the Happy Path

Taking Control of Errors in Swift



Jeff Kelley (@SlaunchaMan)

SwiftFest, July 29th, 2019

Agenda

The History of Error-Handling: Objective-C and Swift

The Swift Error Type

Testing Errors

Error Handling Through Types: Combine and SwiftUI

Agenda

The History of Error-Handling: Objective-C and Swift

The Swift Error Type

Testing Errors

Error Handling Through Types: Combine and SwiftUI

Error Handling in Objective-C

Error Pointers

Objective-C used a pointer to an `NSError` object to vend errors back to the calling code. When the `removeItemAtPath:error:` method finishes, if an error occurred, `success` would be `NO` and `error` might have a pointer to an `NSError` in it.

```
NSError *error;

BOOL success =
[[NSFileManager defaultManager]
 removeItemAtPath:@"/etc/secret_identity"
 error:&error];
```

So, how do we handle this kind of error?

Error Handling in Objective-C

Error Pointers

You might think to do it this way:

```
NSError *error;

BOOL success =
[[NSFileManager defaultManager]
 removeItemAtPath:@"/etc/secret_identity"
 error:&error];

if (error != nil) {
  // Handle the error
}
```

Error Handling in Objective-C

Error Pointers

But in reality, you need to do it this way:

```
NSError *error;

BOOL success =
[[NSFileManager defaultManager]
 removeItemAtPath:@"/etc/secret_identity"
 error:&error];

if (!success) {
  if (error != nil) {
    // Handle the error
  }
}
```

The value in `error` is not sufficient for determining the result of the operation.

Error Handling in Objective-C

Completion Handlers

When writing asynchronous code, we often use completion handlers to convey the results of the operation.

```
HKHealthStore *store = [[HKHealthStore alloc] init];

[store startWatchAppWithWorkoutConfiguration:config
                                  completion:^(BOOL success,
                                               NSError * _Nullable error) {
  if (!success) {
    NSLog(@"Error starting workout: %@", [error localizedDescription]);
    // Alert the user that their workout didn't start
  }
}];
```

This pattern is so common, we have a type for it in Swift.

Errors in Swift

`Result`

`Result` is perfect for asynchronous code—it clearly defines the type you'll have in both the `success` and `failure` cases.

```
@frozen
enum Result<Success, Failure> where Failure : Error {
  case success(Success)
  case failure(Failure)
}
```

`Result` in Practice

Network Requests

A typical place you'll see a `Result` is in fetching something from the network:

```
let url = URL(string: "https://batcave.info/enemies/joker")!

performRequest(url) { result in
  switch result {
  case .success(let data):
    // Parse data into model object
  case .failure(let error):
    // Handle error
  }
}
```

`Result` in Practice

`map()`

We can use `map(_:)` to transform the `success` case of a `Result` into a new value.

```
func map<NewSuccess>(
    _ transform: (Success) -> NewSuccess
) -> Result<NewSuccess, Failure>
```

[1] If you want to transform the `failure` type, there's a corresponding `mapError(_:)` method.

`Result` in Practice

`map(_:)`

A common use of `map(_:)` is to transform data from the network into a model object:

```
let url = URL(string: "https://batcave.info/enemies/joker")!

performRequest(url) { result in
  // result is a Result<Data, Error>

  let mappedResult = result.map(Enemy.init)
  // mappedResult is a Result<Enemy, Error>

  switch result.map {  {
  case .success(let enemy):
    // Handle parsed enemy object
  case .failure(let error):
    // Handle error
  }
}
```

Error-Handling is Unenforceable.

You can lead a developer to error-handling APIs, but you can't make them use them.

`Result` in Actual Practice

How many times have you written code like this?

```
let url = URL(string: "https://batcave.info/enemies/joker")!

performRequest(url) { result in
  switch result {
  case .success(let data):
    // Parse data into model object
  case .failure(let error):
    // TODO: Handle Error
    break
  }
}
```

How many times do you forget about that TODO?

`Result` in Actual Practice

Or even have the `TODO`?

```
let url = URL(string: "https://batcave.info/enemies/joker")!

performRequest(url) { result in
  if case .success(let data) = result {
    // Handle data
  }
}
```

Objective-C Error Handling in Practice

```
NSError *error;

BOOL success =
[[NSFileManager defaultManager]
 removeItemAtPath:@"/etc/secret_identity"
 error:&error];

if (success) {
    // Proceed with what you were doing
}
else {
    // TODO: Handle error.
    NSLog(@"Error: %@", error.localizedDescription);
}
```

I mean, it could be worse. At least we're logging the error.

Objective-C Error Handling in Practice

```
NSError *error;

BOOL success =
[[NSFileManager defaultManager]
 removeItemAtPath:@"/etc/secret_identity"
 error:&error];
```

```
if (success) {
    // Proceed with what you were doing
}
```

Objective-C Error Handling in Practice

```
BOOL success =
[[NSFileManager defaultManager]
 removeItemAtPath:@"/etc/secret_identity"
 error:NULL];

if (success) {
    // Proceed with what you were doing
}
```

Agenda

The History of Error-Handling: Objective-C and Swift

The Swift Error Type

Testing Errors

Error Handling Through Types: Combine and SwiftUI

Swift Errors

Objective-C

Remember the `removeItemAtPath:error:` method we used in Objective-C?

```
- (BOOL)removeItemAtPath:(NSString *)path
                   error:(NSError * _Nullable *)error;
```

Swift Errors

Swift

Here's how that method appears in Swift. Now it's `removeItem(atPath:)`.

```
func removeItem(atPath path: String) throws
```

Notice that the `error` parameter is gone—where did it go?

Swift Errors

If we just call `removeItem(atPath:)`, this won't compile.

```
FileManager.default
  .removeItem(atPath: "/etc/secret_identity")
```

Swift Errors

If we just call `removeItem(atPath:)`, this won't compile.

```
FileManager.default
  .removeItem(atPath: "/etc/secret_identity")
// 🛑 Call can throw but is not marked with 'try'
```

Swift Errors

Any method marked `throws` needs us to `try` calling it first:

```
do {
  try FileManager.default
    .removeItem(atPath: "/etc/secret_identity")
}
catch {
  print(error.localizedDescription)
}
```

There's our `error` parameter.

The Swift `Error` Type

```
public protocol Error {
}
```

The Swift `Error` Type

Custom Error Types

If you know specific error cases you can run into, you can create an `enum` to represent these states.

```
enum EnemyParsingError: Error {
    case invalidData
    case unknown
    case actuallyAMarvelCharacter
}
```

The Swift `Error` Type

Catching Specific Errors

And once you have these cases, you can `catch` them:

```
do {
  try parseEnemy()
}
catch EnemyParsingError.invalidData {
  print("Bad data!")
}
catch {
  // Gotta catch ‘em all!
  print("Some other error: \(error.localizedDescription)")
}
```

Swift Errors

Just like `Result`, you can simply ignore the errors.

```
try? FileManager.default
    .removeItem(atPath: "/etc/secret_identity")

// or

try! FileManager.default
    .removeItem(atPath: "/etc/secret_identity")
```

`throws`

You can of course use `throws` in your own code:

```
func deleteSecretIdentity() throws {
  try FileManager.default
    .removeItem(atPath: "/etc/secret_identity")
}
```

throws

You can also use this with methods that return values:

```
func retrieveSecretIdentity() throws -> String? {
  let path = "/etc/secret_identity"

  let identity = try String(contentsOfFile: path)
  try FileManager.default.removeItem(atPath: path)

  return identity
}
```

Advanced `throws`

We use `map(_:)` all the time without `try`, thanks to the `rethrows` keyword:

```
func map<T>(
   _ transform: (Self.Element) throws -> T
) rethrows -> [T]
```

Advanced `throws`

You can write this yourself, just use `try` inside of your `rethrows` method:

```
extension Collection {

  public func map<T>(
    _ transform: (Element) throws -> T
  ) rethrows -> [T] {
    var result: [T] = []

    for item in self {
      let transformed = try transform(item)
      result.append(transformed)
    }

    return result
  }

}
```

The `Error` Type

The `Error` protocol has a `localizedDescription` property:

```
protocol Error {
  var localizedDescription: String { get }
}
```

You can use this property when you need to display the error:

```
BatLog("Error parsing enemy: \(error.localizedDescription)")
```

Effective Swift Errors

Logging

You may find yourself writing code like this a lot:

```
healthStore.add(samples, to: workout) { success, error in
  if let error = error {
    Log("Error adding samples to workout: " +
        error.localizedDescription)
  }
}
```

Effective Swift Errors

Logging

The pattern involves logging the error if there is one with some additional context.

```
if let error = error {
  Log("Error <some context here>: " +
    error.localizedDescription)
}
```

Effective Swift `Errors`

Logging

We can use an extension on `Error` to wrap this code:

```
extension Error {

    func log(context: String,
             filename: String = #file,
             lineNumber: Int = #line) {
        Log("Error \(context): \(localizedDescription)",
            type: .error,
            filename: filename,
            lineNumber: lineNumber)
    }

}
```

Effective Swift Errors

Logging

Here it is in use:

```
healthStore.add(samples, to: workout) { success, error in
  error?.log(context: "adding samples to workout")
}
```

Agenda

The History of Error-Handling: Objective-C and Swift

The Swift Error Type

Testing Errors

Error Handling Through Types: Combine and SwiftUI

Testing `Result`

The Happy Path

Since `Result` is `Equatable` when its `Success` and `Failure` types are `Equatable`, we can use `XCTAssertEqual(_:_:_:file:line:)`

```
class ResultTests: XCTestCase {

  func testTheHappyPath() {
    let url = URL(string: "https://batcave.info/enemies/joker")!

    let expectation = self.expectation("The request finishes")

    performRequest(url) { result in
      XCTAssertEqual(result, .success)
      expectation.fulfill()
    }

    waitForExpectations(timeout: 2)
  }

}
```

Testing `Result`

The Failure Case

We can do the same thing for the `failure` case:

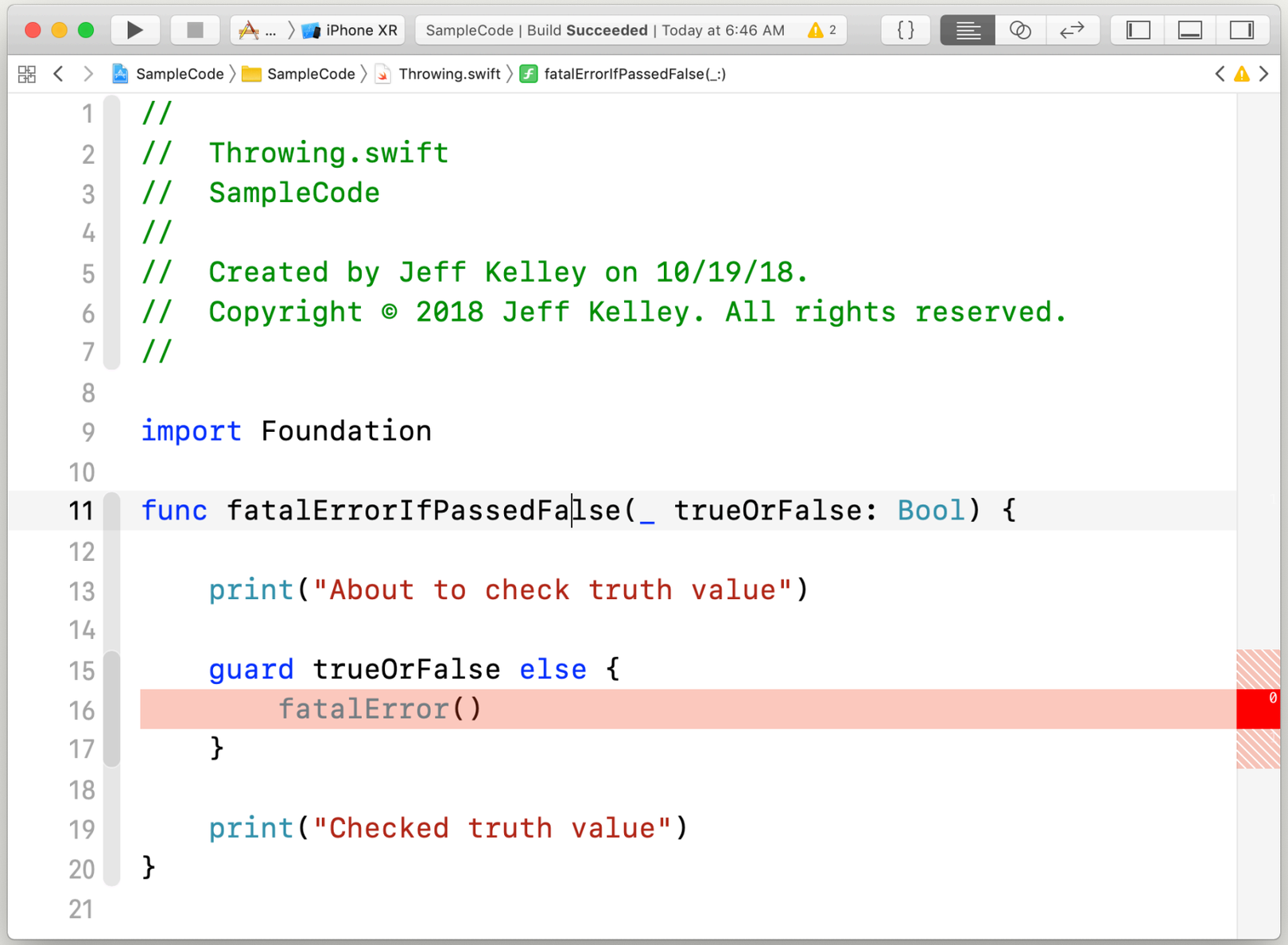
```
class ResultTests: XCTestCase {

  func testTheFailurePath() {
    let url = URL(string: "https://batcave.info/heroes/shazaam")!

    let expectation = self.expectation("The request finishes")

    performRequest(url) { result in
      XCTAssertEqual(result, .failure(NetworkError.notFound))
      expectation.fulfill()
    }

    waitForExpectations(timeout: 2)

}
```

`fatalError()`

If you can't handle an error, sometimes the right thing to do is crash:

```
do {
   try somethingThatCouldThrow()
}
catch {
   fatalError(error.localizedDescription)
}
```

Testing `fatalError()`



Testing `throws`

Consider the following code:

```
enum ThrowingError: Error {
  case passedFalse
}

func throwIfPassedFalse(
  _ trueOrFalse: Bool
) throws {
  guard trueOrFalse else {
    throw ThrowingError.passedFalse
  }
}
```

Testing `throws`

All we need to do is assert that the error is thrown.

```
func testThrowingError() {
  XCTAssertThrowsError(
    try throwIfPassedFalse(false),
    "Expected method to throw") { error in
      XCTAssertEqual(
        error as? ThrowingError,
        ThrowingError.passedFalse)
    }
}
```

Testing `fatalError()`

Replacing the call to `fatalError()`

If we put this method in our app code, it'll take precedence over the original `fatalError()`:

```
func fatalError(
  _ message: @autoclosure () -> String = "",
  file: StaticString = #file,
  line: UInt = #line
) -> Never {
  Swift.fatalError(message,
                   file: file,
                   line: line)
}
```

Testing `fatalError()`

Replacing `fatalError()` With a Closure

```
func fatalError(
  _ message: @autoclosure () -> String = "",
  file: StaticString = #file,
  line: UInt = #line
) -> Never {
  FatalErrorUtilities.fatalErrorClosure(message(), file, line)
}
```

```
struct FatalErrorUtilities {

  typealias FatalErrorClosure = (String, StaticString, UInt) -> Never

  fileprivate static var fatalErrorClosure = defaultFatalErrorClosure

  private static let defaultFatalErrorClosure = {
    (message: String, file: StaticString, line: UInt) -> Never in
    Swift.fatalError(message, file: file, line: line)
  }

}
```

Testing `fatalError()`

Swapping Out the Implementation

```
extension FatalErrorUtilities {

  internal static func replaceFatalError(
    closure: @escaping FatalErrorClosure
  ) {
    fatalErrorClosure = closure
  }

  internal static func restoreFatalError() {
    fatalErrorClosure = defaultFatalErrorClosure
  }

}
```

Testing `fatalError()`

Expect the Error

```
extension XCTestCase {

  func expectFatalError(file: StaticString = #file,
                        line: UInt = #line,
                        testcase: @escaping () -> Void) {
    let expectation = self.expectation(description: "expecting fatal error")

    FatalErrorUtilities.replaceFatalError { (_, _, _) -> Never in
      expectation.fulfill()
      self.unreachable()
    }

    DispatchQueue.global().async(execute: testcase)

    waitForExpectations(timeout: 2) { _ in
      FatalErrorUtilities.restoreFatalError()
    }
  }

  func unreachable() -> Never {
    while true { RunLoop.current.run() }
  }

}
```

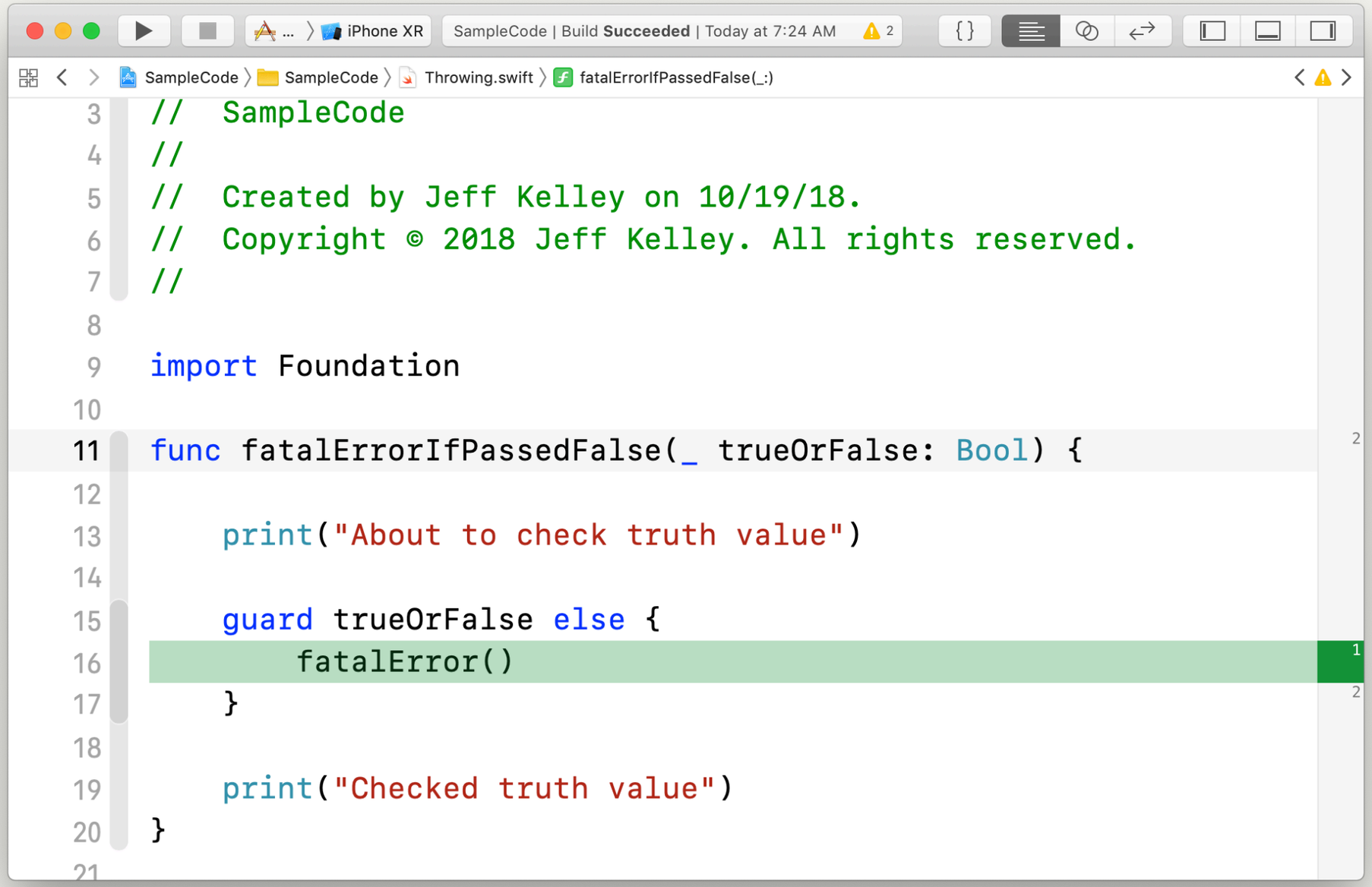
Testing `fatalError()`

Putting it All Together

Now we can write this test, and it'll pass without crashing!

✅
```
func testCrashing() {
    expectFatalError {
        fatalErrorIfPassedFalse(false)
    }
}
```

Testing `fatalError()`



When to Use `fatalError()`

One common use of `fatalError()` is to stop execution if you get into a situation you shouldn't be in:

```
func tableView(
  _ tableView: UITableView,
  cellForRowAt indexPath: IndexPath
) -> UITableViewCell {
  guard let cell = tableView.dequeueReusableCell(
    withIdentifier: "cell")
    as? MyTableViewCell
    else { fatalError() }

  return cell
}
```

Swift Assertion Methods

`Assert.swift` in the Swift Standard Library

Assert Type	Debug	Release	Unchecked
fatalError	✅	✅	✅
precondition	✅	✅	✕
assert	✅	✕	✕

Swift Assertion Methods

`assert()` and `precondition()`

These are functionally equivalent:

```
if (!someCondition) {
  assertionFailure()
}
```

```
assert(someCondition)
```

Effective Swift Errors

ProgrammerError

We can use `throws` and a custom `Error` type to wrap the code for dequeueing table view cells:

```
extension UITableView {

  enum ProgrammerError: Error {
    case noCellReturned
  }

  func dequeue<T: UITableViewCell>(
    _ type: T.Type,
    identifier reuseIdentifier: String
  ) throws -> T {
    guard let cell = dequeueReusableCell(
      withIdentifier: reuseIdentifier) as? T
      else { throw ProgrammerError.noCellReturned }

    return cell
  }

}
```

Effective Swift Errors

ProgrammerError

Now, we can use this in our code:

```
func tableView(
  _ tableView: UITableView,
  cellForRowAt indexPath: IndexPath
) -> UITableViewCell {
  do {
    return try tableView.dequeue(
      MyTableViewCell.self,
      identifier: "cell")
  }
  catch {
    fatalError(error.localizedDescription)
  }
}
```

Effective Swift Errors

Passing Errors to `fatalError()` and Friends

Let's write another replacement for `fatalError()` that takes an `Error` instead:

```
func fatalError(
  _ error: Error,
    file: StaticString = #file,
    line: UInt = #line
) -> Never {
    fatalError(error.localizedDescription,
               file: file,
               line: line)
}
```

Effective Swift Errors

Passing Errors to `fatalError()` and Friends

Now we can just pass the `error` through directly:

```
func tableView(
  _ tableView: UITableView,
  cellForRowAt indexPath: IndexPath
) -> UITableViewCell {
  do {
    return try tableView.dequeue(
      MyTableViewCell.self,
      identifier: "cell")
  }
  catch {
    fatalError(error)
  }
}
```

Effective Swift Errors

Testing Specific Error Conditions

We can use this in our tests, too:

```
func expectFatalError<T: Error>(expectedError: T,
                                file: StaticString = #file,
                                line: UInt = #line,
                                testcase: @escaping () -> Void) where T: Equatable {
  let expectation = self.expectation(description: "expecting fatal error")
  var assertionError: T? = nil

  FatalErrorUtilities.replaceFatalError { error, _, _ in
    assertionError = error as? T
    expectation.fulfill()
    self.unreachable()
  }

  DispatchQueue.global().async(execute: testcase)

  waitForExpectations(timeout: 2) { _ in
    XCTAssertEqual(assertionError,
                   expectedError,
                   file: file,
                   line: line)
    FatalErrorUtilities.restoreFatalError()
  }
}
```

Agenda

The History of Error-Handling: Objective-C and Swift

The Swift Error Type

Testing Errors

Error Handling Through Types: Combine and SwiftUI

Combine

Values That Change Over Time

```
public protocol Publisher {

  /// The kind of values published by this publisher.
  associatedtype Output

  /// The kind of errors this publisher might publish.
  ///
  /// Use `Never` if this `Publisher` does not publish errors.
  associatedtype Failure : Error

  /// This function is called to attach the specified `Subscriber` to this
  /// `Publisher` by `subscribe(_:)`
  ///
  /// - SeeAlso: `subscribe(_:)`
  /// - Parameters:
  ///     - subscriber: The subscriber to attach to this `Publisher`.
  ///                   once attached it can begin to receive values.
  func receive<S>(subscriber: S) where
    S : Subscriber,
    Self.Failure == S.Failure,
    Self.Output == S.Input
}
```

Combine

URLSession's DataTaskPublisher

```
extension URLSession {

  public struct DataTaskPublisher : Publisher {

    /// The kind of values published by this publisher.
    public typealias Output = (data: Data, response: URLResponse)

    /// The kind of errors this publisher might publish.
    ///
    /// Use `Never` if this `Publisher` does not publish errors.
    public typealias Failure = URLError

    /// This function is called to attach the specified `Subscriber` to this
    /// `Publisher` by `subscribe(_:)`
    ///
    /// - SeeAlso: `subscribe(_:)`
    /// - Parameters:
    ///     - subscriber: The subscriber to attach to this `Publisher`.
    ///                   once attached it can begin to receive values.
    public func receive<S>(subscriber: S) where
      S : Subscriber,
      S.Failure == URLSession.DataTaskPublisher.Failure,
      S.Input == URLSession.DataTaskPublisher.Output
  }

}
```

Combine Operators

```
let url = URL(string: "https://batcave.info/enemies/joker")!
let publisher = URLSession.shared.dataTaskPublisher(for: url)

let enemyPublisher: AnyPublisher<Enemy, Error> = publisher.tryMap {
  let decoder = JSONDecoder()

  let enemy = try decoder.decode(Enemy.self, from: $0.data)
  return enemy
}
.eraseToAnyPublisher()
```

Combine Publishers

The `Never` Type

```
NotificationCenter
   .default
   .publisher(for: UIResponder.keyboardWillShowNotification)
   .sink { notification in
     // Process notification
}
```

Combine Publishers

The `Never` Type

```
extension NotificationCenter {

  /// A publisher that emits elements when broadcasting notifications.
  public struct Publisher : Publisher {

    /// The kind of values published by this publisher.
    public typealias Output = Notification

    /// The kind of errors this publisher might publish.
    ///
    /// Use `Never` if this `Publisher` does not publish errors.
    public typealias Failure = Never
  }

}
```

SwiftUI and Errors

```
public protocol BindableObject : AnyObject, … {

  /// A type that publishes an event when the object has changed.
  associatedtype PublisherType : Publisher where Self.PublisherType.Failure == Never

  /// An instance that publishes an event immediately before the
  /// object changes.
  ///
  /// A `View`'s subhierarchy is forcibly invalidated whenever
  /// the `willChange` of its `model` publishes an event.
  var willChange: Self.PublisherType { get }
}
```

SwiftUI and Errors

Enforcing Error-Handling Through Types

```
struct EnemyView: View {

  @ObjectBinding var enemy: Enemy

  var lengthFormatter: LengthFormatter {
    let formatter = LengthFormatter()
    formatter.isForPersonHeightUse = true
    return formatter
  }

  var body: some View {
    VStack {
      Text(enemy.name)
      Text("\(enemy.height, formatter: lengthFormatter)")
    }
  }

}
```

SwiftUI and Errors

Placeholder Values

```
extension Enemy {
   static var placeholder: Enemy {
      return Enemy(name: "Unknown")
   }
}
```

```
let publisher = enemyPublisher
   .catch { _ in Just(.placeholder) }
```

Agenda

The History of Error-Handling: Objective-C and Swift

The Swift Error Type

Testing Errors

Error Handling Through Types: Combine and SwiftUI

Questions?

Contact info:

Jeff Kelley

@SlaunchaMan
jeff@detroitlabs.com