Access Control

The general guiding principle of Swift access control:

> ***No entity can be defined in terms of another entity that has a lower access level.***

There are four levels of access: "private", "fileprivate", "internal", and "public". Private entities can only be accessed from within the lexical scope where they are defined. File-private entities can only be accessed from within the source file where they are defined. Internal entities can be accessed anywhere within the module they are defined. Public entities can be accessed from anywhere within the module and from any other context that imports the current module.

The names `public` and `private` have precedent in many languages; `internal` comes from C# and `fileprivate` from the Swift community. In the future, `public` may be used for both API and SPI, at which point we may design additional annotations to distinguish the two.

By default, most entities in a source file have `internal` access. This optimizes for the most common case – a single-target application project – while not accidentally revealing entities to clients of a framework module.

Warning

This document has not yet been updated for SE-0117, which adds the "open" level of access.

Rules

Access to a particular entity is considered relative to the current *access scope*. The access scope of an entity is its immediate lexical scope (if `private`), the current file (if `fileprivate`), the current module (if `internal`), or the current program (if `public`). A reference to an entity may only be written within the entity's access scope.

If a particular entity is not accessible, it does not appear in name lookup, unlike in C++. However, access control does not restrict access to members via runtime reflection (where applicable), nor does it necessarily restrict visibility of symbols in a linked binary.

Globals and Members

All globals and members have a default access level of `internal`, except within extensions (as described below).

A declaration may have any access level less than or equal to the access level of its type. That is, a `private` constant can have `public` type, but not the other way around. It is legal for a member to have greater access than its enclosing type, but this has no effect.

Accessors for variables have the same access level as their associated variable. The setter may be explicitly annotated with an access level less than or equal to the access level of the variable; this is written as `private(set)` or `internal(set)` before the `var` introducer.

An initializer, method, subscript, or property may have any access level less than or equal to the access level of its type (including the implicit 'Self' type), with a few additional rules:

- If a member is used to satisfy a protocol requirement, its access level must be at least as high as the protocol conformance's; see "Protocols" below.
- If an initializer is `required` by a superclass, its access level must be at least as high as the access level of the subclass itself.
- Accessors for subscripts follow the same rules as accessors for variables.
- A member may be overridden whenever it is accessible.

The implicit memberwise initializer for a struct has the minimum access level of all of the struct's stored properties, except that if all properties are `public` the initializer is `internal`. The implicit no-argument initializer for structs and classes follows the default access level for the type.

Currently, enum cases always have the same access level as the enclosing enum.

Deinitializers are only invoked by the runtime and do not nominally have access. Internally, the compiler represents them as having the same access level as the enclosing type.

Protocols

A protocol may have any access level less than or equal to the access levels of the protocols it refines. That is, a `private` ExtendedWidget protocol can refine a `public` Widget protocol, but not the other way around.

The access level of a requirement is the access level of the enclosing protocol, even when the protocol is `public`. Currently, requirements may not be given a lower access level than the enclosing protocol.

Swift does not currently support private protocol conformances, so for runtime consistency, the access level of the conformance of type T to protocol P is equal to the minimum of T's access level and P's access level; that is, the conformance is accessible whenever both T and P are accessible. This does not change if the protocol is conformed to in an extension. (The access level of a conformance is not currently reflected in the source, but is a useful concept for applying restrictions consistently.)

All members used to satisfy a conformance must have an access level at least as high as the conformance's. This ensures consistency between views of the type; if any member has a *lower* access level than the conformance, then the member could be accessed anyway through a generic function constrained by the protocol.

Note

This rule disallows an `internal` member of a protocol extension to satisfy a `public` requirement for a `public` type. Removing this limitation is not inherently unsafe, but (a) may be unexpected given the lack of explicit reference to the member, and (b) results in references to non-public symbols in the current representation.

A protocol may be used as a type whenever it is accessible. A nominal can conform to a protocol whenever the protocol is accessible.

Structs, Enums, and Classes

A struct, enum, or class may be used as a type whenever it is accessible. A struct, enum, or class may be extended whenever it is accessible.

A class may be subclassed whenever it is accessible. A class may have any access level less than or equal to the access level of its superclass.

Members within constrained extensions must have access less than or equal to the access level of the types used in the constraints.

An extension may be marked with an explicit access modifier (e.g. `private extension`), in which case the default access level of members within the extension is changed to match. No member within such an extension may have broader access than the new default.

Extensions with explicit access modifiers may not add new protocol conformances, since Swift does not support private protocol conformances (see "Protocols" above).

A type may conform to a protocol with lower access than the type itself.

Types

A nominal type's access level is the same as the access level of the nominal declaration itself. A generic type's access level is the minimum of the access level of the base type and the access levels of all generic argument types.

A tuple type's access level is the minimum of the access levels of its elements. A function type's access level is the minimum of the access levels of its input and return types.

A typealias may have any access level up to the access level of the type it aliases. That is, a `private` typealias can refer to a `public` type, but not the other way around. This includes associated types used to satisfy protocol conformances.

Runtime Guarantees

Non-`public` members of a class or extension will not be seen by subclasses or other extensions from outside the module. Therefore, members of a subclass or extension will not conflict with or inadvertently be considered to override non-accessible members of the superclass.

Access levels lower than `public` increase opportunities for devirtualization, though it is still possible to put a subclass of a `private` class within the same scope.

Most information about a non-`public` entity still has to be put into a module file for now, since we don't have resilience implemented. This can be improved later, and is no more revealing than the information currently available in the runtime for pure Objective-C classes.

Interaction with Objective-C

If an entity is exposed to Objective-C, most of the runtime guarantees and optimization opportunities go out the window. We have to use a particular selector for members, everything can be inspected at runtime, and even a private member can cause selector conflicts. In this case, access control is only useful for discipline purposes.

Members explicitly marked `private` or `fileprivate` are *not* exposed to Objective-C unless they are also marked `@objc` (or `@IBAction` or similar), even if declared within a class implicitly or explicitly marked `@objc`.

Any `public` entities will be included in the generated header. In an application or unit test target, `internal` entities will be exposed as well.

Non-Goals: "class-only" and "protected"

This proposal omits two forms of access control commonly found in other languages, a "class-implementation-only" access (often called "private"), and a "class and any subclasses" access (often called "protected"). We chose not to include these levels of access control because they do not add useful functionality beyond `private`, `fileprivate`, `internal`, and `public`.

"class-only"

If "class-only" includes extensions of the class, it is clear that it provides no protection at all, since a class may be extended from any context where it is accessible. So a hypothetical "class-only" must already be limited with regards to extensions. Beyond that, however, a "class-only" limit forces code to be declared within the class that might otherwise naturally be a top-level helper or an extension method on another type.

`private` and `fileprivate` serve the use case of limiting access to the implementation details of a class (even from the rest of the module!) while not tying access to the notion of type.

"protected"

"protected" access provides no guarantees of information hiding, since any subclass can now access the implementation details of its superclass---and expose them publicly, if it so chooses. This interacts poorly with our future plans for resilient APIs. Additionally, it increases the complexity of the access control model for both the compiler and for developers, and like "class-only" it is not immediately clear how it interacts with extensions.

Though it is not compiler-enforced, members that might be considered "protected" are effectively publicly accessible, and thus should be marked `public` in Swift. They can still be documented as intended for overriding rather than for subclassing, but the specific details of this are best dealt with on a case-by-case basis.

Potential Future Directions

- Allowing `private` or `internal` protocol conformances, which are only accessible at compile-time from a particular access scope.
- Limiting particular capabilities, such as marking something `final(public)` to restrict subclassing or overriding outside of the current module.
- Allowing the Swift parts of a mixed-source framework to access private headers.
- Revealing `internal` Swift API in a mixed-source framework in a second generated header.
- Levels of `public`, for example `public("SPI")`.
- Enum cases less accessible than the enum.
- Protocol requirements less accessible than the protocol.