orphan:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)LibraryEvolution.rst, line 3)

Unknown interpreted text role "term".

```
.. default-role:: term
```

Note

This document uses some Sphinx-specific features which are not available on GitHub. For proper rendering, download and build the docs yourself.

Since Swift 5, ABI-stable platforms have supported library evolution, the ability to change a library without breaking source or binary compatibility. This model is intended to serve library designers whose libraries will evolve over time. Such libraries must be both *backwards-compatible*, meaning that existing clients should continue to work even when the library is updated, and *forwards-compatible*, meaning that future clients will be able run using the current version of the library. In simple terms:

- Last year's apps should work with this year's library.
- Next year's apps should work with this year's library.

This document is intended to be a specification for *which* changes can be made without breaking binary compatibility. When a library author wants to make a change, they can jump to the relevant section of this document to see if it's allowed. Anything *not* listed in this document should be assumed unsafe, i.e. changing it will break binary compatibility.

Library evolution was formally described in SE-0260, but this document should be kept up to date as new features are added to the language.

Background

One of Swift's primary design goals has always been to allow efficient execution of code without sacrificing load-time abstraction of implementation.

Abstraction of implementation means that code correctly written against a published interface will correctly function when the underlying implementation changes to anything which still satisfies the original interface. There are many potential reasons to provide

this sort of abstraction. Apple's primary interest is in making it easy and painless for our internal and external developers to improve the ecosystem of Apple products by creating good and secure programs and libraries; subtle deployment problems and/or unnecessary dependencies on the behavior of our implementations would work against these goals.

Our current design in Swift is to provide opt-out load-time abstraction of implementation for all language features. Alone, this would either incur unacceptable cost or force widespread opting-out of abstraction. We intend to mitigate this primarily by designing the language and its implementation to minimize unnecessary and unintended abstraction:

- Avoiding unnecessary language guarantees and taking advantage of that flexibility to limit load-time costs.
- Within the domain that defines an entity, all the details of its implementation are available.
- When entities are not exposed outside their defining module, their implementation is not constrained.
- By default, entities are not exposed outside their defining modules. This is independently desirable to reduce accidental API surface area, but happens to also interact well with the performance design.

This last point is a specific case of a general tenet of Swift: **the default behavior is safe**. Where possible, choices made when an entity is first published should not limit its evolution in the future.

Introduction

This document will frequently refer to a *library* which vends public APIs, and a single *client* that uses them. The same principles apply even when multiple libraries and multiple clients are involved. It also uses the term *ABI-public* introduced in SE-0193.

This document is primarily concerned with *binary compatibility*, i.e. what changes can safely be made to a library between releases that will not break memory-safety or type-safety, or cause clients to fail to run at all. A secondary concern is identifying *binary-compatible source-breaking changes <binary-compatible source-breaking change>*, where clients compiled against the previous version of a library are likely to behave differently than clients compiled against the new version of the library.

> **Note**
>
> These rules do not (and cannot) guarantee that a change is *semantically* backwards-compatible or forwards-compatible. *Any* change to a library's existing API that affects its observable behavior may affect clients. It is the responsibility of a library author to be sure that the changes they are making are *semantically* correct, preserving the preconditions, postconditions, and invariants of previously-published APIs.

This model is largely not of interest to libraries that are bundled with their clients (distribution via source, static library, or embedded/sandboxed dynamic library, as used by the Swift Package Manager). Because a client always uses a particular version of such a library, there is no need to worry about backwards- or forwards-compatibility at the binary level. Just as developers with a single app target are not forced to think about access control, anyone writing a bundled library should (ideally) not be required to use any of the annotations described below in order to achieve full performance.

> **Note**
>
> This model may, however, be useful for library authors that want to preserve *source* compatibility, though this document mostly doesn't discuss that. Additionally, some of these annotations are useful for performance today, such as `@inlinable`.

The term "resilience" comes from the occasional use of "fragile" to describe certain constructs that have very strict binary compatibility rules. For example, a client's use of a C struct is "fragile" in that if the library changes the fields in the struct, the client's use will "break". In Swift, changing the fields in a struct will not automatically cause problems for existing clients, so we say the struct is "resilient".

Supported Evolution

This section describes the various changes that are safe to make when releasing a new version of a library, i.e. changes that will not break binary compatibility. They are organized by declaration type.

Top-Level Functions

An ABI-public top-level function is fairly restricted in how it can be changed. The following changes are permitted:

- Changing the body of the function (as long as it is not `@inlinable`; see below).
- Changing *internal* parameter names (i.e. the names used within the function body, not the labels that are part of the function's full name).
- Reordering generic requirements (but not the generic parameters themselves).
- Adding a default argument expression to a parameter.
- Adding, changing, reordering, or removing property wrappers that either are implementation-detail or in a composition where the outermost wrapper is implementation-detail.
- Changing or removing a default argument is a *binary-compatible source-breaking change*.

- Adding or removing the `@discardableResult` and `@warn_unqualified_access` attributes.

No other changes are permitted; the following are particularly of note:

- An ABI-public function may not change its parameters or return type.
- An ABI-public function may not, in any way, change API-level property wrappers or compositions where the outermost wrapper is API-level.
- An ABI-public function may not change an API-level property-wrapper attribute to an implementation-detail one and vice versa, if it is the only wrapper applied to a given parameter or the outermost wrapper in a composition.
- An ABI-public function may not change its generic requirements.
- An ABI-public function may not change its external parameter names (labels).
- An ABI-public function may not add, remove, or reorder parameters, whether or not they have default arguments.
- An ABI-public function that throws may not become non-throwing or vice versa.
- The `@escaping` attribute may not be added to or removed from a parameter.
- Adding or removing a result builder from a parameter is a *binary-compatible source-breaking change*.

Inlinable Functions

Functions are a very common example of "abstraction of implementation": the function's declaration is published as API, but its body may change between library versions as long as it upholds the same semantic contracts. This applies to other function-like constructs as well: initializers, accessors, and deinitializers.

However, sometimes it is useful to provide the body to clients as well. There are a few common reasons for this:

- The function only performs simple operations, and so inlining it will both save the overhead of a cross-library function call and allow further optimization of callers.
- The function accesses a frozen struct with non-public members; this allows the library author to preserve invariants while still allowing efficient access to the struct.
- The function is generic and its performance may be greatly increased by specialization in the client.

An ABI-public function marked with the `@inlinable` attribute makes its body available to clients as part of the module's public interface. Clients are not required to inline a function marked `@inlinable`.

> **Note**
>
> It is legal to change the implementation of an inlinable function in the next release of the library. However, any such change must be made with the understanding that it will not affect existing clients. This is the standard example of a *binary-compatible source-breaking change*.

Any local functions or closures within an inlinable function are treated as `@_alwaysEmitIntoClient` (see below). A client that inlines the containing function must emit its own copy of the local functions or closures. This is important in case it is necessary to change the inlinable function later.

Removing the `@inlinable` attribute completely---say, to reference private implementation details that should not be ABI-public---is a safe change. However, existing clients will of course not be affected by this change, and any future use of the function must take this into account.

Although they are not a supported feature for arbitrary libraries at this time, public transparent functions are implicitly marked `@inlinable`.

Restrictions on Inlinable Functions

Because the body of an inlinable function (or method, accessor, initializer, or deinitializer) will be inlined into another module, it must not make any assumptions that rely on knowledge of the current module. Here is a trivial example using methods:

```
public struct Point2D {
  var x, y: Double
  public init(x: Double, y: Double) { /*...*/ }
}

extension Point2D {
  @inlinable public func distance(to other: Point2D) -> Double {
    let deltaX = self.x - other.x
    let deltaY = self.y - other.y
    return sqrt(deltaX*deltaX + deltaY*deltaY)
  }
}
```

As written, this `distance` method is not safe to inline. The next release of the library could very well replace the implementation of `Point2D` with a polar representation:

```
public struct Point2D {
  var r, theta: Double
  public init(x: Double, y: Double) { /*...*/ }
}
```

and the `x` and `y` properties have now disappeared. To avoid this, the bodies of inlinable functions have the following restrictions (enforced by the compiler):

- They may not define any local types.
- They must not reference any `private` or `fileprivate` entities.
- They must not reference any `internal` entities except for those that have been declared `@usableFromInline` or `@inlinable`.

Always Emit Into Client

A function, computed property or subscript annotated as `@_alwaysEmitIntoClient` is similar to an `@inlinable` declaration, except the declaration is not part of the module's ABI, meaning that the client must always emit their own copy. As a result:

- Removing a declaration annotated as `@_alwaysEmitIntoClient` is a *binary-compatible source-breaking change*.
- Adding `@_alwaysEmitIntoClient` to a declaration breaks ABI but is a source-compatible change.
- Removing `@_alwaysEmitIntoClient` from a declaration is a binary-compatible change. It also requires updating the availability to at least the OS version where the attribute was removed. As a result, it may be a source-breaking change.

Default Argument Expressions

Default argument expressions for functions that are ABI-public are implicitly `@_alwaysEmitIntoClient`. They are subject to the same restrictions as inlinable functions except that they also must not reference any non-`public` entities, even if they are `@usableFromInline` or `@inlinable`. This is to make sure a default argument expression can always be written explicitly by a caller.

Top-Level Variables and Constants

Given an ABI-public module-scope variable declared with `var`, the following changes are permitted:

- Adding (but not removing) a public setter to a computed variable.
- Adding or removing a non-ABI-public setter.
- Changing from a stored variable to a computed variable, or vice versa, as long as a previously ABI-public setter is not removed.
- As a special case of the above, adding or removing `lazy` from a stored property.
- Changing the body of an accessor, if the property is not marked `@inlinable` (see below).
- Adding or removing an observing accessor (`willSet` or `didSet`) to/from an existing variable. This is effectively the same as modifying the body of a setter.
- Changing the initial value of a stored variable.
- Adding or removing `weak` to/from a variable with `Optional` type.
- Adding or removing `unowned` to/from a variable.
- Adding or removing `@NSCopying` to/from a variable.
- If the variable is get-only, or if it has a non-ABI-public setter, it may be replaced by a `let` constant.
- Adding a property wrapper to a variable, or changing from one property wrapper to another, as long as an ABI-public setter or projected value (`$foo`) is not removed
- Removing a property wrapper from a variable, as long as the property wrapper didn't have a projected value (`$foo`).

For an ABI-public module-scope constant declared with `let`, the following changes are permitted:

- Changing the value of the constant.
- Replacing the constant with a variable.

Giving Up Flexibility

Top-level computed variables can be marked `@inlinable` just like functions. This restricts changes a fair amount:

- Adding an ABI-public setter to a computed variable is still permitted.
- Adding or removing a non-ABI-public setter is still permitted.
- Changing the body of an accessor is a *binary-compatible source-breaking change*.

Any inlinable accessors must follow the rules for inlinable functions, as described above.

Structs

Swift structs are a little more flexible than their C counterparts. By default, the following changes are permitted:

- Reordering any existing members, including stored properties (unless the struct is marked `@frozen`; see below).
- Adding any new members, including stored properties (see below for an exception).
- Changing existing properties from stored to computed or vice versa (unless the struct is marked `@frozen`; see below).
- As a special case of the above, adding or removing `lazy` from a stored property.
- Changing the body of any methods, initializers, or accessors.
- Adding or removing an observing accessor (`willSet` or `didSet`) to/from an existing property (unless the struct is marked `@frozen`; see below). This is effectively the same as modifying the body of a setter.

- Removing any non-ABI-public members, including stored properties.
- Adding a conformance to an ABI-public protocol *that was introduced in the same release* (see below).
- Adding or removing a conformance to a non-ABI-public protocol.
- Adding `@dynamicCallable` to the struct.

The important most aspect of a Swift struct is its value semantics, not its layout. Note that adding a stored property to a struct is *not* a breaking change even with Swift's synthesis of memberwise and no-argument initializers; these initializers are always `internal` and thus not exposed to clients outside the module.

Adding a new stored property with availability newer than the deployment target for the library is an error.

It is not safe to add or remove `mutating` or `nonmutating` from a member or accessor within a struct.

If a conformance is added to a type in version 1.1 of a library, it's important that it isn't accessed in version 1.0. This means that it is only safe to add new conformances to ABI-public protocols when the protocol is introduced, and not after. If the protocol comes from a separate module, there is no safe way to conform to it.

> **TODO**
>
> Coming up with a way to do this, either with availability annotations for protocol conformances or a way to emit a fallback copy of the conformance for clients on older library versions to use, is highly desired.

Methods and Initializers

For the most part struct methods and initializers are treated exactly like top-level functions. They permit all of the same modifications and can also be marked `@inlinable`, with the same restrictions.

Inlinable initializers must always delegate to another initializer or assign an entire value to `self`, since new properties may be added between new releases. For the same reason, initializers declared outside of the struct's module must always delegate to another initializer or assign to `self`. This is enforced by the compiler.

Properties

Struct properties behave largely the same as top-level variables and constants. They permit all of the same modifications, and also allow adding or removing an initial value entirely.

Subscripts

Subscripts behave largely the same as properties, except that there are no stored subscripts. This means that the following changes are permitted:

- Adding (but not removing) a public setter.
- Adding or removing a non-ABI-public setter.
- Changing the body of an accessor.
- Changing index parameter internal names (i.e. the names used within the accessor bodies, not the labels that are part of the subscript's full name).
- Reordering generic requirements (but not the generic parameters themselves).
- Adding a default argument expression to an index parameter.
- Adding, changing, reordering, or removing property wrappers that either are implementation-detail or in a composition where the outermost wrapper is implementation-detail.
- Changing or removing a default argument is a *binary-compatible source-breaking change*.

Like properties, subscripts can be marked `@inlinable`, which makes changing the body of an accessor a *binary-compatible source-breaking change*. Any inlinable accessors must follow the rules for inlinable functions, as described above.

Frozen Structs

To opt out of this flexibility, a struct may be marked `@frozen`. This promises that no stored properties will be added to or removed from the struct, even non-ABI-public ones, and allows the compiler to optimize as such. These stored properties also must not have any observing accessors. In effect:

- Reordering stored instance properties (public or non-public) is not permitted. Reordering all other members is still permitted.
- Adding new stored instance properties (public or non-public) is not permitted. Adding any other new members is still permitted.
- Changing existing instance properties from stored to computed or vice versa is not permitted.
- Similarly, adding or removing `lazy` from a stored property is not permitted.
- Changing the body of any *existing* methods, initializers, or computed property accessors is still permitted.
- Adding observing accessors to any stored instance properties (public or non-public) is not permitted (and is checked by the compiler).
- Removing stored instance properties is not permitted. Removing any other non-ABI-public members is still permitted.
- Adding a new protocol conformance is still permitted, per the usual restrictions.
- Removing conformances to non-ABI-public protocols is still permitted.

- Adding, changing, or removing property wrappers is not permitted.

Additionally, if the type of any stored instance property includes a struct or enum, that struct or enum must be ABI-public. This includes generic parameters, members of tuples, and property wrappers for stored instance properties.

> **Note**
>
> The above restrictions do not apply to `static` properties of `@frozen` structs. Static members effectively behave as top-level functions and variables.

While adding or removing stored properties is forbidden, existing properties may still be modified in limited ways:

- An existing non-ABI-public property may change its access level to any other non-public access level.
- `@usableFromInline` may be added to an `internal` property (with the current availability version, if necessary).
- A `@usableFromInline` property may be made `public`.

Adding or removing `@frozen` from an existing struct is forbidden.

An initializer of a frozen struct may be declared `@inlinable` even if it does not delegate to another initializer.

A `@frozen` struct is *not* guaranteed to use the same layout as a C struct with a similar "shape". If such a struct is necessary, it should be defined in a C header and imported into Swift.

> **Note**
>
> We may add a *different* feature to control layout some day, or something equivalent, but this feature should not restrict Swift from doing useful things like minimizing member padding. While the layout of `@frozen` structs is part of the stable ABI on Apple platforms now, it's not something that can't be revised in the future (with appropriate compatibility considerations). At the very least, Swift structs don't guarantee the same tail padding that C structs do.

Enums

By default, a library owner may add new cases to a public enum between releases without breaking binary compatibility. As with structs, this results in a fair amount of indirection when dealing with enum values, in order to potentially accommodate new values. More specifically, the following changes are permitted:

- Adding a new case (see below for exceptions around `@frozen` and availability).
- Reordering existing cases is a *binary-compatible source-breaking change* (unless the enum is marked `@frozen`; see below). In particular, both CaseIterable and RawRepresentable default implementations may affect client behavior.
- Adding a raw type to an enum that does not have one.
- Adding any other members.
- Removing any non-ABI-public members.
- Adding a new protocol conformance, with the same restrictions as for structs.
- Removing conformances to non-ABI-public protocols.
- Adding `@dynamicCallable` to the enum.

> **Note**
>
> If an enum value has a known case, or can be proven to belong to a set of known cases, the compiler is of course free to use a more efficient representation for the value, just as it may discard fields of structs that are provably never accessed.

Adding a new case with one or more associated values and with availability newer than the deployment target for the library is an error. This limitation is similar to the limitation for stored properties on structs.

Adding or removing the `@objc` attribute from an enum is not permitted; this affects the enum's memory representation and is not backwards-compatible.

Initializers

For the most part enum initializers are treated exactly like top-level functions. They permit all of the same modifications and can also be marked `@inlinable`, with the same restrictions.

Methods and Subscripts

The rules for enum methods and subscripts are identical to those for struct members.

Frozen Enums

A library owner may opt out of this flexibility by marking an ABI-public enum as `@frozen`. A "frozen" enum may not add new cases in the future, guaranteeing to clients that the current set of enum cases is exhaustive. In particular:

- Adding new cases is not permitted.
- Reordering existing cases is not permitted.
- Adding a raw type is still permitted.
- Adding any other members is still permitted.
- Removing any non-ABI-public members is still permitted.
- Adding a new protocol conformance is still permitted.
- Removing conformances to non-ABI-public protocols is still permitted.

> **Note**
>
> Were a public "frozen" enum allowed to have non-public cases, clients of the library would still have to treat the enum as opaque and would still have to be able to handle unknown cases in their `switch` statements.

Adding or removing `@frozen` from an existing enum is forbidden.

Even for default "non-frozen" enums, adding new cases should not be done lightly. Any clients attempting to do an exhaustive switch over all enum cases will likely not handle new cases well.

Protocols

There are very few safe changes to make to protocols and their members:

- A default may be added to an associated type.
- A new optional requirement may be added to an `@objc` protocol.
- All members may be reordered, including associated types.
- Changing *internal* parameter names of function and subscript requirements is permitted.
- Reordering generic requirements is permitted (but not the generic parameters themselves).
- The `@discardableResult` and `@warn_unqualified_access` attributes may be added to or removed from a function requirement.
- A new `associatedtype` requirement may be added (with the appropriate availability), as long as it has a default implementation. If the protocol did not have one or more `associatedtype` requirements before the change, then this is a *binary-compatible source-breaking change*.
- A new non-type requirement may be added (with the appropriate availability), as long as it has an unconstrained default implementation. If the requirement uses `Self` and the protocol has no other requirements using `Self` and no associated types, this is a *binary-compatible source-breaking change* due to restrictions on protocol value types.

All other changes to the protocol itself are forbidden, including:

- Adding or removing refined protocols.
- Removing any existing requirements (type or non-type).
- Removing the default type of an associated type.
- Making an existing requirement optional.
- Making a non-`@objc` protocol `@objc` or vice versa.
- Adding or removing protocols and superclasses from the inheritance clause of an associated type.
- Adding or removing constraints from the `where` clause of the protocol or an associated type.

Protocol extensions may be more freely modified; see below.

Classes

Because class instances are always accessed through references, they are very flexible and can change in many ways between releases. Like structs, classes support all of the following changes:

- Reordering any existing members, including stored properties.
- Changing existing properties from stored to computed or vice versa.
- As a special case of the above, adding or removing `lazy` from a stored property.
- Changing the body of any methods, initializers, accessors, or deinitializers.
- Adding or removing an observing accessor (`willSet` or `didSet`) to/from an existing property. This is effectively the same as modifying the body of a setter.
- Removing any non-ABI-public members, including stored properties.
- Adding a new protocol conformance (subject to the same restrictions as for structs).
- Removing conformances to non-ABI-public protocols.
- Adding `@dynamicCallable` to the class.

Omitted from this list is the free addition of new members. Here classes are a little more restrictive than structs; they only allow the following changes:

- Adding a new convenience initializer.
- Adding a new designated initializer, if the class is not `open` and any `open` subclasses that previously inherited convenience initializers continue to do so.
- Adding a deinitializer.

- Adding new, non-overriding method, subscript, or property.
- Adding a new overriding member, though if the class is `open` the type of the member may not deviate from the member it overrides. Changing the type could be incompatible with existing overrides in subclasses.

Finally, classes allow the following changes that do not apply to structs:

- Removing an explicit deinitializer. (A class with no declared deinitializer effectively has an implicit deinitializer.)
- "Moving" a method, subscript, or property up to its superclass. The declaration of the original member must remain along with its original availability, but its body may consist of simply calling the new superclass implementation.
- A non-final override of a method, subscript, property, or initializer may be removed as long as the generic parameters, formal parameters, and return type *exactly* match the overridden declaration. Any existing callers should automatically use the superclass implementation.
- `final` can be added to or removed from any non-ABI-public class, or any non-ABI-public member of a class.
- `@IBOutlet`, `@IBAction`, `@IBInspectable`, and `@GKInspectable` may be added to a member that is already exposed to Objective-C (either explicitly with `@objc` or implicitly through overriding or protocol requirements). Removing any of these is a *binary-compatible source-breaking change* if the member remains `@objc`, and disallowed if not.
- `@IBDesignable` may be added to a class; removing it is considered a *binary-compatible source-breaking change*.
- Changing a class's superclass `A` to another class `B`, *if* class `B` is a subclass of `A` *and* class `B`, along with any superclasses between it and class `A`, were introduced in the latest version of the library.

Other than those detailed above, no other changes to a class or its members are permitted. In particular:

- `open` cannot be added to or removed from an ABI-public class or member.
- `final` may not be added to or removed from an ABI-public class or its ABI-public members. (The presence of `final` enables optimization.)
- `dynamic` may not be added to *or* removed from any ABI-public members. Existing clients would not know to invoke the member dynamically.
- A `final` override of a member may *not* be removed, even if the type matches exactly; existing clients may be performing a direct call to the implementation instead of using dynamic dispatch.
- `@objc` and `@nonobjc` may not be added to or removed from the class or any existing members, except if the member already was or was not exposed to Objective-C.
- `@NSManaged` may not be added to or removed from any existing ABI-public members.

> **TODO**
>
> `@NSManaged` as it is in Swift 4.2 exposes implementation details to clients in a bad way. If we want to use `@NSManaged` in frameworks with binary compatibility concerns, we need to fix this. rdar://problem/20829214

Initializers

New designated initializers may not be added to an `open` class. This would change the inheritance of convenience initializers, which existing subclasses may depend on. An `open` class also may not change a convenience initializer into a designated initializer or vice versa.

A new `required` initializer may be added to a class only if it is a convenience initializer; that initializer may only call existing `required` initializers. An existing initializer may not be marked `required`.

All of the modifications permitted for top-level functions are also permitted for class initializers. Convenience initializers may be marked `@inlinable`, with the same restrictions as top-level functions; designated initializers may not.

Methods

Both class and instance methods allow all of the modifications permitted for top-level functions, but the potential for overrides complicates things a little. They allow the following changes:

- Changing the body of the method.
- Changing *internal* parameter names (i.e. the names used within the method body, not the labels that are part of the method's full name).
- Reordering generic requirements (but not the generic parameters themselves).
- Adding a default argument expression to a parameter.
- Changing or removing a default argument is a *binary-compatible source-breaking change*.
- Adding or removing the `@discardableResult` and `@warn_unqualified_access` attributes.

Class and instance methods may be marked `@inlinable`, with the same restrictions as struct methods. Additionally, only non-overriding `final` methods may be marked `@inlinable`.

Properties

Class and instance properties allow *most* of the modifications permitted for struct properties, but the potential for overrides complicates things a little. Variable properties (those declared with `var`) allow the following changes:

- Adding (but not removing) a computed setter to a non-`open` property.

- Adding or removing a non-ABI-public setter.
- Changing from a stored property to a computed property, or vice versa, as long as a previously ABI-public setter is not removed.
- Changing the body of an accessor.
- Adding or removing an observing accessor (`willSet` or `didSet`) to/from an existing variable. This is effectively the same as modifying the body of a setter.
- Adding, removing, or changing the initial value of a stored variable.
- Adding or removing `weak` from a variable with `Optional` type.
- Adding or removing `unowned` from a variable.
- Adding or removing `@NSCopying` from a variable.
- Adding a property wrapper to a non-`open` variable, or changing from one property wrapper to another, as long as an ABI-public setter or projected value (`$foo`) is not removed.
- Adding a property wrapper to an `open` variable, or changing from one property wrapper to another, as long as an ABI-public setter or projected value (`$foo`) is not added or removed.
- Removing a property wrapper from a variable, as long as the property wrapper didn't have a projected value (`$foo`).

Adding a public setter to an `open` property is a *binary-compatible source-breaking change*; any existing overrides will not know what to do with the setter and will likely not behave correctly.

Constant properties (those declared with `let`) still permit changing their value, as well as adding or removing an initial value entirely.

Non-overriding `final` computed properties (on both instances and classes) may be marked `@inlinable`. This behaves as described for struct properties.

Subscripts

Subscripts behave much like properties; they inherit the rules of their struct counterparts with a few small changes:

- Adding (but not removing) a public setter to a non-`open` subscript is permitted.
- Adding or removing a non-ABI-public setter is permitted.
- Changing the body of an accessor is permitted.
- Changing index parameter internal names is permitted.
- Reordering generic requirements (but not the generic parameters themselves) is permitted.
- Adding, changing, reordering, or removing property wrappers that either are implementation-detail or in a composition where the outermost wrapper is implementation-detail.
- Adding a default argument expression to an index parameter is permitted.
- Changing or removing a default argument is a *binary-compatible source-breaking change*.

Adding a public setter to an `open` subscript is a *binary-compatible source-breaking change*; any existing overrides will not know what to do with the setter and will likely not behave correctly.

Non-overriding `final` class subscripts may be marked `@inlinable`, which behaves as described for struct subscripts.

Extensions

Extensions largely follow the same rules as the types they extend. The following changes are permitted:

- Adding new extensions and removing empty extensions (that is, extensions that declare neither members nor protocol conformances).
- Moving a member from one extension to another within the same module, as long as both extensions have the exact same constraints.
- Adding any new member.
- Reordering members.
- Removing any non-ABI-public member.
- Changing the body of any methods, initializers, or accessors.

Additionally, non-protocol extensions allow a few additional changes:

- Moving a member from an unconstrained extension to the declaration of the base type, provided that the declaration is in the same module. The reverse is permitted for all members that would be valid to declare in an extension, although note that moving all initializers out of a type declaration may cause a new one to be implicitly synthesized.
- Adding a new protocol conformance (subject to the same restrictions discussed for structs).
- Removing conformances to non-ABI-public protocols.

> **Note**
>
> Although it is not related to evolution, it is worth noting that members of protocol extensions that do *not* satisfy protocol requirements are not overridable, even when the conforming type is a class.

> **Note**
>
> It is an ABI incompatible change to move a member to an extension with different constraints. Similarly, it is an ABI

incompatible change to move a member from a constrained extension back to its base type. Note that this is the case even if the constraints from the extension are restated as constraints in the where clause of e.g. a function or subscript member.

Operators and Precedence Groups

Operator and precedence group declarations are entirely compile-time constructs, so changing them does not have any effect on binary compatibility. However, they do affect *source* compatibility, so it is recommended that existing operators are not changed at all except for the following:

- Making a non-associative precedence group left- or right-associative.

Any other change counts as a *binary-compatible source-breaking change*.

Typealiases

Public typealiases within structs, enums, and protocols may be used for protocol conformances (to satisfy associated type requirements), not only within the library but within client modules as well. Therefore, changing a member typealias in any way is not permitted; while it will not break existing clients, they cannot recompile their code and get correct behavior.

Top-level typealiases only exist at compile-time, so changing the underlying type of one is a *binary-compatible source-breaking change*. However, if the typealias is *used* in the type of any ABI-public declaration in a library, it may be an actual breaking change and would not be permitted.

It is always permitted to change the *use* of a public typealias to its underlying type, and vice versa, at any location in the program.

Typealiases require availability annotations despite being compile-time constructs in order to verify the availability of their underlying types.

`@usableFromInline`

Adding `@usableFromInline` to an `internal` entity promises that the entity will be available at link time in the containing module's binary. This makes it safe to refer to such an entity from an inlinable function or in the stored properties of a frozen struct. `@usableFromInline` declarations shipped as part of an OS should have availability just like `public` declarations; if the entity is ever made `public` or `open`, its availability should not be changed.

> **Note**
>
> Why isn't `@usableFromInline` a special form of `public`? Because we don't want it to imply everything that `public` does, such as requiring overrides to be `public`.

Because a `@usableFromInline` class member may eventually be made `open`, the compiler must assume that new overrides may eventually appear from outside the module if the class is marked `open` unless the member is marked `final`.

For more information, see SE-0193.

Optimization

Allowing a library to evolve inhibits the optimization of client code in several ways. For example:

- A function that currently does not access global memory might do so in the future, so calls to it cannot be freely reordered in client code.
- A stored property may be replaced by a computed property in the future, so client code must not try to access the storage directly.
- A struct may have additional members in the future, so client code must not assume it fits in any fixed-sized allocation.

If the entity is declared within the same module as the code that's using it, then the code is permitted to know all the details of how the entity is declared. After all, if the entity is changed, the code that's using it will be recompiled. However, if the entity is declared in another module, then the code using it must be more conservative, and will therefore receive more conservative answers to its queries. (For example, a stored property may be treated as computed.)

As a special case, inlinable code must be treated as if it is outside the current module, since once it's inlined it will be.

Summary

When possible, Swift gives library authors freedom to evolve their code without breaking binary compatibility. This has implications for both the semantics and performance of client code, and so library owners also have tools to waive the ability to make certain future changes. When shipping libraries as part of the OS, the availability model guarantees that client code will never accidentally introduce implicit dependencies on specific versions of libraries.

Glossary

Glossary

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\(swift-main)(docs)LibraryEvolution.rst, line 958)

Unknown directive type "glossary".

```
.. glossary::

  ABI
    The run-time contract for using a particular API (or for an entire library),
    including things like symbol names, calling conventions, and type layout
    information. Stands for "Application Binary Interface".

  ABI-public
    Describes entities that are part of a library's `ABI`. Marked ``public``,
    ``open``, ``@usableFromInline``, or ``@inlinable`` in Swift. See
    `SE-0193 <SE0193_>`_ for more information.

  API
    An `entity` in a library that a `client` may use, or the collection of all
    such entities in a library. (If contrasting with `SPI`, only those entities
    that are available to arbitrary clients.) Marked ``public`` or ``open`` in
    Swift. Stands for "Application Programming Interface".

  backwards-compatible
    A modification to an API that does not break existing clients. May also
    describe the API in question.

  binary compatibility
    A general term encompassing both backwards- and forwards-compatibility
    concerns. Also known as "ABI compatibility".

  binary-compatible source-breaking change
    A change that does not break `binary compatibility`, but which is known to
    either change the behavior of existing clients or potentially result in
    errors when a client is recompiled. In most cases, a client that *hasn't*
    been recompiled may use the new behavior or the old behavior, or even a
    mix of both; however, this will always be deterministic (same behavior when
    a program is re-run) and will not break Swift's memory-safety and
    type-safety guarantees. It is recommended that these kinds of changes are
    avoided just like those that break binary compatibility.

  client
    A target that depends on a particular library. It's usually easiest to
    think of this as an application, but it could be another library.
    (In certain cases, the "library" is itself an application, such as when
    using Xcode's unit testing support.)

  duck typing
    In Objective-C, the ability to treat a class instance as having an
    unrelated type, as long as the instance handles all messages sent to it.
    (Note that this is a dynamic constraint.)

  entity
    A type, function, member, or global in a Swift program. Occasionally the
    term "entities" also includes conformances, since these have a run-time
    presence and are depended on by clients.

  forwards-compatible
    An API that is designed to handle future clients, perhaps allowing certain
    changes to be made without changing the ABI.

  module
    The primary unit of code sharing in Swift. Code in a module is always built
    together, though it may be spread across several source files.

  SPI
    A subset of `API` that is only available to certain clients. Stands for
    "System Programming Interface".

  target
    In this document, a collection of code in a single Swift module that is
    built together; a "compilation unit". Roughly equivalent to a target in
    Xcode.
```