Access Control

On This Page

Access control restricts access to parts of your code from code in other source files and modules. This feature enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

You can assign specific access levels to individual types (classes, structures, and enumerations), as well as to properties, methods, initializers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.

In addition to offering various levels of access control, Swift reduces the need to specify explicit access control levels by providing default access levels for typical scenarios. Indeed, if you are writing a single-target app, you may not need to specify explicit access control levels at all.

> NOTE
> The various aspects of your code that can have access control applied to them (properties, types, functions, and so on) are referred to as “entities” in the sections below, for brevity.

Modules and Source Files

Swift’s access control model is based on the concept of modules and source files.

A *module* is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift’s `import` keyword.

Each build target (such as an app bundle or framework) in Xcode is treated as a separate module in Swift. If you group together aspects of your app’s code as a stand-alone framework—perhaps to encapsulate and reuse that code across multiple applications—then everything you define within that framework will be part of a separate module when it’s imported and used within an app, or when it’s used within another framework.

A *source file* is a single Swift source code file within a module (in effect, a single file within an app or framework). Although it’s common to define individual types in separate source files, a single source file can contain definitions for multiple types, functions, and so on.

Access Levels

Swift provides five different *access levels* for entities within your code. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

- *Open access* and *public access* enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use open or public access when specifying the public interface to a framework. The difference between open and public access is described below.
- *Internal access* enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app’s or a framework’s internal structure.
- *File-private access* restricts the use of an entity to its own defining source file. Use file-private access to hide the implementation details of a specific piece of functionality when those details are used within an entire file.
- *Private access* restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file. Use private access to hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.

Open access is the highest (least restrictive) access level and private access is the lowest (most restrictive) access level.

Open access applies only to classes and class members, and it differs from public access as follows:

- Classes with public access, or any more restrictive access level, can be subclassed only within the module where they're defined.
- Class members with public access, or any more restrictive access level, can be overridden by subclasses only within the module where they're defined.
- Open classes can be subclassed within the module where they're defined, and within any module that imports the module where they're defined.
- Open class members can be overridden by subclasses within the module where they're defined, and within any module that imports the module where they're defined.

Marking a class as open explicitly indicates that you've considered the impact of code from other modules using that class as a superclass, and that you've designed your class's code accordingly.

Guiding Principle of Access Levels

Access levels in Swift follow an overall guiding principle: *No entity can be defined in terms of another entity that has a lower (more restrictive) access level.*

For example:

- A public variable can't be defined as having an internal, file-private, or private type, because the type might not be available everywhere that the public variable is used.
- A function can't have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are unavailable to the surrounding code.

The specific implications of this guiding principle for different aspects of the language are covered in detail below.

Default Access Levels

All entities in your code (with a few specific exceptions, as described later in this chapter) have a default access level of internal if you don't specify an explicit access level yourself. As a result, in many cases you don't need to specify an explicit access level in your code.

Access Levels for Single-Target Apps

When you write a simple single-target app, the code in your app is typically self-contained within the app and doesn't need to be made available outside of the app's module. The default access level of internal already matches this requirement. Therefore, you don't need to specify a custom access level. You may, however, want to mark some parts of your code as file private or private in order to hide their implementation details from other code within the app's module.

Access Levels for Frameworks

When you develop a framework, mark the public-facing interface to that framework as open or public so that it can be viewed and accessed by other modules, such as an app that imports the framework. This public-facing interface is the application programming interface (or API) for the framework.

> NOTE
>
> Any internal implementation details of your framework can still use the default access level of internal, or can be marked as private or file private if you want to hide them from other parts of the framework's internal code. You need to mark an entity as open or public only if you want it to become part of your framework's API.

Access Levels for Unit Test Targets

When you write an app with a unit test target, the code in your app needs to be made available to that module in order to be tested. By default, only entities marked as open or public are accessible to other modules. However, a unit test target can access any internal entity, if you mark the import declaration for a product module with the `@testable` attribute and compile that product module with testing enabled.

Access Control Syntax

Define the access level for an entity by placing one of the `open`, `public`, `internal`, `fileprivate`, or `private` modifiers before the entity's introducer:

```
public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
fileprivate func someFilePrivateFunction() {}
private func somePrivateFunction() {}
```

Unless otherwise specified, the default access level is internal, as described in Default Access Levels. This means that `SomeInternalClass` and `someInternalConstant` can be written without an explicit access-level modifier, and will still have an access level of internal:

```
class SomeInternalClass {}              // implicitly internal
let someInternalConstant = 0            // implicitly internal
```

Custom Types

If you want to specify an explicit access level for a custom type, do so at the point that you define the type. The new type can then be used wherever its access level permits. For example, if you define a file-private class, that class can only be used as the type of a property, or as a function parameter or return type, in the source file in which the file-private class is defined.

The access control level of a type also affects the default access level of that type's *members* (its properties, methods, initializers, and subscripts). If you define a type's access level as private or file private, the default access level of its members will also be private or file private. If you define a type's access level as internal or public (or use the default access level of internal without specifying an access level explicitly), the default access level of the type's members will be internal.

> IMPORTANT
> A public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such. This requirement ensures that the public-facing API for a type is something you opt in to publishing, and avoids presenting the internal workings of a type as public API by mistake.

```
public class SomePublicClass {                  // explicitly public class
    public var somePublicProperty = 0            // explicitly public class member
    var someInternalProperty = 0                 // implicitly internal class
  member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class
  member
    private func somePrivateMethod() {}          // explicitly private class member
}
```

```swift

class SomeInternalClass {                       // implicitly internal class
    var someInternalProperty = 0                 // implicitly internal class member
    fileprivate func someFilePrivateMethod() {}  // explicitly file-private class member
    private func somePrivateMethod() {}          // explicitly private class member
}

fileprivate class SomeFilePrivateClass {        // explicitly file-private class
    func someFilePrivateMethod() {}              // implicitly file-private class member
    private func somePrivateMethod() {}          // explicitly private class member
}

private class SomePrivateClass {                // explicitly private class
    func somePrivateMethod() {}                  // implicitly private class member
}
```

Tuple Types

The access level for a tuple type is the most restrictive access level of all types used in that tuple. For example, if you compose a tuple from two different types, one with internal access and one with private access, the access level for that compound tuple type will be private.

> NOTE
>
> Tuple types don't have a standalone definition in the way that classes, structures, enumerations, and functions do. A tuple type's access level is deduced automatically when the tuple type is used, and can't be specified explicitly.

Function Types

The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type. You must specify the access level explicitly as part of the function's definition if the function's calculated access level doesn't match the contextual default.

The example below defines a global function called `someFunction()`, without providing a specific access-level modifier for the function itself. You might expect this function to have the default access level of "internal", but this isn't the case. In fact, `someFunction()` won't compile as written below:

```swift
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

The function's return type is a tuple type composed from two of the custom classes defined above in Custom Types. One of these classes is defined as internal, and the other is defined as private. Therefore, the overall access level of the compound tuple type is private (the minimum access level of the tuple's constituent types).

Because the function's return type is private, you must mark the function's overall access level with the `private` modifier for the function declaration to be valid:

```swift
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

It's not valid to mark the definition of `someFunction()` with the `public` or `internal` modifiers, or to use the default setting of internal, because public or internal users of the function might not have appropriate access to the private class used in the function's return type.

Enumeration Types

The individual cases of an enumeration automatically receive the same access level as the enumeration they belong to. You can't specify a different access level for individual enumeration cases.

In the example below, the `CompassPoint` enumeration has an explicit access level of public. The enumeration cases `north`, `south`, `east`, and `west` therefore also have an access level of public:

```
public enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

Raw Values and Associated Values

The types used for any raw values or associated values in an enumeration definition must have an access level at least as high as the enumeration's access level. You can't use a private type as the raw-value type of an enumeration with an internal access level, for example.

Nested Types

Nested types defined within a private type have an automatic access level of private. Nested types defined within a file-private type have an automatic access level of file private. Nested types defined within a public type or an internal type have an automatic access level of internal. If you want a nested type within a public type to be publicly available, you must explicitly declare the nested type as public.

Subclassing

You can subclass any class that can be accessed in the current access context. A subclass can't have a higher access level than its superclass—for example, you can't write a public subclass of an internal superclass.

In addition, you can override any class member (method, property, initializer, or subscript) that is visible in a certain access context.

An override can make an inherited class member more accessible than its superclass version. In the example below, class `A` is a public class with a file-private method called `someMethod()`. Class `B` is a subclass of `A`, with a reduced access level of "internal". Nonetheless, class `B` provides an override of `someMethod()` with an access level of "internal", which is *higher* than the original implementation of `someMethod()`:

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {}
}
```

It's even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass's member takes place within an allowed access level context (that is, within the same source file as the superclass for a file-private member call, or within the same module as the superclass for an internal member call):

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {
        super.someMethod()
    }
}
```

Because superclass `A` and subclass `B` are defined in the same source file, it's valid for the `B` implementation of `someMethod()` to call `super.someMethod()`.

Constants, Variables, Properties, and Subscripts

A constant, variable, or property can't be more public than its type. It's not valid to write a public property with a private type, for example. Similarly, a subscript can't be more public than either its index type or return type.

If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as `private`:

```
private var privateInstance = SomePrivateClass()
```

Getters and Setters

Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.

You can give a setter a *lower* access level than its corresponding getter, to restrict the read-write scope of that variable, property, or subscript. You assign a lower access level by writing `fileprivate(set)`, `private(set)`, or `internal(set)` before the `var` or `subscript` introducer.

> NOTE
>
> This rule applies to stored properties as well as computed properties. Even though you don't write an explicit getter and setter for a stored property, Swift still synthesizes an implicit getter and setter for you to provide access to the stored property's backing storage. Use `fileprivate(set)`, `private(set)`, and `internal(set)` to change the access level of this synthesized setter in exactly the same way as for an explicit setter in a computed property.

The example below defines a structure called `TrackedString`, which keeps track of the number of times a string property is modified:

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
}
```

The `TrackedString` structure defines a stored string property called `value`, with an initial value of `""` (an empty string). The structure also defines a stored integer property called `numberOfEdits`, which is used to track the number of times that `value` is modified. This modification tracking is implemented with a `didSet` property observer on the `value` property, which increments `numberOfEdits` every time the `value` property is set to a new value.

The `TrackedString` structure and the `value` property don’t provide an explicit access-level modifier, and so they both receive the default access level of internal. However, the access level for the `numberOfEdits` property is marked with a `private(set)` modifier to indicate that the property’s getter still has the default access level of internal, but the property is settable only from within code that’s part of the `TrackedString` structure. This enables `TrackedString` to modify the `numberOfEdits` property internally, but to present the property as a read-only property when it’s used outside the structure’s definition.

If you create a `TrackedString` instance and modify its string value a few times, you can see the `numberOfEdits` property value update to match the number of modifications:

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment numberOfEdits."
stringToEdit.value += " So will this one."
print("The number of edits is \(stringToEdit.numberOfEdits)")
// Prints "The number of edits is 3"
```

Although you can query the current value of the `numberOfEdits` property from within another source file, you can’t *modify* the property from another source file. This restriction protects the implementation details of the `TrackedString` edit-tracking functionality, while still providing convenient access to an aspect of that functionality.

Note that you can assign an explicit access level for both a getter and a setter if required. The example below shows a version of the `TrackedString` structure in which the structure is defined with an explicit access level of public. The structure’s members (including the `numberOfEdits` property) therefore have an internal access level by default. You can make the structure’s `numberOfEdits` property getter public, and its property setter private, by combining the `public` and `private(set)` access-level modifiers:

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

Initializers

Custom initializers can be assigned an access level less than or equal to the type that they initialize. The only exception is for required initializers (as defined in Required Initializers). A required initializer must have the same access level as the class it belongs to.

As with function and method parameters, the types of an initializer’s parameters can’t be more private than the initializer’s own access level.

Default Initializers

As described in Default Initializers, Swift automatically provides a *default initializer* without any arguments for any structure or base class that provides default values for all of its properties and doesn’t provide at least one initializer itself.

A default initializer has the same access level as the type it initializes, unless that type is defined as `public`. For a type that is defined as `public`, the default initializer is considered internal. If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type’s definition.

Default Memberwise Initializers for Structure Types

The default memberwise initializer for a structure type is considered private if any of the structure’s stored properties are private. Likewise, if any of the structure’s stored properties are file private, the initializer is file private. Otherwise, the initializer has an access level of internal.

As with the default initializer above, if you want a public structure type to be initializable with a memberwise initializer when used in another module, you must provide a public memberwise initializer yourself as part of the type’s definition.

Protocols

If you want to assign an explicit access level to a protocol type, do so at the point that you define the protocol. This enables you to create protocols that can only be adopted within a certain access context.

The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol. You can’t set a protocol requirement to a different access level than the protocol it supports. This ensures that all of the protocol’s requirements will be visible on any type that adopts the protocol.

> NOTE
> If you define a public protocol, the protocol’s requirements require a public access level for those requirements when they’re implemented. This behavior is different from other types, where a public type definition implies an access level of internal for the type’s members.

Protocol Inheritance

If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. You can’t write a public protocol that inherits from an internal protocol, for example.

Protocol Conformance

A type can conform to a protocol with a lower access level than the type itself. For example, you can define a public type that can be used in other modules, but whose conformance to an internal protocol can only be used within the internal protocol’s defining module.

The context in which a type conforms to a particular protocol is the minimum of the type’s access level and the protocol’s access level. If a type is public, but a protocol it conforms to is internal, the type’s conformance to that protocol is also internal.

When you write or extend a type to conform to a protocol, you must ensure that the type’s implementation of each protocol requirement has at least the same access level as the type’s conformance to that protocol. For example, if a public type conforms to an internal protocol, the type’s implementation of each protocol requirement must be at least “internal”.

> NOTE
> In Swift, as in Objective-C, protocol conformance is global—it isn’t possible for a type to conform to a protocol in two different ways within the same program.

Extensions

You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available. Any type members added in an extension have the same default access level as type members declared in the original type being extended. If you extend a public or internal type, any new type members you add have a default access level of internal. If you extend a file-private type, any new type members you add have a default access level of file private. If you extend a private type, any new type members you add have a default access level of private.

Alternatively, you can mark an extension with an explicit access-level modifier (for example, `private extension`) to set a new default access level for all members defined within the extension. This new default can still be overridden within the extension for individual type members.

You can't provide an explicit access-level modifier for an extension if you're using that extension to add protocol conformance. Instead, the protocol's own access level is used to provide the default access level for each protocol requirement implementation within the extension.

Private Members in Extensions

Extensions that are in the same file as the class, structure, or enumeration that they extend behave as if the code in the extension had been written as part of the original type's declaration. As a result, you can:

- Declare a private member in the original declaration, and access that member from extensions in the same file.
- Declare a private member in one extension, and access that member from another extension in the same file.
- Declare a private member in an extension, and access that member from the original declaration in the same file.

This behavior means you can use extensions in the same way to organize your code, whether or not your types have private entities. For example, given the following simple protocol:

```swift
protocol SomeProtocol {
    func doSomething()
}
```

You can use an extension to add protocol conformance, like this:

```swift
struct SomeStruct {
    private var privateVariable = 12
}

extension SomeStruct: SomeProtocol {
    func doSomething() {
        print(privateVariable)
    }
}
```

Generics

The access level for a generic type or generic function is the minimum of the access level of the generic type or function itself and the access level of any type constraints on its type parameters.

Type Aliases

Any type aliases you define are treated as distinct types for the purposes of access control. A type alias can have an access level less than or equal to the access level of the type it aliases. For example, a private type alias can alias a private, file-private, internal, public, or open type, but a public type alias can't alias an internal, file-private, or private type.

> NOTE
>
> This rule also applies to type aliases for associated types used to satisfy protocol conformances.