ON THIS PAGE

Attributes

There are two kinds of attributes in Swift—those that apply to declarations and those that apply to types. An attribute provides additional information about the declaration or type. For example, the `discardableResult` attribute on a function declaration indicates that, although the function returns a value, the compiler shouldn't generate a warning if the return value is unused.

You specify an attribute by writing the `@` symbol followed by the attribute's name and any arguments that the attribute accepts:

```
@attribute name
@attribute name(attribute arguments)
```

Some declaration attributes accept arguments that specify more information about the attribute and how it applies to a particular declaration. These *attribute arguments* are enclosed in parentheses, and their format is defined by the attribute they belong to.

Declaration Attributes

You can apply a declaration attribute to declarations only.

available

Apply this attribute to indicate a declaration's life cycle relative to certain Swift language versions or certain platforms and operating system versions.

The `available` attribute always appears with a list of two or more comma-separated attribute arguments. These arguments begin with one of the following platform or language names:

- `iOS`
- `iOSApplicationExtension`
- `macOS`
- `macOSApplicationExtension`
- `watchOS`
- `watchOSApplicationExtension`
- `tvOS`
- `tvOSApplicationExtension`
- `swift`

You can also use an asterisk (`*`) to indicate the availability of the declaration on all of the platform names listed above. An `available` attribute that specifies availability using a Swift version number can't use the asterisk.

The remaining arguments can appear in any order and specify additional information about the declaration's life cycle, including important milestones.

- The `unavailable` argument indicates that the declaration isn't available on the specified platform. This argument can't be used when specifying Swift version availability.

- The `introduced` argument indicates the first version of the specified platform or language in which the declaration was introduced. It has the following form:

  ```
  introduced: version number
  ```

  The *version number* consists of one to three positive integers, separated by periods.

- The `deprecated` argument indicates the first version of the specified platform or language in which the declaration was deprecated. It has the following form:

  ```
  deprecated: version number
  ```

The optional *version number* consists of one to three positive integers, separated by periods. Omitting the version number indicates that the declaration is currently deprecated, without giving any information about when the deprecation occurred. If you omit the version number, omit the colon (`:`) as well.

- The `obsoleted` argument indicates the first version of the specified platform or language in which the declaration was obsoleted. When a declaration is obsoleted, it's removed from the specified platform or language and can no longer be used. It has the following form:

  ```
  obsoleted: version number
  ```

  The *version number* consists of one to three positive integers, separated by periods.

- The `message` argument provides a textual message that the compiler displays when emitting a warning or error about the use of a deprecated or obsoleted declaration. It has the following form:

  ```
  message: message
  ```

  The *message* consists of a string literal.

- The `renamed` argument provides a textual message that indicates the new name for a declaration that's been renamed. The compiler displays the new name when emitting an error about the use of a renamed declaration. It has the following form:

  ```
  renamed: new name
  ```

  The *new name* consists of a string literal.

  You can apply the `available` attribute with the `renamed` and `unavailable` arguments to a type alias declaration, as shown below, to indicate that the name of a declaration changed between releases of a framework or library. This combination results in a compile-time error that the declaration has been renamed.

  ```
  1 // First release
  ```

```
protocol MyProtocol {
    // protocol definition
}
```

```
// Subsequent release renames MyProtocol
protocol MyRenamedProtocol {
    // protocol definition
}

@available(*, unavailable, renamed: "MyRenamedProtocol")
typealias MyProtocol = MyRenamedProtocol
```

You can apply multiple `available` attributes on a single declaration to specify the declaration's availability on different platforms and different versions of Swift. The declaration that the `available` attribute applies to is ignored if the attribute specifies a platform or language version that doesn't match the current target. If you use multiple `available` attributes, the effective availability is the combination of the platform and Swift availabilities.

If an `available` attribute only specifies an `introduced` argument in addition to a platform or language name argument, you can use the following shorthand syntax instead:

```
@available(platform name version number, *)
@available(swift version number)
```

The shorthand syntax for `available` attributes concisely expresses availability for multiple platforms. Although the two forms are functionally equivalent, the shorthand form is preferred whenever possible.

```
@available(iOS 10.0, macOS 10.12, *)
class MyClass {
    // class definition
}
```

An `available` attribute that specifies availability using a Swift version number can't additionally specify a declaration's platform availability. Instead, use separate `available` attributes to specify a Swift version availability and one or more platform availabilities.

```
@available(swift 3.0.2)
@available(macOS 10.12, *)
struct MyStruct {
    // struct definition
}
```

discardableResult

Apply this attribute to a function or method declaration to suppress the compiler warning when the function or method that returns a value is called without using its result.

dynamicCallable

Apply this attribute to a class, structure, enumeration, or protocol to treat instances of the type as callable functions. The type must implement either a `dynamicallyCall(withArguments:)` method, a `dynamicallyCall(withKeywordArguments:)` method, or both.

You can call an instance of a dynamically callable type as if it's a function that takes any number of arguments.

```
@dynamicCallable
struct TelephoneExchange {
    func dynamicallyCall(withArguments phoneNumber: [Int]) {
        if phoneNumber == [4, 1, 1] {
            print("Get Swift help on forums.swift.org")
        } else {
            print("Unrecognized number")
        }
    }
}

let dial = TelephoneExchange()

// Use a dynamic method call.
dial(4, 1, 1)
// Prints "Get Swift help on forums.swift.org"

```

```
dial(8, 6, 7, 5, 3, 0, 9)
// Prints "Unrecognized number"

// Call the underlying method directly.
dial.dynamicallyCall(withArguments: [4, 1, 1])
```

The declaration of the `dynamicallyCall(withArguments:)` method must have a single parameter that conforms to the `ExpressibleByArrayLiteral` protocol—like `[Int]` in the example above. The return type can be any type.

You can include labels in a dynamic method call if you implement the `dynamicallyCall(withKeywordArguments:)` method.

```
@dynamicCallable
struct Repeater {
    func dynamicallyCall(withKeywordArguments pairs:
  KeyValuePairs<String, Int>) -> String {
        return pairs
            .map { label, count in
                repeatElement(label, count:
  count).joined(separator: " ")
            }
            .joined(separator: "\n")
    }
}

let repeatLabels = Repeater()
print(repeatLabels(a: 1, b: 2, c: 3, b: 2, a: 1))
// a
// b b
// c c c
// b b
// a
```

The declaration of the `dynamicallyCall(withKeywordArguments:)` method must have a single parameter that conforms to the `ExpressibleByDictionaryLiteral` protocol, and the return type can be any type. The parameter's `Key` must be `ExpressibleByStringLiteral`. The previous example uses `KeyValuePairs` as the parameter type so that callers can include duplicate parameter labels—`a` and `b` appear multiple times in the call to `repeat`.

If you implement both `dynamicallyCall` methods, `dynamicallyCall(withKeywordArguments:)` is called when the method call includes keyword arguments. In all other cases, `dynamicallyCall(withArguments:)` is called.

You can only call a dynamically callable instance with arguments and a return value that match the types you specify in one of your `dynamicallyCall` method implementations. The call in the following example doesn't compile because there isn't an implementation of `dynamicallyCall(withArguments:)` that takes `KeyValuePairs<String, String>`.

```
repeatLabels(a: "four") // Error
```

dynamicMemberLookup

Apply this attribute to a class, structure, enumeration, or protocol to enable members to be looked up by name at runtime. The type must implement a `subscript(dynamicMemberLookup:)` subscript.

In an explicit member expression, if there isn't a corresponding declaration for the named member, the expression is understood as a call to the type's `subscript(dynamicMemberLookup:)` subscript, passing information about the member as the argument. The subscript can accept a parameter that's either a key path or a member name; if you implement both subscripts, the subscript that takes key path argument is used.

An implementation of `subscript(dynamicMemberLookup:)` can accept key paths using an argument of type `KeyPath`, `WritableKeyPath`, or `ReferenceWritableKeyPath`. It can accept member names using an argument of a type that conforms to the `ExpressibleByStringLiteral` protocol—in most cases, `String`. The subscript's return type can be any type.

Dynamic member lookup by member name can be used to create a wrapper type around data that can't be type checked at compile time, such as when bridging data from other languages into Swift. For example:

```
@dynamicMemberLookup
struct DynamicStruct {
    let dictionary = ["someDynamicMember": 325,
                      "someOtherMember": 787]
    subscript(dynamicMember member: String) -> Int {
        return dictionary[member] ?? 1054
    }
}
let s = DynamicStruct()

// Use dynamic member lookup.
let dynamic = s.someDynamicMember
print(dynamic)
// Prints "325"

// Call the underlying subscript directly.
let equivalent = s[dynamicMember: "someDynamicMember"]
print(dynamic == equivalent)
// Prints "true"
```

Dynamic member lookup by key path can be used to implement a wrapper type in a way that supports compile-time type checking. For example:

```
struct Point { var x, y: Int }

@dynamicMemberLookup
struct PassthroughWrapper<Value> {
    var value: Value
    subscript<T>(dynamicMember member: KeyPath<Value, T>) ->
  T {
        get { return value[keyPath: member] }
    }
}

let point = Point(x: 381, y: 431)
```

```
let wrapper = PassthroughWrapper(value: point)
print(wrapper.x)
```

GKInspectable

Apply this attribute to expose a custom GameplayKit component property to the SpriteKit editor UI. Applying this attribute also implies the `objc` attribute.

inlinable

Apply this attribute to a function, method, computed property, subscript, convenience initializer, or deinitializer declaration to expose that declaration's implementation as part of the module's public interface. The compiler is allowed to replace calls to an inlinable symbol with a copy of the symbol's implementation at the call site.

Inlinable code can interact with `public` symbols declared in any module, and it can interact with `internal` symbols declared in the same module that are marked with the `usableFromInline` attribute. Inlinable code can't interact with `private` or `fileprivate` symbols.

This attribute can't be applied to declarations that are nested inside functions or to `fileprivate` or `private` declarations. Functions and closures that are defined inside an inlinable function are implicitly inlinable, even though they can't be marked with this attribute.

nonobjc

Apply this attribute to a method, property, subscript, or initializer declaration to suppress an implicit `objc` attribute. The `nonobjc` attribute tells the compiler to make the declaration unavailable in Objective-C code, even though it's possible to represent it in Objective-C.

Applying this attribute to an extension has the same effect as applying it to every member of that extension that isn't explicitly marked with the `objc` attribute.

You use the `nonobjc` attribute to resolve circularity for bridging methods in a class marked with the `objc` attribute, and to allow overloading of methods and initializers in a class marked with the `objc` attribute.

A method marked with the `nonobjc` attribute can't override a method marked with the `objc` attribute. However, a method marked with the `objc` attribute can override a method marked with the `nonobjc` attribute. Similarly, a method marked with the `nonobjc` attribute can't satisfy a protocol requirement for a method marked with the `objc` attribute.

NSApplicationMain

Apply this attribute to a class to indicate that it's the application delegate. Using this attribute is equivalent to calling the `NSApplicationMain(_:_:)` function.

If you don't use this attribute, supply a `main.swift` file with code at the top level that calls the `NSApplicationMain(_:_:)` function as follows:

```
import AppKit
NSApplicationMain(CommandLine.argc, CommandLine.unsafeArgv)
```

NSCopying

Apply this attribute to a stored variable property of a class. This attribute causes the property's setter to be synthesized with a *copy* of the property's value—returned by the `copyWithZone(_:)` method—instead of the value of the property itself. The type of the property must conform to the `NSCopying` protocol.

The `NSCopying` attribute behaves in a way similar to the Objective-C `copy` property attribute.

NSManaged

Apply this attribute to an instance method or stored variable property of a class that inherits from `NSManagedObject` to indicate that Core Data dynamically provides its implementation at runtime, based on the associated entity description. For a property marked with the `NSManaged` attribute, Core Data also provides the storage at runtime. Applying this attribute also implies the `objc` attribute.

objc

Apply this attribute to any declaration that can be represented in Objective-C—for example, nonnested classes, protocols, nongeneric enumerations (constrained to integer raw-value types), properties and methods (including getters and setters) of classes, protocols and optional members of a protocol, initializers, and subscripts. The `objc` attribute tells the compiler that a declaration is available to use in Objective-C code.

Applying this attribute to an extension has the same effect as applying it to every member of that extension that isn't explicitly marked with the `nonobjc` attribute.

The compiler implicitly adds the `objc` attribute to subclasses of any class defined in Objective-C. However, the subclass must not be generic, and must not inherit from any generic classes. You can explicitly add the `objc` attribute to a subclass that meets these criteria, to specify its Objective-C name as discussed below. Protocols that are marked with the `objc` attribute can't inherit from protocols that aren't marked with this attribute.

The `objc` attribute is also implicitly added in the following cases:

- The declaration is an override in a subclass, and the superclass's declaration has the `objc` attribute.
- The declaration satisfies a requirement from a protocol that has the `objc` attribute.
- The declaration has the `IBAction`, `IBSegueAction`, `IBOutlet`, `IBDesignable`, `IBInspectable`, `NSManaged`, or `GKInspectable` attribute.

If you apply the `objc` attribute to an enumeration, each enumeration case is exposed to Objective-C code as the concatenation of the enumeration name and the case name. The first letter of the case name is capitalized. For example, a case named `venus` in a Swift `Planet` enumeration is exposed to Objective-C code as a case named `PlanetVenus`.

The `objc` attribute optionally accepts a single attribute argument, which consists of an identifier. The identifier specifies the name to be exposed to Objective-C for the entity that the `objc` attribute applies to. You can use this argument to name classes, enumerations, enumeration cases, protocols, methods, getters, setters, and initializers. If you specify the Objective-C name for a class, protocol, or enumeration, include a three-letter prefix on the name, as described in Conventions in Programming with Objective-C. The example below exposes the getter for the `enabled` property of the `ExampleClass` to Objective-C code as `isEnabled` rather than just as the name of the property itself.

```
class ExampleClass: NSObject {
    @objc var enabled: Bool {
        @objc(isEnabled) get {
            // Return the appropriate value
        }
    }
}
```

objcMembers

Apply this attribute to a class declaration, to implicitly apply the `objc` attribute to all Objective-C compatible members of the class, its extensions, its subclasses, and all of the extensions of its subclasses.

Most code should use the `objc` attribute instead, to expose only the declarations that are needed. If you need to expose many declarations, you can group them in an extension that has the `objc` attribute. The `objcMembers` attribute is a convenience for libraries that make heavy use of the introspection facilities of the Objective-C runtime. Applying the `objc` attribute when it isn't needed can increase your binary size and adversely affect performance.

requires_stored_property_inits

Apply this attribute to a class declaration to require all stored properties within the class to provide default values as part of their definitions. This attribute is inferred for any class that inherits from `NSManagedObject`.

testable

Apply this attribute to an `import` declaration to import that module with changes to its access control that simplify testing the module's code. Entities in the imported module that are marked with the `internal` access-level modifier are imported as if they were declared with the `public` access-level modifier. Classes and class members that are marked with the `internal` or `public` access-level modifier are imported as if they were declared with the `open` access-level modifier. The imported module must be compiled with testing enabled.

UIApplicationMain

Apply this attribute to a class to indicate that it's the application delegate. Using this attribute is equivalent to calling the `UIApplicationMain` function and passing this class's name as the name of the delegate class.

If you don't use this attribute, supply a `main.swift` file with code at the top level that calls the `UIApplicationMain(_:_:_:_:)` function. For example, if your app uses a custom subclass of `UIApplication` as its principal class, call the `UIApplicationMain(_:_:_:_:)` function instead of using this attribute.

usableFromInline

Apply this attribute to a function, method, computed property, subscript, initializer, or deinitializer declaration to allow that symbol to be used in inlinable code that's defined in the same module as the declaration. The declaration must have the `internal` access level modifier.

Like the `public` access level modifier, this attribute exposes the declaration as part of the module's public interface. Unlike `public`, the compiler doesn't allow declarations marked with `usableFromInline` to be referenced by name in code outside the module, even though the declaration's symbol is exported. However, code outside the module might still be able to interact with the declaration's symbol by using runtime behavior.

Declarations marked with the `inlinable` attribute are implicitly usable from inlinable code. Although either `inlinable` or `usableFromInline` can be applied to `internal` declarations, applying both attributes is an error.

warn_unqualified_access

Apply this attribute to a top-level function, instance method, or class or static method to trigger warnings when that function or method is used without a preceding qualifier, such as a module name, type name, or instance variable or constant. Use this attribute to help discourage ambiguity between functions with the same name that are accessible from the same scope.

For example, the Swift standard library includes both a top-level `min(_:_:)` function and a `min()` method for sequences with comparable elements. The sequence method is declared with the `warn_unqualified_access` attribute to help reduce confusion when attempting to use one or the other from within a `Sequence` extension.

Declaration Attributes Used by Interface Builder

Interface Builder attributes are declaration attributes used by Interface Builder to synchronize with Xcode. Swift provides the following Interface Builder attributes: `IBAction`, `IBSegueAction`, `IBOutlet`, `IBDesignable`, and `IBInspectable`. These attributes are conceptually the same as their Objective-C counterparts.

You apply the `IBOutlet` and `IBInspectable` attributes to property declarations of a class. You apply the `IBAction` and `IBSegueAction` attribute to method declarations of a class and the `IBDesignable` attribute to class declarations.

Applying the `IBAction`, `IBSegueAction`, `IBOutlet`, `IBDesignable`, or `IBInspectable` attribute also implies the `objc` attribute.

Type Attributes

You can apply type attributes to types only.

autoclosure

Apply this attribute to delay the evaluation of an expression by automatically wrapping that expression in a closure with no arguments. You apply it to a parameter's type in a method or function declaration, for a parameter whose type is a function type that takes no arguments and that returns a value of the type of the expression. For an example of how to use the `autoclosure` attribute, see Autoclosures and Function Type.

convention

Apply this attribute to the type of a function to indicate its calling conventions.

The `convention` attribute always appears with one of the following arguments:

- The `swift` argument indicates a Swift function reference. This is the standard calling convention for function values in Swift.
- The `block` argument indicates an Objective-C compatible block reference. The function value is represented as a reference to the block object, which is an `id`-compatible Objective-C object that embeds its invocation function within the object. The invocation function uses the C calling convention.
- The `c` argument indicates a C function reference. The function value carries no context and uses the C calling convention.

With a few exceptions, a function of any calling convention can be used when a function any other calling convention is needed. A nongeneric global function, a local function that doesn't capture any local variables or a closure that doesn't capture any local variables can be converted to the C calling convention. Other Swift functions can't be converted to the C calling convention. A function with the Objective-C block calling convention can't be converted to the C calling convention.

escaping

Apply this attribute to a parameter's type in a method or function declaration to indicate that the parameter's value can be stored for later execution. This means that the value is allowed to outlive the lifetime of the call. Function type parameters with the `escaping` type attribute require explicit use of `self.` for properties or methods. For an example of how to use the `escaping` attribute, see Escaping Closures.

Switch Case Attributes

You can apply switch case attributes to switch cases only.

unknown

Apply this attribute to a switch case to indicate that it isn't expected to be matched by any case of the enumeration that's known at the time the code is compiled. For an example of how to use the `unknown` attribute, see Switching Over Future Enumeration Cases.

GRAMMAR OF AN ATTRIBUTE

attribute → **@** attribute-name attribute-argument-clause$_{opt}$
attribute-name → identifier
attribute-argument-clause → **(** balanced-tokens$_{opt}$ **)**
attributes → attribute attributes$_{opt}$

balanced-tokens → balanced-token balanced-tokens$_{opt}$
balanced-token → **(** balanced-tokens$_{opt}$ **)**
balanced-token → **[** balanced-tokens$_{opt}$ **]**
balanced-token → **{** balanced-tokens$_{opt}$ **}**
balanced-token → Any identifier, keyword, literal, or operator
balanced-token → Any punctuation except **(**, **)**, **[**, **]**, **{**, or **}**

‹ Declarations Patterns ›