
Error Handling in
Swift 2.0

Agenda

- Kinds of failure.
- Approaches to error handling.
- Swift 2.0 model.
- Interoperability with Obj-C.
- Best practises.

Error handling is the process of responding to and recovering from error conditions in your program.

– The Swift Programming Language (Swift 2 Prerelease)

Kinds of failure

- Some functions and methods can't be guaranteed to always complete execution or provide useful output.
- For example the task of reading a file from disk can fail in multiple ways;
  - File does not exist.
  - File not having read permissions.
  - File not encoded in an compatible format.

Kinds of failure

1. A simple, obvious failure

- Obvious why failure occurred so don't need a detailed error.
- For example, parsing an integer from a string.
- Handled really well with Optional results (Swift).

Kinds of failure
2. Logical failures

- Failure cannot be recovered from.
- Should not attempt recovery - what state will your program be in?
- Recovering from these make program less stable, less secure etc
- For example, array index out of bounds, buffer overflow.
- Handled with assertions and NSException in Cocoa.

Kinds of failure

3. Detailed, recoverable failures

- When there is a rich set of reasons for failure. For example;
  - File not found.
  - Network failure
  - User cancellation.
- Can and should be recoverable.
- Handled with NSError (Cocoa) and Exceptions (Java, C#

Exceptions

Exceptions

- Methods that can fail defer handling of the failure to their caller.
- A method throws one or more exceptions for a caller to catch, or to throw to it's caller.
- The exception object details the nature of the failure.

```java
// Java
public void writeToFile(String path) throws IOException {
    if (validPath) {
        ...
    }
    else {
        throw new IOException("invalid path");
    }
}
```

```
// Java

    String name = textField.text();
    Data nameBytes = name.getBytes("UTF-8");
    nameBytes.writeToFile(path);
    proceedWithName(name);
```

```
// Java
try {
    String name = textField.text();
    Data nameBytes = name.getBytes("UTF-8");
    nameBytes.writeToFile(path);
    proceedWithName(name);
}
catch(IOException exception) {
    // handle the exception
}
```

Exceptions - the Good

- `try-catch` syntax is simple.
- Approach used across many languages: Java C#, Obj-C etc.
- Subclassing can be leveraged to customise exceptions.

Exceptions - the Bad

- Syntax makes it unclear which line actually throws the exception.
- Encourages as little code in the try block as possible.
- A try-catch block for each throwing method.

```
// Java
try {
    String name = textField.text(); // here?
    Data nameBytes = name.getBytes("UTF-8"); // how about here?
    nameBytes.writeToFile(path); // or maybe here?
    proceedWithName(name); // or possibly here?
}
```

Exceptions - the Bad

- Conflate concepts of recoverable and unrecoverable failures.
- For example, nothing to stop you catching and (incorrectly) recovering from `NullPointerException`.
- No compiler help to differentiate.
- Programmer must know which exceptions types indicate which kind of failure.

Exceptions - the Bad

- Calling a throwing method must always be enclosed in a try-catch.
- Even when *you know*
  - it cannot fail = empty `catch` block
  - it cannot be recovered from it = manual `abort()`
- No syntax to model these scenarios.
- Compiler cannot help (force) you do the right thing.

NSError

NSError

- Available in Cocoa and Cocoa Touch.
- Methods that can fail take a error pointer as the last parameter.
- If the method fails
  - `false` or `nil` is returned,
  - and the error pointer contains an `NSError` object detailing the nature of the failure.

```swift
// Swift
func writeToFile(_ path: String, inout error: NSError?) -> Bool {
    if (validPath) {
        ...
    }
    else {
        error = NSError(domain:FileWritingErrorDomain,
                        code:ErrorCode.InvalidPath,
                    userInfo:[NSLocalizedDescriptionKey : "\(path) is invalid"])
        return false
    }

    return true
}
```

```swift
// Swift
let name = textField.text
let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding)!
var writeError: NSError?
if (!nameBytes.writeToFile(path, error:&writeError)) {
    if let error = writeError {
        // handle the error
        return
    }
}
proceedWithName(name)
```

```swift
// Swift
let name = textField.text
let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding)!
nameBytes.writeToFile(path, error: nil) // don't handle
proceedWithName(name)
```

NSError
Customisation during initialisation

- `domain` string
  - Which sort of error, e.g `NSSQLiteErrorDomain`
- `code` integer
  - Which specific error, e.g. `NSPersistentStoreSaveError`
- `userInfo` dictionary
  - Container for custom information about the error.

NSError - the Good

- Flexible `userInfo` dictionary allows for custom key-values.
- Some special keys predefined for you:
  - `NSLocalizedDescriptionKey`
  - `NSLocalizedFailureReasonErrorKey`
  - `NSLocalizedRecoverySuggestionErrorKey`
  - `NSFilePathErrorKey`
  - etc

NSError - the Bad

- The implicit, default behaviour is to *ignore* errors.
- Easy to be lazy:
  - Pass a `nil` error pointer.
  - Improperly define `domain`, `code`, `userInfo`.
- Generally does not leverage type system - not subclassed.

NSError - the Bad

- A strict convention you must follow *on your own*.
  - Compiler cannot help you.
  - Only a convention - not an enforceable requirement.
- Repetitive error prone - easy to get wrong.
- Adds a lot of boilerplate code.
- Adds noise to the original code.

Handling multiple NSErrors

```
// Swift
var encodingError: NSError?
let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding, error:&encodingError)
if (!nameBytes) {
    if let error = encodingError {
    // handle the encoding error
    return
    }
}
var writeError: NSError?
if (!nameBytes.writeToFile(path, error:&encodingError)) {
    if let error = writeError {
    // handle the write error
    return
    }
}
proceedWithName(name)
```

Handling multiple NSErrors

```
// Swift
var encodingError: NSError?
let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding, error:&encodingError)
if (!nameBytes) {
    if let error = encodingError {
    // handle the encoding error
    return
    }
}
var writeError: NSError?
if (!nameBytes.writeToFile(path, error:&writeError)) {
    if let error = writeError {
    // handle the write error
    return
    }
}
proceedWithName(name)
```

Swift 2.0 Error Handling

Swift 2.0 Error Handling

The goal

- Provide an expressive way to handle errors.
- Be a safe, reliable programming model.
- Make error handling readable and maintainable.

Swift 2.0 Error Handling

- A method that can fail throws an error.
- Calling a throwing method is prefixed with `try` inside a `do` block.
- Errors are caught in a corresponding `catch` block.

```
// Swift 2.0

    let name = textField.text!
    let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding)!
    nameBytes.writeToFile(path)
    proceedWithName(name)
```

```
// Swift 2.0
do {
    let name = textField.text!
    let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding)!
    try nameBytes.writeToFile(path)
    proceedWithName(name)
}
catch {
    // handle the error
}
```

`ErrorType`

- `ErrorType` is a new protocol.
- Any object conforming to `ErrorType` can be thrown and caught.
- `NSError` already conforms to `ErrorType` - convenient!

`ErrorType`

- Conform an `enum` to `ErrorType`.
- Carry data for each case in an associated value.
  - Embed invalid state causing the failure.
- Compiler handles protocol conformance details automatically.
  - Easier for a type to conform to `ErrorType` than to subclass `NSError`.

```
// Swift 2.0
enum WriteError : ErrorType {
    case InvalidPath(path: String)
    case InsufficientPermissions
}
```

```
// Swift 2.0
func writeToFile(path: String) throws {
    if (validPath) {
    }
    else {
        throw WriteError.InvalidPath(path: path)
    }
    if (sufficientPermissions) {
        // write data
    }
    else {
        throw WriteError.InsufficientPermissions
    }
}
```

```
// Swift 2.0 improved
func writeToFile(path: String) throws {
    guard validPath else {
        throw WriteError.InvalidPath(path: path)
    }
    guard sufficientPermissions else {
        throw WriteError.InsufficientPermissions
    }
    // write data
}
```

Handling multiple errors

```
// Swift 2.0
do {
    let name = textField.text!
    try let nameBytes = name.dataUsingEncoding(NSUTF8StringEncoding)
    try nameBytes.writeToFile(path)
    proceedWithName(name)
}
catch EncodingError.InvalidEncoding(let encoding) {
    // handle the encoding error
}
catch WriteError.InvalidPath(let path) {
    // handle the path error
}
catch WriteError.InsufficientPermissions {
    // handle the permissions error
}
```

`try!`

- Can use `try!` to assert that an error is *not* thrown.
- Similar to force unwrapping an optional - will cause a runtime crash.
- Use when
  - you know an error *cannot* be thrown, or
  - it would be impossible to recover from.
- No need for an empty catch block.

Obj-C Interoperability

Obj-C Interoperability

```
// Swift 2.0
func writeToFile(path: String) throws
```

↑

Bidirectional mapping between Swift and Objective-C.

↓

```
// Obj-C
- (BOOL)writeToFilePath:(NSString *)path error:(NSError **)error;
```

Obj-C Interoperability

- Relies on the NSError convention.
  - Obj-C methods following the NSError convention are exposed to Swift as a throwing method.
  - Swift methods that throw are exposed to Obj-C as a methods that follow the NSError convention.
- Feels "right" when working in each language and still maintains compatibility.

Obj-C Interoperability
Method signature changes

- When calling in Swift:
  - `inout error: NSError?` removed
  - `throws` added
- When calling in Objective-C:
  - `error:(NSError **)error` added
  - `throws` removed

Swift to Objective-C

- Use `@objc` for `ErrorType` enums so they are printed into the generated header.

```swift
// Swift 2.0
@objc enum WriteError : Int, ErrorType {
    case InvalidPath = 1337
    case InsufficientPermissions = 1338
}

// Obj-C generated header
typedef NS_ENUM(NSInteger, WriteError) {
    InvalidPath = 1337,
    InsufficientPermissions = 1338
}
static NSString * const WriteErrorDomain = @"MyProject.WriteError"
```

Objective-C to Swift

- NSError automatically conforms to `ErrorType`
- Common error types can be used with Swift 2.0 `catch` syntax

```
// Swift 2.0
catch NSURLError.FileDoesNotExist {
}
```

Swift 2.0 Errors - the Good

- Concise, expressive and understandable syntax.
- Defining new error types is as easy as making another `enum` case.
- Compiler tells you to handle errors.
- Obvious which methods in a `do` block can throw.
- Pattern matching supported in the `catch` statement.
- Bidirectional compatibility with Obj-C.

Swift 2.0 Errors - the Bad

- Deceptively similar syntax to existing try-catch approach.
  - Could be confusing to go between approaches.
- Method signature doesn't explicitly say what sorts of error are thrown.
- Existing `do` block renamed to `repeat`.

Swift 2.0 Best Practises

What to use and when

1. Simple, obvious failures: return `Optional`
2. Unrecoverable, logical failures: use `assert()`
3. Recoverable failures: mark method as `throws`
    - Conform an enum to `ErrorType` and `throw` it
    - Define a case for each failure
    - Embed any failure state as an associated value

Any Questions?

References - The Web

- Java Exceptions Documentation
- NSHipster - NSError
- Mike Ash - The Best of What's New in Swift
- Big Nerd Ranch - Error Handling in Swift 2.0
- Nick Lockwood - Thoughts on Swift 2 Errors
- sketchyTech - Deferring and Delegating In Swift 2

References - WWDC

- WWDC 2015 Session 106: What's New In Swift
- WWDC 2015 Session 401: Swift and Objective-C Interoperability

References - Books

- The Swift Programming Language (Swift 2 Prerelease)
- Using Swift with Cocoa and Objective-C (Swift 2 Prerelease)

@lukestringer90
@SheffieldSwift