Error Handling in Swift 2.0

As a tentpole feature for Swift 2.0, we are introducing a new first-class error handling model. This feature provides standardized syntax and language affordances for throwing, propagating, catching, and manipulating recoverable error conditions.

Error handling is a well-trod path, with many different approaches in other languages, many of them problematic in various ways. We believe that our approach provides an elegant solution, drawing on the lessons we've learned from other languages and fixing or avoiding some of the pitfalls. The result is expressive and concise while still feeling explicit, safe, and familiar; and we believe it will work beautifully with the Cocoa APIs.

We're intentionally not using the term "exception handling", which carries a lot of connotations from its use in other languages. Our proposal has some similarities to the exceptions systems in those languages, but it also has a lot of important differences.

Kinds of Error

What exactly is an "error"? There are many possible error conditions, and they don't all make sense to handle in exactly the same way, because they arise in different circumstances and programmers have to react to them differently.

We can break errors down into four categories, in increasing order of severity:

A **simple domain error** arises from an operation that can fail in some obvious way and which is often invoked speculatively. Parsing an integer from a string is a really good example. The client doesn't need a detailed description of the error and will usually want to handle the error immediately. These errors are already well-modeled by returning an optional value; we don't need a more complex language solution for them.

A **recoverable error** arises from an operation which can fail in complex ways, but whose errors can be reasonably anticipated in advance. Examples including opening a file or reading from a network connection. These are the kinds of errors that Apple's APIs use NSError for today, but there are close analogues in many other APIs, such as `errno` in POSIX.

Ignoring this kind of error is usually a bad idea, and it can even be dangerous (e.g. by introducing a security hole). Developers should be strongly encouraged to write code that handles the error. It's common for developers to want to handle errors from different operations in the same basic way, either by reporting the error to the user or passing the error back to their own clients.

These errors will be the focus of this proposal.

The final two classes of error are outside the scope of this proposal. A **universal error** is theoretically recoverable, but by its nature the language can't help the programmer anticipate where it will come from. A **logic failure** arises from a programmer mistake and should not be recoverable at all. In our system, these kinds of errors are reported either with Objective-C/C++ exceptions or simply by logging a message and calling `abort()`. Both kinds of error are discussed extensively in the rationale. Having considered them carefully, we believe that we can address them in a later release without significant harm.

Aspects of the Design

This approach proposed here is very similar to the error handling model manually implemented in Objective-C with the `NSError` convention. Notably, the approach preserves these advantages of this convention:

- Whether a method produces an error (or not) is an explicit part of its API contract.
- Methods default to *not* producing errors unless they are explicitly marked.
- The control flow within a function is still mostly explicit: a maintainer can tell exactly which statements can produce an error, and a simple inspection reveals how the function reacts to the error.
- Throwing an error provides similar performance to allocating an error and returning it -- it isn't an expensive, table-based stack unwinding process.
- Cocoa APIs using standard `NSError` patterns can be imported into this world automatically. Other common patterns (e.g. `CFError`, `errno`) can be added to the model in future versions of Swift.

In addition, we feel that this design improves on Objective-C's error handling approach in a number of ways:

- It eliminates a lot of boilerplate control-flow code for propagating errors.
- The syntax for error handling will feel familiar to people used to exception handling in other languages.
- Defining custom error types is simple and ties in elegantly with Swift enums.

As to basic syntax, we decided to stick with the familiar language of exception handling. We considered intentionally using different terms (like `raise` / `handle`) to try to distinguish our approach from other languages. However, by and large, error propagation in this proposal works like it does in exception handling, and people are inevitably going to make the connection. Given that, we couldn't find a compelling reason to deviate from the `throw` / `catch` legacy.

This document just contains the basic proposal and will be very light on rationale. We considered many different languages and programming environments as part of making this proposal, and there's an extensive discussion of them in the separate rationale document. For example, that document explains why we don't simply allow all functions to throw, why we don't propagate errors using simply an `ErrorOr<T>` return type, and why we don't just make error propagation part of a general monad feature. We encourage you to read that rationale if you're interested in understanding why we made the decisions we did.

With that out of the way, let's get to the details of the proposal.

Typed propagation

Whether a function can throw is part of its type. This applies to all functions, whether they're global functions, methods, or closures.

By default, a function cannot throw. The compiler statically enforces this: anything the function does which can throw must appear in a context which handles all errors.

A function can be declared to throw by writing `throws` on the function declaration or type:

```
func foo() -> Int {  // This function is not permitted to throw.
func bar() throws -> Int {   // This function is permitted to throw.
```

`throws` is written before the arrow to give a sensible and consistent grammar for function types and implicit `()` result types, e.g.:

```
func baz() throws {

// Takes a 'callback' function that can throw.
// 'fred' itself can also throw.
func fred(_ callback: (UInt8) throws -> ()) throws {

// These are distinct types.
let a : () -> () -> ()
let b : () throws -> () -> ()
let c : () -> () throws -> ()
let d : () throws -> () throws -> ()
```

For curried functions, `throws` only applies to the innermost function. This function has type `(Int) -> (Int) throws -> Int`:

```
func jerry(_ i: Int)(j: Int) throws -> Int {
```

`throws` is tracked as part of the type system: a function value must also declare whether it can throw. Functions that cannot throw are a subtype of functions that can, so you can use a function that can't throw anywhere you could use a function that can:

```
func rachel() -> Int { return 12 }
func donna(_ generator: () throws -> Int) -> Int { ... }

donna(rachel)
```

The reverse is not true, since the caller would not be prepared to handle the error.

A call to a function which can throw within a context that is not allowed to throw is rejected by the compiler.

It isn't possible to overload functions solely based on whether the functions throw. That is, this is not legal:

```
func foo() {
func foo() throws {
```

A throwing method cannot override a non-throwing method or satisfy a non-throwing protocol requirement. However, a non-throwing method can override a throwing method or satisfy a throwing protocol requirement.

It is valuable to be able to overload higher-order functions based on whether an argument function throws, so this is allowed:

```
func foo(_ callback: () throws -> Bool) {
func foo(_ callback: () -> Bool) {
```

`rethrows`

Functions which take a throwing function argument (including as an autoclosure) can be marked as `rethrows`:

```
extension Array {
  func map<U>(_ fn: ElementType throws -> U) rethrows -> [U]
}
```

It is an error if a function declared `rethrows` does not include a throwing function in at least one of its parameter clauses.

`rethrows` is identical to `throws`, except that the function promises to only throw if one of its argument functions throws.

More formally, a function is *rethrowing-only* for a function *f* if:

- it is a throwing function parameter of *f*,
- it is a non-throwing function, or
- it is implemented within *f* (i.e. it is either *f* or a function or closure defined therein) and it does not throw except by either:
  - calling a function that is rethrowing-only for *f* or
  - calling a function that is `rethrows`, passing only functions that are rethrowing-only for *f*.

It is an error if a `rethrows` function is not rethrowing-only for itself.

A `rethrows` function is considered to be a throwing function. However, a direct call to a `rethrows` function is considered to not throw if it is fully applied and none of the function arguments can throw. For example:

```
// This call to map is considered not to throw because its
// argument function does not throw.
let absolutePaths = paths.map { "/" + $0 }

// This call to map is considered to throw because its
// argument function does throw.
let streams = try absolutePaths.map { try InputStream(filename: $0) }
```

For now, `rethrows` is a property of declared functions, not of function values. Binding a variable (even a constant) to a function loses the information that the function was `rethrows`, and calls to it will use the normal rules, meaning that they will be considered to throw regardless of whether a non-throwing function is passed.

For the purposes of override and conformance checking, `rethrows` lies between `throws` and non-`throws`. That is, an ordinary throwing method cannot override a `rethrows` method, which cannot override a non-throwing method; but an ordinary throwing method can be overridden by a `rethrows` method, which can be overridden by a non-throwing method. Equivalent rules apply for protocol conformance.

Throwing an error

The `throw` statement begins the propagation of an error. It always takes an argument, which can be any value that conforms to the `Error` protocol (described below).

```
if timeElapsed > timeThreshold {
  throw HomeworkError.Overworked
}

throw NSError(domain: "whatever", code: 42, userInfo: nil)
```

As mentioned above, attempting to throw an error out of a function not marked `throws` is a static compiler error.

Catching errors

A `catch` clause includes an optional pattern that matches the error. This pattern can use any of the standard pattern-matching tools provided by `switch` statements in Swift, including boolean `where` conditions. The pattern can be omitted; if so, a `where` condition is still permitted. If the pattern is omitted, or if it does not bind a different name to the error, the name `error` is automatically bound to the error as if with a `let` pattern.

The `try` keyword is used for other purposes which it seems to fit far better (see below), so `catch` clauses are instead attached to a generalized `do` statement:

```
// Simple do statement (without a trailing while condition),
// just provides a scope for variables defined inside of it.
do {
   let x = foo()
}

// do statement with two catch clauses.
do {
  ...

} catch HomeworkError.Overworked {
  // a conditionally-executed catch clause

} catch _ {
  // a catch-all clause.
}
```

As with `switch` statements, Swift makes an effort to understand whether catch clauses are exhaustive. If it can determine it is, then the compiler considers the error to be handled. If not, the error automatically propagates out of scope, either to a lexically enclosing `catch` clause or out of the containing function (which must be marked `throws`).

We expect to refine the `catch` syntax with usage experience.

`Error`

The Swift standard library will provide `Error`, a protocol with a very small interface (which is not described in this proposal). The standard pattern should be to define the conformance of an `enum` to the type:

```
enum HomeworkError : Error {
  case Overworked
  case Impossible
  case EatenByCat(Cat)
  case StopStressingMeWithYourRules
}
```

The `enum` provides a namespace of errors, a list of possible errors within that namespace, and optional values to attach to each

option.

Note that this corresponds very cleanly to the `NSError` model of an error domain, an error code, and optional user data. We expect to import system error domains as enums that follow this approach and implement `Error`. `NSError` and `CFError` themselves will also conform to `Error`.

The physical representation (still being nailed down) will make it efficient to embed an `NSError` as an `Error` and vice-versa. It should be possible to turn an arbitrary Swift `enum` that conforms to `Error` into an `NSError` by using the qualified type name as the domain key, the enumerator as the error code, and turning the payload into user data.

Automatic, marked, propagation of errors

Once an error is thrown, Swift will automatically propagate it out of scopes (that permit it), rather than relying on the programmer to manually check for errors and do their own control flow. This is just a lot less boilerplate for common error handling tasks. However, doing this naively would introduce a lot of implicit control flow, which makes it difficult to reason about the function's behavior. This is a serious maintenance problem and has traditionally been a considerable source of bugs in languages that heavily use exceptions.

Therefore, while Swift automatically propagates errors, it requires that statements and expressions that can implicitly throw be marked with the `try` keyword. For example:

```
func readStuff() throws {
  // loadFile can throw an error.  If so, it propagates out of readStuff.
  try loadFile("mystuff.txt")

  // This is a semantic error; the 'try' keyword is required
  // to indicate that it can throw.
  var y = stream.readFloat()

  // This is okay; the try covers the entire statement.
  try y += stream.readFloat()

  // This try applies to readBool().
  if try stream.readBool() {
    // This try applies to both of these calls.
    let x = try stream.readInt() + stream.readInt()
  }

  if let err = stream.getOutOfBandError() {
    // Of course, the programmer doesn't have to mark explicit throws.
    throw err
  }
}
```

Developers can choose to "scope" the `try` very tightly by writing it within parentheses or on a specific argument or list element:

```
// Ok.
let x = (try stream.readInt()) + (try stream.readInt())

// Semantic error: the try only covers the parenthesized expression.
let x2 = (try stream.readInt()) + stream.readInt()

// The try applies to the first array element.  Of course, the
// developer could cover the entire array by writing the try outside.
let array = [ try foo(), bar(), baz() ]
```

Some developers may wish to do this to make the specific throwing calls very clear. Other developers may be content with knowing that something within a statement can throw. The compiler's fixit hints will guide developers towards inserting a single `try` that covers the entire statement. This could potentially be controlled someday by a coding style flag passed to the compiler.

`try!`

To concisely indicate that a call is known to not actually throw at runtime, `try` can be decorated with `!`, turning the error check into a runtime assertion that the call does not throw.

For the purposes of checking that all errors are handled, a `try!` expression is considered to handle any error originating from within its operand.

`try!` is otherwise exactly like `try`: it can appear in exactly the same positions and doesn't affect the type of an expression.

Manual propagation and manipulation of errors

Taking control over the propagation of errors is important for some advanced use cases (e.g. transporting an error result across threads when synchronizing a future) and can be more convenient or natural for specific use cases (e.g. handling a specific call differently within a context that otherwise allows propagation).

As such, the Swift standard library should provide a standard Rust-like `Result<T>` enum, along with API for working with it, e.g.:

- A function to evaluate an error-producing closure and capture the result as a `Result<T>`.
- A function to unpack a `Result<T>` by either returning its value or propagating the error in the current context.

This is something that composes on top of the basic model, but that has not been designed yet and details aren't included in this proposal.

The name `Result<T>` is a stand-in and needs to be designed and reviewed, as well as the basic operations on the type.

`defer`

Swift should provide a `defer` statement that sets up an *ad hoc* clean-up action to be run when the current scope is exited. This replicates the functionality of a Java-style `finally`, but more cleanly and with less nesting.

This is an important tool for ensuring that explicitly-managed resources are released on all paths. Examples include closing a network connection and freeing memory that was manually allocated. It is convenient for all kinds of error-handling, even manual propagation and simple domain errors, but is especially nice with automatic propagation. It is also a crucial part of our long-term vision for universal errors.

`defer` may be followed by an arbitrary statement. The compiler should reject a `defer` action that might terminate early, whether by throwing or with `return`, `break`, or `continue`.

Example:

```
if exists(filename) {
  let file = open(filename, O_READ)
  defer close(file)

  while let line = try file.readline() {
    ...
  }

  // close occurs here, at the end of the formal scope.
}
```

If there are multiple defer statements in a scope, they are guaranteed to be executed in reverse order of appearance. That is:

```
let file1 = open("hello.txt")
defer close(file1)
let file2 = open("world.txt")
defer close(file2)
...
// file2 will be closed first.
```

A potential extension is to provide a convenient way to mark that a defer action should only be taken if an error is thrown. This is a convenient shorthand for controlling the action with a flag. We will evaluate whether adding complexity to handle this case is justified based on real-world usage experience.

Importing Cocoa

If possible, Swift's error-handling model should transparently work with the SDK with a minimal amount of effort from framework owners.

We believe that we can cover the vast majority of Objective-C APIs with `NSError**` out-parameters by importing them as `throws` and removing the error clause from their signature. That is, a method like this one from `NSAttributedString`:

```
- (NSData *)dataFromRange:(NSRange)range
       documentAttributes:(NSDictionary *)dict
                    error:(NSError **)error;
```

would be imported as:

```
func dataFromRange(_ range: NSRange,
                   documentAttributes dict: NSDictionary) throws -> NSData
```

There are a number of cases to consider, but we expect that most can be automatically imported without extra annotation in the SDK, by using a couple of simple heuristics:

- The most common pattern is a `BOOL` result, where a false value means an error occurred. This seems unambiguous.
- Also common is a pointer result, where a `nil` result usually means an error occurred. This appears to be universal in Objective-C; APIs that can return `nil` results seem to do so via out-parameters. So it seems to be safe to make a policy decision that it's okay to assume that a `nil` result is an error by default.

  If the pattern for a method is that a `nil` result means it produced an error, then the result can be imported as a non-optional type.
- A few APIs return `void`. As far as I can tell, for all of these, the caller is expected to check for a non-`nil` error.

For other sentinel cases, we can consider adding a new clang attribute to indicate to the compiler what the sentinel is:

- There are several APIs returning `NSInteger` or `NSUInteger`. At least some of these return 0 on error, but that doesn't seem like a reasonable general assumption.
- `AVFoundation` provides a couple methods returning `AVKeyValueStatus`. These produce an error if the API returned `AVKeyValueStatusFailed`, which, interestingly enough, is not the zero value.

The clang attribute would specify how to test the return value for an error. For example:

```
+ (NSInteger)writePropertyList:(id)plist
                      toStream:(NSOutputStream *)stream
                        format:(NSPropertyListFormat)format
                       options:(NSPropertyListWriteOptions)opt
                         error:(out NSError **)error
  NS_ERROR_RESULT(0);

- (AVKeyValueStatus)statusOfValueForKey:(NSString *)key
                                  error:(NSError **)
  NS_ERROR_RESULT(AVKeyValueStatusFailed);
```

We should also provide a Clang attribute which specifies that the correct way to test for an error is to check the out-parameter. Both of these attributes could potentially be used by the static analyzer, not just Swift. (For example, they could try to detect an invalid error check.)

Cases that do not match the automatically imported patterns and that lack an attribute would be left unmodified (i.e., they'd keep their NSErrorPointer argument) and considered "not awesome" in the SDK auditing tool. These will still be usable in Swift: callers will get the NSError back like they do today, and have to throw the result manually.

For initializers, importing an initializer as throwing takes precedence over importing it as failable. That is, an imported initializer with a nullable result and an error parameter would be imported as throwing. Throwing initializers have very similar constraints to failable initializers; in a way, it's just a new axis of failability.

One limitation of this approach is that we need to be able to reconstruct the selector to use when an overload of a method is introduced. For this reason, the import is likely to be limited to methods where the error parameter is the last one and the corresponding selector chunk is either `error:` or the first chunk (see below). Empirically, this seems to do the right thing for all but two sets of APIs in the public API:

- The `ISyncSessionDriverDelegate` category on `NSObject` declares half-a-dozen methods like this:

  ```
  - (BOOL)sessionDriver:(ISyncSessionDriver *)sender
        didRegisterClientAndReturnError:(NSError **)outError;
  ```

  Fortunately, these delegate methods were all deprecated in Lion, and are thus unavailable in Swift.

- `NSFileCoordinator` has half a dozen methods where the `error:` clause is second-to-last, followed by a block argument. These methods are not deprecated as far as I know.

The above translation rule would import methods like this one from `NSDocument`:

```
- (NSDocument *)duplicateAndReturnError:(NSError **)outError;
```

like so:

```
func duplicateAndReturnError() throws -> NSDocument
```

The `AndReturnError` bit is common but far from universal; consider this method from `NSManagedObject`:

```
- (BOOL)validateForDelete:(NSError **)error;
```

This would be imported as:

```
func validateForDelete() throws
```

This is a really nice import, and it's somewhat unfortunate that we can't import `duplicateAndReturnError:` as `duplicate()`.

Potential future extensions to this model

We believe that the proposal above is sufficient to provide a huge step forward in error handling in Swift programs, but there is always more to consider in the future. Some specific things we've discussed (and may come back to in the future) but don't consider to be core to the Swift 2.0 model are:

Higher-order polymorphism

We should make it easy to write higher-order functions that behave polymorphically with respect to whether their arguments throw. This can be done in a fairly simple way: a function can declare that it throws if any of a set of named arguments do. As an example (using strawman syntax):

```
func map<T, U>(_ array: [T], fn: T -> U) throwsIf(fn) -> [U] {
```

```
  ...
}
```

There's no need for a more complex logical operator than disjunction for normal higher-order stuff.

This feature is highly desired (e.g. it would allow many otherwise redundant overloads to be collapsed into a single definition), but it may or may not make it into Swift 2.0 based on schedule limitations.

Generic polymorphism

For similar reasons to higher-order polymorphism, we should consider making it easier to parameterize protocols on whether their operations can throw. This would allow the writing of generic algorithms, e.g. over `Sequence`, that handle both conformances that cannot throw (like `Array`) and those that can (like a hypothetical cloud-backed implementation).

However, this would be a very complex feature, yet to be designed, and it is far out-of-scope for Swift 2.0. In the meantime, most standard protocols will be written to not allow throwing conformances, so as to not burden the use of common generic algorithms with spurious error-handling code.

Statement-like functions

Some functions are designed to take trailing closures that feel like sub-statements. For example, `autoreleasepool` can be used this way:

```
autoreleasepool {
  foo()
}
```

The error-handling model doesn't cause major problems for this. The compiler can infer that the closure throws, and `autoreleasepool` can be overloaded on whether its argument closure throws; the overload that takes a throwing closures would itself throw.

There is one minor usability problem here, though. If the closure contains throwing expressions, those expressions must be explicitly marked within the closure with `try`. However, from the compiler's perspective, the call to `autoreleasepool` is also a call that can throw, and so it must also be marked with `try`:

```
try autoreleasepool {    // 'try' is required here...
  let string = try parseString() // ...and here.
  ...
}
```

This marking feels redundant. We want functions like `autoreleasepool` to feel like statements, but marks inside built-in statements like `if` don't require the outer statement to be marked. It would be better if the compiler didn't require the outer `try`.

On the other hand, the "statement-like" story already has a number of other holes: for example, `break`, `continue`, and `return` behave differently in the argument closure than in statements. In the future, we may consider fixing that; that fix will also need to address the error-propagation problem.

`using`

A `using` statement would acquire a resource, holds it for a fixed period of time, optionally binds it to a name, and then releases it whenever the controlled statement exits. `using` has many similarities to `defer`. It does not subsume `defer`, which is useful for many ad-hoc and tokenless clean-ups. But it could be convenient for the common pattern of a type-directed clean-up.

Automatically importing CoreFoundation and C functions

CF APIs use `CFErrorRef` pretty reliably, but there are several problems here: 1) the memory management rules for CFErrors are unclear and potentially inconsistent. 2) we need to know when an error is raised.

In principle, we could import POSIX functions into Swift as throwing functions, filling in the error from `errno`. It's nearly impossible to imagine doing this with an automatic import rule, however; much more likely, we'd need to wrap them all in an overlay.

In both cases, it is possible to pull these into the Swift error handling model, but because this is likely to require massive SDK annotations it is considered out of scope for iOS 9/OS X 10.11 & Swift 2.0.

Unexpected and universal errors

As discussed above, we believe that we can extend our current model to support untyped propagation for universal errors. Doing this well, and in particular doing it without completely sacrificing code size and performance, will take a significant amount of planning and insight. For this reason, it is considered well out of scope for Swift 2.0.